

Einführung in die Programmierung mit R

*Andrew Ellis
Boris Mayer*

2019-03-05

Contents

Vorwort	5
0.1 Voraussetzungen	5
0.2 Was ist R?	5
0.3 Weiterführende Literatur	5
0.4 Typographische Konventionen	6
0.5 License	6
1 RStudio Workflow	7
1.1 Graphische Benutzeroberfläche	7
1.2 Packages	9
1.3 Help	11
1.4 Arbeiten mit RStudio	12
2 Die R Sprache	19
2.1 Operatoren und Funktionen	19
2.2 Variable definieren	26
2.3 Funktionen aufrufen	27
2.4 Datentypen	29
2.5 Übungsaufgaben	43
3 Datensätze	49
3.1 Datensätze selber erstellen	49
3.2 Daten importieren	57
3.3 Übungsaufgaben	61
4 Daten transformieren	65
4.1 Tidy data	65
4.2 Der Pipe Operator	65
4.3 Reshaping: <code>tidyverse</code>	68
4.4 Daten manipulieren: <code>dplyr</code>	72
4.5 Übungsaufgaben	83
5 Grafiken mit ggplot2	89
5.1 Schritt 1: Plot-Objekt erstellen	90
5.2 Schritt 2: Aesthetic mappings	91
5.3 Schritt 3: geoms hinzufügen	92
5.4 Geoms für verschiedene Datentypen	97
5.5 Facets	114
5.6 Farben und Themes	117
5.7 Beschriftungen	120
5.8 Grafiken speichern	120
5.9 Übungsaufgabe	121

6 Statistische Verteilungen	123
6.1 Normalverteilung	123
6.2 <i>t</i> -Verteilung	128
6.3 Chi-Quadrat-Verteilung	131
6.4 <i>F</i> -Verteilung	132
6.5 Übungsaufgaben	134
7 Funktionen	137
7.1 Schiefe (skewness)	137
7.2 Eigene Funktionen definieren	141
7.3 Skewness Funktion	142
7.4 Weitere Beispiele	143
7.5 Übungsbeispiele	144
8 Deskriptive Statistik	145
8.1 Zusammenfassung von Verteilungskennwerten aller Variablen in der Datenmatrix	145
8.2 Deskriptivstatistik für nominalskalierte Variablen	148
8.3 Deskriptivstatistik für ordinale und metrische Variablen	154
8.4 Kovarianz- und Korrelationsanalyse	161
8.5 Reliabilitätsanalyse mit dem psych-Package	168
9 Mittelwertsvergleiche	175
9.1 Vergleich eines Stichprobenmittelwerts mit einem fixen Wert (μ_0) - <i>t</i> -Test für eine Stichprobe	175
9.2 Vergleich zweier Stichprobenmittelwerte	176
9.3 Vergleich mehrerer Stichprobenmittelwerte	184
10 Lineare Modelle	201
10.1 Faktorielle ANOVA	201
10.2 Multiple Regression	206
10.3 Übungsaufgaben:	219

Vorwort

0.1 Voraussetzungen

Bitte installieren Sie sowohl die aktuelle Version von R: R version 3.5.2 (2018-12-20) als auch RStudio: RStudio Desktop

Wir verwenden für diese Vorlesung RStudio, ein **integrated development environment** (IDE), welches die Arbeit mit R sehr angenehm macht. Das R Programm muss separat installiert werden, wir werden aber nicht direkt damit arbeiten.

0.2 Was ist R?

R ist sowohl eine Programmiersprache als auch eine Statistikumgebung. R ist open-source, d.h. der Source Code ist unter der GNU Public License frei verfügbar. Ausserdem ist R kostenlos.

Wo kommt der Name “R” her? R wurde als open-source Variante einer kommerziellen Sprache entwickelt, welche S heisst (Programmiersprachen haben oft nur einen Buchstaben als Namen, z.B. C). Die beiden R Entwickler (Ross Ihaka und Robert Gentleman) nannten angeblich die Sprache R, weil ihre Vornamen beide mit dem Buchstaben R beginnen.

Wir werden R primär als Statistikumgebung kennenlernen; wir werden uns jedoch auch teilweise mit R als Programmiersprache beschäftigen. Das bedeutet, dass wir am Anfang verschiedene Datentypen und ein wenig R Syntax kennenlernen, damit wir richtig damit arbeiten können.

Die R Sprache gilt als relativ “schwierig” zu lernen, unter anderem, weil man sich viele verschiedene Funktionsnamen merken muss, und diese eine etwas inkonsistente Namensgebung haben. Wir orientieren uns deswegen, soweit möglich, an einer Sammlung von modernen R Packages (Erweiterungspakete), welche von RStudio, insbesondere von Hadley Wickham, entwickelt wurden.

Aber bitte lassen Sie sich nicht abschrecken. Diese Packages repräsentieren den ‘state-of-the-art’, was Datenanalyse anbelangt, und die Arbeit damit ist einfach erlernbar. Es gibt beinahe für alle Probleme ein dafür entsprechendes Packet mit einer Lösung. Nach wenigen Anwendungen wird die “Programmierung” intuitiv.

0.3 Weiterführende Literatur

Wir orientieren uns inhaltlich teilweise an dem Buch R für Einsteiger: Einführung in die Statistiksoftware für die Sozialwissenschaften von Maike Luhmann, verwenden aber nicht deren R Code.

In Bezug auf R Code orientieren uns wir und an dem online frei verfügbaren Buch R for Data Science von Garrett Grolemund und Hadley Wickham. Dieses Buch ist jedoch weit umfangreicher, als wir für diese Vorlesung brauchen.

Für diejenigen, welche sich mit R als Programmiersprache auseinandersetzen wollen, empfehlen wir die Bücher Hands-On Programming with R von Garrett Grolemund und Advanced R von Hadley Wickham. Das letztere ist jedoch wirklich nur für Vertiefer.

DataCamp bietet verschiedene Online-Kurse an (teilweise kostenpflichtig). Dieser Einführungskurs ist jedoch kostenlos.

0.4 Typographische Konventionen

Wir verwenden zusätzlich zum Haupttext folgende Textblöcke:

- In diesem Block stehen Kommentare und Erläuterungen. ■
- In diesem Block stehen Zusatzinformationen. Oft sind diese für Leute gedacht, welche ihr Wissen vertiefen möchten, und sind **nicht** prüfungsrelevant. ■
- In diesem Block stehen Übungen. ■

Diese Unterlagen bestehen zu einem grossen Teil aus R Code. Code chunks sehen so aus:

```
x <- seq(from = 1, to = 10, by = 1)
```

Dieser Code kann in der R Konsole ausgeführt werden. Code chunks können auch einen Output haben:

```
x  
#> [1] 1 2 3 4 5 6 7 8 9 10
```

In einem solchen Block ist x der Input und #> [1] 1 2 3 4 5 6 7 8 9 10 der Output (in diesem Beispiel haben wir eine Variable x kreiert und ihr die Sequenz 1 bis 10 zugewiesen).

0.5 License

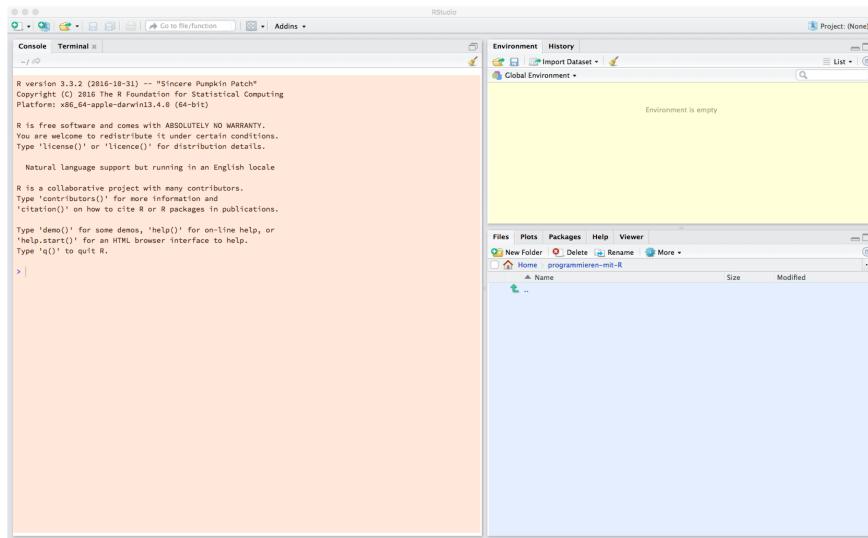
This work is licensed under a Creative Commons Attribution 4.0 International License.

Chapter 1

RStudio Workflow

1.1 Graphische Benutzeroberfläche

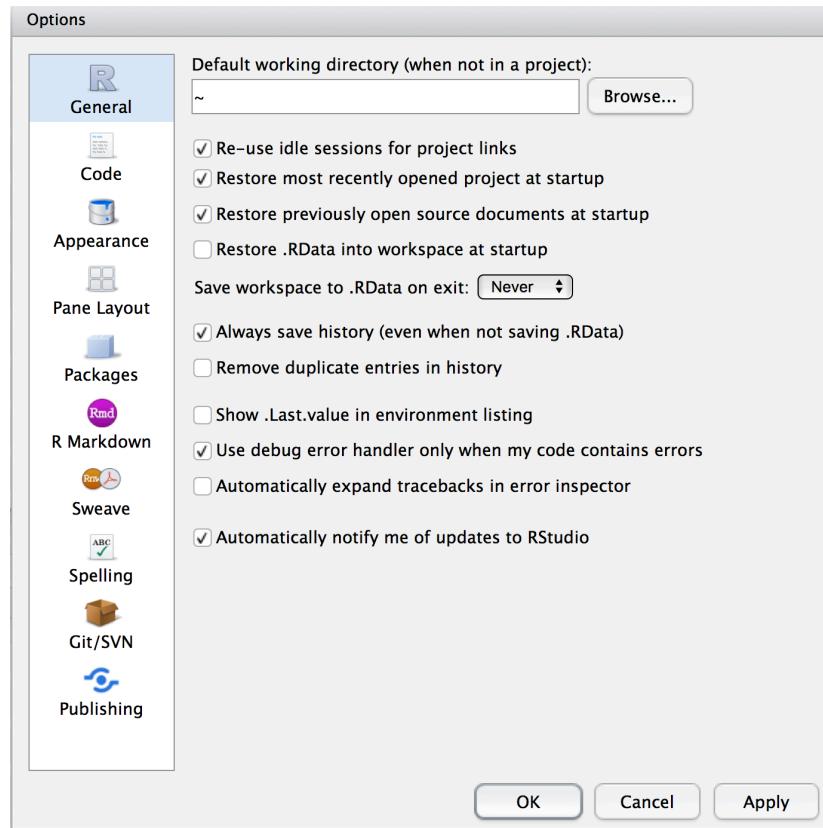
Bevor wir mit dem Inhalt anfangen, sollten wir uns ein wenig mit RStudio anfreunden. Öffnen Sie nun RStudio. Das Programm sollte ungefähr so aussehen:



Unter dem Menu **Preferences** oder **Global Options** kann man RStudio den eigenen Präferenzen anpassen. Wir empfehlen aber dringend, folgende zwei Optionen **nicht** zu aktivieren:

- 1) Restore .RData into workspace at startup (sollte nicht ausgewählt sein)
- 2) Save workspace to .RData on exit (Never)

Diese Optionen sind zwar manchmal verlockend, da damit alle Daten und Variablen automatisch wieder geladen werden. Es ist aber meistens besser, mit einem leeren Workspace zu beginnen, und die ausgeführten R Befehle in einem Script oder Notebook festzuhalten.



Das RStudio GUI (Graphical User Interface) besteht aus mehreren Bereichen. Auf der linken Seite (rot markiert) erscheint die **R Konsole**. Hier kann R Code eingegeben werden, und wird dann von R interpretiert (ausgeführt). Das > Zeichen ist die R Prompt (Aufforderungszeichen). Wir können so interaktiv mit R arbeiten, und als Taschenrechner benützen, z.B.

```
2 + 3
#> [1] 5
exp(1)
#> [1] 2.718282
3/4
#> [1] 0.75
```

- Wenn wir $3/4$ eingeben, erscheint im Output [1] 0.75. Dies bedeutet, dass der Output mit dem ersten Element eines Vektor beginnt. Dies ist gleichzeitig das einzige Element, und ist somit ein Skalar (ein Skalar in R ist einfach ein Vektor mit der Länge 1). ■

Rechts oben (gelb markiert) erscheinen 2 Bereiche, **Environment** und **History**. Im **Environment**-Bereich sehen wir alle Variablen, Datensätze und Funktionen, welche wir definiert haben; diese erscheinen im **Global Environment** (drop-down menu). Wenn Sie auf dieses Menu klicken, sehen Sie welche **Packages** geladen sind. Wenn Sie z.B. das **stats** Package auswählen, sehen Sie alle Funktionen, welche durch dieses Package verfügbar gemacht werden. Wir werden in Kürze mehr über Packages erfahren.

The screenshot shows the RStudio interface with the 'Environment' tab selected. At the top, there are icons for creating a new file, opening an existing one, and importing datasets. Below that, a dropdown menu shows 'package:stats'. The main area is titled 'Values' and lists various R functions along with their types:

acf	<Promise>
acf2AR	<Promise>
add.scope	<Promise>
addmargins	<Promise>
aggregate.data.frame	<Promise>
aggregate.ts	<Promise>
aov	<Promise>
approx	<Promise>
approxfun	<Promise>
ar	<Promise>
ar.mle	<Promise>
ar.ols	<Promise>
arima	<Promise>

Im **History**-Bereich sehen Sie alle Befehle, die Sie bisher ausgeführt haben.

Rechts unten (blau markiert) erscheinen die Bereiche **Files** (Dateimanager), **Plots**, **Packages** und der **Help Viewer**.

Im Dateimanager können Sie das Arbeitsverzeichnis (working directory) wechseln, und Dateien öffnen:

The screenshot shows the RStudio 'Files' tab. A context menu is open over a folder named 'programmieren-mit-R'. The menu options are: Copy..., Copy To..., Move..., Set As Working Directory (which is highlighted in blue), Go To Working Directory, and Show Folder in New Window. The folder contains two files: 'demo-notebook.Rmd' and 'demo-notebook.nb.html'. To the right, a list of files in the folder shows their sizes and modification dates:

Size	Modified
869 B	Feb 14, 2017, 1:01 PM
773.4 KB	Feb 14, 2017, 1:01 PM

- Wenn Sie im Dateimanager das Arbeitsverzeichnis wechseln, erscheint in der Konsole die Syntax für den Befehl, den R soeben ausgeführt hat. In diesem Fall ist das `setwd(...)`. Mit `getwd()` kann man R fragen, in welchem Arbeitsverzeichnis man sich befindet. ■

1.2 Packages

Bevor wir anfangen, mit RStudio zu arbeiten, müssen wir einige **Packages** installieren. Packages stellen zusätzliche Funktionen zur Verfügung, welche nicht in der Basisausstattung von R (base R) enthalten sind. Wir installieren zunächst eine Sammlung von Packages zur Datenmanipulation (`tidyverse`, `dplyr`), zum Im-

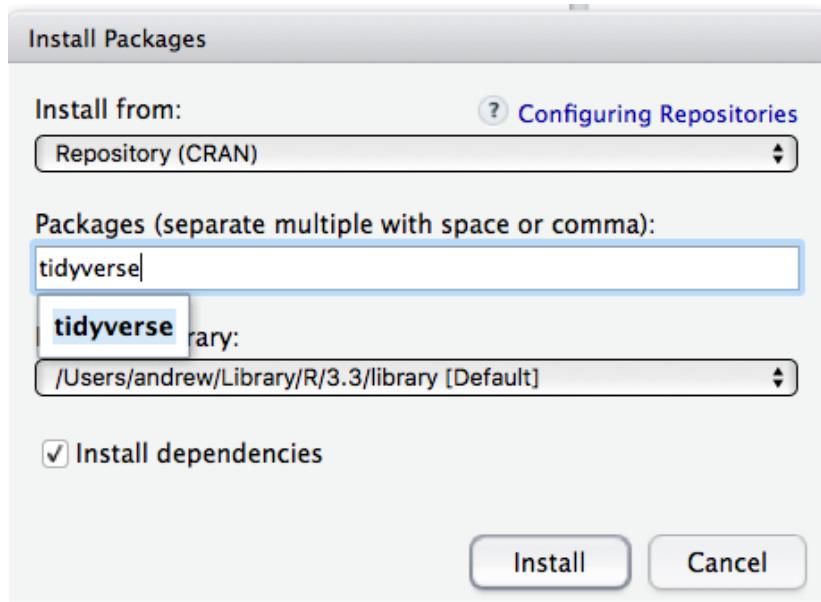
portieren von SPSS Files (`haven`) und für Grafiken (`ggplot2`). Diese können alle zusammen mit diesem Befehl installiert werden, welchen wir in der Konsole eingeben:

```
install.packages("tidyverse")
```

Anschliessend können wir die Packages so laden:

```
library(tidyverse)
#> -- Attaching packages ----- tidyverse 1.2.1 --
#> v ggplot2 3.1.0      v purrrr   0.3.1
#> v tibble  2.0.1      v dplyr    0.8.0.1
#> v tidyr   0.8.3      v stringr  1.4.0
#> v readr   1.3.1      v forcats  0.4.0
#> -- Conflicts ----- tidyverse_conflicts() --
#> x dplyr::filter() masks stats::filter()
#> x dplyr::lag()   masks stats::lag()
```

Als Alternative dazu können Packages über das GUI installiert



und geladen werden.

	Name	Description	Version	X
<input checked="" type="checkbox"/>	tidyverse	Easily Install and Load 'Tidyverse' Packages	1.1.1	X
<input type="checkbox"/>	broom	Convert Statistical Analysis Objects into Tidy Data Frames	0.4.1	X
<input checked="" type="checkbox"/>	tidyshiny	Interface for manipulating data with the tidyverse package	0.1.0	X
<input checked="" type="checkbox"/>	tidyverse	Easily Tidy Data with 'spread()' and 'gather()' Functions	0.6.1	X

Ausserdem können so alle Packages aktualisiert werden (**Update**) - es empfiehlt sich, dies regelmässig zu tun.

- Bitte installieren Sie nun die `tidyverse` Packages. Weitere Packages werden wir installieren, wenn wir sie benötigen. ■

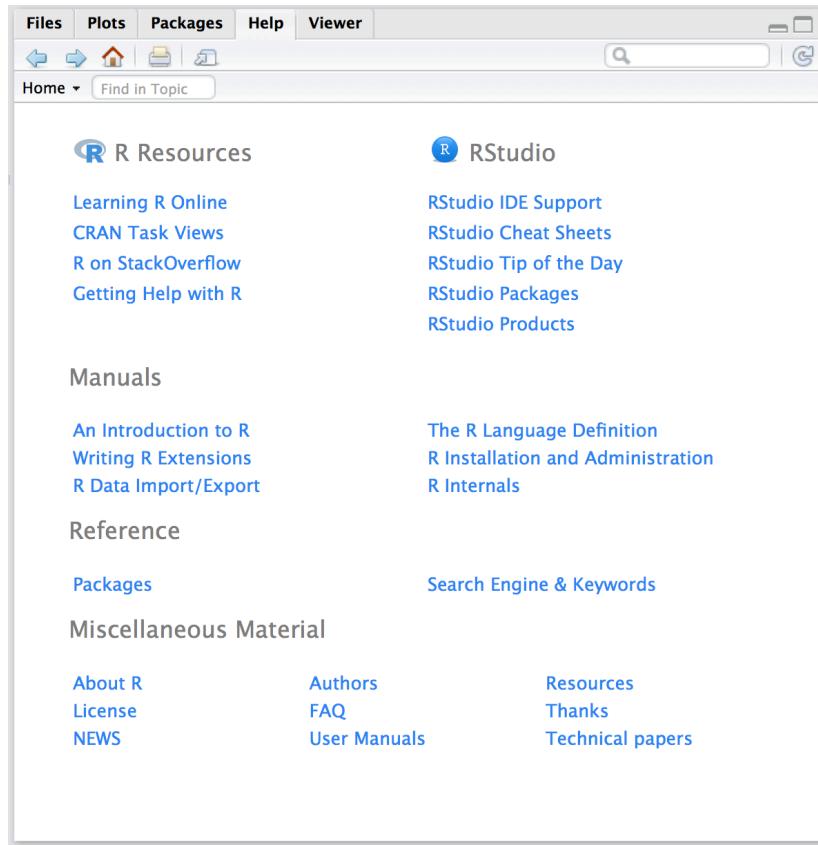
R Packages werden auf einem Server bereitgestellt: The Comprehensive R Archive Network, oder CRAN.

Unter Task Views sehen Sie Sammlungen von Packages zu bestimmten Themen; die Task View zum Thema Psychometrik finden Sie hier: CRAN Task View: Psychometric Models and Methods.

Da die meisten Statistiker mit R arbeiten, gibt es fast zu jedem Thema ein R Package, oder zumindest R Code, den man mit einer gezielten Suche im Internet finden kann (siehe auch nächster Abschnitt).

1.3 Help

Selbst wenn man täglich mit R arbeitet, ist es (fast) unmöglich, sich alle Funktionen zu merken. R hat ein sehr gutes, eingebautes Hilfesystem, welches aber für Anfänger nicht ganz leicht verständlich ist. Dies ist im Bereich **Help** zugänglich:



Sehr empfehlenswert ist die Question und Answer website Stackoverflow - hier finden sich Antworten auf (fast) alle möglichen Fragen (es gibt immer jemanden, der das gleiche Problem hatte).

Wenn man gezielt suchen will, kann man im Suchfenster des **Help** Viewers einen Begriff eingeben. Wenn man einfach herausfinden möchte, welche Funktionen in einem installierten Package verfügbar sind, kann man im **Packages**-Bereich nach einem Package suchen, und dann auf dieses klicken, um die Hilfeseiten zu sehen.

■ Probieren Sie es selber aus: suchen Sie nach dem Package `dplyr`. Wenn Sie darauf klicken, sehen sie zuoberst einen Link zu `User guides`, `package vignettes` and `other documentation`. Vignettes sind als Einführungen gedacht. ■

Wenn man in der Konsole schnell Hilfe zu einer Funktion erhalten will, kann man so vorgehen:

```
help(mean)
```

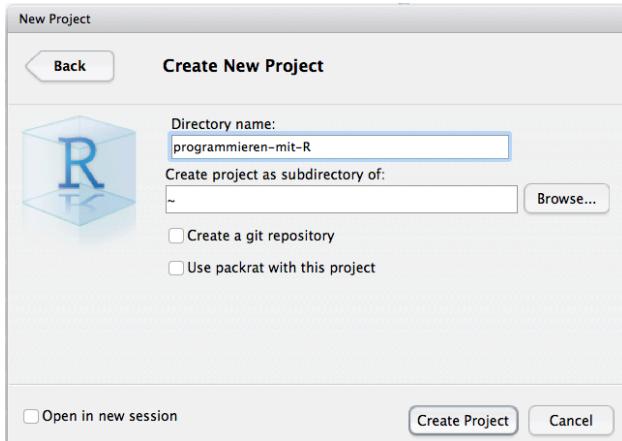
Dies öffnet die Hilfeseite für die `mean` Funktion. Alternativ dazu kann man auch `?mean` eingeben.

Man kann aber auch zu einem Thema Hilfe suchen. Wenn wir z.B. Hilfe zum Thema Zahlen auf- oder abrunden brauchen, können wir `help("round")` eingeben. Weitere Hilfe zur `help()` Funktion finden Sie so: `help(help)`.

1.4 Arbeiten mit RStudio

1.4.1 Projekte

Nun können wir mit der Arbeit beginnen. Es empfiehlt sich, ein **Projekt** anzulegen, über das Menu **File > New Project**. Sie können das Projekt nennen, wie Sie wollen - wir haben den Namen **programmieren-mit-R** gewählt.



Der Vorteil eines Projekts (im Vergleich zu einer einzelnen neuen R Script-Datei) besteht zunächst darin, dass RStudio beim einem Neustart wieder alle Files öffnet. Außerdem kann man beliebig viele Projekte erstellen, und entweder gleichzeitig offen haben, oder zwischen ihnen wechseln. Des weiteren wird beim Öffnen eines Projekts das Arbeitsverzeichnis automatisch auf den Ordner definiert, in dem sich die Projekt-Datei befindet. Man muss sich also dann nicht mehr um das *working directory* kümmern und kann z.B. Daten- oder Skriptdateien direkt vom Projektordner aus aufrufen, ohne jedes Mal den vollständigen Pfad definieren zu müssen.

1.4.2 Konsole

Es gibt im Wesentlichen zwei Vorgehensweisen, wenn man mit R arbeiten will: man kann 1) entweder Befehle direkt in die Konsole eintippen, oder 2) eine Textdatei öffnen, Befehle in die Datei schreiben, und diese dann an die Konsole schicken.

Mit der Konsole arbeiten ist zu empfehlen, wenn man schnell etwas ausprobieren will, oder wenn man Befehle wiederholen will (dies kann man mit den Up/Down-Pfeiltasten machen), aber es ist meistens besser, wenn man mit einer Textdatei arbeitet.

- Die **History** der eingegebenen Befehle kann man in der Konsole mit der Tastenkombination **CMD + Up** auf macOS oder **CTRL + Up** anzeigen lassen. ■

Manchmal passiert es, dass man einen Befehl unvollständig eingibt. R wartet dann auf weiteren Input, und zeigt dies mit einem `+` an.

Hier wurde z.B. die Klammer vergessen:

```
> mean(x
+
```

Nun muss man entweder die fehlende Klammer eingeben, um den Befehl zu vervollständigen, oder man kann mit der **ESCAPE**-Taste abbrechen.

Für den Fall, dass R abstürzt, kann man mit Hilfe des Menus **Session > Interrupt R** oder **Session > Terminate R** auswählen.

1.4.3 R Script

Wir öffnen und speichern ein neues R Script (Textdatei mit dem Suffix `.R`). Geben Sie hier nun z.B.

```
2 + 3
#> [1] 5
```

ein, wählen sie den Text aus, und klicken sie auf **Run**. Sie haben auch im Menü **Code** verschiedene Möglichkeiten (und Tastaturkürzel), um Code auszuführen.

Der Output erscheint in der Konsole.

- Geben Sie folgendes ein: `x <- c(101, 105, 99, 87, 102, 98)`, und führen sie den Code aus. ■

Sie haben gerade einen Vektor definiert. Im Output erscheint

```
> x <- c(101, 105, 99, 87, 102, 98)
```

und im **Environment** sehen Sie, dass `x` ein `num [1:6]` ist. Dies bedeutet, dass `x` ein numerischer Vektor der Länge 6 ist.

- Geben Sie nun auch folgendes ein: `boxplot(x)`, und führen sie den Code aus. ■

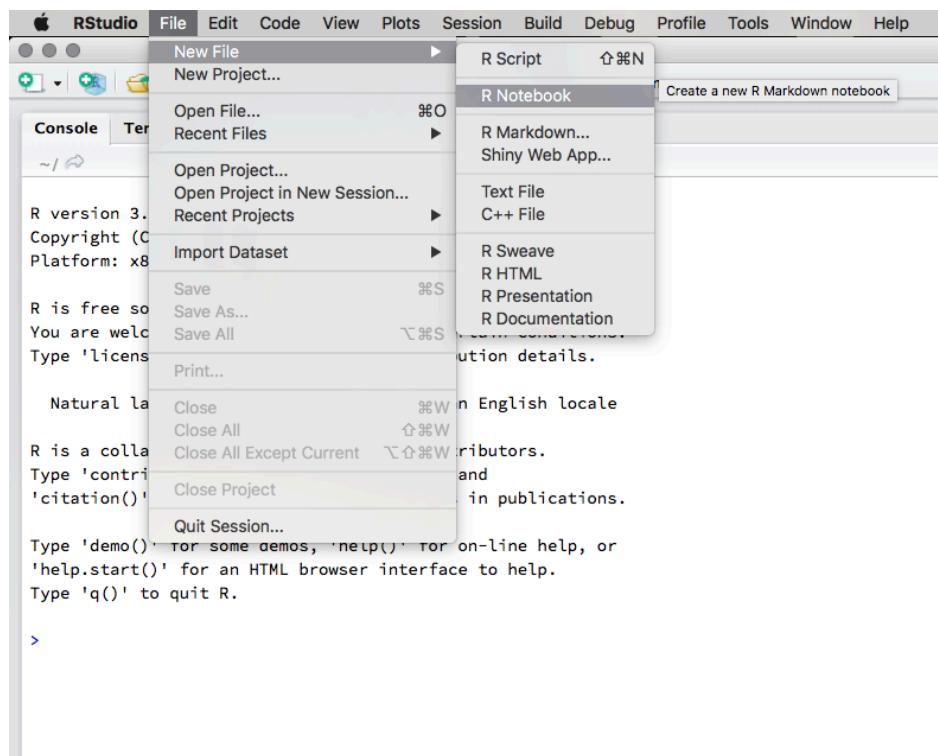
Der Boxplot erscheint im separaten **Plots** Viewer.

R Scripts eignen sich am besten, wenn man längere Programme schreiben oder ganze Datenanalysen speichern will. Für die interaktive Arbeit gibt es jedoch seit kurzem R Notebooks, welche wir uns als nächstes ansehen werden.

1.4.4 R Notebooks verwenden

R Notebooks sind “RMarkdown-interaktiv” (RMarkdown ist eine simple Markup-Sprache), und zeigen Text (Prosa), R Code und Grafiken in demselben Dokument an.

Öffnen Sie eine neue Notebook File:



Sie können dies nun speichern. Notebooks (und alle RMarkdown Dokumente) erhalten das Suffix `.Rmd`.

Das Notebook sollte so aussehen:

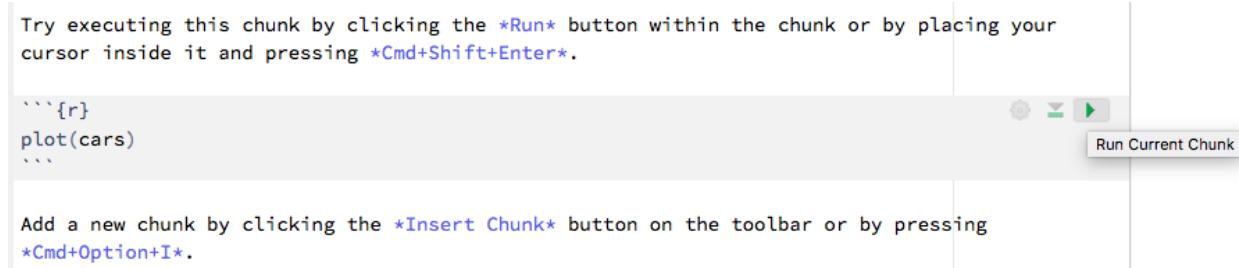
The screenshot shows the RStudio code editor with an R Markdown file open, titled 'test.Rmd'. The file content is as follows:

```

1 ---
2 title: "R Notebook"
3 output: html_notebook
4 ---
5
6 This is an [R Markdown](http://rmarkdown.rstudio.com) Notebook. When you execute code within
the notebook, the results appear beneath the code.
7
8 Try executing this chunk by clicking the *Run* button within the chunk or by placing your
cursor inside it and pressing *Cmd+Shift+Enter*.
9
10 ```{r}
11 plot(cars)
12 ```
13
14 Add a new chunk by clicking the *Insert Chunk* button on the toolbar or by pressing
*Cmd+Option+I*.
15
16 When you save the notebook, an HTML file containing the code and output will be saved
alongside it (click the *Preview* button or press *Cmd+Shift+K* to preview the HTML file).
17
18 The preview shows you a rendered HTML copy of the contents of the editor. Consequently, unlike
*Knit*, *Preview* does not run any R code chunks. Instead, the output of the chunk when it was
last run in the editor is displayed.
19
20

```

Im Notebook sehen Sie sowohl Text mit der R Markdown Markup-Sprache, als auch R code chunks. Diese können ausgeführt werden.



The screenshot shows the RStudio Notebook interface. At the top, there is a message: "Try executing this chunk by clicking the *Run* button within the chunk or by placing your cursor inside it and pressing *Cmd+Shift+Enter*." Below this, a code chunk is displayed:

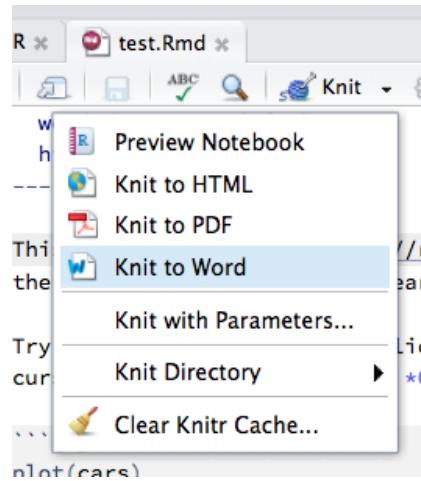
```
```{r}
plot(cars)
...```

```

To the right of the code, there is a toolbar with several icons. One of the icons is a green arrow pointing to the right, labeled "Run Current Chunk". Below the toolbar, another message reads: "Add a new chunk by clicking the \*Insert Chunk\* button on the toolbar or by pressing \*Cmd+Option+I\*."

Wenn Sie auf den grünen Pfeil klicken erscheint der Output direkt unter dem code chunk.

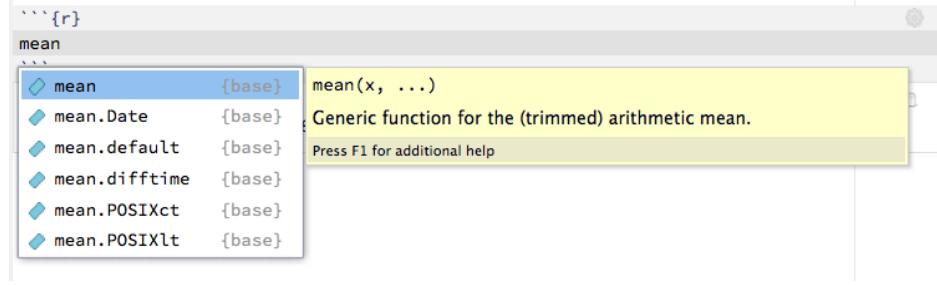
Klicken Sie nun auf **Preview**, oder benützen Sie die entsprechende Tastenkombination: **CMD + Shift + K** (macOS) oder **CTRL + Shift + K** (Windows). RStudio wird Sie nun darauf hinweisen, dass fehlende R Packages installiert werden müssen (dies ist nur beim ersten mal der Fall). Stimmen Sie der Installation zu, das Notebook wird dann anschliessend kompiliert, und es erscheint ein HTML Dokument im **Viewer**. Dieses beinhaltet sowohl Text als auch den evaluierten R Code, und kann als eigenständiges Dokument weitergegeben werden. Ein weiterer Vorteil eines Notebooks: Sie können daraus ein Word-Dokument erstellen:



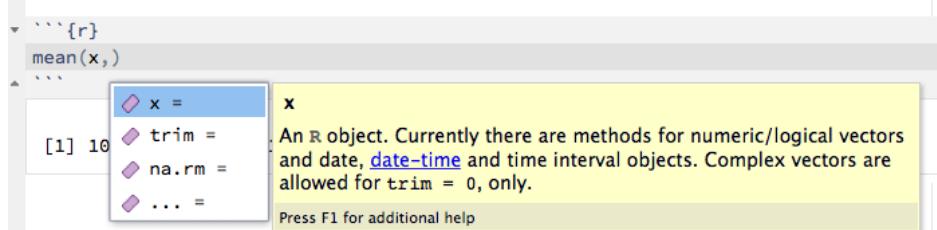
Probieren Sie es selber aus.

### 1.4.5 Tab completion

RStudio verfügt über eine weitere sehr hilfreiche Funktion: **Tab completion**. Wenn man in der Konsole oder in einem Script/Notebook den Anfang eines Befehls eingibt, und dann die **Tabulator**-Taste drückt, erscheint ein Menu mit möglichen **completions**. Wenn wir z.B. einen Mittelwert berechnen wollen, schreiben wir `mean`, und drücken dann Tab. Es erscheint eine List mit möglichen Optionen.



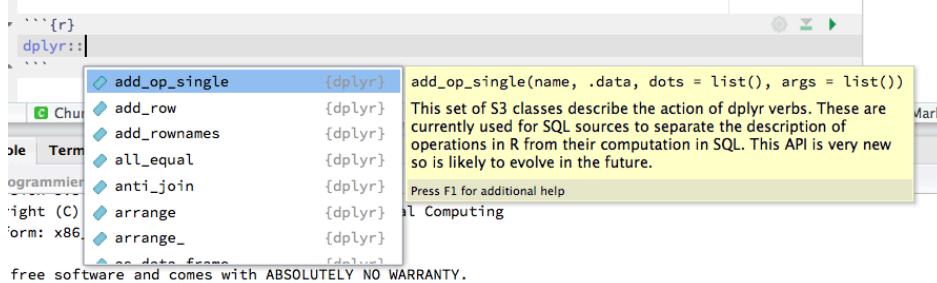
Wenn man ausserdem eine Funktion eingegeben hat, und innerhalb der Klammern Tab drückt, erscheint eine Liste mit den Argumenten dieser Funktion.



Eine weitere nützliche Funktion: wenn wir nicht sicher sind, wie eine Funktion heisst, aber wir wissen, in welchem Package wir sie finden, können wir den Package Namen schreiben, gefolgt von `:::`, und dann Tab drücken. RStudio gibt uns eine Liste mit allen Funktionen dieses Packages. Wenn Sie wissen, dass sie eine Funktion im Package `dplyr` suchen, welche mit `f` anfängt, schreiben Sie einfach

`dplyr::f`

drücken Tab, und RStudio übernimmt den Rest.



#### 1.4.6 Tasten

Wenn wir mit R arbeiten, werden wir folgende Zeichen häufig brauchen:

```
[] Square braces (eckige Klammern)
{ } Curly braces (geschweifte Klammern)
$ Dollar key (Dollarzeichen - für das Subsetting/Auswählen von Variablen benötigt)
Hash (pound) key (Rautezeichen - für Kommentare in R Script Dateien)
~ Tilde (für die Model Notation/Modellformulierung)
| Vertical bar (senkrechter Strich - als logischer Operator benötigt, siehe nächstes Kapitel)
` Backtick (Gravis-Akzent - wird hauptsächlich für code chunks benötigt, muss selten manuell eingegeben)
```

Leider sind manche dieser Zeichen auf Tastaturen mit Schweizerischem Tastaturlayout nicht leicht auffindbar.

- Nehmen Sie sich ein paar Momente Zeit, diese Zeichen auf Ihrer Tastatur zu suchen.  
Tipp: Die öffnende eckige Klammer erhält man mit **ALT + 5** (praktischerweise fügt R die öffnende und die schliessende Klammer in freudiger Erwartung eines Befehls gleich zusammen ein), die schliessende mit **ALT + 6**. Für geschweifte Klammern benutzen Sie analog **ALT + 8** und **ALT + 9**. Das Rautezeichen erhalten Sie mit **ALT + 3**, die Tilde mit **ALT + N**, den senkrechten Strich mit **ALT + 7**.



# Chapter 2

## Die R Sprache

### 2.1 Operatoren und Funktionen

Als erste Anwendung können wir R als Taschenrechner benützen. Zuvor müssen wir uns aber noch ein wenig Vokabular aneignen, und dann ein bisschen Syntax lernen.

Zum Basisvokabular gehören einige eingebaute Operatoren (sowohl arithmetische als auch logische).

#### 2.1.1 Arithmetische Operatoren

Die ersten fünf Operatoren sollten selbsterklärend sein:

+	Addition
-	Subtraktion
*	Multiplikation
/	Division
<sup>^</sup> oder <sup>**</sup>	Potenz

x %*% y	Matrixmultiplikation c(5, 3) %*% c(2, 4) == 22
x %% y	Modulo (x mod y) 5 %% 2 == 1
x %/% y	Ganzzahlige Teilung: 5 %/% 2 == 2

■ Die letzten drei Operatoren sind vielleicht nicht allen geläufig.  
%\*% ist der Operator für die Matrixmultiplikation. Der einfachste Fall ist die Multiplikation zweier gleich langer Vektoren (eines Zeilen- und eines Spaltenvektors). Dabei werden die Elemente der beiden Vektoren elementweise miteinander multipliziert und anschliessend die Produkte aufsummiert. Ergebnis ist ein Skalar (eine einzelne Zahl bzw. ein Vektor mit nur einem Element bzw. eine Matrix mit nur einer Zeile und einer Spalte). %% ist der Modulo Operator und gibt den Rest nach einer ganzzahligen Division an. So gibt z.B. 5 %% 2 (sprich 5 modulo 2) den Wert 1. %/% gibt das Resultat der ganzzahligen Division, d.h. 5 %/% 2 gibt den Wert 2 (wie oft kann man 2 von 5 subtrahieren?). Diese Operatoren werden beim Programmieren oft gebracht. ■

#### 2.1.2 Logische Operatoren und Funktionen

Die letzten beiden Funktionen, xor() und isTRUE(), werden Sie wahrscheinlich selten brauchen, sie sind nur der Vollständigkeit halber aufgelistet.

<	Kleiner
<=	Kleiner gleich
>	Grösser
>=	Grösser gleich

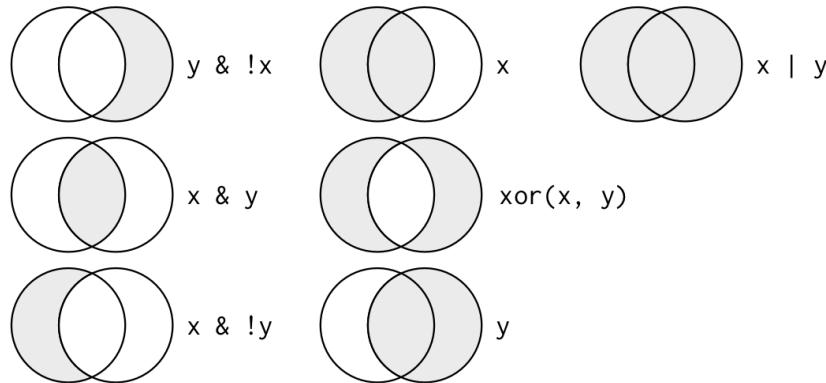


Figure 2.1: Veranschaulichung der logischen Operatoren. Aus [R for Data Science](<http://r4ds.had.co.nz>)

```
== Gleich (testet auf Äquivalenz)
!= Ungleich
!x Nicht x (Verneinung)
x | y x ODER y
x & y x UND y
xor(x, y) Exklusives ODER (entweder in x oder y, aber nicht in beiden)
isTRUE(x) testet ob x wahr ist
```

Die folgende Grafik zeigt die Verwendung der logischen Operatoren anhand von Venn-Diagrammen. x bezieht sich dabei immer den linken Kreis, y auf den rechten. Der ausgewählte Bereich ist immer dunkel dargestellt.

Es kommen noch weitere numerische Funktionen hinzu. Diese werden jedoch als Funktionen gebraucht, und nicht als Operatoren.

- Operatoren werden so gebraucht: `1 + 2`; sie stehen zwischen den Operanden. Funktionen sehen so aus: `abs(x)`; sie werden auf Argumente angewandt. ■

### 2.1.3 Numerische Funktionen

```
abs(x) Betrag
sqrt(x) Quadratwurzel
ceiling(x) Aufrunden: ceiling(3.475) ist 4
floor(x) Abrunden: floor(3.475) ist 3
round(x, digits = n) Runden: round(3.475, digits = 2) ist 3.48
log(x) Natürlicher Logarithmus
log(x, base = n) Logarithmus zur Basis n
log2(x) Logarithmus zur Basis 2
log10(x) Logarithmus zur Basis 10
exp(x) Exponentialfunktion: e^x
```

- **Wirklich nur für Nerds:** Es gibt in R keinen Unterschied zwischen Operatoren und Funktionen. Jeder Operator ist eigentlich eine Funktion mit einer speziellen Infix-Notation. So kann z.B. der Operator `+` auch als Funktion benutzt werden, allerdings muss dieser dann mit `\``` (backticks) umgeben werden. Sie können dies in der Konsole ausprobieren. ■

```
Als Operator mit Infix Notation
2 + 3
#> [1] 5

Als Funktionsaufruf
\`+`\(2, 3)
```

```
#> [1] 5

Die beiden Schreibweisen sind äquivalent
2 + 3 == `+`(2, 3)
#> [1] TRUE
```

#### 2.1.4 R als Taschenrechner

Wir können nun mit R rechnen. Im nächsten code chunk sehen Sie einige Beispiele. R verwendet den hash key # als Kommentarzeichen, d.h. Zeilen, welche mit einem # beginnen, werden nicht evaluiert, und werden dazu benutzt, den Code zu kommentieren.

Führen Sie diese Befehle aus, in dem Sie die einzelnen Zeilen auswählen, und dann gleichzeitig die **CMD + Enter** (macOS) oder **CTRL + Enter** (Windows) Tasten drücken.

```
Addition
5 + 5
#> [1] 10
99 + 89
#> [1] 188
12321 + 34324324
#> [1] 34336645

Subtraktion
6 - 5
#> [1] 1
5 - 89
#> [1] -84

Multiplikation
3 * 5
#> [1] 15
34 * 54
#> [1] 1836

Division
4 / 9
#> [1] 0.4444444
(5 + 5) / 2
#> [1] 5

Operatorpräzedenz: Klammern beachten
Punktrechnung vor Strichrechnung

(3 + 7 + 2 + 8) / (4 + 11 + 3)
#> [1] 1.111111

1/2 * (12 + 14 + 10)
#> [1] 18

1/2 * 12 + 14 + 10
#> [1] 30
```

```

Potenzieren
3^2
#> [1] 9
2^12
#> [1] 4096

Exponentialfunktion
exp(5)
#> [1] 148.4132

Das nächste Resultat ist so gross,
dass es in wissenschaftlicher Notation erscheint:
1.068647 * 10^13
exp(30)
#> [1] 1.068647e+13

Ganzzahlige Division
28 ist vier mal durch 6 teilbar, mit Rest 4
28 %% 6 # Rest: 4
#> [1] 4
28 %/% 6 # vier mal teilbar
#> [1] 4

5 %% 2 # Rest: 1
#> [1] 1
5 %/% 2 # zwei mal teilbar
#> [1] 2

Logische Operatoren
3 > 2
#> [1] TRUE
4 > 5
#> [1] FALSE
4 < 4
#> [1] FALSE
4 <= 4
#> [1] TRUE
5 >= 5
#> [1] TRUE
6 != 6
#> [1] FALSE
9 == 5 + 4
#> [1] TRUE

!(3 > 2)
#> [1] FALSE
(3 > 2) & (4 > 5) # UND
#> [1] FALSE
(3 > 2) | (4 > 5) # ODER
#> [1] TRUE
xor((3 > 2), (4 > 5))
#> [1] TRUE

```

- Bei xor darf nur *entweder der eine oder der andere* Ausdruck wahr sein, damit der gesamte Ausdruck

wahr ist (exklusives ODER, ausschliessende Disjunktion), während bei `|` *mindestens einer* der Ausdrücke wahr sein muss, damit der Gesamtausdruck wahr ist (inklusives ODER, nicht-ausschliessende Disjunktion). Während also `(3 > 2) | (4 > 5)` und `xor((3 > 2), (4 > 5))` beide TRUE ergeben, ergibt `(3 > 2) | (5 > 4)` TRUE, aber `xor((3 > 2), (5 > 4))` ergibt FALSE. ■

■ Berechnen Sie: ■

- 1)  $\frac{1}{3} \frac{1+3+5+7+2}{3+5+4}$
  - 2)  $e$  (wie lässt sich das berechnen?)
  - 3)  $\sqrt{2}$
  - 4)  $\sqrt[3]{8}$
  - 5)  $\log_2(8)$
- ■

### 2.1.5 Statistische Funktionen

Hier ist eine Auflistung statistischer Funktionen. Diese Funktionen haben alle das Argument `na.rm`, welches standardmässig den Wert FALSE annimmt. Damit können fehlende Werte berücksichtigt werden, d.h. fehlende Werte (na = not available) werden in diesem Fall nicht entfernt (rm = remove). Diese Funktionen können alle auf einen Vektor angewandt werden (siehe weiter unten).

```
mean(x, na.rm = FALSE) Mittelwert
sd(x) Standardabweichung
var(x) Varianz

median(x) Median
quantile(x, probs) Quantile von x. probs: Vektor mit Wahrscheinlichkeiten

sum(x) Summe
min(x) Minimalwert x_min
max(x) Maximalwert x_max
range(x) x_min und x_max

wenn center = TRUE: zentrieren
wenn scale = TRUE: durch SD teilen
scale(x, center = TRUE, scale = TRUE) Zentrieren und Standardisieren

mit prob können wir gewichtet ziehen:
sample(x, size, replace = FALSE, prob) Ziehen mit/ohne Zurücklegen.
 prob: Vektor mit Wahrscheinlichkeiten
```

### 2.1.6 Weitere nützliche Funktionen

```
c() Combine: kreiert einen Vektor
seq(from, to, by) Generiert eine Sequenz
: Colon Operator: generiert eine reguläre Sequenz
 (d.h. Sequenz in Einerschritten)
rep(x, times, each) Wiederholt x
 times: die Sequenz wird n-mal wiederholt
 each: jedes Element wird n-mal wiederholt
```

<code>head(x, n = 6)</code>	Zeigt die n ersten Elemente von x an
<code>tail(x, n = 6)</code>	Zeigt die n letzten Elemente von x an

### 2.1.7 Beispiele

```
c(1, 2, 3, 4, 5, 6)
#> [1] 1 2 3 4 5 6

mean(c(1, 2, 3, 4, 5, 6))
#> [1] 3.5

mean(c(1, NA, 3, 4, 5, 6), na.rm = TRUE)
#> [1] 3.8

mean(c(1, NA, 3, 4, 5, 6), na.rm = FALSE)
#> [1] NA

sd(c(1, 2, 3, 4, 5, 6))
#> [1] 1.870829

sum(c(1, 2, 3, 4, 5, 6))
#> [1] 21

min(c(1, 2, 3, 4, 5, 6))
#> [1] 1

range(c(1, 2, 3, 4, 5, 6))
#> [1] 1 6

scale(c(1, 2, 3, 4, 5, 6), center = TRUE, scale = FALSE)
#> [,1]
#> [1,] -2.5
#> [2,] -1.5
#> [3,] -0.5
#> [4,] 0.5
#> [5,] 1.5
#> [6,] 2.5
#> attr(,"scaled:center")
#> [1] 3.5
Der Mittelwert des zentrierten Vektors wird zur Information mit ausgegeben!

scale(c(1, 2, 3, 4, 5, 6), center = TRUE, scale = TRUE)
#> [,1]
#> [1,] -1.3363062
#> [2,] -0.8017837
#> [3,] -0.2672612
#> [4,] 0.2672612
#> [5,] 0.8017837
#> [6,] 1.3363062
#> attr(,"scaled:center")
#> [1] 3.5
```

```
#> attr("scaled:center")
#> [1] 1.870829
Mittelwert und Standardabweichung des z-standardisierten Vektors
werden zur Information mit ausgegeben!

Ziehen mit Zurücklegen
sample(c(1, 2, 3, 4, 5, 6), size = 1, replace = TRUE)
#> [1] 1

sample(c(1, 2, 3, 4, 5, 6), size = 12, replace = TRUE)
#> [1] 6 4 1 1 3 3 2 5 5 6 2 1

Ziehen mit Zurücklegen, ungleich gewichtet:
sample(c(1, 2, 3, 4, 5, 6), size = 1, replace = TRUE,
 prob = c(4/12, 1/12, 1/12, 2/12, 2/12, 2/12))
#> [1] 1

sample(c(1, 2, 3, 4, 5, 6), size = 12, replace = TRUE,
 prob = c(4/12, 1/12, 1/12, 2/12, 2/12, 2/12))
#> [1] 6 1 6 1 6 2 1 5 5 1 2 5

c(1, 2, 3, 4, 5, 6)
#> [1] 1 2 3 4 5 6

seq(from = 1, to = 6, by = 1)
#> [1] 1 2 3 4 5 6

1:6
#> [1] 1 2 3 4 5 6

rep(1:6, times = 2)
#> [1] 1 2 3 4 5 6 1 2 3 4 5 6

rep(1:6, each = 2)
#> [1] 1 1 2 2 3 3 4 4 5 5 6 6

rep(1:6, times = 2, each = 2)
#> [1] 1 1 2 2 3 3 4 4 5 5 6 6
```

■ Machen Sie bitte die folgenden Übungsbeispiele:

- 1) Erzeugen Sie eine Sequenz von 0 bis 100 in 5-er Schritten.
- 2) Berechnen Sie den Mittelwert des Vektors  $[1, 3, 4, 7, 11, 2]$ .
- 3) Lassen Sie sich die Spannweite  $x_{max} - x_{min}$  dieses Vektors ausgeben.
- 4) Berechnen Sie die Summe dieses Vektors.
- 5) Zentrieren Sie diesen Vektor.
- 6) Simulieren Sie einen Münzwurf mit der Funktion `sample()`. Tipp: nehmen Sie für Kopf 1 und für Zahl 0. Simulieren Sie 100 Münzwürfe.
- 7) Simulieren Sie eine Trick-Münze mit  $p \neq 0.5$
- 8) Generieren Sie einen Vektor, der aus 100 Wiederholungen der Zahl 3 besteht.

## 2.2 Variable definieren

Bisher haben wir die Resultate unserer Berechnungen nicht gespeichert. Selbstverständlich können wir in R Variablen definieren, und diesen einen Wert zuweisen.

Variablen werden in R so definiert: `my_var <- 4.` `<-` ist hier ein spezieller Zuweisungspfeil, und besteht aus einem `<` Zeichen und einem `-`. Es gibt in RStudio dafür eine Tastenkombination `ALT + -`. Hier weisen wir also der neuen Variablen `my_var` den Wert 4 zu.

■ Man kann in R sowohl `<-` als auch `=` für die Zuweisung verwenden. Auf den ersten Blick erscheint das verwirrend (es ist aus historischen Gründen so), aber `=` ist außerdem das Symbol für die Zuweisung von Argumenten (für Funktionen), und wenn man `<-` verwendet, ist es klar, dass man eine Variable definiert. Außerdem haben R Puristen `<-` lieber. ■

### 2.2.1 Variablennamen

Eine Variable muss also einen Namen haben - dieser besteht aus Buchstaben, Zahlen und den Zeichen `_` und/oder `.` Ein Variablenname muss mit einem Buchstaben beginnen, und darf keine Leerschläge enthalten.

Es gibt ein paar Konventionen, an die man sich halten sollte, um R Code lesbar und verständlich zu machen - vor allem, wenn man diesen mit anderen teilt. Wir empfehlen, für die Namensgebung `snake_case` zu verwenden, d.h. wir trennen Wörter innerhalb eines Namens mit einem Unterstrich: `my_var`.

Andere Möglichkeiten wären:

```
snake_case_variable
camelCaseVariable
variable.with.periods
variable.With_noConventions

gute Variablennamen
x_mean
x_sd

anzahl_personen
alter

weniger gute Variablennamen
p
a

unmögliche Variablennamen
x_mittelwert
sd von x
```

In vielen Texten, vor allem in älteren, findet man Variablennamen mit `.` anstelle von `_`, wie z.B. im Buch von Luhmann. Moderne Style Guides empfehlen `_`.

Wenn wir eine Variable definiert haben:

```
my_var <- 4
```

können wir uns deren Wert in der Konsole ausgeben lassen:

```
print(my_var)
#> [1] 4
```

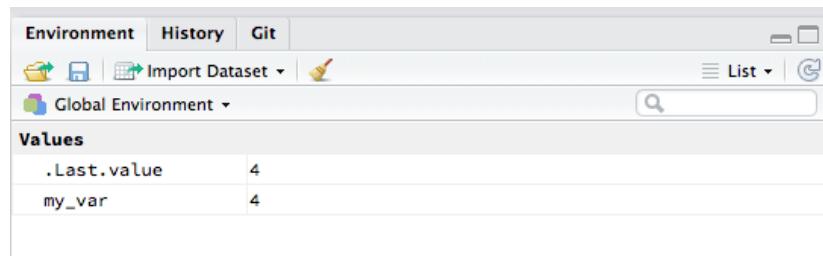
```
oder einfach

my_var
#> [1] 4
```

Anstelle von `print(my_var)` kann man einfach den Namen `my_var` eingeben. Wenn man will, kann man gleich bei der Definition der Variablen die Zuweisung in Klammern schreiben, und das Resultat wird gleichzeitig in der Konsole angezeigt:

```
(my_var <- 4)
#> [1] 4
```

- Wenn Sie die Variable `my_var` definieren, sehen Sie in RStudio im **Environment**-Bereich eine neue Variable und deren Wert unter **Values**.



- Wiederholen Sie einige der Beispiele von oben, aber speichern Sie diesmal die Resultate in neuen Variablen. Sie können dann natürlich die Variablen wieder verwenden.

```
vektor <- c(1, 3, 4, 7, 11, 2)
summe <- sum(vektor)

mittelwert <- mean(vektor)
mittelwert
#> [1] 4.666667

gerundeter_mittelwert <- round(mittelwert, digits = 1)
gerundeter_mittelwert
#> [1] 4.7
```

Diese Variablen existieren nun im **Global Environment**, aber nur solange die aktuelle R Session bestehen bleibt. Wenn sie R neu starten, sind diese Variablen nicht mehr vorhanden. Deshalb sollte man immer in einem R script/notebook festhalten, was man gemacht hat.

## 2.3 Funktionen aufrufen

Wir haben nun schon einige Funktionen verwendet. Hier schauen wir uns nun die Syntax eines Funktionaufrufs (function call) an.

Die Funktion, welche unten angezeigt wird, besteht aus einem Namen `function_name` und hat zwei Argumente, `arg1` und `arg2`. Die Argumente können, aber müssen nicht, default (voreingestellte) Werte haben. In diesem Beispiel hat `arg1` keinen default Wert. `arg2` hat den default Wert `val2`. Argumente ohne default Werte müssen zwingend angegeben werden. Wenn man Argumente mit default Werten nicht explizit angibt, wird der jeweilige default Wert für das Argument verwendet. Typische default Werte für Argumente sind `TRUE` oder `FALSE`, z.B. gibt es in vielen Funktionen das Argument `na.rm` mit der Voreinstellung (default Wert) `na.rm = TRUE`. Wenn dieses Argument also NICHT explizit angegeben wird, werden fehlende Werte per Voreinstellung entfernt, bevor die jeweilige Funktion angewendet wird.

```
function_name(arg1, arg2 = val2)
```

Eine Funktion kann beliebig viele Argumente haben.

- Wir verwenden in R den `=` Operator, wenn wir einem Argument einen Wert zuweisen. Der Unterschied zwischen `<-` und `=` ist am Anfang nicht ganz leicht nachvollziehbar, und wird hier noch einmal illustriert. ■

```
Zuweisungspfeil:
`<-` weist einem Objekt, z.B. einer Variablen, einen Wert zu:
x <- c(23.192, 21.454, 24.677)

Gleichheitszeichen:
`=` weist bei einem Funktionsaufruf den Argumenten der Funktion einen Wert zu.
Z.B. die round Funktion:
round(x, digits = 1)
#> [1] 23.2 21.5 24.7
```

- Das Argument `digits` bestimmt, auf wieviele Nachkommastellen gerundet werden soll, und wird innerhalb der Funktion `round()` verwendet. Mit `=` weisen wir diesem Argument den Wert 1 zu. ■

Wenn man wissen will, welche Argumente eine Funktion hat, kann man dies am besten mit TAB Completion herausfinden.

- Tippen Sie in der Konsole `scale()` und drücken Sie TAB. Sie sehen, dass diese Funktion drei Argumente hat, `x`, `center` und `scale`. Schreiben Sie `scale(vektor, scale =` gefolgt von TAB. Sie sehen, dass `scale = TRUE` den default Wert `TRUE` hat. ■
- Was sind die Argumente der Funktion `round()`? Hat eines der Argumente einen default Wert? ■
- Schauen Sie im **Help** Viewer nach, was die Funktion `rnorm()` macht. Was sind die Argumente. Was bedeuten die default Werte? ■
- Schauen Sie im **Help** Viewer nach, welche Argumente die `seq()` Funktion hat. ■
- Was machen folgende Funktionsaufrufe? ■

```
seq()
seq(1, 10)
seq(1, 10, 2)
seq(1, 10, 2, 20)
seq(1, 10, length.out = 20)
```

### 2.3.1 Verschachtelung von Funktionen

Wir können beliebig viele Funktion ineinander verschachteln, d.h. wir können den Output einer Funktion einer weiteren Funktion als Input übergeben. Wir sehen uns diese Verschachtelung an folgendem Beispiel

an.

Wir definieren zuerst einen Vektor, berechnen davon den Mittelwert und runden diesen auf zwei Dezimalstellen:

```
definiert einen Vektor:
c(34.444, 45.853, 21.912, 29.261, 31.558)
#> [1] 34.444 45.853 21.912 29.261 31.558

berechnet den Mittelwert:
mean(c(34.444, 45.853, 21.912, 29.261, 31.558))
#> [1] 32.6056

rundet auf 2 Dezimalstellen:
round(mean(c(34.444, 45.853, 21.912, 29.261, 31.558)),
 digits = 2)
#> [1] 32.61
```

Die Funktionen werden immer in der Reihenfolge *von innen nach aussen* ausgeführt, d.h. zuerst `mean()`, und dann `round()`.

Wenn wir mehrere Funktionen ineinander verschachteln, kann unser Code schnell unlesbar werden. Natürlich könnten wir die einzelnen Zwischenschritte speichern, wie im Beispiel weiter oben, aber dann definieren wir eine Menge Variablen, welche wir vielleicht gar nicht benötigen. Wir werden im Kapitel über Datentransformation einen neuen Operator kennenlernen, welcher eine sehr elegante Lösung für dieses Problem bietet.

## 2.4 Datentypen

Wir haben bisher mit Vektoren gearbeitet. Diese stellen den fundamentalen Datentyp dar. Alle weiteren Datentypen bauen auf diesem auf. Vektoren selber können in folgende Typen unterschieden werden in:

- **numeric vectors:** Mit diesen hatten wir es bisher zu tun. Numerische Vektoren werden weiter in `integer` (ganze Zahlen) und `double` (reelle Zahlen) unterteilt.
- **character vectors:** Die Elemente dieses Typs bestehen aus Zeichen, welche von Anführungszeichen umgeben werden (entweder ' oder " ), z.B. 'Wort' oder "Wort".
- **logical vectors:** Die Elemente dieses Typs können nur 3 Werte annehmen: `TRUE`, `FALSE` oder `NA`.

Vektoren müssen aus denselben Elementen bestehen, d.h. wir können nicht `logical` und `character` Vektoren mischen. Vektoren haben 3 Eigenschaften:

- Typ: `typeof()`: Was ist es?
- Länge: `length()`: Wie viele Elemente?
- Attribute: `attributes()`: Zusätzliche Information (Metadaten)

Vektoren werden entweder mit der `c()` (Abkürzung für `combine`) Funktion erstellt, oder mit speziellen Funktion, wie z.B. `seq()` oder `rep()`. Wir schauen uns nun die verschiedenen Typen von Vektoren an.

### 2.4.1 Numeric vectors

Numerische Vektoren bestehen aus Zahlen. Und zwar entweder aus natürlichen Zahlen (`integer`) oder aus reellen Zahlen (`double`).

```
numbers <- c(1, 2.5, 4.5)
typeof(numbers)
#> [1] "double"
```

```
length(numbers)
#> [1] 3
```

Wir können die einzelnen Elemente eines Vektor mit [] auswählen (dies nennt man im Fachjargon **subsetting**):

```
Das erste Element:
numbers[1]
#> [1] 1

Das zweite Element:
numbers[2]
#> [1] 2.5

Das letzte Element:
numbers hat die Länge 3
length(numbers)
#> [1] 3

Wir können damit numbers indizieren
numbers[length(numbers)]
#> [1] 4.5

Mit - (Minus) können wir ein Element weglassen, z.B. alle Elemente außer das Erste:
numbers[-1]
#> [1] 2.5 4.5

Wir können eine Sequenz verwenden, z.B. die ersten zwei Elemente:
numbers[1:2]
#> [1] 1.0 2.5

Wir können das erste und dritte Element weglassen:
numbers[-c(1, 3)]
#> [1] 2.5
```

## Matrizen

Wie bereits erwähnt, ist eigentlich alles in R ein Vektor. Ein Skalar ist ein Vektor der Länge 1. Eine Matrix ist im Prinzip auch ein Vektor, aber einer mit einem **dim** (dimension) Attribut:

```
x ist ein Vektor
x <- 1:8

Wir weisen nun x ein dim Attribut zu, d.h. wir machen aus
x eine Matrix (hier mit 2 Zeilen und 4 Spalten)
dim(x) <- c(2, 4)
x
#> [,1] [,2] [,3] [,4]
#> [1,] 1 3 5 7
#> [2,] 2 4 6 8

Wir können Matrizen auch so erstellen:
m <- matrix(x <- 1:8, nrow = 2, ncol = 4, byrow = FALSE)
m
#> [,1] [,2] [,3] [,4]
```

```
#> [1,] 1 3 5 7
#> [2,] 2 4 6 8

Was sind die Dimensionen?
dim(m)
#> [1] 2 4
```

Beachten Sie das Argument `byrow`, welches den default Wert `FALSE` hat. Wenn wir das Argument `byrow = TRUE` setzen, dann erhalten wir:

```
m2 <- matrix(x <- 1:8, nrow = 2, ncol = 4, byrow = TRUE)
m2
#> [,1] [,2] [,3] [,4]
#> [1,] 1 2 3 4
#> [2,] 5 6 7 8
```

Dies führt dazu, dass die Zeilen zuerst aufgefüllt werden, während oben `byrow = FALSE` die Spalten zuerst aufgefüllt werden.

Wir können Matrizen transponieren, d.h. die Zeilen werden zu Spalten, und die Spalten werden zu Zeilen:

```
m_transponiert <- t(m)
m_transponiert
#> [,1] [,2]
#> [1,] 1 2
#> [2,] 3 4
#> [3,] 5 6
#> [4,] 7 8
```

Es gibt zwei weitere Funktionen, welche wir kennenlernen sollten: `cbind()` und `rbind()`. Diese dienen dazu, Vektoren oder Matrizen zusammenzufügen.

`cbind()` kombiniert die Spalten (column-bind) von Vektoren/Matrizen zu einem Objekt:

```
x1 <- 1:3
x1 ist ein Vektor
x1
#> [1] 1 2 3

x2 <- 10:12
x2 ist ein Vektor
x2
#> [1] 10 11 12

m1 <- cbind(x1, x2)
m1 ist eine Matrix mit den Dimensionen [3, 2]
m1
#> x1 x2
#> [1,] 1 10
#> [2,] 2 11
#> [3,] 3 12
```

Daraus entsteht eine Matrix `m1` mit den Ausgangsvektoren als Spalten.

`rbind()` kombiniert die Zeilen (row-bind) von Vektoren/Matrizen zu einem Objekt:

```
m2 <- rbind(x1, x2)
m2 ist eine Matrix mit den Dimensionen [2, 3]
m2
#> [,1] [,2] [,3]
#> x1 1 2 3
#> x2 10 11 12
```

Hier resultiert eine Matrix `m2` mit den Ausgangsvektoren als Zeilen.

Auch Matrizen können mit `[]` indiziert werden. Wir müssen hier aber angeben, welche Zeile(n) und Spalte(n) wir erhalten wollen, und zwar mit `[zeilennummer, spaltennummer]`.

Die Zeilennummer und Spaltennummer sind durch ein Komma getrennt. Wenn `zeilennummer` oder `spaltennummer` leer gelassen werden, heisst das: "Wähle alle Zeilen/Spalten aus."

```
Die erste Zellen: Zeile 1, Spalte 1
m1[1, 1]
#> x1
#> 1

Zeile 1, Spalte 2
m1[1, 2]
#> x2
#> 10

Zeilen 2 bis 3, Spalte 1
m1[2:3, 1]
#> [1] 2 3

Alle Zeilen, Spalte 1
m1[, 1]
#> [1] 1 2 3

Zeile 2, alle Spalten
m1[2,]
#> x1 x2
#> 2 11
```

## Vektorisierung

Alle Operatoren in R sind vektorisiert, d.h. sie operieren elementweise auf Vektoren:

```
x1 <- 1:10
x1 + 2
#> [1] 3 4 5 6 7 8 9 10 11 12

x2 <- 11:20

x1 + x2
#> [1] 12 14 16 18 20 22 24 26 28 30

x1 * x2
#> [1] 11 24 39 56 75 96 119 144 171 200
```

Dasselbe gilt für Funktionen:

```
x1 <- 1:10

x1^2
#> [1] 1 4 9 16 25 36 49 64 81 100

exp(x1)
#> [1] 2.718282 7.389056 20.085537 54.598150 148.413159
#> [6] 403.428793 1096.633158 2980.957987 8103.083928 22026.465795
```

## Recycling

Etwas Besonderes in R ist das Vektor **recycling**: dies bedeutet, dass ein kürzerer Vektor wiederholt wird, wenn wir z.B. zwei Vektoren addieren. Dies kann oft nützlich sein, birgt aber auch Gefahren, wenn man sich dessen nicht bewusst ist.

```
Der kürzere Vektor wird rezykliert:
1:10 + 1:2
#> [1] 2 4 4 6 6 8 8 10 10 12
```

Was hier genau passiert, kann so dargestellt werden:

```
1 2 3 4 5 6 7 8 9 10
1 2 1 2 1 2 1 2 1 2
```

Der kürzere Vektor 1:2 wird so oft wie nötig wiederholt.

Was passiert, wenn der längere Vektor nicht ein Vielfaches des kürzeren ist?

```
1:10 + 1:3
#> Warning in 1:10 + 1:3: longer object length is not a multiple of shorter
#> object length
#> [1] 2 4 6 5 7 9 8 10 12 11
```

R führt zwar den Befehl aus, gibt uns aber auch eine Warnung.

```
1 2 3 4 5 6 7 8 9 10
1 2 3 1 2 3 1 2 3 1
```

## Missing Values

Fehlende Werte werden mit **NA** deklariert. Wir werden diese im nächsten Kapitel kennenlernen.

```
zahlen <- c(12, 13, 15, 11, NA, 10)
zahlen
#> [1] 12 13 15 11 NA 10
```

Mit der Funktion **is.na()** kann man testen, ob etwas tatsächlich ein fehlender Wert ist:

```
is.na(zahlen)
#> [1] FALSE FALSE FALSE FALSE TRUE FALSE
```

■ Fehlende Werte sind nicht zu verwechseln mit den Werten **Inf** (infinity) und **NaN** (not a number). Diese entstehen z.B. bei Division durch Null:  $1/0$  oder  $0/0$ . Es gibt ausserdem noch den Datentyp **NULL**: diesen benutzen wir, wenn etwas undefiniert bleiben soll. ■

```
1/0
#> [1] Inf
0/0
#> [1] NaN
```

## 2.4.2 Character vectors

Character vectors (strings) werden benutzt, um Text darzustellen.

```
text <- c("these are", "some strings")
text
#> [1] "these are" "some strings"
typeof(text)
#> [1] "character"

text hat 2 Elemente:
length(text)
#> [1] 2
```

Wie numerische Vektoren können auch character Vektoren indiziert werden, d.h. wir können einzelne Elemente auswählen.

`letters` und `LETTERS` sind sogenannte `built-in constants`. Dies sind Vektoren mit allen Buchstaben der englischen Sprache.

```
?letters
letters[1:3]
#> [1] "a" "b" "c"
letters[10:15]
#> [1] "j" "k" "l" "m" "n" "o"
LETTERS[24:26]
#> [1] "X" "Y" "Z"
```

Es gibt eine Funktion, mit welcher wir character vectors zusammenfügen können:

```
paste(LETTERS[1:3], letters[24:26], sep = "_")
#> [1] "A_x" "B_y" "C_z"

Spezialfall mit sep = ""
paste0(1:3, letters[5:7])
#> [1] "1e" "2f" "3g"

vorname <- "Ronald Aylmer"
nachname <- "Fisher"
paste("Mein Name ist:", vorname, nachname, sep = " ")
#> [1] "Mein Name ist: Ronald Aylmer Fisher"

zahl <- 7
zahl ist zwar eine ganze Zahl, wird aber durch paste() zu einem character
paste(zahl, "ist eine Zahl", sep = " ")
#> [1] "7 ist eine Zahl"
```

## 2.4.3 Logical vectors

Logische Vektoren können 3 Werte annehmen; entweder `TRUE`, `FALSE` oder `NA`.

```
log_var <- c(TRUE, FALSE, TRUE)
log_var
#> [1] TRUE FALSE TRUE
```

Logische Vektoren werden vor allem dazu benutzt, um numerische Vektoren zu indizieren, z.B. um alle positiven Elemente eines Vektors auszuwählen.

```
x ist ein Vektor von standardnormalverteilten Zufallszahlen (d.h. wir ziehen
hier eine Zufallsstichprobe der Grösse n = 24 aus einer normalverteilten
Population mit dem Mittelwert 0 und der Standardabweichung 1). Die Funktion
`rnorm`, mit der wir die Zufallsstichprobe standardnormalverteilter Zahlen
ziehen, werden wir zu einem späteren Zeitpunkt noch ausführlicher behandeln.
set.seed(5434) # macht das Beispiel reproduzierbar

x <- rnorm(24)
x
#> [1] 1.06115528 0.87480990 -0.30032832 1.21965848 0.09860288
#> [6] 1.89862128 -1.54699798 0.96349219 -0.64968432 -1.09672125
#> [11] -0.55326456 -0.29394388 0.58151046 -0.15135071 1.66997280
#> [16] -0.10726874 0.51633289 -0.64741465 0.10489022 -0.95484078
#> [21] 0.22940461 -0.54106301 -0.76310004 1.22446844

Wir brauchen nun einen logischen Vektor, der für jedes Element angibt,
ob dieses Element die Bedingung erfüllt (x soll positiv sein):
x > 0
#> [1] TRUE TRUE FALSE TRUE TRUE TRUE FALSE TRUE FALSE FALSE FALSE
#> [12] FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE FALSE
#> [23] FALSE TRUE

Nun indizieren wir damit den Vektor x:
x[x > 0]
#> [1] 1.06115528 0.87480990 1.21965848 0.09860288 1.89862128 0.96349219
#> [7] 0.58151046 1.66997280 0.51633289 0.10489022 0.22940461 1.22446844

Wir könnten den Index auch speichern
index <- x > 0

und damit x indizieren:
x[index]
#> [1] 1.06115528 0.87480990 1.21965848 0.09860288 1.89862128 0.96349219
#> [7] 0.58151046 1.66997280 0.51633289 0.10489022 0.22940461 1.22446844
```

Wir können auch alle Elemente von `x` suchen, welche eine Standardabweichung über dem Mittelwert liegen:

```
x
#> [1] 1.06115528 0.87480990 -0.30032832 1.21965848 0.09860288
#> [6] 1.89862128 -1.54699798 0.96349219 -0.64968432 -1.09672125
#> [11] -0.55326456 -0.29394388 0.58151046 -0.15135071 1.66997280
#> [16] -0.10726874 0.51633289 -0.64741465 0.10489022 -0.95484078
#> [21] 0.22940461 -0.54106301 -0.76310004 1.22446844

x_mean <- mean(x)
x_sd <- sd(x)
x_mean
#> [1] 0.1182059
x_sd
#> [1] 0.9156615

x > (x_mean + x_sd)
#> [1] TRUE FALSE FALSE TRUE FALSE TRUE FALSE FALSE FALSE FALSE FALSE
```

```
#> [12] FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#> [23] FALSE TRUE

x[x > (x_mean + x_sd)]
#> [1] 1.061155 1.219658 1.898621 1.669973 1.224468

oder in einem Befehl
x[x > (mean(x) + sd(x))]
#> [1] 1.061155 1.219658 1.898621 1.669973 1.224468
```

- Wenn wir wissen wollen, welche Elemente des Vektors die Bedingung erfüllen, können wir die Funktion `which()` benützen.

```
dies zeigt an, welche Elemente von x die Bedingung erfüllen
which(x > (mean(x) + sd(x)))
#> [1] 1 4 6 15 24
```

- In diesem Beispiel liegen die Elemente 1, 4, 6, 15, 24 eine Standardabweichung oder mehr über dem Mittelwert. Wir können auch damit den Vektor indizieren; anstatt einen logischen Vektor zu verwenden, sagen wir einfach explizit, welche Elemente wir haben wollen.

```
x[which(x > (mean(x) + sd(x)))]
#> [1] 1.061155 1.219658 1.898621 1.669973 1.224468

x[which(x > 0)] und x[x > 0] sind äquivalent
x[which(x > (mean(x) + sd(x)))] == x[x > (mean(x) + sd(x))]
#> [1] TRUE TRUE TRUE TRUE TRUE
```

#### 2.4.4 Factors

Bisher haben wir `numeric`, `logical` und `character` Vektoren kennengelernt. Diese werden in R auch `atomic vectors` genannt, weil sie die fundamentalen Datentypen darstellen. Ein weiterer Objekttyp wird benötigt, um kategoriale Daten oder Gruppierungsvariablen darzustellen. Dieser Objekttyp wird `factor` genannt. Ein `factor` ist ein Vektor von natürlichen Zahlen (integer vector), der mit zusätzlicher Information (Metadaten) versehen ist. Diese `attributes` sind die Objektklasse `factor` und die Faktorstufen `levels`. Am besten wir illustrieren dies mit einem Beispiel.

```
geschlecht ist ein character vector
geschlecht <- c("male", "female", "male", "male", "female")
geschlecht
#> [1] "male" "female" "male" "male" "female"

typeof(geschlecht)
#> [1] "character"

attributes(geschlecht)
#> NULL
```

Nun können wir mit der Funktion `factor()` einen Faktor definieren:

```
geschlecht <- factor(geschlecht, levels = c("female", "male"))
geschlecht
#> [1] male female male male female
```

```
#> Levels: female male

geschlecht hat nun den Datentyp integer
typeof(geschlecht)
#> [1] "integer"

aber die `class` factor
class(geschlecht)
#> [1] "factor"

und die Attribute levels und class
attributes(geschlecht)
#> $levels
#> [1] "female" "male"
#>
#> $class
#> [1] "factor"
```

Wir haben bei der Definition die `levels` explizit angegeben. Dies hätten wir nicht tun müssen, da R dies automatisch macht; die Faktorstufen sind standardmäßig alphabetisch geordnet.

- Dass ein `factor` ein `integer vector` ist, erkennen wir, wenn wir das `class` Attribut entfernen, mit `unclass(geschlecht)`. Dies wird auch im **Environment**-Bereich von RStudio angezeigt. ■

```
geschlecht
#> [1] male female male male female
#> Levels: female male
unclass(geschlecht)
#> [1] 2 1 2 2 1
#> attr(,"levels")
#> [1] "female" "male"
```

Faktoren werden wir später oft benötigen, wenn wir Regressionsmodelle mit Kodiervariablen erstellen möchten. Die erste Stufe eines Faktors wird von R automatisch als Referenzkategorie bestimmt, wenn wir den Faktor als Prädiktorvariable benutzen. Manchmal wollen wir jedoch eine andere Stufe als Referenzkategorie. In diesem Fall kann man die Reihenfolge der Faktorstufen ändern. Es gibt drei Möglichkeiten: mit `relevel()`, `levels()` oder mit `factor()`.

Mit `relevel()` kann die Referenzkategorie bestimmt werden.

```
levels(geschlecht)
#> [1] "female" "male"

Wir müssen das Resultat der Variable wieder zuweisen
geschlecht <- relevel(geschlecht, ref = "male")
levels(geschlecht)
#> [1] "male" "female"
```

Mit `factor()` kann man die Reihenfolge genau bestimmen - es müssen aber alle Stufen explizit angegeben werden.

```
geschlecht
#> [1] male female male male female
#> Levels: male female

geschlecht <- factor(geschlecht, levels = c("male", "female"))
geschlecht
#> [1] male female male male female
#> Levels: male female
levels(geschlecht)
#> [1] "male" "female"
```

Die Dummykodierung werden wir im Verlauf dieser Vorlesung kennenlernen - wir lassen diese meistens von R automatisch erstellen, aber wir können selber die Referenzkategorien bestimmen.

Eine weitere nützliche Funktion für Faktoren ist `table()`. Damit können wir eine Häufigkeitstabelle erstellen.

```
table(geschlecht)
#> geschlecht
#> male female
#> 3 2
```

## 2.4.5 Lists

Ein weiterer Datentyp ist `list`. Während Vektoren aus Elementen desselben Typs bestehen, können Listen aus heterogenen Elementen zusammengesetzt werden.

Listen werden mit der Funktion `list()` definiert:

```
x <- list(1:3, "a", c(TRUE, FALSE, TRUE), c(2.3, 5.9))
x
#> [[1]]
#> [1] 1 2 3
#>
#> [[2]]
#> [1] "a"
#>
#> [[3]]
#> [1] TRUE FALSE TRUE
#>
#> [[4]]
#> [1] 2.3 5.9
```

Hier haben wir eine Liste `x` definiert, welche als Elemente einen `numeric` Vektor, einen `character`, einen `logical` Vektor und einen weiteren `numeric` Vektor enthält.

Listen können wie Vektoren indiziert werden:

```
x[1]
#> [[1]]
#> [1] 1 2 3
x[2]
#> [[2]]
#> [1] "a"
x[3]
#> [[3]]
#> [1] TRUE FALSE TRUE
x[4]
```

```
#> [[1]]
#> [1] 2.3 5.9
```

- Listen können auch mit [[ indiziert werden. Damit werden die Elemente noch weiter “entpackt”. Dies wird sehr gut in R for Data Science erklärt.

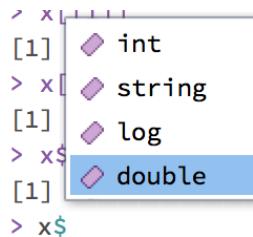
Wir werden in dieser Vorlesung selten selber Listen erstellen. Listen sind aber äusserst wichtig in R, und die statistischen Funktionen haben üblicherweise als Output eine Liste. Diese sind aber **named lists**, was bedeutet, dass die Elemente der Liste einen Namen haben. Erstellt wird eine **named list** so:

```
x <- list(int = 1:3,
 string = "a",
 log = c(TRUE, FALSE, TRUE),
 double = c(2.3, 5.9))

x
#> $int
#> [1] 1 2 3
#>
#> $string
#> [1] "a"
#>
#> $log
#> [1] TRUE FALSE TRUE
#>
#> $double
#> [1] 2.3 5.9

Der Typ einer Liste ist "list"
typeof(x)
#> [1] "list"
```

Die Elemente von x haben nun Namen, und können somit viel einfacher direkt ausgewählt werden. Dafür gibt es den speziellen \$ Operator. Wenn Sie wissen, dass x eine Liste ist, Sie aber die Namen der Elemente nicht kennen, können Sie in der Konsole oder im Editor x\$ schreiben, und dann TAB drücken. RStudio zeigt alle Elemente dieser Liste an.



Diese Namen müssen nicht in Anführungszeichen geschrieben werden.

```
x$string
#> [1] "a"
x$double
#> [1] 2.3 5.9
```

### 2.4.6 Data Frames

Nun kommen wir zu dem für uns wichtigsten Objekt in R, dem Data Frame. Datensätze werden in R durch Data Frames repräsentiert. Ein Data Frame besteht aus Zeilen (rows) und Spalten (columns) und entspricht einem Datensatz in SPSS. Technisch gesehen ist ein Data Frame eine Liste, deren Elemente gleichlange (equal-length) Vektoren sind. Die Vektoren selber können `numeric`, `logical` oder `character` Vektoren sein, oder natürlich Faktoren. Numerische Variablen in einem Datensatz sollten demzufolge `numeric` Vektoren, kategoriale Variablen/Gruppierungsvariablen sollten `factors` sein. Ein Data Frame ist eine 2-dimensionale Struktur, und kann einerseits wie ein Vektor indiziert werden (genauer: wie eine Matrix), andererseits wie eine Liste.

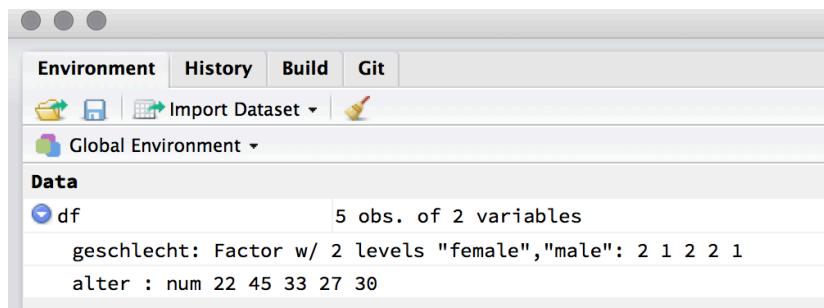
Traditionell werden Data Frames in R mit der Funktion `data.frame()` definiert. In RStudio bzw. dem `tidyverse`-Package werden Data Frames neuerdings auch `tibbles` oder `tbl` genannt. `tibbles` werden mit der Funktion `data_frame()` definiert, und stellen lediglich eine moderne Variante eines Data Frames dar. Sie erleichtern das Arbeiten mit Datensätzen.

■ Genauer gesagt: wenn man `data_frame()` beim Importieren von Datensätzen benutzt, muss man weniger vorsichtig sein. RStudio macht das automatisch. Wir verwenden die Begriffe **Data Frame** und **Tibble** als Synonyme, auch wenn sie sich in einigen Merkmalen unterscheiden (**Tibble** hört sich an wie ‘table’, wenn man es mit einem Neuseeländischen Akzent ausspricht). ■

Ein Data Frame wird so definiert:

```
library(dplyr)
df <- data_frame(geschlecht = factor(c("male", "female",
 "male", "male",
 "female")),
 alter = c(22, 45, 33, 27, 30))
#> Warning: `data_frame()` is deprecated, use `tibble()` .
#> This warning is displayed once per session.
df
#> # A tibble: 5 x 2
#> geschlecht alter
#> <fct> <dbl>
#> 1 male 22
#> 2 female 45
#> 3 male 33
#> 4 male 27
#> 5 female 30
```

`df` ist nun ein Data Frame mit zwei Variablen, `geschlecht` und `alter`. Im **Environment**-Bereich von RStudio erscheinen Data Frames unter **Data**:



Ein Data Frame hat die Attribute `names()`, `colnames()` und `rownames()`, wobei `names()` und `colnames()`

dasselbe bedeuten.

```
attributes(df)
#> $names
#> [1] "geschlecht" "alter"
#>
#> $row.names
#> [1] 1 2 3 4 5
#>
#> $class
#> [1] "tbl_df" "tbl" "data.frame"
```

Die Länge eines Data Frames ist die Länge der Liste, d.h. sie entspricht der Anzahl Spalten. Diese kann mit `ncol()` abgefragt werden; `nrow()` gibt die Anzahl Zeilen.

```
ncol(df)
#> [1] 2
nrow(df)
#> [1] 5
```

## Data Frame Subsetting

Wie oben erwähnt, kann ein Data Frame wie eine Liste indiziert werden, oder wie eine Matrix.

- Wie eine Liste: die einzelnen Spalten können mit `$` ausgewählt werden.
- Wie eine Matrix: die Elemente können mit `[` ausgewählt werden.

```
Spaltennamen (Variablen) auswählen
df$geschlecht
#> [1] male female male male female
#> Levels: female male

df$alter
#> [1] 22 45 33 27 30

df["geschlecht"]
#> # A tibble: 5 x 1
#> geschlecht
#> <fct>
#> 1 male
#> 2 female
#> 3 male
#> 4 male
#> 5 female

df["alter"]
#> # A tibble: 5 x 1
#> alter
#> <dbl>
#> 1 22
#> 2 45
#> 3 33
#> 4 27
#> 5 30

Nach Position auswählen
```

```
df [1]
#> # A tibble: 5 x 1
#> geschlecht
#> <fct>
#> 1 male
#> 2 female
#> 3 male
#> 4 male
#> 5 female

df [2]
#> # A tibble: 5 x 1
#> alter
#> <dbl>
#> 1 22
#> 2 45
#> 3 33
#> 4 27
#> 5 30
```

Ähnlich wie bei Matrizen können Zeilen und Spalten ausgewählt werden, auch hier mit `[zeilennummer, spaltennummer]`.

```
Erste Zeile, erste Spalte
df [1, 1]
#> # A tibble: 1 x 1
#> geschlecht
#> <fct>
#> 1 male

Erste Zeile, alle Spalten
df [1,]
#> # A tibble: 1 x 2
#> geschlecht alter
#> <fct> <dbl>
#> 1 male 22

Alle Zeilen, erste Spalte
df [, 1]
#> # A tibble: 5 x 1
#> geschlecht
#> <fct>
#> 1 male
#> 2 female
#> 3 male
#> 4 male
#> 5 female

Alle Zeilen, alle Spalten
df [,]
#> # A tibble: 5 x 2
#> geschlecht alter
#> <fct> <dbl>
#> 1 male 22
```

```
#> 2 female 45
#> 3 male 33
#> 4 male 27
#> 5 female 30

Wir können auch Sequenzen verwenden
Die ersten drei Zeilen, alle Spalten
df[1:3,]

#> # A tibble: 3 x 2
#> geschlecht alter
#> <fct> <dbl>
#> 1 male 22
#> 2 female 45
#> 3 male 33
```

Da die Spalten Vektoren sind, können wir diese auch indizieren:

```
df$geschlecht[1]
#> [1] male
#> Levels: female male

oder

df$alter[2:3]
#> [1] 45 33
```

## 2.5 Übungsaufgaben

### Zahlen runden

```
x <- rnorm(10, mean = 1, sd = 0.5)
x
#> [1] 0.66851021 1.60952322 0.62064523 0.62563620 1.31433575
#> [6] 1.10078969 0.48313164 -0.02969281 0.71076782 0.85972382
```

- 1) Runden Sie den Vektor x auf 0 Dezimalstellen.
- 2) Runden Sie den Vektor x auf 3 Dezimalstellen.

```
round(x = x, digits = 0)
#> [1] 1 2 1 1 1 0 0 1 1

round(x, digits = 3)
#> [1] 0.669 1.610 0.621 0.626 1.314 1.101 0.483 -0.030 0.711 0.860

(zahl <- 3.45263)
#> [1] 3.45263
```

- 3) Runden Sie zahl auf die nächste natürliche Zahl auf/ab.

```
ceiling(zahl)
#> [1] 4
floor(zahl)
#> [1] 3
```

## Mittelwert berechnen

- 1) Berechnen Sie den Mittelwert der Variable `alter` in folgendem Data Frame:

```
df <- data_frame(geschlecht = sample(c("male", "female"),
 size = 24,
 replace = TRUE),
 alter = runif(24, min = 19, max = 45))
df
#> # A tibble: 24 x 2
#> geschlecht alter
#> <chr> <dbl>
#> 1 male 22.1
#> 2 male 22.6
#> 3 male 24.4
#> 4 female 40.7
#> 5 male 22.0
#> 6 male 42.1
#> # ... with 18 more rows

mean(df$alter)
#> [1] 31.17342
```

- 2) Lassen Sie sich deskriptive Statistiken ausgeben (`summary()`).

```
summary(df)
#> geschlecht alter
#> Length:24 Min. :20.27
#> Class :character 1st Qu.:24.29
#> Mode :character Median :30.15
#> Mean :31.17
#> 3rd Qu.:37.94
#> Max. :44.08
```

## Matrizen

- 1) Kombinieren Sie die beiden Matrizen `m1` und `m2` (beide mit den Dimensionen [12, 4]), so dass eine neue Matrix `m3` entsteht, mit den Dimensionen [24, 4].

```
m1 <- matrix(rnorm(48, mean = 110, sd = 5), ncol = 4)
m2 <- matrix(rnorm(48, mean = 100, sd = 10), ncol = 4)

m1
#> [,1] [,2] [,3] [,4]
#> [1,] 112.6577 110.3748 114.1209 104.0900
#> [2,] 116.7024 120.9622 111.6725 113.6405
#> [3,] 114.4603 109.6374 107.9318 102.1479
#> [4,] 109.8077 109.1152 108.8644 114.3867
#> [5,] 113.5108 110.4932 100.2663 112.1465
#> [6,] 119.0698 110.9733 113.7455 111.6653
#> [7,] 109.4984 114.9592 106.6855 106.4838
#> [8,] 110.6606 112.5256 111.0061 114.9715
#> [9,] 108.3435 101.4709 107.8447 107.2151
#> [10,] 108.1350 110.0513 112.4929 101.0572
#> [11,] 110.8037 113.1775 113.9552 102.4492
#> [12,] 104.0074 103.4657 110.5090 110.2850

m2
```

```

#> [,1] [,2] [,3] [,4]
#> [1,] 107.06283 97.57565 100.09972 106.05653
#> [2,] 94.31888 121.33598 89.61459 120.11837
#> [3,] 94.44192 102.61623 113.93752 89.38039
#> [4,] 82.87758 108.59534 101.22006 94.78172
#> [5,] 97.84665 114.72505 104.53691 97.47717
#> [6,] 108.32954 89.14035 107.05284 88.84799
#> [7,] 112.39876 120.85322 93.22207 116.04059
#> [8,] 111.42864 98.94615 113.08818 99.33227
#> [9,] 111.44163 99.61579 102.23889 94.39533
#> [10,] 110.92366 92.24573 89.61547 115.96388
#> [11,] 95.51570 114.81185 91.85183 102.81788
#> [12,] 113.37321 88.37608 109.54577 100.70044
m3 <- rbind(m1, m2)
m3
#> [,1] [,2] [,3] [,4]
#> [1,] 112.65772 110.37480 114.12094 104.09003
#> [2,] 116.70244 120.96217 111.67248 113.64050
#> [3,] 114.46026 109.63743 107.93183 102.14786
#> [4,] 109.80768 109.11518 108.86436 114.38675
#> [5,] 113.51080 110.49324 100.26626 112.14648
#> [6,] 119.06984 110.97326 113.74550 111.66534
#> [7,] 109.49837 114.95920 106.68548 106.48376
#> [8,] 110.66056 112.52559 111.00613 114.97145
#> [9,] 108.34351 101.47088 107.84474 107.21505
#> [10,] 108.13497 110.05134 112.49292 101.05717
#> [11,] 110.80370 113.17754 113.95516 102.44920
#> [12,] 104.00743 103.46569 110.50899 110.28505
#> [13,] 107.06283 97.57565 100.09972 106.05653
#> [14,] 94.31888 121.33598 89.61459 120.11837
#> [15,] 94.44192 102.61623 113.93752 89.38039
#> [16,] 82.87758 108.59534 101.22006 94.78172
#> [17,] 97.84665 114.72505 104.53691 97.47717
#> [18,] 108.32954 89.14035 107.05284 88.84799
#> [19,] 112.39876 120.85322 93.22207 116.04059
#> [20,] 111.42864 98.94615 113.08818 99.33227
#> [21,] 111.44163 99.61579 102.23889 94.39533
#> [22,] 110.92366 92.24573 89.61547 115.96388
#> [23,] 95.51570 114.81185 91.85183 102.81788
#> [24,] 113.37321 88.37608 109.54577 100.70044

```

- 2) Wählen Sie aus m3 die Elemente so aus (mit Matrix subsetting), dass Sie die ursprüngliche Matrix m1 erhalten.

```

m3[1:12,]
#> [,1] [,2] [,3] [,4]
#> [1,] 112.6577 110.3748 114.1209 104.0900
#> [2,] 116.7024 120.9622 111.6725 113.6405
#> [3,] 114.4603 109.6374 107.9318 102.1479
#> [4,] 109.8077 109.1152 108.8644 114.3867
#> [5,] 113.5108 110.4932 100.2663 112.1465
#> [6,] 119.0698 110.9733 113.7455 111.6653
#> [7,] 109.4984 114.9592 106.6855 106.4838
#> [8,] 110.6606 112.5256 111.0061 114.9715

```

```
#> [9,] 108.3435 101.4709 107.8447 107.2151
#> [10,] 108.1350 110.0513 112.4929 101.0572
#> [11,] 110.8037 113.1775 113.9552 102.4492
#> [12,] 104.0074 103.4657 110.5090 110.2850
m1
#> [,1] [,2] [,3] [,4]
#> [1,] 112.6577 110.3748 114.1209 104.0900
#> [2,] 116.7024 120.9622 111.6725 113.6405
#> [3,] 114.4603 109.6374 107.9318 102.1479
#> [4,] 109.8077 109.1152 108.8644 114.3867
#> [5,] 113.5108 110.4932 100.2663 112.1465
#> [6,] 119.0698 110.9733 113.7455 111.6653
#> [7,] 109.4984 114.9592 106.6855 106.4838
#> [8,] 110.6606 112.5256 111.0061 114.9715
#> [9,] 108.3435 101.4709 107.8447 107.2151
#> [10,] 108.1350 110.0513 112.4929 101.0572
#> [11,] 110.8037 113.1775 113.9552 102.4492
#> [12,] 104.0074 103.4657 110.5090 110.2850
m3[1:12,] == m1
#> [,1] [,2] [,3] [,4]
#> [1,] TRUE TRUE TRUE TRUE
#> [2,] TRUE TRUE TRUE TRUE
#> [3,] TRUE TRUE TRUE TRUE
#> [4,] TRUE TRUE TRUE TRUE
#> [5,] TRUE TRUE TRUE TRUE
#> [6,] TRUE TRUE TRUE TRUE
#> [7,] TRUE TRUE TRUE TRUE
#> [8,] TRUE TRUE TRUE TRUE
#> [9,] TRUE TRUE TRUE TRUE
#> [10,] TRUE TRUE TRUE TRUE
#> [11,] TRUE TRUE TRUE TRUE
#> [12,] TRUE TRUE TRUE TRUE
```

## Character vectors

Generieren Sie aus den Variablen ID, Initialen und Alter eine neue Variable, welche so aussieht:

```
"1-RS-44" "2-MM-78" "3-PD-22" "4-PG-34" "5-DK-67" "1-RS-59"
ID <- c(1, 2, 3, 4, 5)
Initialen <- c("RS", "MM", "PD", "PG", "DK")
Alter <- c(44, 78, 22, 34, 67, 59)

personen <- paste(ID, Initialen, Alter, sep = "-")
personen
#> [1] "1-RS-44" "2-MM-78" "3-PD-22" "4-PG-34" "5-DK-67" "1-RS-59"
```

## Data Frame

Ändern Sie die Reihenfolge der Faktorstufen des Datensatzes alk\_aggr, so dass placebo die Referenzkategorie bildet.

```
library(dplyr)
library(tidyr)
```

```

kein_alkohol <- c(64, 58, 64)
placebo <- c(74, 79, 72)
anti_placebo <- c(71, 69, 67)
alkohol <- c(69, 73, 74)

alk_aggr <- data_frame(kein_alkohol = kein_alkohol,
 placebo = placebo,
 anti_placebo = anti_placebo,
 alkohol = alkohol)

alk_aggr <- alk_aggr %>%
 gather(key = alkoholbedingung, value = aggressivitaet) %>%
 mutate(alkoholbedingung = factor(alkoholbedingung))

alk_aggr
#> # A tibble: 12 x 2
#> alkoholbedingung aggressivitaet
#> <fct> <dbl>
#> 1 kein_alkohol 64
#> 2 kein_alkohol 58
#> 3 kein_alkohol 64
#> 4 placebo 74
#> 5 placebo 79
#> 6 placebo 72
#> # ... with 6 more rows

levels(alk_aggr$alkoholbedingung)
#> [1] "alkohol" "anti_placebo" "kein_alkohol" "placebo"

alk_aggr$alkoholbedingung <- factor(alk_aggr$alkoholbedingung,
 levels = c("placebo",
 "anti_placebo",
 "kein_alkohol",
 "alkohol"))

levels(alk_aggr$alkoholbedingung)
#> [1] "placebo" "anti_placebo" "kein_alkohol" "alkohol"

alternative Lösung
alk_aggr$alkoholbedingung <- relevel(alk_aggr$alkoholbedingung, ref = "placebo")
levels(alk_aggr$alkoholbedingung)
#> [1] "placebo" "anti_placebo" "kein_alkohol" "alkohol"

```

## Fortgeschrittene Aufgabe

- 1) Wählen Sie aus einem numerischen Vektor nur die geraden Zahlen:

```

x <- seq(1, 20, by = 1)
x
#> [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

```

Tipp: Sie brauchen dafür den modulo operator `%%`. Geraden Zahlen sind durch 2 teilbar, d.h. es gibt bei der ganzzahligen Teilung keinen Rest.

```

Gerade Zahlen sind durch 2 teilbar. Wir wollen nun für jedes Element in `x` den Rest, wenn wir durch
x %% 2

```

```
#> [1] 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
Dies gibt uns einen logischen Vektor; wir machen daraus eine Indexvariable:
index <- x %% 2 == 0
index
#> [1] FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE FALSE
#> [12] TRUE FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE
Und nun müssen wir damit `x` indizieren:
gerade_zahlen <- x[index]
gerade_zahlen
#> [1] 2 4 6 8 10 12 14 16 18 20
```

2) Machen Sie dasselbe für ungerade Zahlen.

```
x <- seq(1, 20, by = 1)
x
#> [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

wir wählen alle Zahlen aus, deren Rest ungleich Null ist
index <- x %% 2 != 0

und indizieren damit x
ungerade_zahlen <- x[index]

ungerade_zahlen
#> [1] 1 3 5 7 9 11 13 15 17 19

dasselbe in einer Zeile:
x[x %% 2 != 0]
#> [1] 1 3 5 7 9 11 13 15 17 19
```

# Chapter 3

## Datensätze

Wir werden nun anfangen, mit Datensätzen als Data Frames zu arbeiten. Zunächst werden wir diese selber kreieren, und dann Datensätze importieren. In einem nachfolgenden Kapitel werden wir Funktionen des Packages `dplyr` kennenlernen, mit denen es wesentlich einfacher ist, Variablen und Fälle auszuwählen.

### 3.1 Datensätze selber erstellen

#### 3.1.1 Ohne Messwiederholung

Wir erstellen nun einen Datensatz mit einem `between-subjects` Faktor.

■ Erstellen Sie einen Data Frame. Dieser soll zwei Variablen beinhalten:

**Aggressivität:** das Ausmass des aggressiven Verhaltens, gemessen über die Stärke eines elektrischen Schocks, den die Versuchspersonen einer anderen (fiktiven) Person applizieren sollten. Dies könnte die abhängige Variable in einer ANOVA sein.

**Alkoholkonsum:** Vier Experimentalgruppen (Faktorstufen), welche die Alkoholbedingung repräsentieren. Die Stufen sind `kein Alkohol`, `Placebo`, `Anti-Placebo` und `Alkohol`.

Gegeben sind folgende Daten (als Vektoren): ■

```
kein_alkohol <- c(64, 58, 64)
placebo <- c(74, 79, 72)
anti_placebo <- c(71, 69, 67)
alkohol <- c(69, 73, 74)
```

■ Das Ziel ist ein solcher Data Frame: ■

```
alkoholkonsum
#> # A tibble: 12 x 2
#> alkoholbedingung aggressivitaet
#> <fct> <dbl>
#> 1 kein_alkohol 64
#> 2 kein_alkohol 58
#> 3 kein_alkohol 64
#> 4 placebo 74
#> 5 placebo 79
#> 6 placebo 72
#> # ... with 6 more rows
```

Wie erstellen wir aus vier individuellen Vektoren einen solchen Data Frame? Eine mögliche Lösung wäre, zuerst aus jedem der vier Vektoren einen Data Frame zu erstellen und dann die Alkoholbedingung hinzuzufügen. Anschliessend können wir die vier Vektoren zusammenfügen, und die Alkoholbedingung zu einem `factor` konvertieren.

```
Zuerst dplyr package laden:
library(dplyr)

kein_alkohol <- c(64, 58, 64)
kein_alkohol <- data_frame(aggressivitaet = kein_alkohol,
 alkoholbedingung = "kein_alkohol")
kein_alkohol
#> # A tibble: 3 x 2
#> aggressivitaet alkoholbedingung
#> <dbl> <chr>
#> 1 64 kein_alkohol
#> 2 58 kein_alkohol
#> 3 64 kein_alkohol

placebo <- c(74, 79, 72)
placebo <- data_frame(aggressivitaet = placebo,
 alkoholbedingung = "placebo")
placebo
#> # A tibble: 3 x 2
#> aggressivitaet alkoholbedingung
#> <dbl> <chr>
#> 1 74 placebo
#> 2 79 placebo
#> 3 72 placebo

anti_placebo <- c(71, 69, 67)
anti_placebo <- data_frame(aggressivitaet = anti_placebo,
 alkoholbedingung = "anti_placebo")
anti_placebo
#> # A tibble: 3 x 2
#> aggressivitaet alkoholbedingung
#> <dbl> <chr>
#> 1 71 anti_placebo
#> 2 69 anti_placebo
#> 3 67 anti_placebo

alkohol <- c(69, 73, 74)
alkohol <- data_frame(aggressivitaet = alkohol,
 alkoholbedingung = "alkohol")
alkohol
#> # A tibble: 3 x 2
#> aggressivitaet alkoholbedingung
#> <dbl> <chr>
#> 1 69 alkohol
#> 2 73 alkohol
#> 3 74 alkohol
```

Nun können wir diese vier Data Frames mit `rbind()` oder `bind_rows()` zusammenfügen. Die Funktion

`bind_rows()` ist im `dplyr` package, und hat einige Vorteile (in diesem Beispiel spielt es keine Rolle, welche Sie benützen).

```
alk_aggr <- bind_rows(kein_alkohol,
 placebo,
 anti_placebo,
 alkohol)
```

```
alk_aggr
#> # A tibble: 12 x 2
#> aggressivitaet alkoholbedingung
#> <dbl> <chr>
#> 1 64 kein_alkohol
#> 2 58 kein_alkohol
#> 3 64 kein_alkohol
#> 4 74 placebo
#> 5 79 placebo
#> 6 72 placebo
#> # ... with 6 more rows
```

Nun müssen wir noch die Variable `alkoholbedingung` zu einem `factor` konvertieren:

```
alk_aggr$alkoholbedingung <- factor(alk_aggr$alkoholbedingung)
```

```
alk_aggr
#> # A tibble: 12 x 2
#> aggressivitaet alkoholbedingung
#> <dbl> <fct>
#> 1 64 kein_alkohol
#> 2 58 kein_alkohol
#> 3 64 kein_alkohol
#> 4 74 placebo
#> 5 79 placebo
#> 6 72 placebo
#> # ... with 6 more rows
```

- Oft wird auch die Funktion `as.factor()` verwendet. Diese konvertiert ein bestehendes Objekt zu einem `factor`, und würde hier zu demselben Ergebnis führen. ■

Der Datensatz ist nun komplett:

```
alk_aggr
#> # A tibble: 12 x 2
#> aggressivitaet alkoholbedingung
#> <dbl> <fct>
#> 1 64 kein_alkohol
#> 2 58 kein_alkohol
#> 3 64 kein_alkohol
#> 4 74 placebo
#> 5 79 placebo
#> 6 72 placebo
#> # ... with 6 more rows
```

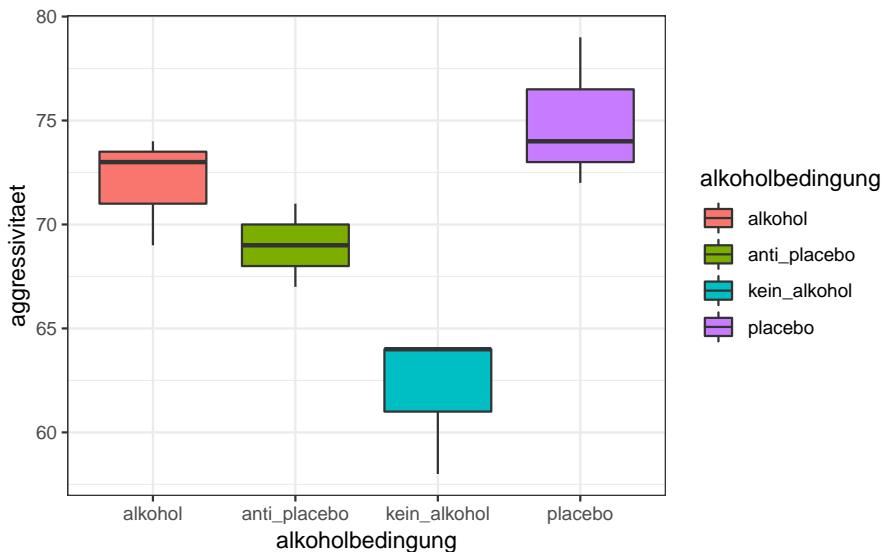
In diesem Datensatz haben wir zwei Variablen - einen Gruppierungsfaktor `alkoholbedingung` und eine Messvariable `aggressivitaet`, und die Stufen des Faktors sind nicht messwiederholt. Jede Beobachtung steht auf einer Zeile, und jede Variable steht in einer Spalte - ein solcher Datensatz ist im *long* Format

dargestellt.

- Welche Faktorstufe ist die Referenzkategorie? ■

- Als kleine Vorschau: wir können diesen Datensatz nun einfach grafisch darstellen, z.B. mit einem Boxplot-Diagramm. ■

```
library(ggplot2)
alk_aggr %>%
 ggplot(aes(x = alkoholbedingung,
 y = aggressivitaet,
 fill = alkoholbedingung)) +
 geom_boxplot() +
 theme_bw()
```



- Die Methode, welche wir hier angewandt haben, um einen Datensatz zu erstellen, ist keine sehr gute Lösung. Wir haben zu viel manuell gemacht, obwohl der Computer eigentlich diese Aufgaben für uns übernehmen sollte. Noch wichtiger: einige Arbeitsschritte haben wir mehrmals ausgeführt, und vielleicht haben wir `copy/paste` benutzt. Dies kann gefährlich sein, da sich Fehler einschleichen.

Wir werden in Kürze eine elegantere Methode kennenlernen, um Data Frames zu erstellen, und zu transformieren. Dabei werden wir so vorgehen, dass wir die vier Vektoren spaltenweise zusammenfügen und danach den Data Frame konvertieren (reshaping). ■

### 3.1.2 Mit Messwiederholung

Bei einem Faktor mit Messwiederholung sind wir von SPSS ein anderes Format gewöhnt. Wenn die Stufen des `repeated measures` Faktors nicht in einer Spalte stehen, sondern in separaten Spalten, ist der Datensatz nicht im *long*, sondern im *wide* Format. Wir veranschaulichen das mit einem weiteren Beispiel.

- Erstellen Sie einen Data Frame mit den folgenden Variablen:

**Vpn:** Die Versuchspersonennummer.

**Zufriedenheit:** Ein Zufriedenheitsrating auf einer Skala von 0-100 Punkten.

**Messzeitpunkt:** Ein Faktor mit zwei Stufen, welche zwei unterschiedliche Messzeitpunkte repräsentieren.

Wir konstruieren nun diesen Datensatz im *wide* Format.

```
Vpn <- c("VP_1", "VP_2", "VP_3", "VP_4")
Vpn <- factor(Vpn)
N ist die Anzahl Vpn
N <- length(Vpn)

Daten werden simuliert - Statistiker machen sowas oft, aber NUR um etwas zu illustrieren,
oder um statistische Verfahren zu testen.

set.seed(1234)

zufriedenheit_t1 <- round(rnorm(N, mean = 60, sd = 10), digits = 2)
zufriedenheit_t1
#> [1] 47.93 62.77 70.84 36.54
zufriedenheit_t2 <- round(rnorm(N, mean = 75, sd = 10), digits = 2)
zufriedenheit_t2
#> [1] 79.29 80.06 69.25 69.53
```

Nun können wir die Variablen mit `data_frame()` zu einem Datensatz zusammenfügen:

```
zufriedenheit <- data_frame(Vpn = Vpn,
 zufriedenheit_t1 = zufriedenheit_t1,
 zufriedenheit_t2 = zufriedenheit_t2)

zufriedenheit
#> # A tibble: 4 x 3
#> Vpn zufriedenheit_t1 zufriedenheit_t2
#> <fct> <dbl> <dbl>
#> 1 VP_1 47.9 79.3
#> 2 VP_2 62.8 80.1
#> 3 VP_3 70.8 69.2
#> 4 VP_4 36.5 69.5
```

Mit diesem Datensatz können wir in R einen T-Test durchführen, aber das wide Format ist in vielerlei Hinsicht nicht optimal; viele Verfahren verlangen einen Datensatz im long Format. Um die Daten mit `ggplot2` grafisch darzustellen, ist es ebenfalls sinnvoll, den Datensatz in ein long Format zu konvertieren. Datenkonvertierung wird oft als *reshaping* bezeichnet. Wir werden im nächsten Kapitel darauf eingehen und die `tidyverse` Packages kennenlernen. Ein Begriff, welchen wir immer häufiger antreffen, ist *tidy* Data. Dies bezieht sich eben auf das Datenformat; das long Format wird bevorzugt (ist *tidier* als das wide Format). Wenn wir uns daran halten, erleichtern wir uns die Arbeit mit R erheblich, und wir können von den `tidyverse` Packages profitieren, welche alle sehr gut aufeinander abgestimmt sind.

Bevor wir das Data *reshaping* automatisieren, ist es aber sinnvoll, einen Datensatz manuell von wide zu long zu konvertieren.

### 3.1.3 Manuelle Konversion von wide zu long

Jede Person gibt zu zwei Messzeitpunkten ein Rating - diese Ratings stellen die Beobachtungen dar. In einem long Datensatz stehen die *Beobachtungen* in den Zeilen, daher erhält jede Beobachtung eine Zeile, und nicht mehr jede Versuchsperson. Die Versuchsperson wird gemäss der Anzahl Beobachtungen pro Person dupliziert.

```
Wir nennen den Datensatz nun zufriedenheit_wide
zufriedenheit_wide <- zufriedenheit

Vpn <- rep(zufriedenheit_wide$Vpn, each = 2)
Vpn
```

```
#> [1] VP_1 VP_1 VP_2 VP_2 VP_3 VP_3 VP_4 VP_4
#> Levels: VP_1 VP_2 VP_3 VP_4
```

Nun müssen wir aus den beiden Variablen `zufriedenheit_t1` und `zufriedenheit_t2` des wide Datensatzes einen Faktor `messzeitpunkt` und eine `rating` Variable (die die ‘geratete’ Zufriedenheit repräsentiert) erstellen:

```
messzeitpunkt <- rep(c("t1", "t2"), times = N)
messzeitpunkt <- as.factor(messzeitpunkt)

messzeitpunkt
#> [1] t1 t2 t1 t2 t1 t2 t1 t2
#> Levels: t1 t2

rating <- c(rbind(zufriedenheit_wide$zufriedenheit_t1,
 zufriedenheit_wide$zufriedenheit_t2))

oder

rating <- as.vector(rbind(zufriedenheit_wide$zufriedenheit_t1,
 zufriedenheit_wide$zufriedenheit_t2))

rating
#> [1] 47.93 79.29 62.77 80.06 70.84 69.25 36.54 69.53
```

■ Dieser Schritt war etwas schwierig nachzuvollziehen. Das macht nichts, Sie müssen das in Zukunft auch nicht selber machen. Wir haben hier lediglich die beiden Spalten `zufriedenheit_t1` und `zufriedenheit_t2` als Zeilenvektoren zu einer Matrix zusammengefügt, und dann mit `c()` oder `as.vector()` zu einem Vektor konvertiert. Dabei haben wir uns die Tatsache zunutze gemacht, dass R zuerst die Spalten einer Matrix nimmt (column-major). ■

```
rbind(zufriedenheit_wide$zufriedenheit_t1,
 zufriedenheit_wide$zufriedenheit_t2)
#> [,1] [,2] [,3] [,4]
#> [1,] 47.93 62.77 70.84 36.54
#> [2,] 79.29 80.06 69.25 69.53
```

■ ■ ■

Wir haben nun alle 3 Variablen, die wir für den Datensatz im long Format brauchen, und können diese nun zusammenfügen:

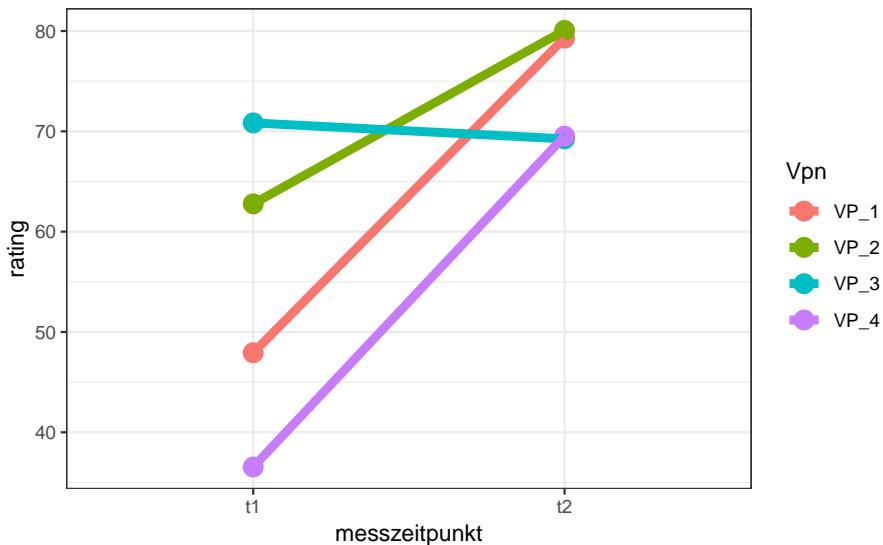
```
zufriedenheit_long <- data_frame(Vpn = Vpn,
 messzeitpunkt = messzeitpunkt,
 rating = rating)

zufriedenheit_long
#> # A tibble: 8 x 3
#> Vpn messzeitpunkt rating
#> <fct> <fct> <dbl>
#> 1 VP_1 t1 47.9
#> 2 VP_1 t2 79.3
#> 3 VP_2 t1 62.8
#> 4 VP_2 t2 80.1
```

```
#> 5 VP_3 t1 70.8
#> 6 VP_3 t2 69.2
#> # ... with 2 more rows
```

Der Datensatz mit Messwiederholung kann jetzt ähnlich wie derjenige ohne Messwiederholung mit `ggplot2` grafisch dargestellt werden.

```
library(ggplot2)
zufriedenheit_long %>%
 ggplot(aes(x = messzeitpunkt,
 y = rating,
 group = Vpn, colour = Vpn)) +
 geom_point(size = 4) +
 geom_line(size = 2) +
 theme_bw() # macht den Hintergrund weiss
```



Manuelles *reshaping* ist eher mühsam, und sollte vermieden werden; das obige Beispiel soll nur zur Illustration dienen. Wir werden in Zukunft dafür eine Funktion aus dem `tidyverse` Package verwenden: `gather()`.

Als kleine Vorschau: die Konvertierung von wide zu long würden wir so machen:

```
library(tidyr)
library(stringr)

zufriedenheit_wide
#> # A tibble: 4 x 3
#> Vpn zufriedenheit_t1 zufriedenheit_t2
#> <fct> <dbl> <dbl>
#> 1 VP_1 47.9 79.3
#> 2 VP_2 62.8 80.1
#> 3 VP_3 70.8 69.2
#> 4 VP_4 36.5 69.5

zufriedenheit_long <- zufriedenheit_wide %>%
 gather(key = messzeitpunkt, value = rating, -Vpn)

zufriedenheit_long
#> # A tibble: 8 x 3
```

```

#> Vpn messzeitpunkt rating
#> <fct> <chr> <dbl>
#> 1 VP_1 zufriedenheit_t1 47.9
#> 2 VP_2 zufriedenheit_t1 62.8
#> 3 VP_3 zufriedenheit_t1 70.8
#> 4 VP_4 zufriedenheit_t1 36.5
#> 5 VP_1 zufriedenheit_t2 79.3
#> 6 VP_2 zufriedenheit_t2 80.1
#> # ... with 2 more rows

Wir wollen nun 'zufriedenheit_' von den Faktorstufen entfernen, so
dass die Stufen 't1' und 't2' heißen. Dafür verwenden wir die Funktion 'str_replace()'

zufriedenheit_long$messzeitpunkt <- zufriedenheit_long$messzeitpunkt %>%
 str_replace(".*_", "") %>%
 as.factor()

zufriedenheit_long
#> # A tibble: 8 x 3
#> Vpn messzeitpunkt rating
#> <fct> <fct> <dbl>
#> 1 VP_1 t1 47.9
#> 2 VP_2 t1 62.8
#> 3 VP_3 t1 70.8
#> 4 VP_4 t1 36.5
#> 5 VP_1 t2 79.3
#> 6 VP_2 t2 80.1
#> # ... with 2 more rows

Die obige Schreibweise benutzt den 'pipe' Operator %>%: wir werden bald
mehr darüber erfahren.
Wir können äquivalent auch die Funktionen verschachtelt
aufrufen:
zufriedenheit_long <- gather(data = zufriedenheit_wide,
 key = messzeitpunkt,
 value = rating, -Vpn)
zufriedenheit_long$messzeitpunkt <-
 as.factor(str_replace(zufriedenheit_long$messzeitpunkt,
 ".*_", ""))

```

### 3.1.4 Long vs. wide

Um den Unterschied zwischen einem long und einem wide Datensatz zu verdeutlichen, zeigen wir hier nochmals beide Varianten:

```

wide
zufriedenheit_wide
#> # A tibble: 4 x 3
#> Vpn zufriedenheit_t1 zufriedenheit_t2
#> <fct> <dbl> <dbl>
#> 1 VP_1 47.9 79.3
#> 2 VP_2 62.8 80.1
#> 3 VP_3 70.8 69.2
#> 4 VP_4 36.5 69.5

```

```
long
zufriedenheit_long
#> # A tibble: 8 x 3
#> Vpn messzeitpunkt rating
#> <fct> <fct> <dbl>
#> 1 VP_1 t1 47.9
#> 2 VP_2 t1 62.8
#> 3 VP_3 t1 70.8
#> 4 VP_4 t1 36.5
#> 5 VP_1 t2 79.3
#> 6 VP_2 t2 80.1
#> # ... with 2 more rows
```

- Versuchen Sie, aus dem *long* Datensatz den ersten Messzeitpunkt für alle Versuchspersonen zu extrahieren. Sie müssen dazu die Zeilen des Data Frames mit einem logischen Vektor indizieren, und alle Spalten auswählen.

Extrahieren Sie die Zufriedenheitsratings zu beiden Messzeitpunkten für die erste Versuchsperson. ■

## 3.2 Daten importieren

Meistens arbeiten wir jedoch nicht mit selber generierten Datensätzen, sondern wollen diese aus Textdateien, Excel-Spreadsheets oder SPSS-Dateien importieren.

Laden Sie bitte folgende Datensätze von ILIAS herunter:

- 1) zufriedenheit.csv
- 2) zufriedenheit-semicolon.csv
- 3) zufriedenheit.sav
- 4) zufriedenheit.xls

Es handelt sich um den Datensatz zur Zufriedenheit, den wir oben erstellt haben. `zufriedenheit.csv` ist eine Textdatei, in dem die Spalten durch Kommata getrennt sind (comma-separated values). `zufriedenheit-semicolon.csv` ist ebenfalls eine Textdatei, die Spalten sind hier aber durch Strichpunkte getrennt (semicolons), wie es im deutschsprachigen Raum üblich ist. `zufriedenheit.sav` und `zufriedenheit.xls` sind SPSS- bzw. Excel-Dateien.

Kreieren Sie in Ihrem Projektordner in RStudio einen neuen Ordner und nennen Sie diesen “data”. Speichern Sie nun die heruntergeladenen Datenfiles in diesem Ordner. Wir werden diese Dateien nun der Reihe nach importieren.

Es gibt in RStudio zwei Möglichkeiten, Datensätze zu importieren:

- 1) Mit Funktionsaufrufen: `read_csv()`, `read_csv2()` (für ‘;’), `read_sav()` und `read_excel()`.
- 2) Mit dem GUI: Das Menu kann entweder via ‘File > Import Dataset’ oder im Environment-Bereich aufgerufen werden.

Die zweite Option ist am Anfang viel einfacher und hat zwei grosse Vorteile. Erstens werden alle Optionen für den Import angezeigt (diese sind auch als Funktionsargumente verfügbar), und zweitens generiert RStudio den entsprechenden R Code. Wenn man sich nicht sicher ist, kann man beim ersten Mal das GUI benützen, und in Zukunft dann den generierten Code benützen. Dies ist oft schneller und hat wiederum den Vorteil, dass man alle Schritte für die Datenanalyse in einem R Script/Notebook festhalten kann.

### 3.2.1 Comma-separated values (CSV) Dateien

Die Funktion, welche wir benötigen, um CSV Dateien zu importieren, befindet sich im `readr` Package. Dieses müssen wir zuerst laden:

```
library(readr)
```

Da dieses Package zu dem Metapackage `tidyverse` gehört, können wir einfach dieses laden:

```
library(tidyverse)
```

Zuerst importieren wir den Datensatz via GUI. Klicken Sie in *Environment* auf *Import Dataset > From Text (readr)* (oder ‘File > Import Dataset > From Text (readr)’). Im Dialogfenster sehen Sie rechts unten einen *Code Preview*. Hier steht am Anfang:

```
library(readr)
dataset <- read_csv(NULL)
View(dataset)
```

Links unten sehen Sie alle Optionen für den Import. Diese Optionen sind Argumente der `read_csv()` oder der allgemeineren `read_delim()` Funktion:

```
args(read_csv)
#> function (file, col_names = TRUE, col_types = NULL, locale = default_locale(),
#> na = c("", "NA"), quoted_na = TRUE, quote = "\"", comment = "",
#> trim_ws = TRUE, skip = 0, n_max = Inf, guess_max = min(1000,
#> n_max), progress = show_progress(), skip_empty_rows = TRUE)
#> NULL
```

Wählen Sie über *File / Browse* die Datei `zufriedenheit.csv` aus. Sie sehen nun den Pfad dieser Datei sowie eine *Data Preview*. Hier werden die Daten gezeigt, mit Angaben zum Datentyp.

Vpn (character)	messzeitpunkt (character)	rating (double)
VP_1	t1	64.29
VP_2	t1	65.06
VP_3	t1	54.25
VP_4	t1	54.53
VP_1	t2	67.24
VP_2	t2	75.64
VP_3	t2	84.59
VP_4	t2	73.90

Sie sollten sehen, dass sowohl die Variable `Vpn` als auch die Variable `messzeitpunkt` als `character` vectors importiert wurden. Diese werden wir nachher zu Faktoren konvertieren.

Bei *Import Options* wird automatisch ein Name für den Data Frame generiert; dieser entspricht dem Dateinamen (ohne das Suffix ‘.csv.’).

Sie probieren am besten selber aus, welche Auswirkung es hat, wenn Sie die Optionen ändern. Wenn sie z.B. “First Row as Names” nicht auswählen, wird R die Variablennamen, welche in der ersten Zeile stehen, nicht verwenden. Eine weitere wichtige Option ist “NA”: hier können Sie angeben, welche Werte in der Datei als fehlende Werte behandelt werden sollen.

Nun klicken Sie bitte auf *Import*. Im *Environment* Bereich erscheint ein Data Frame (**zufriedenheit**), und der Code, der verwendet wurde, erscheint in der Konsole.

```
library(readr)
zufriedenheit <- read_csv("data/zufriedenheit.csv")
#> Parsed with column specification:
#> cols(
#> Vpn = col_character(),
#> messzeitpunkt = col_character(),
#> rating = col_double()
#>)
```

Nun müssen wir die beiden Gruppierungsvariablen zu Faktoren konvertieren:

```
zufriedenheit$Vpn <- as.factor(zufriedenheit$Vpn)
zufriedenheit$messzeitpunkt <- as.factor(zufriedenheit$messzeitpunkt)

zufriedenheit
#> # A tibble: 8 x 3
#> Vpn messzeitpunkt rating
#> <fct> <fct> <dbl>
#> 1 VP_1 t1 64.3
#> 2 VP_2 t1 65.1
#> 3 VP_3 t1 54.2
#> 4 VP_4 t1 54.5
#> 5 VP_1 t2 67.2
#> 6 VP_2 t2 75.6
#> # ... with 2 more rows
```

- Machen Sie nun dasselbe mit der **zufriedenheit-semicolon.csv** Datei. Hier müssen Sie unbedingt bei den Optionen für “Delimiter” “Semicolon” auswählen, da R normalerweise ein Komma als Trennzeichen erwartet.

Sie erhalten nun im *Code Preview* den Code:

```
library(readr)
zufriedenheit_semicolon <- read_delim("data/zufriedenheit-semicolon.csv",
 ";", escape_double = FALSE, trim_ws = TRUE)
```

- Versuchen Sie, die Dateien ohne GUI, d.h. mit dem generierten R Code zu importieren. Für die zweite Datei können Sie auch die Funktion **read\_csv2()** verwenden.

Wenn wir einen Data Frame als **csv** Datei speichern wollen, verwenden wir dafür die Funktion **write\_csv()**:

```
write_csv(x = zufriedenheit_long, path = "data/zufriedenheit.csv")
```

### 3.2.2 SPSS-Dateien

Nun importieren wir denselben Datensatz, aber dieses Mal von einer SPSS-Datei (**zufriedenheit.sav**). Dafür benötigen wir die Funktion **read\_sav()**. Diese befindet sich im **haven** Package:

```
library(haven)
```

aber auch im `tidyverse` Package. Klicken Sie in *Environment* auf *Import Dataset > From SPSS*, und wählen Sie die Datei aus. Sie sehen den *Code Preview*:

```
library(haven)
zufriedenheit_spss <- read_sav("data/zufriedenheit.sav")
```

Im Gegensatz zum Importieren von CSV Dateien gibt es hier keine Optionen mit Ausnahme des Namens; diesen ändern wir zu `zufriedenheit_spss`. Klicken Sie bitte auf *Import*. Im *Environment* sehen Sie jetzt diese Variable. Im Unterschied zu dem von einer CSV-Datei importierten Data Frame haben die Variablen `zufriedenheit_spss` weitere Attribute. Das wichtigste dabei ist das `Labels` Attribut:

```
zufriedenheit_spss$Vpn
#> <Labelled double>
#> [1] 1 2 3 4 1 2 3 4
#>
#> Labels:
#> value label
#> 1 VP_1
#> 2 VP_2
#> 3 VP_3
#> 4 VP_4
```

Dieses beinhaltet die Werte-Labels im SPSS Datensatz. So können wir nachschauen, welche Codierung für kategoriale Variablen (Faktoren) verwendet wurde. Falls z.B. die Werte 0 und 1 für Geschlecht verwendet wurden, können wir herausfinden, welche Stufe den Wert 0 hat, und können die Stufen umcodieren. Wenn wir die Variablen zu Faktoren konvertieren, können wir die Funktion `as_factor()` aus dem `haven` package verwenden. Diese Funktion erlaubt es uns, entweder die Werte oder die Wertelabels als Stufen des Faktors zu benützen. Dazu benützen wir das Argument `levels`; dieses kann die Werte "default", "labels", "values" oder "both" annehmen. Aus den Help Pages, welche mit `?as_factor` aufgerufen werden können, entnehmen wir, dass "default" die sinnvollste Einstellung zu sein scheint. Damit werden Wertelabels benützt, falls sie vorhanden sind, andernfalls wird auf die Werte selber zurückgegriffen. "default" ist zudem auch die Standardeinstellung. Die anderen Optionen sind "both" (Werte und Werte-Labels werden zusammengefügt), "label" (nur Labels, falls nicht vorhanden, NA) und "values" (Nur Werte).

### Argumente der Funktion `as_factor()`

```
levels
 How to create the levels of the generated factor:
 "default": uses labels where available, otherwise the values. Labels are sorted by value.
 "both": like "default", but pastes together the level and value
 "label": use only the labels; unlabelled values become NA
 "values": use only the values

ordered
 If TRUE create an ordered (ordinal) factor, if FALSE (the default) create a regular (nominal) factor
```

Mit dem Argument `ordered`, welches entweder `TRUE` oder `FALSE` (Standardeinstellung) ist, kann ein ordinaler Faktor erstellt werden, d.h. ein Faktor, dessen Stufen in einer Rangfolge stehen.

```
zufriedenheit_spss$Vpn <- as_factor(zufriedenheit_spss$Vpn,
 levels = "default")
zufriedenheit_spss$messzeitpunkt <- as_factor(zufriedenheit_spss$messzeitpunkt,
 levels = "default")
```

Wir können unsere Data Frames im SPSS Format speichern:

```
write_sav(data = zufriedenheit_long, path = "data/zufriedenheit.sav")
```

- Importieren Sie den **Beispieldatensatz** (ebenfalls auf ILIAS). Versuchen Sie herauszufinden, welche Wertelabels die kategorialen Variablen haben.

### 3.2.3 Excel Dateien

Auch Excel Dateien können importiert werden. Wählen Sie *Import Dataset > From Excel*, und wählen Sie dann die Excel Datei. Nennen Sie die Variable `zufriedenheit_xls`. Unter den “Name” Feld sehen sie ein Dropdown-Menu mit den Namen “Sheet”. Hier können Sie angeben, welche “Worksheet” Sie importieren wollen. Wenn Sie hier “zufriedenheit” auswählen, erscheint folgender R Code im *Code Preview*:

```
library(readxl)
zufriedenheit_xls <- read_excel("data/zufriedenheit.xlsx",
 sheet = "zufriedenheit")
```

Die Funktion, welche wir brauchen, heisst `read_excel()`, und befindet sich im `readxl` package. Dieses muss explizit geladen werden, d.h. es wird nicht automatisch mit dem Befehl `library(tidyverse)` geladen.

Auch hier müssen kategoriale Variablen zu Faktoren konvertiert werden:

```
zufriedenheit_xls$Vpn <- as.factor(zufriedenheit_xls$Vpn)
zufriedenheit_xls$messzeitpunkt <- as.factor(zufriedenheit_xls$messzeitpunkt)
```

### 3.2.4 RData-Dateien

Unsere letzte Option ist eine RData-Datei. Dies ist ein **binary** Datei, d.h. es handelt sich in diesem Fall um *keine* Textdatei. Wir können mehrere Objekte aus dem R Workspace in eine RData-Datei speichern, und wieder daraus laden. Der grosse Vorteil daran ist, dass Attribute, wie z.B. die Objektklasse `factor` und die Faktorstufen `levels` mitgespeichert werden, und nicht wie beim Speichern als CSV verloren gehen. Ein weiterer Vorteil ist, dass die Datei komprimiert werden kann. Demgegenüber steht die Tatsache, dass eine RData Datei nicht ohne Weiteres in anderen Programmiersprachen gelesen werden kann. Deshalb sollte man möglichst darauf achten, Textdateien (CSV) zu verwenden, wenn man beabsichtigt, einen Datensatz jemand anderem zur Verfügung zu stellen.

Des Weiteren ist es auch empfehlenswert, alle Schritte der Datenanalyse in einem Script oder Notebook festzuhalten. Der Import von Dateien und die Umwandlung von Variablen kann so automatisiert werden.

Mit der Funktion `save()` können wir Objekte in unserem Workspace als .RData (oder .Rda) Dateien speichern. Z.B. können wir den Data Frame `zufriedenheit` speichern:

```
save(zufriedenheit, file = "data/zufriedenheit.Rda")
```

Wir können dieser Funktion auch eine Liste von Objekten als Argument übergeben:

```
save(zufriedenheit, zufriedenheit_spss, zufriedenheit_xls,
 file = "data/zufriedenheit_alle.Rda")
```

`zufriedenheit_alle.Rda` enthält nun die drei Data Frames.

Mit `load()` können wir diese auch wieder laden:

```
load(file = "data/zufriedenheit_alle.Rda")
```

## 3.3 Übungsaufgaben

### Daten importieren 1

- 1) Laden Sie den Datensatz `Therapy.sav` von ILIAS herunter, und importieren sie ihn.

```
library(haven)
Therapy <- read_sav("data/Therapy.sav")
```

2) Überprüfen Sie Codierung der Gruppenvariable. Welche ist die Referenzkategorie?

```
attributes(Therapy$Gruppe)
#> $format.spss
#> [1] "F8.2"
#>
#> $class
#> [1] "haven_labelled"
#>
#> $labels
#> Kontrollgruppe Therapiegruppe
#> 0 1
```

3) Wandeln Sie die Gruppierungsvariablen zu Faktoren um.

```
Vpnrs wird als numerischer Vektor importiert, und
hat kein 'labels' Attribut. Wir müssen as.factor()
verwenden.
attributes(Therapy$Vpnrs)
#> $format.spss
#> [1] "F3.0"
Therapy$Vpnrs <- as.factor(Therapy$Vpnrs)

Gruppe ist 'labelled'. Hier können wir as_factor()
verwenden.
Therapy$Gruppe <- as_factor(Therapy$Gruppe)
levels(Therapy$Gruppe)
#> [1] "Kontrollgruppe" "Therapiegruppe"
```

```
Therapy
#> # A tibble: 100 x 5
#> Vpnrs Gruppe Pretest Posttest Difference_PrePost
#> <dbl> <fct> <dbl> <dbl> <dbl>
#> 1 1 Kontrollgruppe 4.29 3.21 1.08
#> 2 2 Kontrollgruppe 6.18 5.99 0.190
#> 3 3 Kontrollgruppe 3.93 4.17 -0.239
#> 4 4 Kontrollgruppe 5.06 4.76 0.295
#> 5 5 Kontrollgruppe 6.45 5.64 0.814
#> 6 6 Kontrollgruppe 4.49 4.67 -0.180
#> # ... with 94 more rows
```

4) Ist dieser Datensatz im *long* oder im *wide* Format? Muss dieser Datensatz vielleicht umgewandelt werden? Falls ja, wie würden Sie das machen?

Der Datensatz ist im *wide* Format: Jede Person wurde zu zwei Messzeitpunkten gemessen. Die Variable **Messzeitpunkt** existiert aber nicht in diesem Datensatz. Stattdessen haben wir zwei Variablen, **Pretest** und **Posttest**, welche die beiden Stufen des Faktors **Messzeitpunkt** repräsentieren. Dieses Format wird in SPSS für Test mit Messwiederholten Variablen vorausgesetzt (für einen t-Test kann auch R damit umgehen), aber in R verlangen viele Verfahren, unter anderen auch **ggplot2**, einen Datensatz im *long* Format.

```
library(dplyr)
library(tidyr)
Therapy_long <- Therapy %>%
```

```
Difference_PrePost weglassen
select(-Difference_PrePost) %>%
zu long konvertieren
gather(key = messzeitpunkt,
 value = rating,
 Pretest, Posttest, -Vpnr, -Gruppe) %>%
mutate(messzeitpunkt = as.factor(messzeitpunkt))
```

Wir haben nun die beiden Messzeitpunkte zu einem Faktor mit 2 Stufen zusammengefügt. Außerdem haben wir jetzt nicht mehr eine Zeile pro Person, sondern eine Zeile pro Beobachtung (für jede Person haben wir zwei Beobachtungen).

```
Therapy_long
#> # A tibble: 200 x 4
#> Vpnr Gruppe messzeitpunkt rating
#> <fct> <fct> <fct> <dbl>
#> 1 1 Kontrollgruppe Pretest 4.29
#> 2 2 Kontrollgruppe Pretest 6.18
#> 3 3 Kontrollgruppe Pretest 3.93
#> 4 4 Kontrollgruppe Pretest 5.06
#> 5 5 Kontrollgruppe Pretest 6.45
#> 6 6 Kontrollgruppe Pretest 4.49
#> # ... with 194 more rows

levels(Therapy_long$messzeitpunkt)
#> [1] "Posttest" "Pretest"
```

Nun wollen wir wahrscheinlich eher `Pretest` als Referenzkategorie verwenden:

```
Therapy_long$messzeitpunkt <- relevel(Therapy_long$messzeitpunkt, ref = "Pretest")
```

## Daten importieren 2

- 1) Erstellen Sie einen (simulierten) Datensatz und exportieren Sie diesen entweder zu CSV, XLS oder SAV. Suchen Sie sich einen Partner und tauschen Sie Datensätze aus. Dieser Datensatz sollte mindestens einen Faktor und mindestens eine numerische Variable enthalten. Versuchen Sie auch, fehlende Werte (`NA`) einzufügen.
- 2) Sie erhalten einen Datensatz von Ihrem Partner. Versuchen Sie, diesen zu importieren, und eventuell vorhandene kategoriale Variablen zu Faktoren zu konvertieren. Versuchen Sie, mit fehlenden Werten umzugehen.



# Chapter 4

## Daten transformieren

### 4.1 Tidy data

Im letzten Beispiel haben wir einen Datensatz vom *wide* zum *long* Format konvertiert, so dass jede Variable im Datensatz eine eigene Spalte erhielt und jede Beobachtung eine eigene Zeile. Wenn diese Bedingungen erfüllt sind, sprechen wir von *tidy data*. Abgesehen davon, dass viele R Packages *tidy* Datensätze erfordern, gibt es einen weiteren wichtigen Grund, Datensätze in einem einheitlichen Format zu verwenden: die kognitive Belastung wird für uns geringer. Datenanalyse ist ohnehin schon schwierig genug, so dass wir nicht zusätzlich immer mit den Daten ‘kämpfen’ wollen (dies wird auf English nicht umsonst als ‘data wrangling’ bezeichnet).

Um solche Transformationen von Datensätzen auszuführen, benutzen wir Funktionen der Packages `tidyverse` und `dplyr` - diese gehören zum `tidyverse`. Dieses ist also lediglich eine Art ‘Metapackage’. Die `tidyverse` Packages werden so entwickelt, dass sie zueinander kompatibel sind und stellen sozusagen eine “Grammatik” der Datenverarbeitung zur Verfügung. Die Packages `readr` und `haven` haben wir bereits kennengelernt, in diesem Kapitel kommen `tidyverse` und `dplyr` dazu, und im nächsten Kapitel werden wir `ggplot2` kennenlernen.

**tidyverse:** Dieses Package verwenden wir, um Datensätze von *wide* zu *long* zu transformieren, und natürlich auch von *long* zu *wide*.

**dplyr:** Dieses Packages stellt eine Sammlung von Funktionen zur Verfügung, um Daten zu manipulieren: Fälle/Variablen auswählen, Daten zusammenfassen, neue Variablen kreieren, neue Datensätze erstellen.

Bevor wir nun mit Datentransformationen weiterfahren, lernen wir einen weiteren Operator kennen.

### 4.2 Der Pipe Operator

Wir haben schon festgestellt, dass Code schnell unübersichtlich werden kann, wenn wir eine Sequenz von Operationen ausführen. Dies führt zu verschachtelten Funktionsaufrufen.

Beispiel: Wir haben einen numerischen Vektor von Messwerten (an einer Stichprobe der Grösse  $n = 10$  erhoben oder wie hier zu Übungszwecken durch einen (Pseudo-)Zufallsgenerator generiert) und wollen diese zuerst zentrieren, dann die Standardabweichung berechnen, und anschliessend noch auf zwei Nachkommastellen runden.

```
set.seed(1283)
stichprobe <- rnorm(10, 24, 5)
stichprobe
#> [1] 24.74984 21.91726 23.98551 19.63019 23.96428 22.83092 18.86240
#> [8] 19.08125 23.76589 21.88846
```

Die gewünschte Berechnung der gerundeten Standardabweichung der zentrierten Werte können wir als verschachtelte Funktionsaufrufe durchführen:

```
round(sd(scale(stichprobe,
 center = TRUE,
 scale = FALSE)),
 digits = 2)
#> [1] 2.19
```

Die Funktion `scale()`, `sd()` und `round()` werden nun der Reihe nach ausgeführt (von innen nach aussen), und zwar so, dass der Output einer Funktion der nächsten Funktion als Input übergeben wird.

Die Funktionen `scale()` und `round()` haben zusätzlich noch Argumente: `center = TRUE`, `scale = FALSE`, bzw. `digits = 2`. Dies ist zwar effizient, aber führt zu Code, der schwierig zu lesen ist.

Eine Alternative dazu wäre, die Zwischenschritte als Variablen zu speichern:

```
stichprobe_z <- scale(stichprobe, center = TRUE,
 scale = FALSE)

sd_stichprobe_z <- sd(stichprobe_z)
sd_stichprobe_z_gerundet <- round(sd_stichprobe_z,
 digits = 2)

sd_stichprobe_z_gerundet
#> [1] 2.19
```

So steht jeder der Teilschritte in einer eigenen Zeile und wir verstehen den Code ohne Probleme. Diese Methode erfordert jedoch, dass wir zwei Variablen definieren, die wir eigentlich gar nicht brauchen.

Es gibt nun aber eine sehr elegante Methode, um Funktionen nacheinander aufzurufen, ohne diese Funktionen ineinander verschachtelt schreiben zu müssen: wir benutzen dafür den `pipe` Operator. Dieser wird vom Package `dplyr` zur Verfügung gestellt und sieht so aus:

```
library(dplyr)
```

```
%>%
```

und ist als *Infix* Operator definiert. Das bedeutet, dass er *zwischen* zwei Objekten steht, ähnlich wie ein mathematischer Operator. Der Name `pipe` ist so zu verstehen, dass wir ein Objekt an eine Funktionen ‘weiterleiten’ oder ‘übergeben’.

- Eigentlich befindet sich der `pipe` Operator im Package `magrittr`, und wird von `dplyr` importiert. Wenn sie mehr darüber erfahren wollen, können Sie hier nachschauen. ■

Dieser `pipe` Operator wird so oft verwendet, dass er schon eine eigene Tastenkombination hat: `Cmd+Shift+M` (MacOS) oder `Ctrl+Shift+M` (Windows und Linux).

Unser Beispiel von oben:

```
round(sd(scale(stichprobe,
 center = TRUE,
 scale = FALSE)),
 digits = 2)
#> [1] 2.19
```

wird mit dem `%>%` Operator zu:

```
library(dplyr)
stichprobe %>%
 scale(center = TRUE, scale = FALSE) %>%
 sd() %>%
 round(digits = 2)
#> [1] 2.19
```

Dieser Code ist so zu lesen:

- 1) Wir beginnen mit dem Objekt `stichprobe` und übergeben es mit `%>%` als Argument an die Funktion `scale()`
- 2) Wir wenden `scale()`, mit den zusätzlichen Argumenten `center = TRUE, scale = FALSE` darauf an, und übergeben den Output als Argument an die Funktion `sd()`
- 3) Wir wenden `sd()` an (ohne weitere Argumente) und reichen den Output als Argument weiter an `round()`
- 4) `round()`, mit dem weiteren Argument `digits = 2`, wird ausgeführt. Da kein weiterer pipe folgt, wird der Output in die Konsole geschrieben.

Somit ist klar: wenn wir das Resultat weiterverwenden möchten, müssen wir es einer Variablen zuweisen:

```
sd_stichprobe_z_gerundet <- stichprobe %>%
 scale(center = TRUE, scale = FALSE) %>%
 sd() %>%
 round(digits = 2)

sd_stichprobe_z_gerundet
#> [1] 2.19
```

Wir übergeben also mit `%>%` ein Objekt an eine Funktion. Wenn wir nichts weiter spezifizieren, ist dieses Objekt das erste Argument der Funktion. Die grossen Vorteile sind:

- Unser Code ist lesbarer.
- Wir mussten keine unnötigen Variablen definieren.

### Syntax des Pipe Operators

Der `%>%` Operator wird im Allgemeinen wie folgt verwendet. Nehmen wir an `f()`, `g()` und `h()` seien Funktionen, dann gilt:

```
x %>% f
 oder
x %>% f()

ist äquivalent zu

f(x)
```

Wenn `y` ein weiteres Argument von `f()` ist, dann gilt:

```
x %>% f(y)

ist äquivalent zu

f(x, y)
```

Wenn wir der Reihe nach `f()`, `g()` und `h()` anwenden, dann gilt:

```
x %>% f() %>% g() %>% h()

oder

x %>%
 f() %>%
 g() %>%
 h()

ist äquivalent zu

h(g(f(x)))
```

Wir müssen das Objekt `x` nicht als erstes Argument weitergeben; wir können es an einer beliebigen Stelle verwenden. Dafür brauchen wir den Argument-Platzhalter `.`:

```
x %>% f(y, .)

ist äquivalent zu

f(y, x)

x %>% f(y, z = .)

ist äquivalent zu

f(y, z = x)
```

In den meisten Fällen ist jedoch das Objekt, welches übergeben wird, gleichzeitig auch das erste Argument der nächsten Funktion (vor allem für die `tidyverse` Funktionen), so dass wir diesen Platzhalter selten brauchen.

Wenn wir mit den Funktionen der `tidyverse` und `dplyr` Packages arbeiten werden wir diesen Operator sehr häufig benutzen. Ein weiterer Grund, sich damit anzufreunden, ist, dass dieser immer häufiger Verwendung findet und sehr viele Beispiele im Internet (z.B. auf Stackoverflow) den `%>%` Operator benutzen.

## 4.3 Reshaping: `tidyverse`

Um einen Datensatz zu transformieren (reshaping) brauchen wir zwei Funktionen: `gather()` und `spread()`, beide finden wir im Package `tidyverse`.

```
library(tidyverse)
```

### 4.3.1 `gather()`

Wir benutzen `gather()`, wenn wir einen *wide* Datensatz zu einem *long* Datensatz konvertieren wollen; d.h. `gather()` wird dazu verwendet, mehrere Spalten, welche evtl. Stufen eines Faktors repräsentieren könnten, zu einer Spalte zusammenzufügen, welche den Faktor selber repräsentiert. Die Werte in den ursprünglichen Variablen werden in einer Werte-Variable zusammengefasst.

- `gather()` nimmt als Input mehrere Spalten und macht daraus ein **key-value** Paar. ■

Die Syntax von `gather()` sieht (vereinfacht) so aus:

```

gather(data, key, value, column1, column2, ...)

oder mit %>%

data %>%
 gather(key, value, column1, column2, ...)

```

Die Argumente haben die folgende Bedeutung:

```

data: Ein data frame.
key: Name der 'neuen' Gruppierungsvariable.
value: Name der neuen Messvariable.
column1, column2, ...: Alle Spalten, welche entweder
 gesammelt werden sollen, oder nicht. Nicht zu
 verwendende Spalten können mit '-' ausgeschlossen
 werden (-Spaltenname).

```

Am besten sehen wir uns ein kleines Beispiel an. Wir haben einen *wide* data frame mit den Faktorstufen "A" und "B".

```

bsp_wide <- data_frame(
 ID = factor(as.character(1:5)),
 A = floor(rnorm(5, 70, 1)),
 B = floor(rnorm(5, 50, 2)))

bsp_wide
#> # A tibble: 5 x 3
#> ID A B
#> <fct> <dbl> <dbl>
#> 1 1 71 50
#> 2 2 71 48
#> 3 3 70 52
#> 4 4 69 49
#> 5 5 69 50

```

Wir wollen nun die Spalten A und B zu einem Faktor "sammeln". Die ID Variable hingegen soll ignoriert werden, da sie nicht zu dem Faktor gehört. Der neue Faktor soll **Bedingung** heißen, und die neue Messvariable soll **Score** heißen.

```

Nicht vergessen, den Output einer neuen
Variablen zuzuweisen
library(tidyr)
bsp_long <- bsp_wide %>%
 gather(key = Bedingung, value = Score, A, B, -ID)

bsp_long
#> # A tibble: 10 x 3
#> ID Bedingung Score
#> <fct> <chr> <dbl>
#> 1 1 A 71
#> 2 2 A 71
#> 3 3 A 70
#> 4 4 A 69
#> 5 5 A 69
#> 6 1 B 50
#> # ... with 4 more rows

```

Bedingung sollte ein Faktor sein:

```
bsp_long$Bedingung <- factor(bsp_long$Bedingung)
```

### Beispiel

Als weiteres Beispiel sehen wir uns den “Therapy”-Datensatz an:

```
library(haven)
Therapy <- read_sav("data/Therapy.sav")
Therapy$Vpnr <- as.factor(Therapy$Vpnr)
Therapy$Gruppe <- as_factor(Therapy$Gruppe)
```

```
Therapy
#> # A tibble: 100 x 5
#> Vpnr Gruppe Pretest Posttest Difference_PrePost
#> <dbl> <fct> <dbl> <dbl> <dbl>
#> 1 1 Kontrollgruppe 4.29 3.21 1.08
#> 2 2 Kontrollgruppe 6.18 5.99 0.190
#> 3 3 Kontrollgruppe 3.93 4.17 -0.239
#> 4 4 Kontrollgruppe 5.06 4.76 0.295
#> 5 5 Kontrollgruppe 6.45 5.64 0.814
#> 6 6 Kontrollgruppe 4.49 4.67 -0.180
#> # ... with 94 more rows
```

In diesem Datensatz wollen wir die Variablen `Pretest` und `Posttest` zu einem Faktor `messzeitpunkt` zusammenfassen. `Difference_PrePost` soll weggelassen werden und wird daher in einem ersten Schritt aus dem Datensatz entfernt. `Vpnr` und `Gruppe` sollen für das Reshaping ignoriert werden, da sie nicht zum Faktor `messzeitpunkt` gehören.

```
Therapy <- Therapy[, c("Vpnr", "Gruppe", "Pretest", "Posttest")]

Therapy_long <- Therapy %>%
 gather(key = messzeitpunkt,
 value = rating,
 Pretest, Posttest, -Vpnr, -Gruppe)

messzeitpunkt muss ein Faktor sein
Therapy_long$messzeitpunkt <- factor(Therapy_long$messzeitpunkt,
 levels = c("Pretest", "Posttest"))

Therapy_long
#> # A tibble: 200 x 4
#> Vpnr Gruppe messzeitpunkt rating
#> <dbl> <fct> <fct> <dbl>
#> 1 1 Kontrollgruppe Pretest 4.29
#> 2 2 Kontrollgruppe Pretest 6.18
#> 3 3 Kontrollgruppe Pretest 3.93
#> 4 4 Kontrollgruppe Pretest 5.06
#> 5 5 Kontrollgruppe Pretest 6.45
#> 6 6 Kontrollgruppe Pretest 4.49
#> # ... with 194 more rows
```

### 4.3.2 `spread()`

`spread()` ist quasi das Gegenteil von `gather()`. Diese Funktion nimmt einen Faktor und eine Messvariable und “verteilt” die Werte der Messvariable über neue Spalten, welche die Stufen des Faktors repräsentieren.

Dies bedeutet, dass wir `spread()` verwenden, wenn wir aus einem *long* Datensatz einen *wide* Datensatz machen wollen.

Die `spread()` Syntax sieht (vereinfacht) so aus:

```
spread(data, key, value)

oder

data %>%
 spread(key, value)
```

An unserem einfachen Beispiel illustriert bedeutet dies, dass wir in `bsp_long` als `key` den Faktor `Bedingung` aufteilen möchten, und pro Stufe eine `value` Variable mit den Werten der `Score` Variablen machen möchten.

```
bsp_long
#> # A tibble: 10 x 3
#> ID Bedingung Score
#> <fct> <fct> <dbl>
#> 1 1 A 71
#> 2 2 A 71
#> 3 3 A 70
#> 4 4 A 69
#> 5 5 A 69
#> 6 1 B 50
#> # ... with 4 more rows
```

```
bsp_wide_2 <- bsp_long %>%
 spread(key = Bedingung, value = Score)
```

```
bsp_wide_2
#> # A tibble: 5 x 3
#> ID A B
#> <fct> <dbl> <dbl>
#> 1 1 71 50
#> 2 2 71 48
#> 3 3 70 52
#> 4 4 69 49
#> 5 5 69 50
```

Nun verifizieren wir noch, dass der neue und der alte *wide* Datensatz identisch sind:

```
all.equal(bsp_wide, bsp_wide_2)
#> [1] TRUE
```

`gather()` und `spread()` sind also komplementär.

### Beispiel

Nun konvertieren wir den “Therapy”-Datensatz von *long* zurück zu *wide*:

```
Therapy_long
#> # A tibble: 200 x 4
#> Vpnr Gruppe messzeitpunkt rating
#> <fct> <fct> <fct> <dbl>
#> 1 1 Kontrollgruppe Pretest 4.29
#> 2 2 Kontrollgruppe Pretest 6.18
#> 3 3 Kontrollgruppe Pretest 3.93
```

```
#> 4 4 Kontrollgruppe Pretest 5.06
#> 5 5 Kontrollgruppe Pretest 6.45
#> 6 6 Kontrollgruppe Pretest 4.49
#> # ... with 194 more rows

Therapy_wide <- Therapy_long %>%
 spread(key = messzeitpunkt, value = rating)

Therapy_wide
#> # A tibble: 100 x 4
#> Vpnr Gruppe Pretest Posttest
#> <fct> <fct> <dbl> <dbl>
#> 1 1 Kontrollgruppe 4.29 3.21
#> 2 2 Kontrollgruppe 6.18 5.99
#> 3 3 Kontrollgruppe 3.93 4.17
#> 4 4 Kontrollgruppe 5.06 4.76
#> 5 5 Kontrollgruppe 6.45 5.64
#> 6 6 Kontrollgruppe 4.49 4.67
#> # ... with 94 more rows

haben wir den wide Datensatz wieder hergestellt?
all.equal(Therapy, Therapy_wide)
#> Warning: Column `Pretest` has different attributes on LHS and RHS of join
#> Warning: Column `Posttest` has different attributes on LHS and RHS of join
#> [1] TRUE
```

### 4.3.3 Fehlende Werte ausschliessen: drop\_na()

Eine ganz wichtige Funktion im `tidyverse` Package ist `drop_na()`. Damit können wir alle Zeilen löschen, welche fehlende Werte haben.

Als Illustration dient dieses Beispiel:

```
df <- data_frame(var1 = c(1, 2, NA), var2 = c("a", NA, "b"))
df
#> # A tibble: 3 x 2
#> var1 var2
#> <dbl> <chr>
#> 1 1 a
#> 2 2 <NA>
#> 3 NA b

df %>% drop_na()
#> # A tibble: 1 x 2
#> var1 var2
#> <dbl> <chr>
#> 1 1 a
```

## 4.4 Daten manipulieren: dplyr

Nun sind wir in der Lage, Datensätze von *wide* zu *long* und umgekehrt zu transformieren, aber damit ist unsere Arbeit noch nicht getan. Die meisten Datensätze müssen bearbeitet werden, bevor sie analysiert werden können: Wir müssen z.B. Fälle und/oder Variablen auswählen, Werte recodieren, Variablen umbenennen, nach bestimmten Variablen sortieren, neue Variablen bilden, Datensätze nach Gruppierungsvariablen aufteilen oder Variablen zusammenfassen.

Das `dplyr` Package stellt Funktionen für alle diese Aufgaben zur Verfügung (und noch viele mehr, wir betrachten hier nur eine kleine Auswahl). `dplyr` besteht sozusagen aus Verben (Funktionen) für all diese Operationen, und diese Funktionen können - je nach Bedarf - auf sehr elegante Weise zusammengesetzt werden.

- Wir werden in dieser Vorlesung nur einen kleinen Teil der Funktionalität von `dplyr` kennenlernen. Wer mehr wissen will, kann dies in den Help Pages nachschauen: `help(package = "dplyr")`. ■

Für den Rest dieses Kapitels arbeiten wir mit *long* Datensätzen. Falls ein Datensatz *wide* ist, sollte er konvertiert werden.

Wir laden zuerst das `dplyr` Package:

```
library(dplyr)
```

Wir sehen uns nun der Reihe nach die verschiedenen Funktionen und deren Verwendung an. Wir verwenden immer den `%>%` Operator. Der Input Dataframe ist dabei immer als erstes Argument der Funktion zu verstehen. Für die Beispiele verwenden wir die Datensätze `bsp_long`, `Therapy_long` und `alk_aggr`. Letzteren finden Sie im Datenordner auf ILIAS.

```
library(readr)
alk_aggr <- read_csv("data/alkohol-aggression.csv")
#> Parsed with column specification:
#> cols(
#> aggressivitaet = col_double(),
#> alkoholbedingung = col_character()
#>)
alk_aggr$alkoholbedingung <- factor(alk_aggr$alkoholbedingung)
```

Bei allen unten stehenden Beispielen gilt: wenn wir das Resultat weiterverwenden möchten, müssen wir den Output einer neuen Variablen zuweisen.

#### 4.4.1 Variablen auswählen mit `select()`

Mit der Funktion `select()` wählen wir Variablen aus einem Datensatz aus.

Syntax:

```
df steht für data frame
select(df, variable1, variable2, -variable3)

oder

df %>% select(variable1, variable2, -variable3)
```

`select()` wählt hier aus dem Dataframe `df` die Variablen `variable1` und `variable2` aus, `variable3` wird weggelassen.

#### Beispiele

```
nur ID
bsp_long %>% select(ID)
#> # A tibble: 10 x 1
#> ID
#> <fct>
#> 1 1
#> 2 2
```

```
#> 3 3
#> 4 4
#> 5 5
#> 6 1
#> # ... with 4 more rows

Bedingung und Score
bsp_long %>% select(Bedingung, Score)
#> # A tibble: 10 x 2
#> Bedingung Score
#> <fct> <dbl>
#> 1 A 71
#> 2 A 71
#> 3 A 70
#> 4 A 69
#> 5 A 69
#> 6 B 50
#> # ... with 4 more rows

oder

bsp_long %>% select(-ID)
#> # A tibble: 10 x 2
#> Bedingung Score
#> <fct> <dbl>
#> 1 A 71
#> 2 A 71
#> 3 A 70
#> 4 A 69
#> 5 A 69
#> 6 B 50
#> # ... with 4 more rows

Therapy_long %>%
 select(Vpntr, Gruppe, rating)
#> # A tibble: 200 x 3
#> Vpntr Gruppe rating
#> <fct> <fct> <dbl>
#> 1 1 Kontrollgruppe 4.29
#> 2 2 Kontrollgruppe 6.18
#> 3 3 Kontrollgruppe 3.93
#> 4 4 Kontrollgruppe 5.06
#> 5 5 Kontrollgruppe 6.45
#> 6 6 Kontrollgruppe 4.49
#> # ... with 194 more rows

wählt alle Variablen von Gruppe bis rating aus (also auch die Variablen dazwischen)
Therapy_long %>%
 select(Gruppe:rating)
#> # A tibble: 200 x 3
#> Gruppe messzeitpunkt rating
#> <fct> <fct> <dbl>
#> 1 Kontrollgruppe Pretest 4.29
#> 2 Kontrollgruppe Pretest 6.18
```

```
#> 3 Kontrollgruppe Pretest 3.93
#> 4 Kontrollgruppe Pretest 5.06
#> 5 Kontrollgruppe Pretest 6.45
#> 6 Kontrollgruppe Pretest 4.49
#> # ... with 194 more rows
```

### Reihenfolge der Variablen

Wir können mit `select()` auch die Reihenfolge der Variablen verändern:

```
Therapy_long %>%
 select(rating, messzeitpunkt, Gruppe, Vpn)
#> # A tibble: 200 x 4
#> rating messzeitpunkt Gruppe Vpn
#> <dbl> <fct> <fct> <fct>
#> 1 4.29 Pretest Kontrollgruppe 1
#> 2 6.18 Pretest Kontrollgruppe 2
#> 3 3.93 Pretest Kontrollgruppe 3
#> 4 5.06 Pretest Kontrollgruppe 4
#> 5 6.45 Pretest Kontrollgruppe 5
#> 6 4.49 Pretest Kontrollgruppe 6
#> # ... with 194 more rows
```

### “Hilfe”-Funktionen für `select()`

Es gibt eine Reihe von sogenannten “helper functions” um Variablen auszuwählen:

```
einschliessen
Therapy_long %>% select(starts_with("Gr"))
#> # A tibble: 200 x 1
#> Gruppe
#> <fct>
#> 1 Kontrollgruppe
#> 2 Kontrollgruppe
#> 3 Kontrollgruppe
#> 4 Kontrollgruppe
#> 5 Kontrollgruppe
#> 6 Kontrollgruppe
#> # ... with 194 more rows
```

```
Therapy_long %>% select(ends_with("ng"))
#> # A tibble: 200 x 1
#> rating
#> <dbl>
#> 1 4.29
#> 2 6.18
#> 3 3.93
#> 4 5.06
#> 5 6.45
#> 6 4.49
#> # ... with 194 more rows
```

```
Therapy_long %>% select(contains("u"))
#> # A tibble: 200 x 2
#> Gruppe messzeitpunkt
```

```

#> <fct> <fct>
#> 1 Kontrollgruppe Pretest
#> 2 Kontrollgruppe Pretest
#> 3 Kontrollgruppe Pretest
#> 4 Kontrollgruppe Pretest
#> 5 Kontrollgruppe Pretest
#> 6 Kontrollgruppe Pretest
#> # ... with 194 more rows

vars <- c("Gruppe", "rating")
einschliessendes ODER mit one_of()
Therapy_long %>% select(one_of(vars))
#> # A tibble: 200 x 2
#> Gruppe rating
#> <fct> <dbl>
#> 1 Kontrollgruppe 4.29
#> 2 Kontrollgruppe 6.18
#> 3 Kontrollgruppe 3.93
#> 4 Kontrollgruppe 5.06
#> 5 Kontrollgruppe 6.45
#> 6 Kontrollgruppe 4.49
#> # ... with 194 more rows

ausschliessen
Therapy_long %>% select(-starts_with("Gr"))
#> # A tibble: 200 x 3
#> Vpnr messzeitpunkt rating
#> <fct> <fct> <dbl>
#> 1 1 Pretest 4.29
#> 2 2 Pretest 6.18
#> 3 3 Pretest 3.93
#> 4 4 Pretest 5.06
#> 5 5 Pretest 6.45
#> 6 6 Pretest 4.49
#> # ... with 194 more rows

Therapy_long %>% select(-ends_with("ng"))
#> # A tibble: 200 x 3
#> Vpnr Gruppe messzeitpunkt
#> <fct> <fct> <fct>
#> 1 1 Kontrollgruppe Pretest
#> 2 2 Kontrollgruppe Pretest
#> 3 3 Kontrollgruppe Pretest
#> 4 4 Kontrollgruppe Pretest
#> 5 5 Kontrollgruppe Pretest
#> 6 6 Kontrollgruppe Pretest
#> # ... with 194 more rows

Therapy_long %>% select(-contains("u"))
#> # A tibble: 200 x 2
#> Vpnr rating
#> <fct> <dbl>
#> 1 1 4.29

```

```
#> 2 2 6.18
#> 3 3 3.93
#> 4 4 5.06
#> 5 5 6.45
#> 6 6 4.49
#> # ... with 194 more rows

vars <- c("Gruppe", "rating")
Therapy_long %>% select(-one_of(vars))
#> # A tibble: 200 x 2
#> Vpnrs messzeitpunkt
#> <fct> <fct>
#> 1 1 Pretest
#> 2 2 Pretest
#> 3 3 Pretest
#> 4 4 Pretest
#> 5 5 Pretest
#> 6 6 Pretest
#> # ... with 194 more rows
```

Damit können Variablen anhand von Suchkriterien ausgewählt oder ausgeschlossen werden. Wir werden in späteren Kapiteln weitere Beispiele dafür sehen.

#### 4.4.2 Variablen umbenennen mit `rename()`

Variablen können mit `rename()` umbenannt werden. Die nicht umbenannten Variablen werden im Datensatz gelassen.

**Syntax:**

```
df %>% rename(neuer_name = alter_name)
```

#### Beispiel

```
"Vpnrs" umbenennen in "ID"
Therapy_long %>%
 rename(ID = Vpnrs)
#> # A tibble: 200 x 4
#> ID Gruppe messzeitpunkt rating
#> <fct> <fct> <fct> <dbl>
#> 1 1 Kontrollgruppe Pretest 4.29
#> 2 2 Kontrollgruppe Pretest 6.18
#> 3 3 Kontrollgruppe Pretest 3.93
#> 4 4 Kontrollgruppe Pretest 5.06
#> 5 5 Kontrollgruppe Pretest 6.45
#> 6 6 Kontrollgruppe Pretest 4.49
#> # ... with 194 more rows
```

#### 4.4.3 Beobachtungen (Fälle) auswählen mit `filter()`

Beobachtungen oder Fälle (Zeilen) werden mit `filter()` ausgewählt, d.h. wir können damit Fälle auswählen, welche gewisse Bedingungen erfüllen (oder nicht).

Wir können mehrere Bedingungen mit logischen Operatoren verknüpfen. **Syntax:**

```
df %>% filter(variable1 < WERT1 & variable2 == WERT2)
```

### Beispiele

```
nur Kontrollgruppe auswählen
Therapy_long %>%
 filter(Gruppe == "Kontrollgruppe")
#> # A tibble: 100 x 4
#> Vpnrs Gruppe messzeitpunkt rating
#> <fct> <fct> <fct> <dbl>
#> 1 1 Kontrollgruppe Pretest 4.29
#> 2 2 Kontrollgruppe Pretest 6.18
#> 3 3 Kontrollgruppe Pretest 3.93
#> 4 4 Kontrollgruppe Pretest 5.06
#> 5 5 Kontrollgruppe Pretest 6.45
#> 6 6 Kontrollgruppe Pretest 4.49
#> # ... with 94 more rows

nur Posttest
Therapy_long %>%
 filter(messzeitpunkt == "Posttest")
#> # A tibble: 100 x 4
#> Vpnrs Gruppe messzeitpunkt rating
#> <fct> <fct> <fct> <dbl>
#> 1 1 Kontrollgruppe Posttest 3.21
#> 2 2 Kontrollgruppe Posttest 5.99
#> 3 3 Kontrollgruppe Posttest 4.17
#> 4 4 Kontrollgruppe Posttest 4.76
#> 5 5 Kontrollgruppe Posttest 5.64
#> 6 6 Kontrollgruppe Posttest 4.67
#> # ... with 94 more rows

nur Pretest
Therapy_long %>%
 filter(messzeitpunkt != "Posttest")
#> # A tibble: 100 x 4
#> Vpnrs Gruppe messzeitpunkt rating
#> <fct> <fct> <fct> <dbl>
#> 1 1 Kontrollgruppe Pretest 4.29
#> 2 2 Kontrollgruppe Pretest 6.18
#> 3 3 Kontrollgruppe Pretest 3.93
#> 4 4 Kontrollgruppe Pretest 5.06
#> 5 5 Kontrollgruppe Pretest 6.45
#> 6 6 Kontrollgruppe Pretest 4.49
#> # ... with 94 more rows

nur ratings >= 5
Therapy_long %>%
 filter(rating >= 5)
#> # A tibble: 66 x 4
#> Vpnrs Gruppe messzeitpunkt rating
#> <fct> <fct> <fct> <dbl>
#> 1 2 Kontrollgruppe Pretest 6.18
#> 2 4 Kontrollgruppe Pretest 5.06
```

```

#> 3 5 Kontrollgruppe Pretest 6.45
#> 4 10 Kontrollgruppe Pretest 5.12
#> 5 11 Kontrollgruppe Pretest 6.04
#> 6 19 Kontrollgruppe Pretest 5.20
#> # ... with 60 more rows

nur ratings zwischen 3 und 5
Therapy_long %>%
 filter(rating >= 3 & rating <= 5)
#> # A tibble: 130 x 4
#> Vpnr Gruppe messzeitpunkt rating
#> <fct> <fct> <fct> <dbl>
#> 1 1 Kontrollgruppe Pretest 4.29
#> 2 3 Kontrollgruppe Pretest 3.93
#> 3 6 Kontrollgruppe Pretest 4.49
#> 4 7 Kontrollgruppe Pretest 4.60
#> 5 8 Kontrollgruppe Pretest 4.46
#> 6 9 Kontrollgruppe Pretest 4.76
#> # ... with 124 more rows

nur Vpn 3 und 5
Therapy_long %>%
 filter(Vpnr == 3 | Vpnr == 5)
#> # A tibble: 4 x 4
#> Vpnr Gruppe messzeitpunkt rating
#> <fct> <fct> <fct> <dbl>
#> 1 3 Kontrollgruppe Pretest 3.93
#> 2 5 Kontrollgruppe Pretest 6.45
#> 3 3 Kontrollgruppe Posttest 4.17
#> 4 5 Kontrollgruppe Posttest 5.64

Alternative dazu (mit dem %in% Operator)
Therapy_long %>%
 filter(Vpnr %in% c(3, 5))
#> # A tibble: 4 x 4
#> Vpnr Gruppe messzeitpunkt rating
#> <fct> <fct> <fct> <dbl>
#> 1 3 Kontrollgruppe Pretest 3.93
#> 2 5 Kontrollgruppe Pretest 6.45
#> 3 3 Kontrollgruppe Posttest 4.17
#> 4 5 Kontrollgruppe Posttest 5.64

```

#### 4.4.4 Beobachtungen (Fälle) sortieren mit `arrange()`

Mit der `arrange()` Funktion können wir Beobachtungen sortieren, entweder in aufsteigender oder in absteigender Reihenfolge.

```

aufsteigend
Therapy_long %>%
 arrange(Vpnr)
#> # A tibble: 200 x 4
#> Vpnr Gruppe messzeitpunkt rating
#> <fct> <fct> <fct> <dbl>
#> 1 1 Kontrollgruppe Pretest 4.29
#> 2 1 Kontrollgruppe Posttest 3.21

```

```
#> 3 2 Kontrollgruppe Pretest 6.18
#> 4 2 Kontrollgruppe Posttest 5.99
#> 5 3 Kontrollgruppe Pretest 3.93
#> 6 3 Kontrollgruppe Posttest 4.17
#> # ... with 194 more rows

absteigend
Therapy_long %>%
 arrange(desc(Vpntr))
#> # A tibble: 200 x 4
#> Vpntr Gruppe messzeitpunkt rating
#> <fct> <fct> <fct> <dbl>
#> 1 100 Therapiegruppe Pretest 4.77
#> 2 100 Therapiegruppe Posttest 4.50
#> 3 99 Therapiegruppe Pretest 4.66
#> 4 99 Therapiegruppe Posttest 3.80
#> 5 98 Therapiegruppe Pretest 4.36
#> 6 98 Therapiegruppe Posttest 3.91
#> # ... with 194 more rows
```

#### 4.4.5 Neue Variablen erstellen mit `mutate()`

Neue Variablen können mit `mutate()` aus schon bestehenden Variablen gebildet werden.

Syntax:

```
df %>% mutate(neue_variable_1 = FORMEL_1,
 neue_variable_2 = FORMEL_2)
```

#### Beispiele

```
Wir wollen die Ratings in %
Therapy_long %>%
 mutate(rating_p = round(rating/7 * 100, digits = 1))
#> # A tibble: 200 x 5
#> Vpntr Gruppe messzeitpunkt rating rating_p
#> <fct> <fct> <fct> <dbl> <dbl>
#> 1 1 Kontrollgruppe Pretest 4.29 61.2
#> 2 2 Kontrollgruppe Pretest 6.18 88.2
#> 3 3 Kontrollgruppe Pretest 3.93 56.2
#> 4 4 Kontrollgruppe Pretest 5.06 72.3
#> 5 5 Kontrollgruppe Pretest 6.45 92.2
#> 6 6 Kontrollgruppe Pretest 4.49 64.1
#> # ... with 194 more rows
```

- Wenn wir die alten Variablen ersetzen wollen, können wir anstelle von `mutate()` die Funktion `transmute()` verwenden. Diese Funktion gibt nur die neue Variable als Output, und nicht den ganzen Dataframe.

```
Therapy_long %>%
 transmute(rating_p = round(rating/7 * 100, digits = 1))
#> # A tibble: 200 x 1
#> rating_p
#> <dbl>
#> 1 61.2
```

```
#> 2 88.2
#> 3 56.2
#> 4 72.3
#> 5 92.2
#> 6 64.1
#> # ... with 194 more rows
```

#### 4.4.6 Daten gruppieren mit group\_by()

Nun ist es oft der Fall, dass wir bestimmte Operationen nicht auf den ganzen Datensatz anwenden wollen, sondern nur auf Subgruppen, welche durch Faktorstufen definiert sind. Dafür gibt es die Funktion `group_by()` - diese teilt den Datensatz anhand einer Gruppierungsvariable, wendet eine Funktion auf jeden Teil an, und setzt den Datensatz danach wieder zusammen (split-apply-combine). `group_by()` wird deshalb meistens in Kombination mit anderen Funktionen verwendet.

**Syntax:**

```
df <- group_by(gruppierung_1, gruppierung_2, gruppierung_3)
```

#### Beispiele

```
Wir bilden eine gruppenzentrierte Aggressionsvariable
library(readr)
alk_aggr <- read_csv("data/alkohol-aggression.csv")
#> Parsed with column specification:
#> cols(
#> aggressivitaet = col_double(),
#> alkoholbedingung = col_character()
#>)
alk_aggr$alkoholbedingung <- factor(alk_aggr$alkoholbedingung)

alk_aggr %>%
 group_by(alkoholbedingung) %>%
 mutate(group_mean = mean(aggressivitaet),
 aggr_c = scale(aggressivitaet, scale = FALSE))
#> # A tibble: 12 x 4
#> # Groups: alkoholbedingung [4]
#> aggressivitaet alkoholbedingung group_mean aggr_c
#> <dbl> <fct> <dbl> <dbl>
#> 1 64 kein_alkohol 62 2
#> 2 58 kein_alkohol 62 -4
#> 3 64 kein_alkohol 62 2
#> 4 74 placebo 75 -1
#> 5 79 placebo 75 4
#> 6 72 placebo 75 -3
#> # ... with 6 more rows
```

#### 4.4.7 Werte rekodieren mit recode()

Mit `mutate()` und `recode()` oder `recode_factor()` können wir Variablen rekodieren:

**Syntax:**

```

recode(variable,
 alter_wert_1 = "neuer_wert_1",
 alter_wert_2 = "neuer_wert_2")

Wir wollen die Kontrollgruppe zu "control" umbenennen,
und die Therapiegruppe zu "treatment"
Therapy_long %>%
 mutate(Gruppe = recode_factor(Gruppe,
 Kontrollgruppe = "control",
 Therapiegruppe = "treatment"))

#> # A tibble: 200 x 4
#> Vpnr Gruppe messzeitpunkt rating
#> <dbl> <fct> <fct> <dbl>
#> 1 1 control Pretest 4.29
#> 2 2 control Pretest 6.18
#> 3 3 control Pretest 3.93
#> 4 4 control Pretest 5.06
#> 5 5 control Pretest 6.45
#> 6 6 control Pretest 4.49
#> # ... with 194 more rows

```

#### 4.4.8 Variablen zusammenfassen mit `summarize()`

Mit `summarize()` oder `summarise()` können wir Variablen zusammenfassen und deskriptive Kennzahlen berechnen. Im Gegensatz zu `mutate()` gibt `summarize()` nicht einen Wert für jede Beobachtung als Output, sondern einen Wert für jede Gruppe.

Syntax:

```
df %>% summarize(kennzahl = FUNKTION(variable))
```

#### Beispiele

```

Gruppenmittelwerte pro Messzeitpunkt
Therapy_long %>%
 group_by(Gruppe, messzeitpunkt) %>%
 summarize(mean_rating = mean(rating))

#> # A tibble: 4 x 3
#> # Groups: Gruppe [2]
#> Gruppe messzeitpunkt mean_rating
#> <fct> <fct> <dbl>
#> 1 Kontrollgruppe Pretest 5.06
#> 2 Kontrollgruppe Posttest 4.65
#> 3 Therapiegruppe Pretest 4.82
#> 4 Therapiegruppe Posttest 4.23

```

#### 4.4.9 Zusammenfassung

Wir können nun damit beginnen, alle Funktionen zusammenzusetzen. Zum Beispiel haben wir weiter oben im `Therapy` Datensatz die Differenzvariable `Pretest - Posttest` gelöscht. Diese können wir nun aber problemlos aus dem `long` Datensatz wieder berechnen:

```

Therapy_long %>%
 spread(key = messzeitpunkt, value = rating) %>%
 mutate(Delta = Pretest - Posttest)

```

```
#> # A tibble: 100 x 5
#> Vpn Gruppe Pretest Posttest Delta
#> <fct> <fct> <dbl> <dbl> <dbl>
#> 1 1 Kontrollgruppe 4.29 3.21 1.08
#> 2 2 Kontrollgruppe 6.18 5.99 0.190
#> 3 3 Kontrollgruppe 3.93 4.17 -0.239
#> 4 4 Kontrollgruppe 5.06 4.76 0.295
#> 5 5 Kontrollgruppe 6.45 5.64 0.814
#> 6 6 Kontrollgruppe 4.49 4.67 -0.180
#> # ... with 94 more rows
```

## 4.5 Übungsaufgaben

Laden Sie die Datensätze “RTdata.csv”, “Beispieldatensatz.sav” und “honeymoon.csv” von ILIAS herunter.

```
library(tidyr)
library(dplyr)
```

### Reaktionszeiten

In diesem (simulierten) Datensatz mussten drei Vpn in drei Bedingungen eine Reaktionszeitaufgabe erledigen. In der einen Bedingung sollten die Vpn langsam sein, in der andere Bedingung sollten sie schnell sein. Die dritte Bedingung ist eine Kontrollbedingung.

```
library(readr)
rt_wide <- read_csv("data/RTdata.csv")
#> Parsed with column specification:
#> cols(
#> ID = col_double(),
#> control = col_double(),
#> slow = col_double(),
#> fast = col_double()
#>)
rt_wide
#> # A tibble: 300 x 4
#> ID control slow fast
#> <dbl> <dbl> <dbl> <dbl>
#> 1 1 1.55 3.62 3.31
#> 2 1 0.782 1.50 1.57
#> 3 1 3.10 2.30 1.13
#> 4 1 2.97 1.83 1.97
#> 5 1 1.38 3.53 3.26
#> 6 1 0.329 3.44 5.34
#> # ... with 294 more rows
```

- Dieser Datensatz muss zuerst ins *long* Format konvertiert werden. Konvertieren Sie Gruppierungsvariablen falls nötig zu Faktoren.

```
rt_long <- rt_wide %>%
 gather(key = condition, value = rt,
 control, slow, fast, -ID) %>%
 mutate(ID = factor(ID),
 condition = factor(condition))
```

```
rt_long
#> # A tibble: 900 x 3
#> ID condition rt
#> <fct> <fct> <dbl>
#> 1 1 control 1.55
#> 2 1 control 0.782
#> 3 1 control 3.10
#> 4 1 control 2.97
#> 5 1 control 1.38
#> 6 1 control 0.329
#> # ... with 894 more rows
```

- b) Berechnen Sie die mittlere Reaktionszeit und Standardabweichung für jede Person in jeder Bedingung.

```
summary_rt <- rt_long %>%
 group_by(ID, condition) %>%
 summarise(mean_rt = mean(rt),
 sd_rt = sd(rt))

summary_rt
#> # A tibble: 9 x 4
#> # Groups: ID [3]
#> ID condition mean_rt sd_rt
#> <fct> <fct> <dbl> <dbl>
#> 1 1 control 1.68 1.02
#> 2 1 fast 2.15 1.58
#> 3 1 slow 3.20 1.93
#> 4 2 control 2.12 1.31
#> 5 2 fast 1.69 1.21
#> 6 2 slow 3.23 1.69
#> # ... with 3 more rows
```

- c) Logarithmieren Sie die RT und fügen Sie dem Datensatz diese als neue Variable hinzu.

```
rt_long <- rt_long %>%
 mutate(log_rt = log(rt))

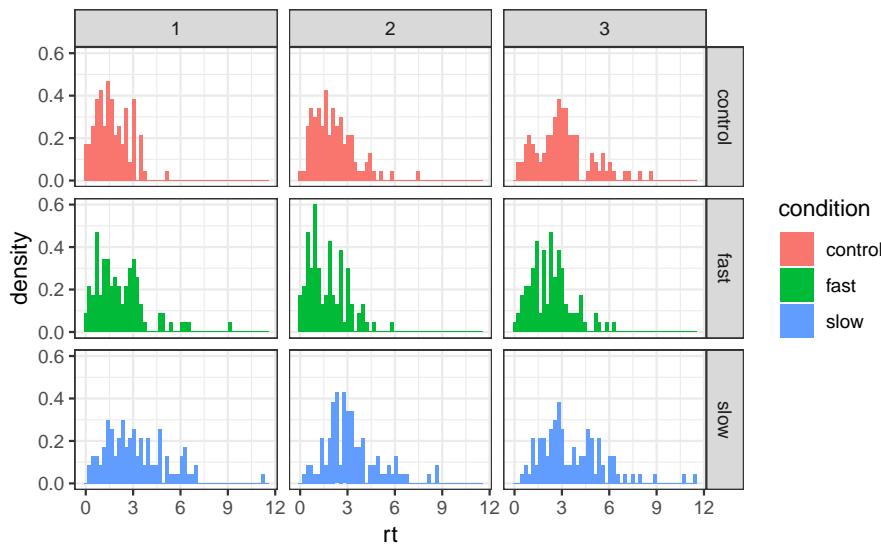
rt_long
#> # A tibble: 900 x 4
#> ID condition rt log_rt
#> <fct> <fct> <dbl> <dbl>
#> 1 1 control 1.55 0.439
#> 2 1 control 0.782 -0.246
#> 3 1 control 3.10 1.13
#> 4 1 control 2.97 1.09
#> 5 1 control 1.38 0.319
#> 6 1 control 0.329 -1.11
#> # ... with 894 more rows
```

- Wenn wir einen Datensatz im *long* Format haben, ist es relativ einfach mit `ggplot2` Grafiken zu erstellen. Dieses Package schauen wir uns im nächsten Kapitel an. ■

```
library(ggplot2)

rt_long %>%
 ggplot(aes(x = rt, y = ..density.., fill = condition)) +
 geom_histogram(bins = 50) +
 facet_grid(condition ~ ID) +
```

```
theme_bw()
```



## Arbeitszufriedenheit: Honeymoon oder Hangover?

Wir untersuchen nun einen weiteren Datensatz, den Sie bereits aus den Statistikübungen kennen. Es wurde untersucht, ob die Arbeitszufriedenheit von Job-Einsteigern einen U-förmigen Verlauf aufweist. Am Anfang ist man zufrieden (honeymoon), nach einiger Zeit ist man wegen der Schwierigkeiten in der Eingewöhnungsphase frustriert und die Zufriedenheit bricht ein (hangover). Letztendlich sollte sich die Zufriedenheit wieder normalisieren. Gemessen wurde die Arbeitszufriedenheit zu drei Messzeitpunkten (am Anfang, nach drei Monaten und nach sechs Monaten). Es wurden Job-Einsteiger in zwei Firmen befragt.

Laden Sie den Datensatz `honeymoon.csv` von ILIAS herunter und importieren Sie ihn. Beachten Sie dabei die Trennzeichen ("delimiter").

```
library(readr)
honeymoon <- read_delim("data/honeymoon.csv",
 delim = ";")
#> Parsed with column specification:
#> cols(
#> Firma = col_double(),
#> Anfang = col_double(),
#> Drei_Monate = col_double(),
#> Sechs_Monate = col_double()
#>)
```

```
honeymoon
#> # A tibble: 10 x 4
#> Firma Anfang Drei_Monate Sechs_Monate
#> <dbl> <dbl> <dbl> <dbl>
#> 1 1 9 4 5
#> 2 1 9 4 8
#> 3 1 9 7 14
#> 4 1 10 9 5
#> 5 1 8 1 3
```

```
#> 6 2 10 10 10
#> # ... with 4 more rows
```

Die Variable `Firma` ist natürlich ein Faktor, und sollte konvertiert werden.

```
honeymoon <- honeymoon %>% mutate(Firma = as.factor(Firma))
```

Wir wollen nun einen *long* Datensatz erstellen, mit einem messwiederholten Faktor `Messzeitpunkt` und einer Messvariablen `Arbeitszufriedenheit`.

Im Kapitel über Mittelwertsvergleiche werden wir mit diesem Datensatz eine ANOVA mit Messwiederholung durchführen. Davor wollen wir schon einmal den Personenmittelwert über alle Messzeitpunkte selber berechnen, sowie den Bedingungsmittelwert über alle Personen. Beide Mittelwerte sollen für die beiden Firmen separat berechnet werden.

Zuerst kreieren wir eine Personenvariable. Es hat fünf Personen pro Firma, also weisen wir den Personen einfach die Zahlen 1 bis 10 zu.

```
nrow(.) bedeutet hier nrow(honeymoon) [pipe operator]
honeymoon <- honeymoon %>%
 mutate(ID = 1:nrow(.))
```

```
honeymoon_long <- honeymoon %>%
 gather(key = Messzeitpunkt, value = Arbeitszufriedenheit,
 -Firma, -ID) %>%
 mutate(Messzeitpunkt = factor(Messzeitpunkt))
```

Die Reihenfolge der Faktorstufen stimmt glücklicherweise schon, sonst könnten wir diese mit `relevel()` ändern.

```
levels(honeymoon_long$Messzeitpunkt)
#> [1] "Anfang" "Drei_Monate" "Sechs_Monate"
```

Vielleicht wollen wir die Faktorstufen umbenennen: von "Anfang", "Drei\_Monate", "Sechs\_Monate" zu "0", "3", "6".

```
honeymoon_long <- honeymoon_long %>%
 mutate(Messzeitpunkt = recode_factor(Messzeitpunkt,
 Anfang = "0",
 Drei_Monate = "3",
 Sechs_Monate = "6"))
```

```
honeymoon_long
#> # A tibble: 30 x 4
#> Firma ID Messzeitpunkt Arbeitszufriedenheit
#> <fct> <int> <fct> <dbl>
#> 1 1 1 0 9
#> 2 1 2 0 9
#> 3 1 3 0 9
#> 4 1 4 0 10
#> 5 1 5 0 8
#> 6 2 6 0 10
#> # ... with 24 more rows
```

Wir sortieren nun nach Firma und ID.

```
honeymoon_long <- honeymoon_long %>%
 # zuerst nach Firma, dann nach ID
 arrange(Firma, ID)
```

```
honeymoon_long
#> # A tibble: 30 x 4
#> Firma ID Messzeitpunkt Arbeitszufriedenheit
#> <fct> <int> <fct> <dbl>
#> 1 1 1 0 9
#> 2 1 1 3 4
#> 3 1 1 6 5
#> 4 1 2 0 9
#> 5 1 2 3 4
#> 6 1 2 6 8
#> # ... with 24 more rows
```

Nun berechnen wir den Personenmittelwert. Wir gruppieren zuerst nach `Firma` und `ID`, und berechnen dann den Mittelwert der `Arbeitszufriedenheit`.

```
honeymoon_long <- honeymoon_long %>%
 group_by(Firma, ID) %>%
 mutate(Personenmittelwert = mean(Arbeitszufriedenheit))
```

```
honeymoon_long
#> # A tibble: 30 x 5
#> # Groups: Firma, ID [10]
#> Firma ID Messzeitpunkt Arbeitszufriedenheit Personenmittelwert
#> <fct> <int> <fct> <dbl> <dbl>
#> 1 1 1 0 9 6
#> 2 1 1 3 4 6
#> 3 1 1 6 5 6
#> 4 1 2 0 9 7
#> 5 1 2 3 4 7
#> 6 1 2 6 8 7
#> # ... with 24 more rows
```

Wir wollen nun auch noch den Bedingungsmittelwert berechnen. Jetzt gruppieren wir nach `Firma` und `Messzeitpunkt`.

```
honeymoon_long_bedingung <-
 honeymoon_long %>%
 group_by(Firma, Messzeitpunkt) %>%
 summarise(Bedingungsmittelwert = mean(Arbeitszufriedenheit))
```

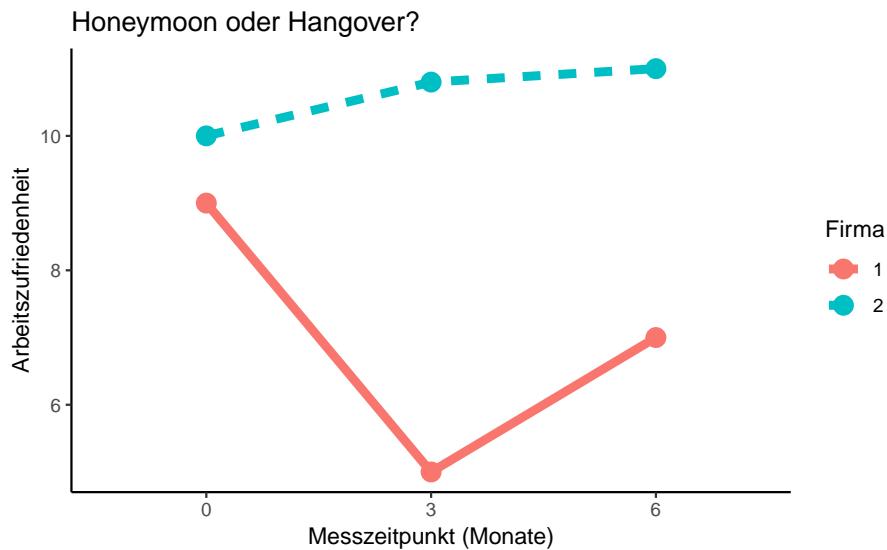
```
honeymoon_long_bedingung
#> # A tibble: 6 x 3
#> # Groups: Firma [2]
#> Firma Messzeitpunkt Bedingungsmittelwert
#> <fct> <fct> <dbl>
#> 1 1 0 9
#> 2 1 3 5
#> 3 1 6 7
#> 4 2 0 10
#> 5 2 3 10.8
#> 6 2 6 11
```

```
honeymoon_long_bedingung %>%
 ggplot(aes(x = Messzeitpunkt,
 y = Bedingungsmittelwert,
 color = Firma,
```

```

group = Firma,
linetype = Firma)) +
geom_point(size = 4) +
geom_line(size = 2) +
labs(x = "Messzeitpunkt (Monate)",
y = "Arbeitszufriedenheit",
title = "Honeymoon oder Hangover?") +
theme_classic()

```



- **Wirklich nur für Vertiefer:** Wir haben nun schon mehrmals die Funktion `str_replace()` benutzt, um Faktorstufen umzubennen. Diese Funktion, welche in dem Package `stringr` zu finden ist, dient dazu, text ("strings") zu ersetzen. Die Funktion hat die Argumente `str_replace(string, pattern, replacement)`. `string` ist das Objekt, in welchem Text ersetzt werden soll, `pattern` ist ein Suchbegriff (*regular expression*) und `replacement` ist der Text, welcher den Suchbegriff ersetzt. In unserem Beispiel wollten wir in den Werten einer Variablen das Präfix `leben_` löschen, oder durch "" (keinen Character) ersetzen. Nehmen Sie an, die Variable hat als Wert den String "`leben_freunde`"; dann suchen wir in diesem String den Suchbegriff `".*_"` und ersetzen ihn durch `". ".*_ "` bedeutet: Wir suchen irgendeinen Character (.), n-Mal wiederholt (\*), gefolgt von einem \_.

```

str_replace("leben_freunde", ".*_", "")
#> [1] "freunde"

```

- 
-

# Chapter 5

## Grafiken mit ggplot2

Grafiken sind für die Datenanalyse sehr wichtig. Einerseits können wir sie für explorative Datenanalyse einsetzen, um eventuell verborgene Zusammenhänge zu entdecken oder uns einfach einen Überblick zu verschaffen. Andererseits brauchen wir Grafiken, um Resultate darzustellen und anderen zu kommunizieren.

Wir haben schon mehrmals in diesem Skript Grafiken mit dem Package `ggplot2` erstellt, ohne uns den Code genauer anzuschauen. In diesem Kapitel werden wir nun die Syntax von `ggplot2` kennenlernen.

- Es gibt in R verschiedene Möglichkeiten, Grafiken zu erstellen. Mit dem ursprünglichen Grafiksystem (R Base Graphics) kann man sehr schnell einfache Grafiken erstellen. Es ist auch sehr mächtig und flexibel, aber das Problem ist, dass die Syntax etwas archaisch erscheint, und es für Anfänger schwierig ist, Grafiken selber anzupassen. ■

Im Gegensatz dazu basiert `ggplot2` auf einer intuitiven Syntax, der sogenannten Grammar of graphics. Sobald man sich daran gewöhnt hat, kann man mit einer eleganten und konsistenten “Grammatik” sehr komplexe Grafiken erstellen. `ggplot2` ist darauf ausgelegt, mit `tidy` Data zu arbeiten, d.h. wir brauchen Datensätze im *long* Format. Grafiken werden nun immer nach demselben Prinzip erstellt:

**Schritt 1:** Wir beginnen mit einem Datensatz und erstellen ein Plot-Objekt mit der Funktion `ggplot()`.

**Schritt 2:** Wir definieren sogenannte “aesthetic mappings”, d.h. wir bestimmen welche Variablen auf den X-, bzw. Y-Achsen dargestellt werden sollen, und welche Variablen benutzt werden, um die Daten zu gruppieren. Die Funktion, welche wir dafür benutzen heisst `aes()`.

**Schritt 3:** Wir fügen dem Plot eine oder mehrere “Layers” oder “Schichten” hinzu. Diese Layers definieren, wie etwas dargestellt werden soll, z.B. als Linie oder als Histogramm. Die Funktionen beginnen mit dem Präfix `geom_`, z.B. `geom_line()`.

Um `ggplot2` zu benutzen brauchen wir nun noch einen zusätzlichen Operator: `+`. Diesen kennen Sie bereits als mathematischen Operator, aber in diesem Zusammenhang bedeutet die Verwendung von `+`, dass wir einzelne Elemente eines Plot-Objektes zusammenfügen.

Nach dieser etwas abstrakten Einführung illustrieren wir diese Schritte an einem praktischen Beispiel.

Am Ende des letzten Kapitels haben wir den Zusammenhang zwischen psychischem Stress und Geschlecht untersucht. Wir laden nun nochmals den Datensatz:

```
library(dplyr)
library(tidyr)
library(stringr)
library(haven)
```

```
westost <- read_sav("data/Beispieldatensatz.sav")
```

und erstellen einen Datensatz, der nur die Variablen ID, geschlecht und stress\_psych enthält.

```
stress_psych <- westost %>%
 select(ID, geschlecht, stress_psych) %>%
 mutate(geschlecht = factor(geschlecht),
 ID = factor(ID)) %>%
 drop_na()
stress_psych
#> # A tibble: 284 x 3
#> ID geschlecht stress_psych
#> <dbl> <fct> <dbl>
#> 1 2 0 3.5
#> 2 14 0 1
#> 3 15 0 2.5
#> 4 17 0 1.67
#> 5 18 0 2.5
#> 6 19 0 6.83
#> # ... with 278 more rows
```

In diesem Datensatz haben wir eine numerische Variable, stress\_psych und eine Gruppierungsvariable, geschlecht. Unsere Frage lautete, ob die Variable stress\_psych mit der Gruppierungsvariablen geschlecht zusammenhängt. Diesen Zusammenhang könnten wir auf verschiedene Arten grafisch darstellen: mit Punkten, einem Boxplot oder einem Violin-Plot. Diese drei Methoden sind in der Sprache von ggplot2 verschiedene geoms und können so benutzt werden: geom\_point(), geom\_boxplot() oder geom\_violin(). Zusätzlich gibt es noch eine Funktion geom\_jitter(), welche die Punkte in einem Punktdiagramm nicht aufeinander zeichnet, sondern mit einem räumlichen “jittering” (Flackern).

Das ggplot2 Package können wir entweder individuell oder als Teil des tidyverse laden:

```
library(ggplot2)

oder

library(tidyverse)
```

## 5.1 Schritt 1: Plot-Objekt erstellen

Wir beginnen mit einem Datensatz und erstellen ein Plot-Objekt mit der Funktion ggplot(). Diese Funktion hat als erstes Argument einen Dataframe. Dies bedeutet, dass wir den pipe Operator verwenden können:

```
>
> ggplot()
```

<code>◆ data =</code>	<b>data</b>
<code>◆ mapping =</code>	Default dataset to use for plot. If not already a data.frame, will be converted to one by <code>fortify</code> . If not specified, must be supplied in each layer added to the plot.
<code>◆ ... =</code>	
<code>◆ environment =</code>	Press F1 for additional help

Wir haben also zwei Möglichkeiten. Wir bevorzugen hier die `pipe` Notation, aber es ist selbstverständlich auch möglich, den Dataframe innerhalb der Funktion als Argument anzugeben. Gleichzeitig weisen wir das Objekt einer Variablen zu, und nennen diese `p`.

```
1. Variante
p <- ggplot(data = stress_psych)

2. Variante
p <- stress_psych %>%
 ggplot()
```

## 5.2 Schritt 2: Aesthetic mappings

Nun definieren wir mit dem zweiten Argument `mapping` die “aesthetic mappings”. Diese bestimmen, wie die Variablen benutzt werden, um die Daten darzustellen, und werden mit der Funktion `aes()` definiert. Wir wollen die Gruppierungsvariable `geschlecht` auf der X-Achse darstellen und `stress_psych` soll auf der Y-Achse angezeigt werden. Zusätzlich kann `aes()` weitere Argumente haben: `fill`, `color`, `shape`, `linetype`, `group`. Diese werden dazu benutzt, um den Stufen der Gruppierungsvariablen unterschiedliche Farben, Formen, Linien, etc. zuzuweisen.

In diesem Beispiel haben wir die Gruppierungsvariable `geschlecht` und wir wollen, dass die beiden Stufen von `geschlecht` verschiedene Farben haben und mit verschiedenen Farben “ausgefüllt” werden.

- `color` ist ein Attribut von Linien oder Punkten, `fill` ist ein Attribut von Flächen. ■

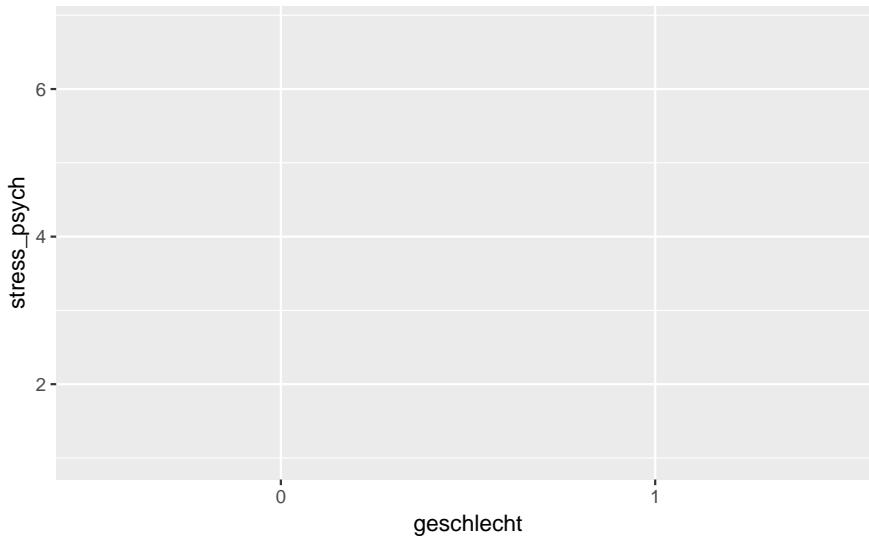
Wenn wir die “aesthetic mappings” innerhalb der Funktion `ggplot()` definieren, gelten sie für alle “Layers”, d.h. für alle Elemente des Plots. Wir könnten diese mappings auch für jede “Layer” separat definieren.

```
p <- stress_psych %>%
 ggplot(mapping = aes(x = geschlecht,
 y = stress_psych,
 color = geschlecht,
 fill = geschlecht))
```

`p` ist nun ein “leeres” Plot-Objekt. Wir können es uns anschauen, aber es wird noch nichts angezeigt, da es noch keine “Layers” enthält. Ein `ggplot2` Objekt wird angezeigt, indem wir das Objekt auf der Konsole ausgeben lassen, entweder mit oder ohne `print()`.

```
p

oder print(p)
```



Wir sehen, dass `ggplot2` für uns schon die Achsen anhand der Variablennamen beschriftet hat.

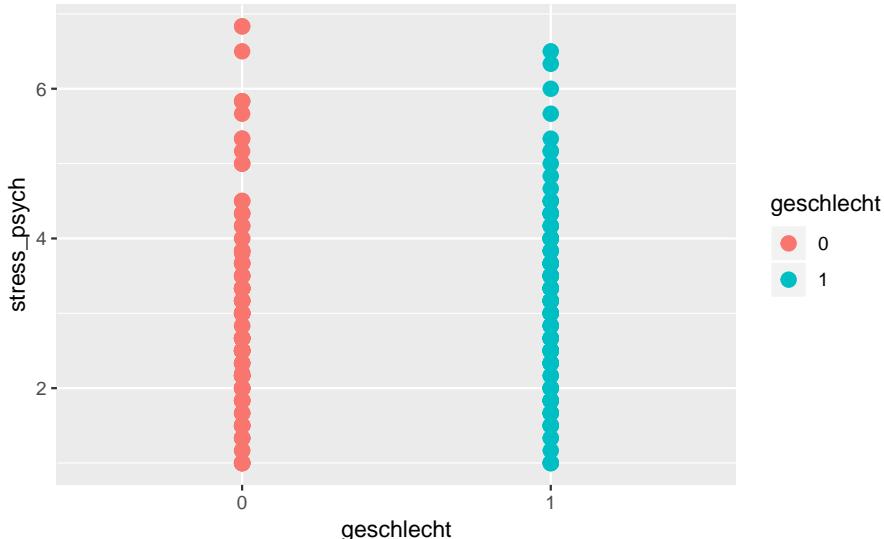
### 5.3 Schritt 3: geoms hinzufügen

Dem Plot-Objekt `p` können wir nun mit `geom_`-Funktionen "Layers" hinzufügen. Die Syntax funktioniert so: Wir "addieren" zu dem Plot-Objekt `p` ein `geom`: `p + geom_`.

#### 5.3.1 Punktdiagramm

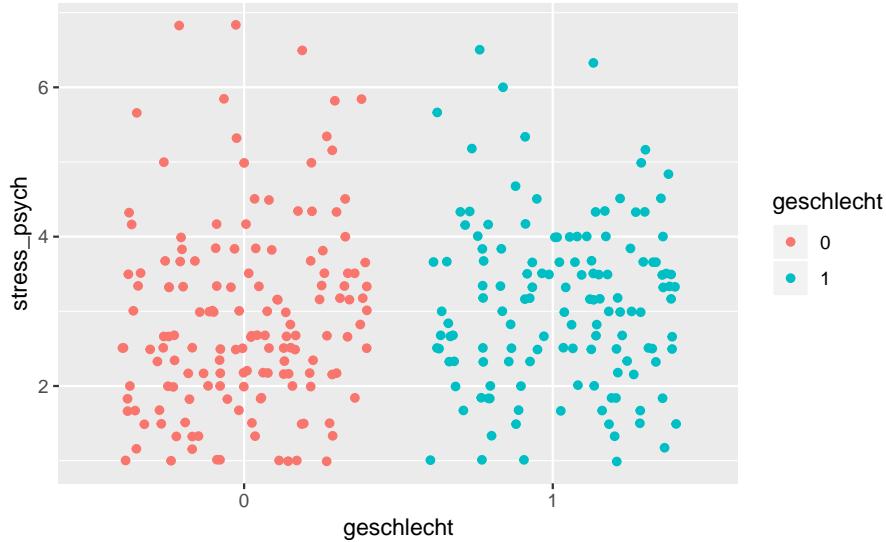
Wir versuchen zuerst, die Beobachtungen als Punkte darzustellen:

```
die Funktion geom_point() hat ein size Argument
p + geom_point(size = 3)
```



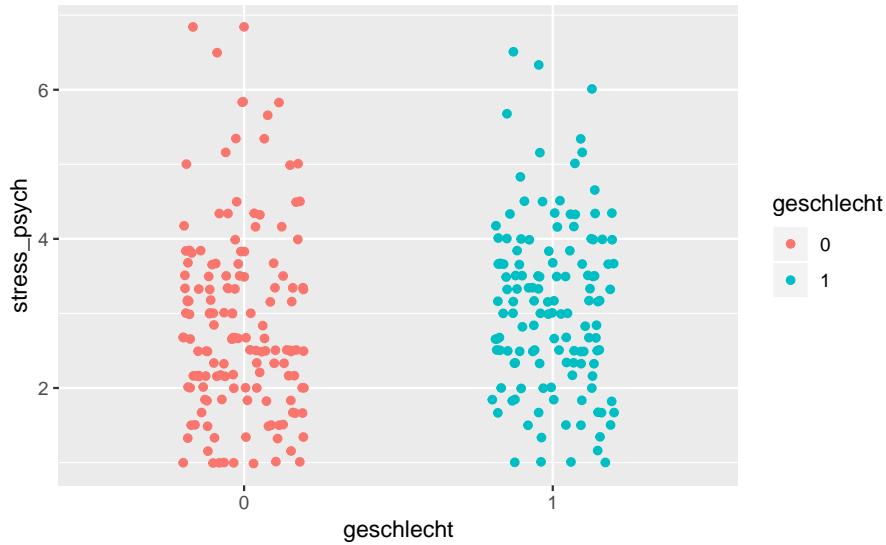
Die Punkte werden nun in verschiedenen Farben dargestellt, aber innerhalb eines Geschlechts werden Punkte eventuell übereinander geplottet, wenn sie denselben Wert haben (overplotting). Für diesen Fall gibt es die Funktion `geom_jitter()`, welche Punkte mit einem "jittering" nebeneinander zeichnet:

```
p + geom_jitter()
```



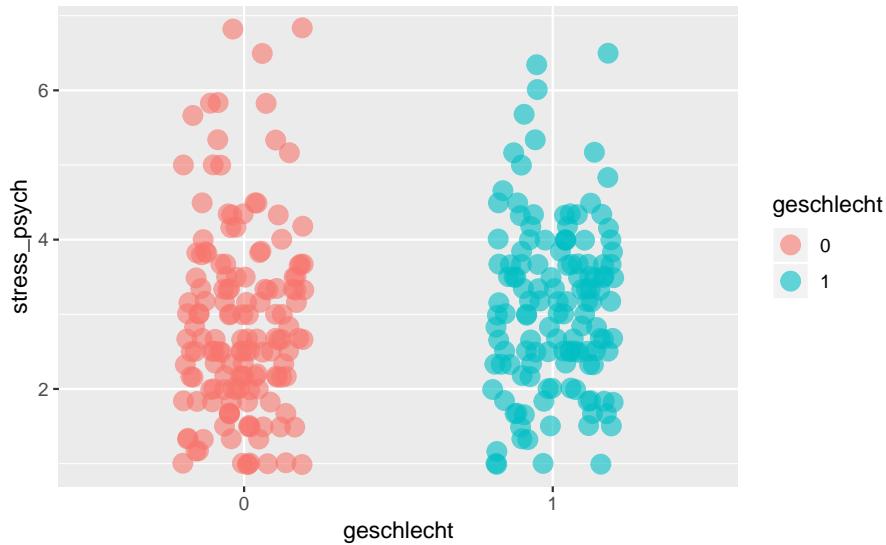
`geom_jitter()` hat ein Argument `width`, mit dem wir bestimmen können, wie breit die Streuung der Punkte ist.

```
p + geom_jitter(width = 0.2)
```



`geom_jitter()` hat weitere Argumente: `size` bestimmt den Durchmesser der Punkte, und `alpha` bestimmt die Transparenz.

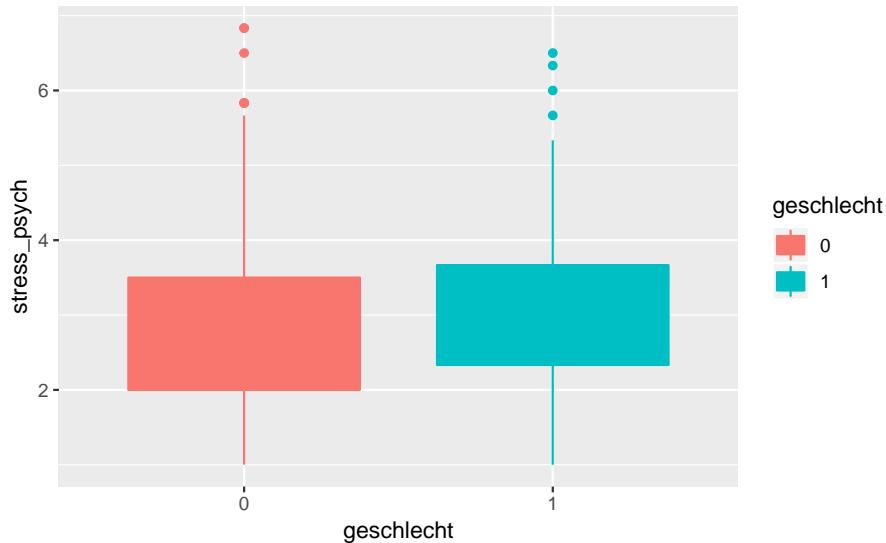
```
p + geom_jitter(width = 0.2, size = 4, alpha = 0.6)
```



### 5.3.2 Verteilung grafisch darstellen

Eine weitere Möglichkeit wäre, die zentrale Tendenz und Streuung der Daten mit einem Boxplot- oder Violin-Diagramm darzustellen.

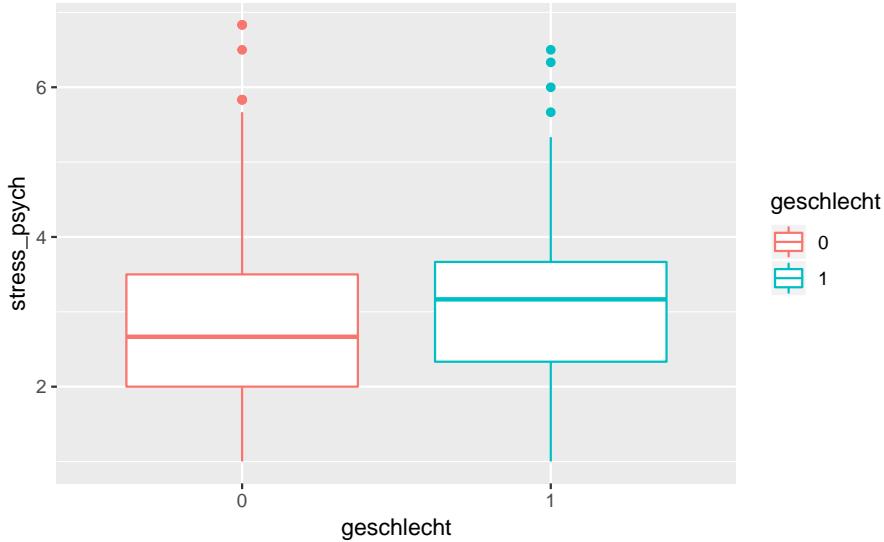
```
p + geom_boxplot()
```



In einem Boxplot wird der Median dargestellt, das Rechteck repräsentiert die mittleren 50%, und die “whiskers” zeigen  $1.5 * \text{Interquartilsbereich}$ . Ausreißer werden mit Punkten dargestellt. Um den Median zu sehen, ist es besser, wenn wir das `fill` Attribut weglassen:

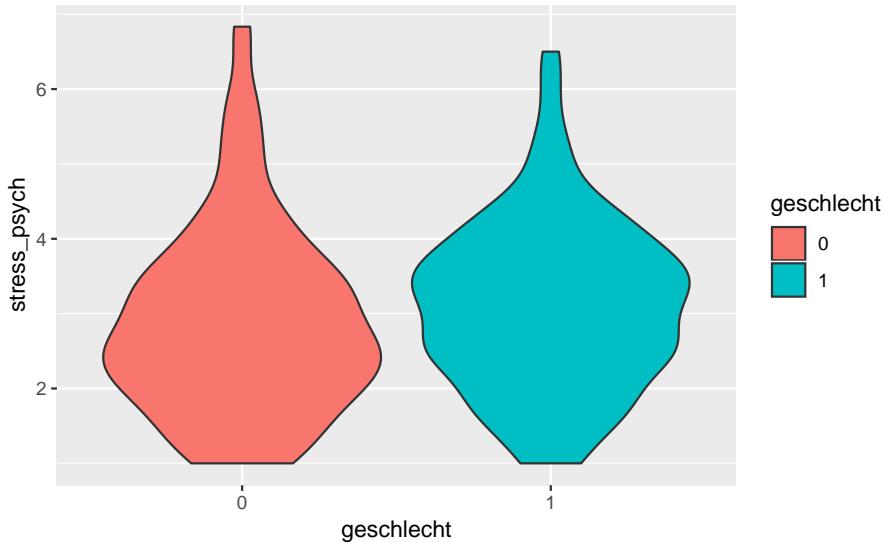
```
p <- stress_psych %>%
 ggplot(mapping = aes(x = geschlecht,
 y = stress_psych,
 color = geschlecht))
```

```
p + geom_boxplot()
```



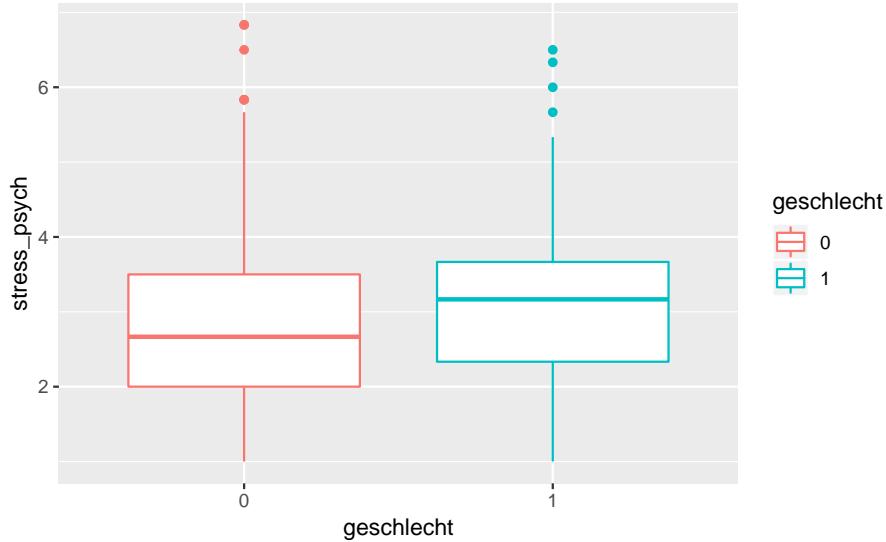
Ein Violin-Plot ist ähnlich wie ein Boxplot, zeigt aber nicht die Quantile, sondern ein “kernel density estimate”. Ein Violin-Plot sieht am besten aus, wenn wir das `fill` Attribut verwenden.

```
p <- stress_psych %>%
 ggplot(mapping = aes(x = geschlecht,
 y = stress_psych,
 fill = geschlecht))
p + geom_violin()
```

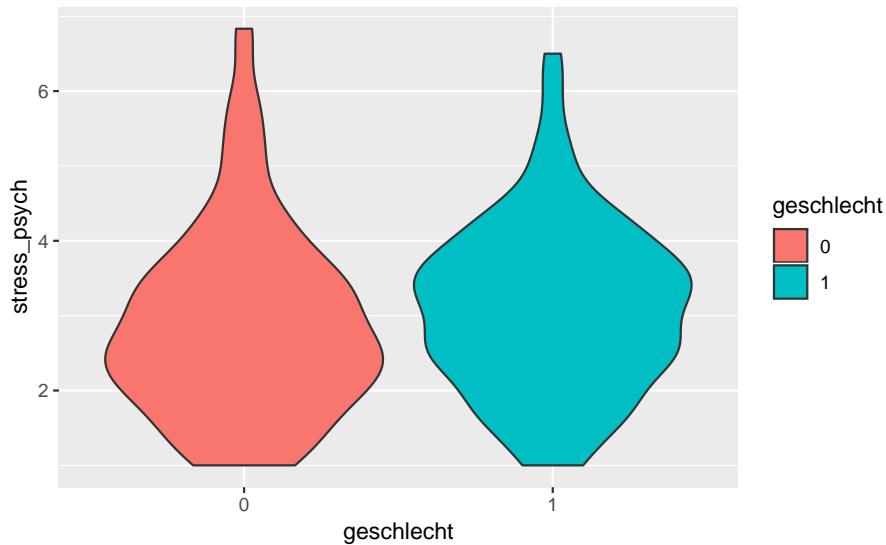


Wenn wir feststellen, dass ein Mapping nicht für alle “Layers” gelten soll, dann können wir es für jede “Layer” individuell definieren, anstatt in der `ggplot()` Funktion:

```
p <- stress_psych %>%
 ggplot(mapping = aes(x = geschlecht,
 y = stress_psych))
p + geom_boxplot(mapping = aes(color = geschlecht))
oder einfacher
p + geom_boxplot(aes(color = geschlecht))
```



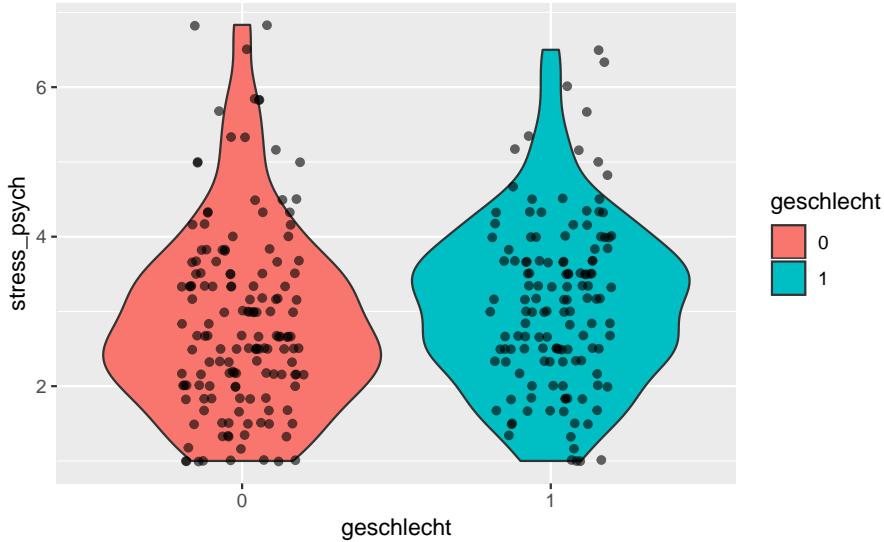
```
p + geom_violin(aes(fill = geschlecht))
```



### 5.3.3 Mehrere Layers kombinieren

Wir können auch mehrere “Layers” verwenden. Wir müssen lediglich mehrere `geom_` Funktionen mit einem `+` zusammenfügen:

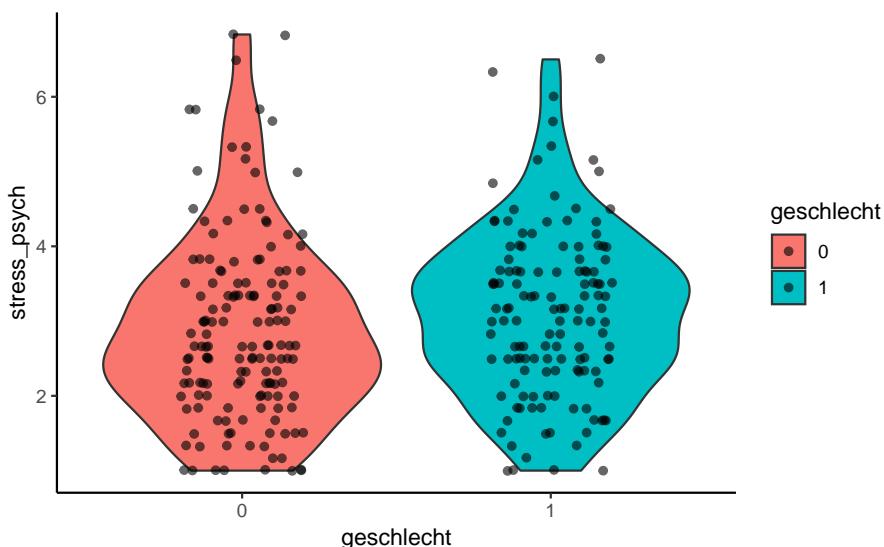
```
p +
 geom_violin(aes(fill = geschlecht)) +
 geom_jitter(width = 0.2, alpha = 0.6)
```



- Schauen Sie sich die Grafikbeispiele in den vorangegangenen Kapiteln an. Verstehen Sie nun den Code?
- 

In den bisherigen Beispielen haben wir kein Plot-Objekt erstellt, sondern den Datensatz mit dem `pipe` Operator an die `ggplot()` Funktion geschickt, und dann mit `+` direkt die `geoms` hinzugefügt. Außerdem haben wir weitere Funktionen verwendet, wie z.B. `theme_classic()`, um den Hintergrund weiß darzustellen.

```
stress_psych %>%
 ggplot(mapping = aes(x = geschlecht,
 y = stress_psych,
 fill = geschlecht)) +
 geom_violin() +
 geom_jitter(width = 0.2, alpha = 0.6) +
 theme_classic()
```



## 5.4 Geoms für verschiedene Datentypen

Wir fassen zusammen: bisher haben wir gelernt, dass wir einen Plot in mehreren Schritten zusammenstellen. Wir beginnen mit einem Dataframe und definieren mit der `ggplot()` Funktion ein `ggplot2` Objekt. Mit der

`aes()` Funktion weisen wir Variablen eines Dataframes der X-, bzw. der Y-Achse zu und definieren weitere “aesthetic mappings”, z.B. eine farbliche Codierung anhand einer Gruppierungsvariablen. Anschliessen fügen wir dem Plot-Objekt Grafikelemente mit `geom_*` Funktionen als “Layers” hinzu.

Nun schauen wir uns eine Auswahl an `geoms` für verschiedene Kombination von Variablen an. Wir können dabei entweder eine Variable auf der X-Achse oder zwei Variablen auf den X- und Y-Achsen darstellen und diese Variablen können entweder kontinuierlich oder kategorial sein.

- Wir werden hier nur eine kleine Auswahl der möglichen `ggplot2` Funktionen betrachten. Das Package ist sehr umfangreich und hat eine sehr übersichtliche Website, auf der alles dokumentiert ist: `ggplot2` Dokumentation

Nachdem Sie dieses Kapitel durchgearbeitet haben, sind Sie in der Lage, selber Lösungen für grafische Darstellung zu finden. Datenvisualisierung kann ein sehr kreativer Prozess sein und macht Spass! Weitere Beispiele für spezifische Datenanalysemethoden sehen Sie in den nachfolgenden Kapiteln.

Für die folgenden Beispiele verwenden wir die West/Ost- und Kinderwunsch-Datensätze:

```
library(dplyr)
library(tidyr)
library(stringr)
library(haven)

westost <- read_sav("data/Beispieldatensatz.sav")

westost <- westost %>%
 mutate(westost = as_factor(westost),
 geschlecht = as_factor(geschlecht),
 bildung_mutter = as_factor(bildung_mutter,
 levels = "values"),
 bildung_vater = as_factor(bildung_vater,
 levels = "values"),
 ID = as.factor(ID))

kinderwunsch <- read_sav("data/Kinderwunsch_Schweiz.sav")

kinderwunsch <- kinderwunsch %>%
 mutate(geschl = as_factor(geschl))
```

### 5.4.1 Eine Variable

Wenn wir nur eine Variable auf der X-Achse grafisch darstellen möchten, müssen wir aber dennoch Werte auf der Y-Achse darstellen. Dies wird oft eine deskriptive Zusammenfassung wie z.B. Häufigkeiten sein.

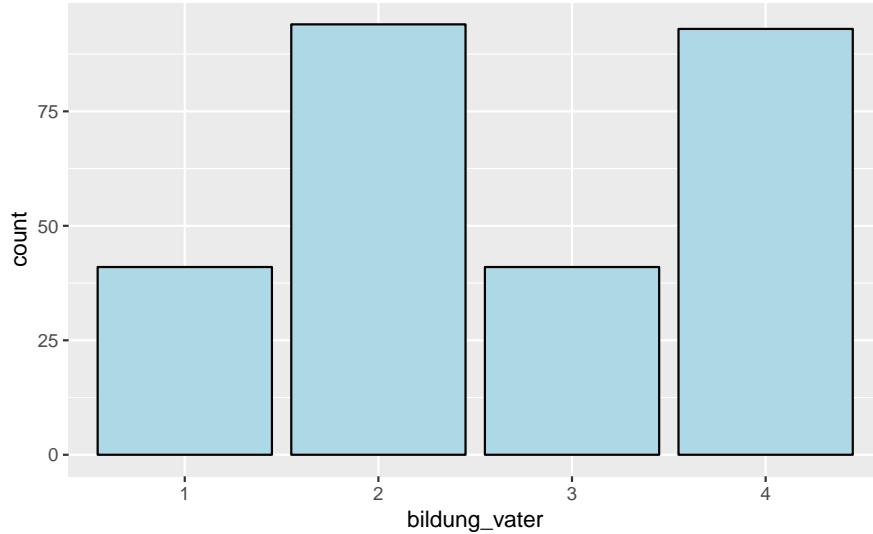
#### Kategoriale Variablen

Wenn wir eine kategoriale Variable grafisch darstellen, verwenden wir oft einen **bar chart** or **bar graph**. Dieser stellt z.B. Häufigkeiten der verschiedenen Kategorien anhand eines Rechtecks (rectangular bar) dar. Die Funktion, welche dafür verwendet wird, heißt `geom_bar()`.

Als Beispiel wollen wir die Häufigkeiten der vier Bildungsstufen des Vaters plotten.

- Wenn wir `fill = 'lightblue'`, `color = 'black'` nicht innerhalb der `aes()` Funktion verwenden, dann werden diese Argumente nicht als Gruppierungsanweisung aufgefasst. Wir können z.B. mit `fill = 'lightblue'` einfach alle Elemente hellblau einfärben.

```
p <- westost %>%
 select(bildung_vater) %>%
 drop_na() %>%
 ggplot(aes(x = bildung_vater))
p + geom_bar(fill = 'lightblue', color = 'black')
```

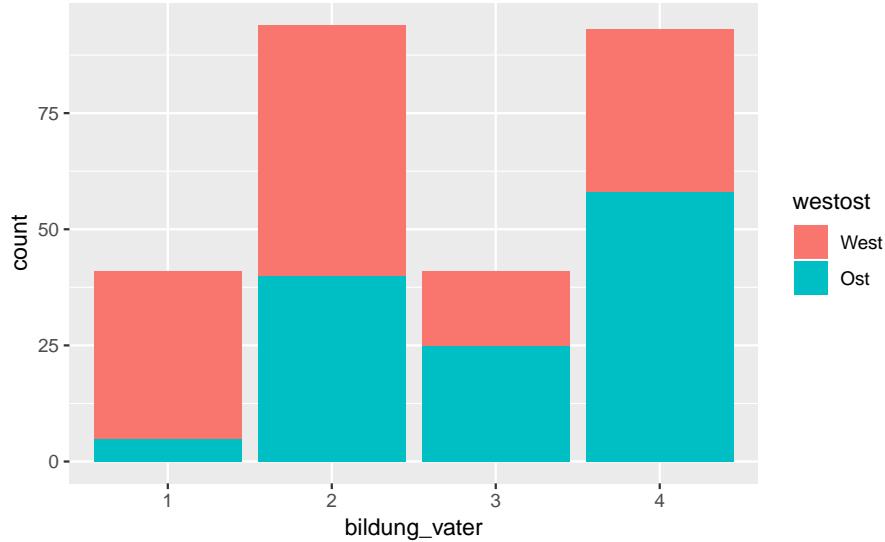


Eine Übersicht über die möglichen Farbnamen erhalten Sie mit der Funktion `colors()`. Es gibt 657 davon, wir zeigen hier mit `sample(15)` nur 15 zufällig ausgewählte an:

```
colors() %>% sample(15)
#> [1] "darkturquoise" "cyan4" "khaki2"
#> [4] "rosybrown" "slategray3" "lightsteelblue3"
#> [7] "gray76" "antiquewhite3" "grey19"
#> [10] "mediumpurple3" "grey6" "grey95"
#> [13] "darksalmon" "blue2" "yellow"
```

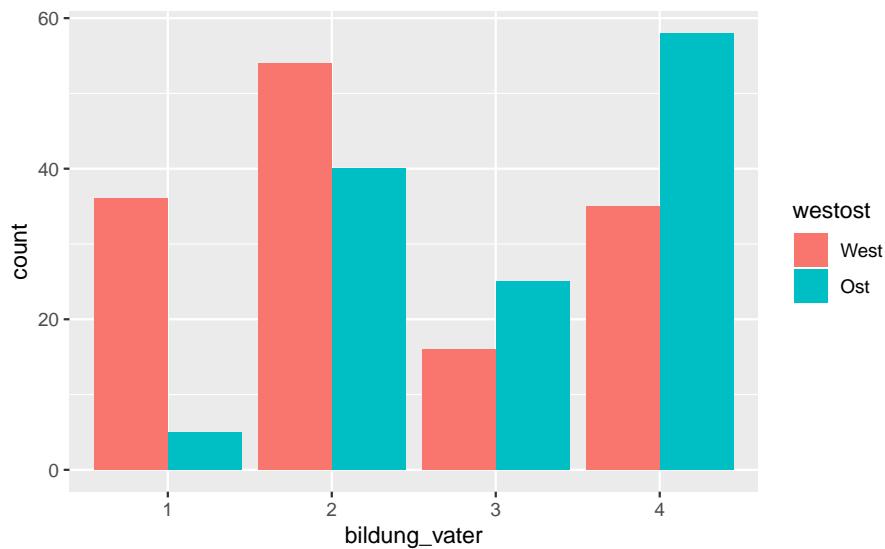
Auch hier können wir zusätzlich eine Gruppierungsvariable angeben, anhand derer wir die Rechtecke farblich kodieren.

```
p <- westost %>%
 select(bildung_vater, westost) %>%
 drop_na() %>%
 ggplot(aes(x = bildung_vater, fill = westost))
p + geom_bar()
```



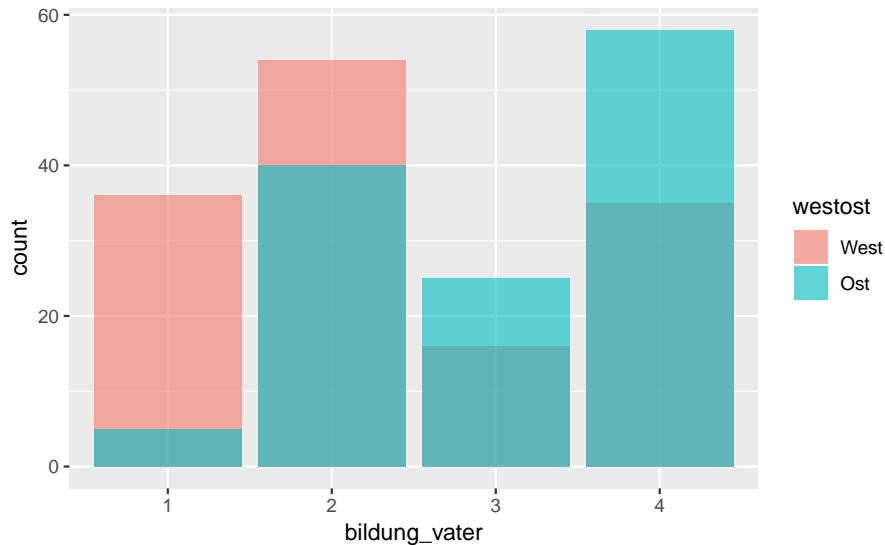
Standardmäßig kreiert `ggplot2` einen *stacked* Bar Chart, d.h. die Rechtecke werden aufeinander gestapelt. Wenn dies nicht erwünscht ist, können wir das Argument `position = "dodge"` der Funktion `geom_bar()` verwenden. Damit teilen wir mit, dass die Bars nebeneinander gezeichnet werden sollen.

```
p + geom_bar(position = "dodge")
```



Als dritte Variante können wir `position = "identity"` verwenden; so werden die Bars übereinander gezeichnet. Da das hintere Rechteck nicht mehr sichtbar ist, verwenden wir das `alpha` Argument, um die Bars transparent zu machen.

```
p + geom_bar(position = "identity", alpha = 0.6)
```



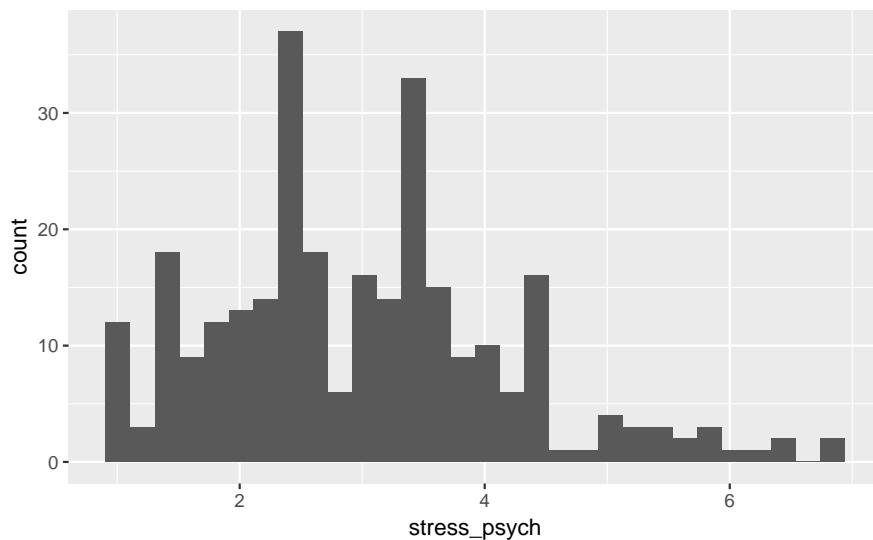
### Kontinuierliche Variablen

Falls die Variable, welche wir grafisch darstellen wollen, nicht kategorial, sondern kontinuierlich ist, bietet sich ein Histogramm an; dies erzeugen wir mit der Funktion `geom_histogram()`. Als Beispiel betrachten wir die psychische Symptomatik.

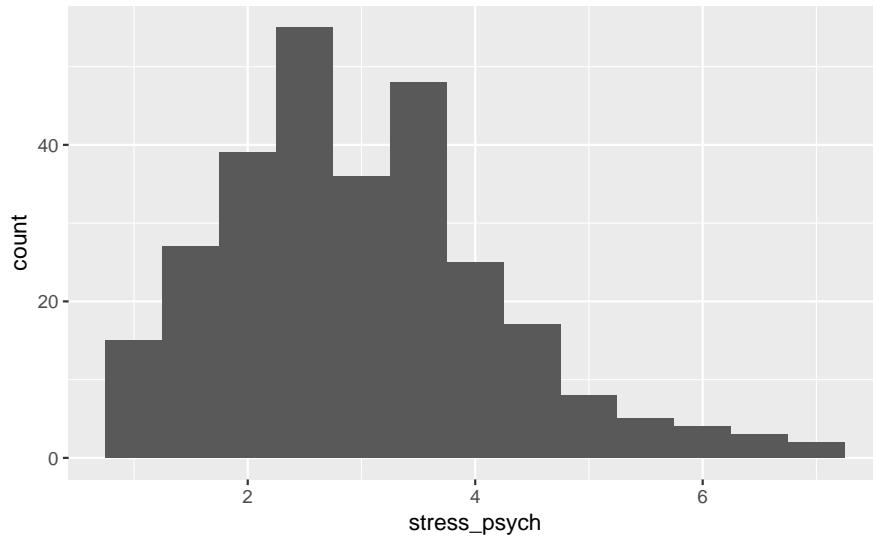
Ein Histogramm bietet eine grafische Darstellung der Verteilung einer numerischen Variablen. Dazu werden die Werte dieser Variablen in diskrete Intervalle, oder `bins`, unterteilt. Auf der Y-Achse werden dann, analog zu einem Bar Chart, die Häufigkeiten in den jeweiligen Intervallen dargestellt. Die Bestimmung der Grösse der Intervalle (`binwidth`) ist kritisch. Wenn wir nichts spezifizieren, wählt `ggplot2` selber eine `binwidth` aus, aber wir können diese mit dem Argument `binwidth` auch selber angeben.

```
p <- westost %>%
 select(stress_psych) %>%
 drop_na() %>%
 ggplot(mapping = aes(x = stress_psych))

Wir lassen die binwidth automatisch auswählen
p + geom_histogram()
#> `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



```
Wir bestimmen die binwidth selber
p + geom_histogram(binwidth = 0.5)
```



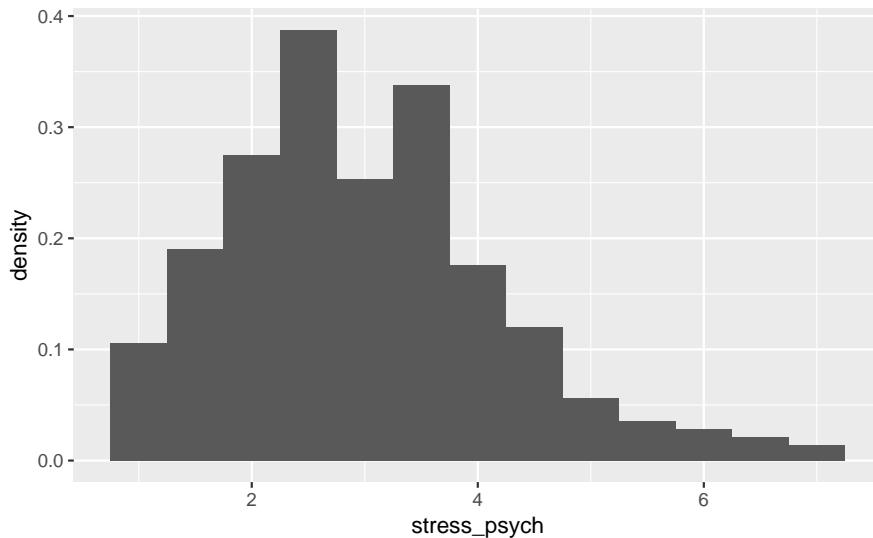
Die Bestimmung der `binwidth` hängt natürlich von der Skala der Variablen ab und sollte weder zu fein noch zu grob sein.

■ Probieren Sie mehrere Werte für die `binwidth` aus. Was ist optimal? ■

Wenn wir auf der Y-Achse anstelle der absoluten die relativen Häufigkeiten sehen wollen, können wir als `y = ..density..` als Argument der `aes()` Funktion verwenden.

```
p <- westost %>%
 select(stress_psych) %>%
 drop_na() %>%
 ggplot(mapping = aes(x = stress_psych, y = ..density..))

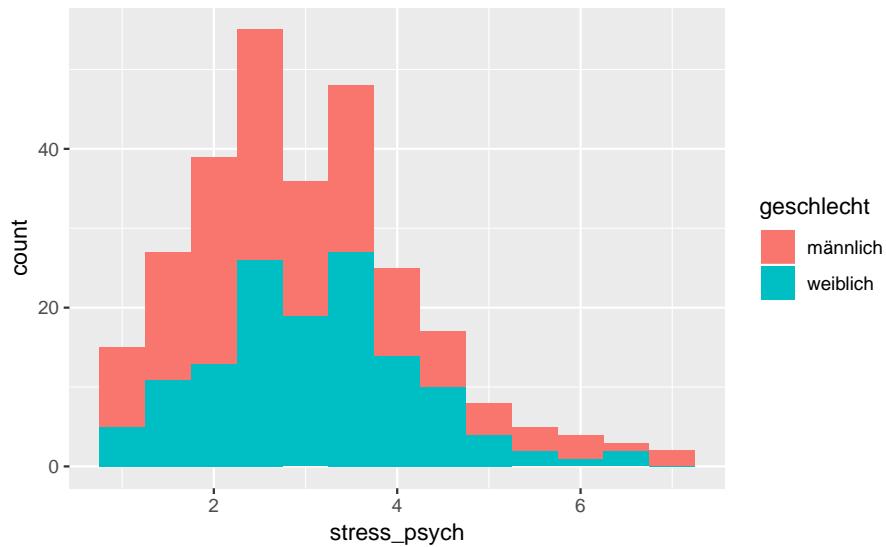
p + geom_histogram(binwidth = 0.5)
```



Selbsverständlich gibt es auch für Histogramme die Möglichkeit, eine Gruppierungsvariable zu verwenden.

```
p <- westost %>%
 select(stress_psych, geschlecht) %>%
 drop_na() %>%
 ggplot(mapping = aes(x = stress_psych, fill = geschlecht))

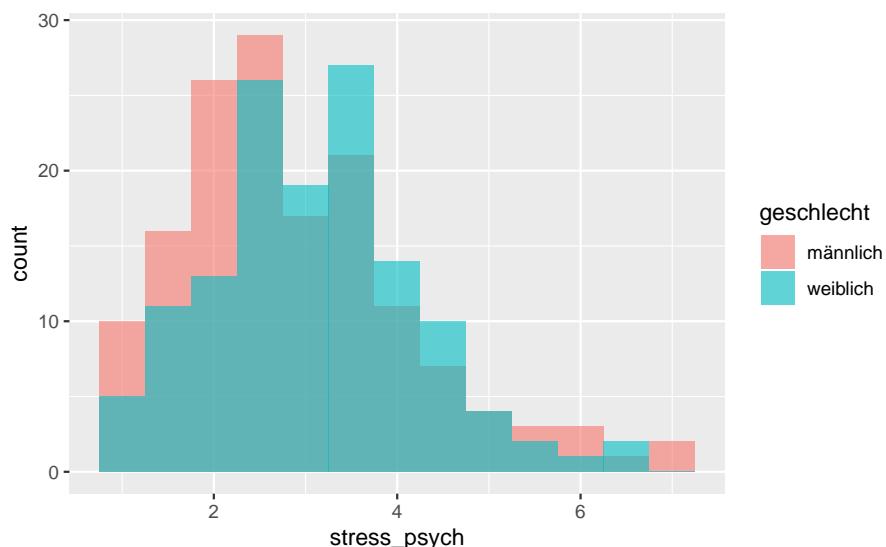
p + geom_histogram(binwidth = 0.5)
```



Wie beim Bar Chart werden die Histogramme übereinander geplottet (“stacked”). Wollen wir sie aufeinander, verwenden wir `position = "identity"`.

```
p <- westost %>%
 select(stress_psych, geschlecht) %>%
 drop_na() %>%
 ggplot(mapping = aes(x = stress_psych, fill = geschlecht))

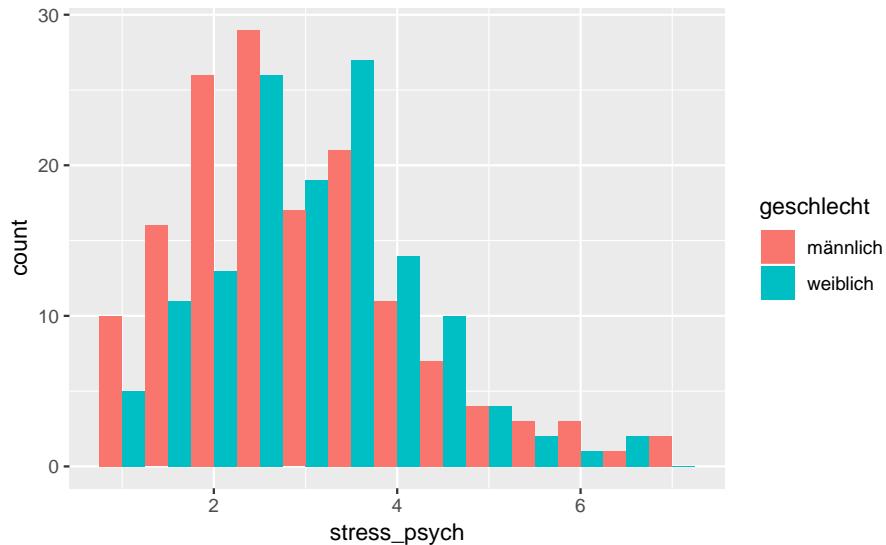
p + geom_histogram(binwidth = 0.5,
 position = "identity",
 alpha = 0.6)
```



Nebeneinander geht auch:

```
p <- westost %>%
 select(stress_psych, geschlecht) %>%
 drop_na() %>%
 ggplot(mapping = aes(x = stress_psych, fill = geschlecht))

p + geom_histogram(binwidth = 0.5,
 position = "dodge")
```



### 5.4.2 Zwei Variablen

Nun stellen wir zwei Variablen eines Datensatzes gemeinsam dar. Auch hier hängen die möglichen `geoms` vom Datentyp der Variablen ab.

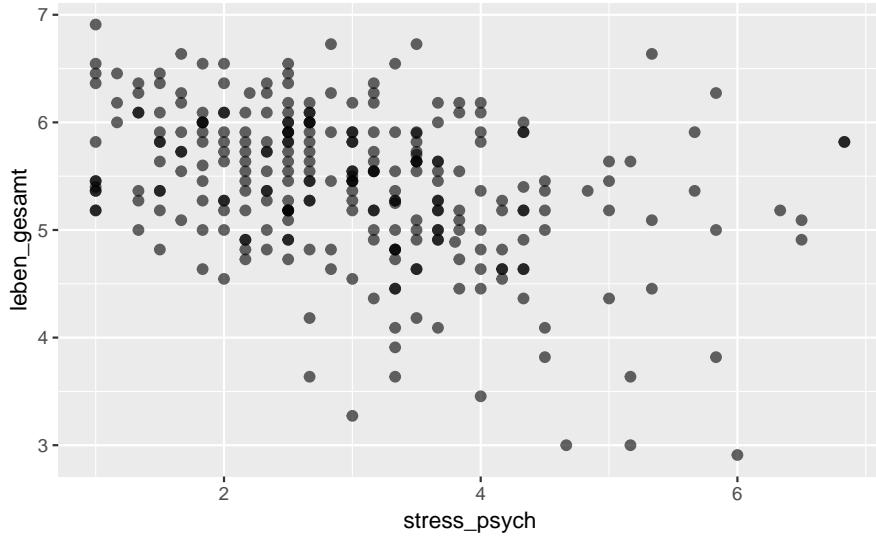
#### X und Y kontinuierlich

Wenn beide Variablen kontinuierlich sind, können wir deren Zusammenhang anhand eines ‘Scatterplots’ oder eines Liniendiagramms darstellen. Wir verwenden die Funktionen `geom_point()`, bzw. `geom_line()`.

Als Beispiel wollen wir den Zusammenhang zwischen psychische Symptomatik und Lebenszufriedenheit visualisieren.

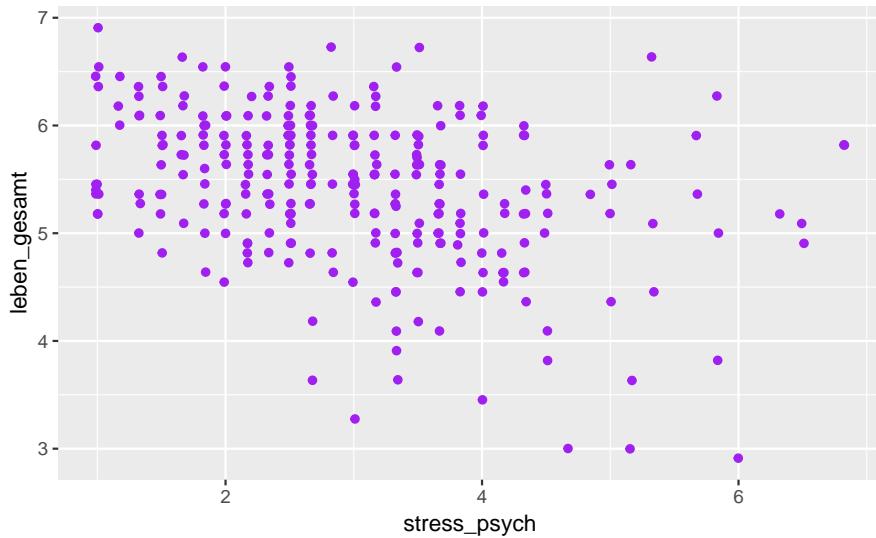
```
p <- westost %>%
 select(stress_psych, leben_gesamt) %>%
 drop_na() %>%
 ggplot(mapping = aes(x = stress_psych, y = leben_gesamt))

p + geom_point(size = 2, alpha = 0.6)
```



Die `size` und `alpha` Argumente haben wir weiter oben bereits kennengelernt, sowie die Möglichkeit, ‘overplotting’ mit der Funktion `geom_jitter()` zu vermeiden. Sowohl `geom_jitter()` als auch `geom_point()` haben auch eine `colour` oder ein `color` Argument.

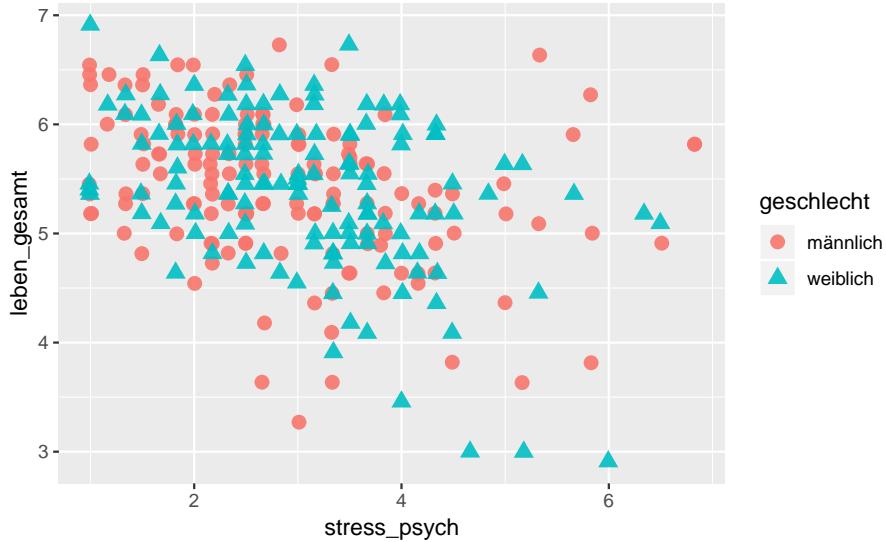
```
p + geom_jitter(colour = "purple")
```



Die Gruppierung anhand einer kategorialen Variablen funktioniert auch hier. Wir verwenden sowohl die Farbe als auch die Form der Punkte, um die Kategorien besser unterscheiden zu können.

```
p <- westost %>%
 select(stress_psych, leben_gesamt, geschlecht) %>%
 drop_na() %>%
 ggplot(mapping = aes(x = stress_psych,
 y = leben_gesamt,
 color = geschlecht,
 shape = geschlecht))

p + geom_jitter(size = 3, alpha = 0.9)
```



Mit der Funktion `geom_line()` können wir Liniendiagramme erstellen. Als Beispiel wollen wir in einem neuen Dataframe die mittlere Schulnote für die verschiedenen Bildungsniveaus des Vaters berechnen, und dann grafisch darstellen. Bevor wir die durchschnittliche Note plotten, konvertieren wir den Faktor `bildung_vater` zu einer numerischen Variable.

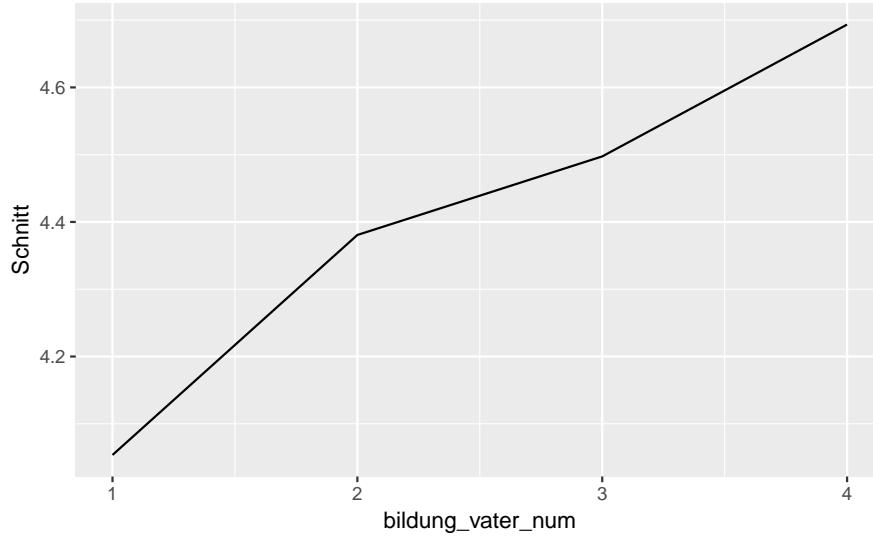
- Wir könnten `bildung_vater` auch als Faktor verwenden, müssten dann aber eine Gruppierungsvariable für `geom_line()` definieren. In diesem würden wir `group = 1` verwenden, da wir nur eine Gruppe haben:  
`p + geom_line(group = 1)`

```
bildung_vater <- westost %>%
 select(Schnitt, bildung_vater) %>%
 drop_na() %>%
 group_by(bildung_vater) %>%
 summarise(Schnitt = mean(Schnitt)) %>%
 mutate(bildung_vater_num = as.numeric(bildung_vater))

bildung_vater
#> # A tibble: 4 x 3
#> bildung_vater Schnitt bildung_vater_num
#> <fct> <dbl> <dbl>
#> 1 1 4.05 1
#> 2 2 4.38 2
#> 3 3 4.50 3
#> 4 4 4.69 4

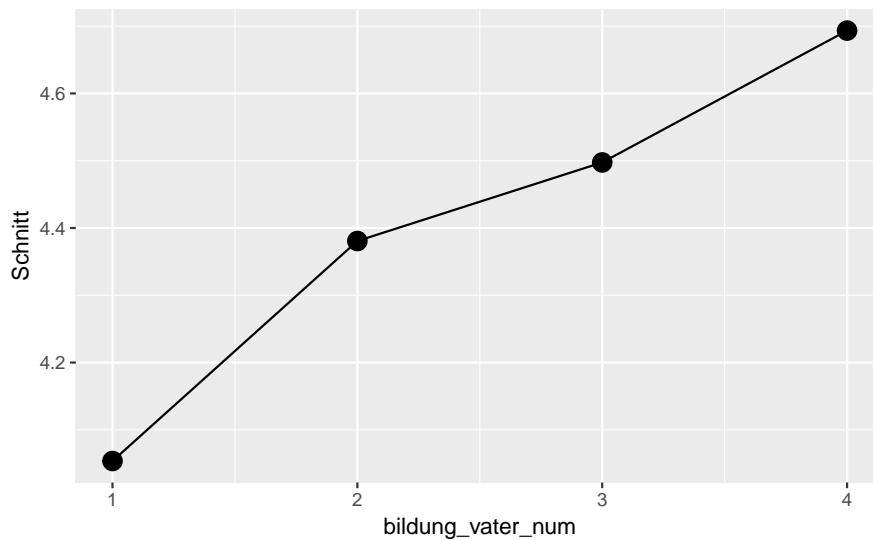
p <- bildung_vater %>%
 ggplot(aes(x = bildung_vater_num,
 y = Schnitt))

p + geom_line()
```



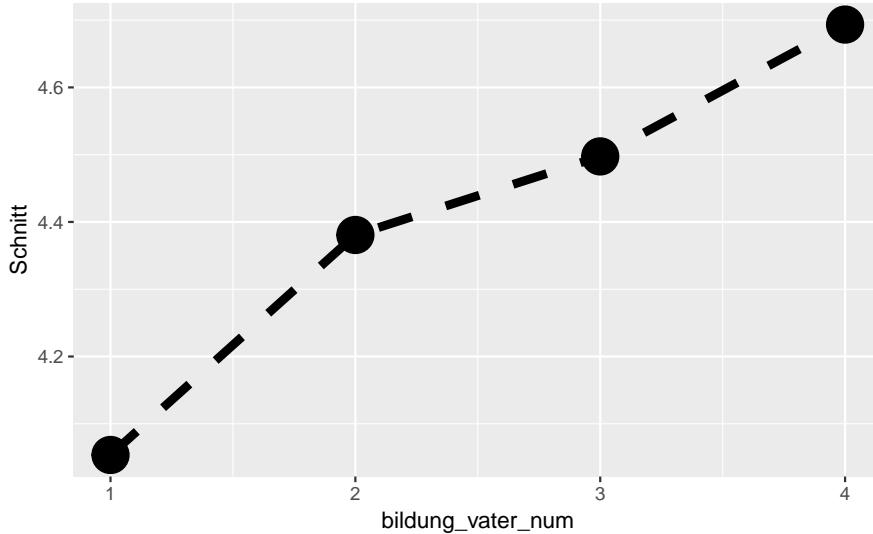
Wir können nun wie im “Honeymoon oder Hangover”-Beispiel das Liniendiagramm um Punkte ergänzen:

```
p + geom_line() +
 geom_point(size = 4)
```



Auch `geom_line()` hat Argumente, um die Eigenschaften zu ändern. In diesem Fall benutzen wir das Argument `linetype`, welches die Werte "blank", "solid", "dashed", "dotted", "dotdash", "longdash" oder "twodash" annehmen kann.

```
p + geom_line(linetype = "dashed", size = 2) +
 geom_point(size = 8)
```



### X kategorial und Y kontinuierlich

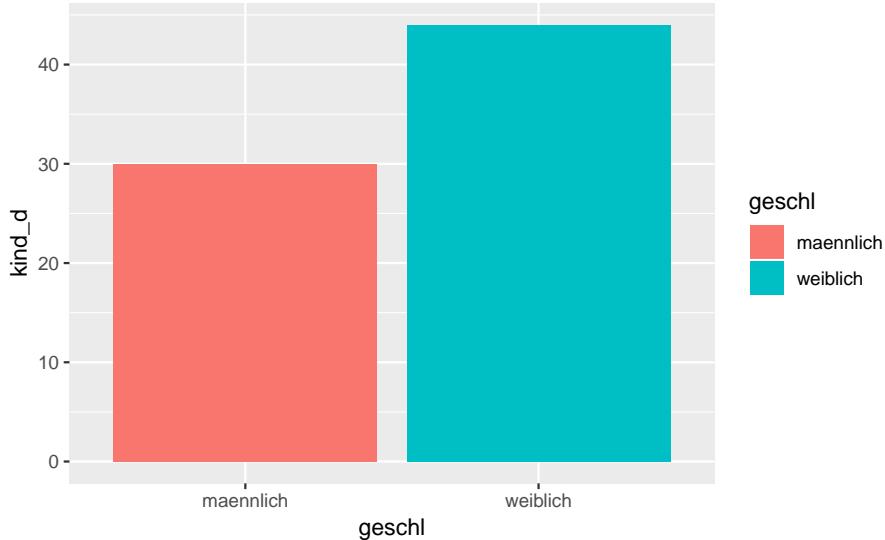
Wenn eine der Variablen kategorial ist, können wir diese, anstatt sie als Gruppierungsvariable zu verwenden, auf einer Achse darstellen.

Beispiele dafür haben wir oben schon gesehen: dort haben wir die Variablen `geschlecht` und `psychische Symptomatik` dargestellt und die Funktionen `geom_boxplot()` und `geom_violin()` benutzt. Wir können aber auch die Funktion `geom_bar()` für zwei Variablen verwenden. Die Variable auf der Y-Achse wird in diesem Fall für alle Beobachtungen in den Kategorien auf der X-Achse summiert. Da dies keine statistische Transformation benötigt, verwenden wir das Argument `stat = 'identity'`.

Als Beispiel betrachten wir den Kinderwunsch-Datensatz. In diesem wurden Versuchspersonen gefragt, ob sie ein Kind wollen oder nicht (binäre Antwort). Zusätzlich wurden die Intimität zur eigenen Mutter, die emotionale Einstellung zu Kindern und das Geschlecht erhoben. Auf der Y-Achse stellen wir die absoluten Häufigkeiten einer „Ja“-Antwort dar.

```
p <- kinderwunsch %>%
 ggplot(aes(x = geschl,
 y = kind_d,
 fill = geschl))

p + geom_bar(stat = 'identity')
```



Um diese Grafik besser verstehen zu können, berechnen wir zusätzlich noch die relativen Häufigkeiten einer „Ja“-Antwort pro Geschlecht.

```
kinderwunsch %>%
 group_by(geschl) %>%
 summarise(N = length(kind_d),
 Ja = sum(kind_d),
 prop_Ja = sum(kind_d)/N)
#> # A tibble: 2 x 4
#> geschl N Ja prop_Ja
#> <fct> <int> <dbl> <dbl>
#> 1 maennlich 44 30 0.682
#> 2 weiblich 56 44 0.786
```

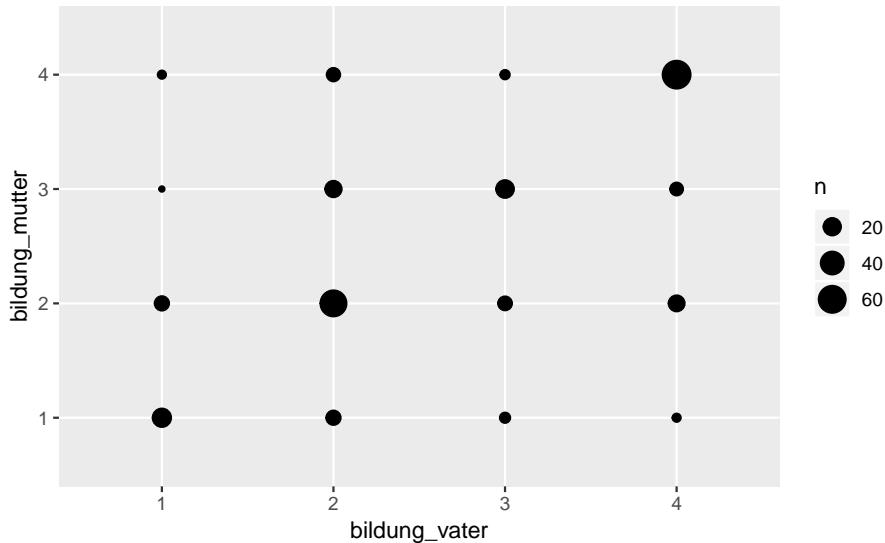
## X und Y kategorial

Zuletzt können die Variablen sowohl auf der X- als auch auf der Y-Achse kategorial sein. In diesem Fall wäre es sinnvoll, die gemeinsamen Häufigkeiten grafisch darzustellen. Dafür gibt es die Funktion `geom_count()`.

Als Beispiel wollen wir die gemeinsame Häufigkeitsverteilung der Bildung des Vaters und der Bildung der Mutter betrachten.

```
p <- westost %>%
 select(starts_with("bildung")) %>%
 drop_na() %>%
 ggplot(aes(x = bildung_vater,
 y = bildung_mutter))

p + geom_count()
```



`geom_count()` zählt die gemeinsamen Häufigkeiten der Kategorien der beiden Variablen und stellt diese als Durchmesser der Punkte dar.

■ Die Häufigkeitstabelle erhalten wir mit der Funktion `table()`. ■

```
table(westost$bildung_vater, westost$bildung_mutter)
#>
#> 1 2 3 4
#> 1 23 12 2 3
#> 2 12 55 17 10
#> 3 5 11 21 4
#> 4 3 16 9 65
```

## Beispiele

Wir betrachten nun 2 Übungsbeispiele.

### Arbeitszufriedenheit

Im ersten Beispiel laden wir den Arbeitszufriedenheitsdatensatz vom letzten Kapitel nochmals und schauen uns die Grafik genauer an.

```
honeymoon <- read_delim("data/honeymoon.csv",
 delim = ";")
#> Parsed with column specification:
#> cols(
#> Firma = col_double(),
#> Anfang = col_double(),
#> Drei_Monate = col_double(),
#> Sechs_Monate = col_double()
#>)

Wiederholung vom letzten Kapitel:
honeymoon_long_bedingung <- honeymoon %>%
 mutate(Firma = as.factor(Firma)) %>%
```

```

 mutate(ID = 1:nrow(.)) %>%
 gather(key = Messzeitpunkt, value = Arbeitszufriedenheit,
 -Firma, -ID) %>%
 mutate(Messzeitpunkt = factor(Messzeitpunkt)) %>%
 arrange(Firma, ID) %>%
 group_by(Firma, ID) %>%
 mutate(Personenmittelwert = mean(Arbeitszufriedenheit)) %>%
 group_by(Firma, Messzeitpunkt) %>%
 summarise(Bedingungsmittelwert = mean(Arbeitszufriedenheit))

honeymoon_long_bedingung
#> # A tibble: 6 x 3
#> # Groups: Firma [2]
#> Firma Messzeitpunkt Bedingungsmittelwert
#> <fct> <fct> <dbl>
#> 1 1 Anfang 9
#> 2 1 Drei_Monate 5
#> 3 1 Sechs_Monate 7
#> 4 2 Anfang 10
#> 5 2 Drei_Monate 10.8
#> 6 2 Sechs_Monate 11

```

Wir stellen nun anhand eines Liniendiagrams die mittlere Arbeitszufriedenheit über die Messzeitpunkte hinweg dar, und zwar benützen wir als Gruppierungsfaktor die `Firma`. `group = Firma` ist hier wichtig, die anderen beiden Argumente, `color = Firma` und `linetype = Firma` sind nur aus ästhetischen Gründen da und könnten weggelassen werden.

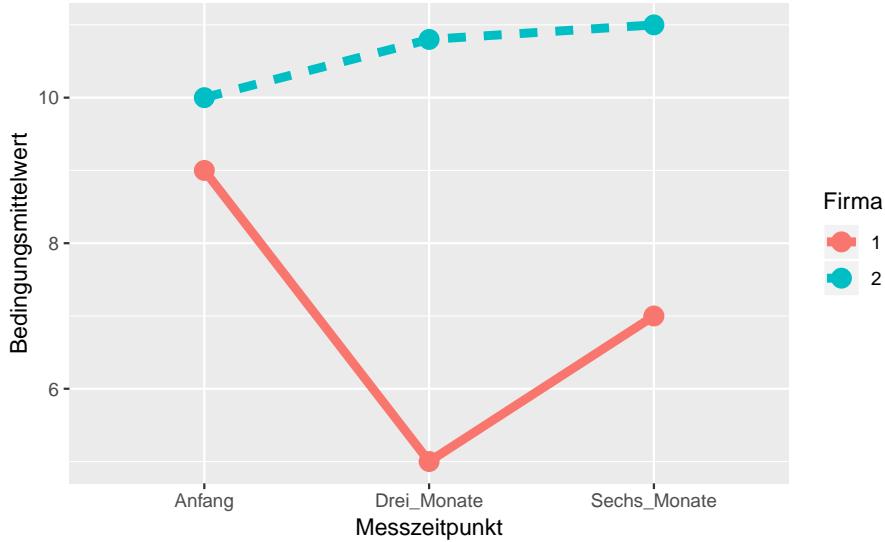
- Wie weiter oben schon angedeutet braucht `ggplot2` das `group` Argument, wenn wir ein Liniendiagramm mit einer kategorialen Variable auf der X-Achse erstellen wollen. ■

```

p <- honeymoon_long_bedingung %>%
 ggplot(aes(x = Messzeitpunkt,
 y = Bedingungsmittelwert,
 color = Firma,
 linetype = Firma,
 group = Firma))

p + geom_point(size = 4) +
 geom_line(size = 2)

```



Liniendiagramme werden oft benutzt, um den zeitlichen Verlauf einer Variablen darzustellen. Dies bedeutet, dass wir auf der X-Achse die Zeit darstellen, wie in diesem Beispiel. Meistens wird Zeit jedoch als kontinuierliche Variable verwendet und nicht, wie hier, als Faktor.

### Wide vs. Long: Bildung der Eltern

Anhand des nächsten Beispiels betrachten wir den Unterschied zwischen dem *long* und dem *wide* Format. Wir haben, als wir die gemeinsame Häufigkeitsverteilung der Bildung des Vaters und der Mutter dargestellt haben, `bildung_vater` und `bildung_mutter` als separate Variablen verwendet, um sie auf separaten Achsen darzustellen. Wir könnten jedoch auch `bildung_vater` und `bildung_mutter` als Stufen eines Messwiederholungsfaktors `eltern` (`key`) zusammenfassen, und die Bildungsniveaus als Messvariable `bildung` (`value`), d.h. als `key/value` Paar. Dies machen wir, wenn wir `bildung` als Variable auf einer Achse verwenden wollen und `eltern` als Gruppierungsvariable.

Dies ist vielleicht nicht ganz einfach zu verstehen, deshalb betrachten wir gleich ein konkretes Beispiel. Wir wollen nun, ähnlich wie oben, die mittlere Schulnote für die verschiedenen Bildungsniveaus der Eltern grafisch darstellen. Diesmal machen wir dies für beide Elternteile. Wir wollen jedoch unterschiedliche Linien für *Vater* und *Mutter*. Nun ist es wichtig, dass wir einen *long* Datensatz bilden.

```
westost <- read_sav("data/Beispieldatensatz.sav")

westost <- westost %>%
 mutate(westost = as_factor(westost),
 geschlecht = as_factor(geschlecht),
 bildung_mutter = as_factor(bildung_mutter,
 levels = "values"),
 bildung_vater = as_factor(bildung_vater,
 levels = "values"),
 ID = as.factor(ID))

bildung <- westost %>%
 # Variablen auswählen
 select(Schnitt, bildung_vater, bildung_mutter) %>%
 # Fehlende Werte ausschliessen
 drop_na() %>%
 # wide zu long
 gather(key = eltern, value = bildung, -Schnitt) %>%
 # zu Faktoren konvertieren
```

```

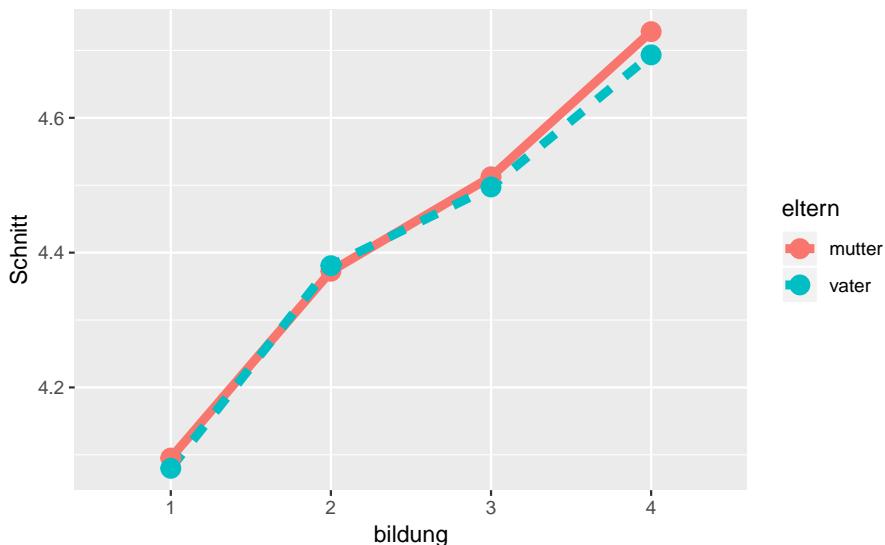
 mutate(eltern = as.factor(eltern),
 bildung = as.factor(bildung)) %>%
 mutate(eltern = str_replace(eltern, ".*_", ""))
 # Gruppieren: zuerst Eltern, dann Bildungsniveaus
 group_by(eltern, bildung) %>%
 # Mittlere Note berechnen
 summarise(Schnitt = mean(Schnitt))

bildung
#> # A tibble: 8 x 3
#> # Groups: eltern [2]
#> eltern bildung Schnitt
#> <chr> <fct> <dbl>
#> 1 mutter 1 4.10
#> 2 mutter 2 4.37
#> 3 mutter 3 4.51
#> 4 mutter 4 4.73
#> 5 vater 1 4.08
#> 6 vater 2 4.38
#> # ... with 2 more rows

p <- bildung %>%
 ggplot(aes(x = bildung,
 y = Schnitt,
 colour = eltern,
 linetype = eltern,
 group = eltern))

p + geom_line(size = 2) +
 geom_point(size = 4)

```



- An diesem Beispiel wird deutlich, dass ein grosser Teil der Arbeit mit ggplot2 darin besteht, die Daten zuerst ins ‘richtige’ Format zu bringen. Wenn dies getan ist, ist es jedoch relativ einfach, eine Grafik zu erstellen. Dies ist ein nicht zu unterschätzender Vorteil von ggplot2. ■

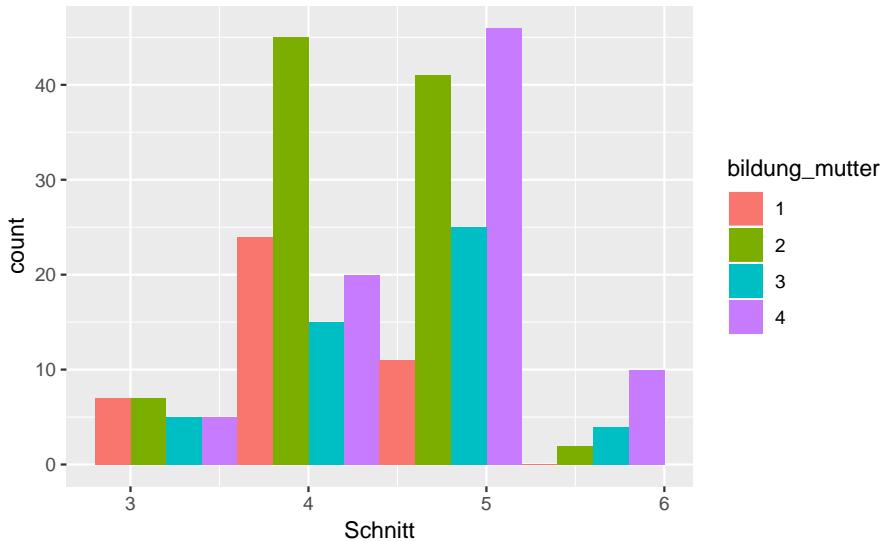
## 5.5 Facets

Bisher haben wir Gruppierungsvariablen dazu benutzt, um unterschiedliche Farben/Formen/Linien für die Kategorien der Gruppierungsvariable innerhalb eines Plots zu erzeugen. Manchmal ist dies jedoch zu unübersichtlich.

Wollen wir zum Beispiel ein Histogramm der Schulnoten erstellen, und zwar für jede Stufe der Bildung der Mutter, dann wäre die Grafik völlig überladen.

```
p <- westost %>%
 select(Schnitt, bildung_mutter) %>%
 drop_na() %>%
 ggplot(mapping = aes(x = Schnitt,
 fill = bildung_mutter))

p + geom_histogram(binwidth = 0.8,
 position = "dodge")
```



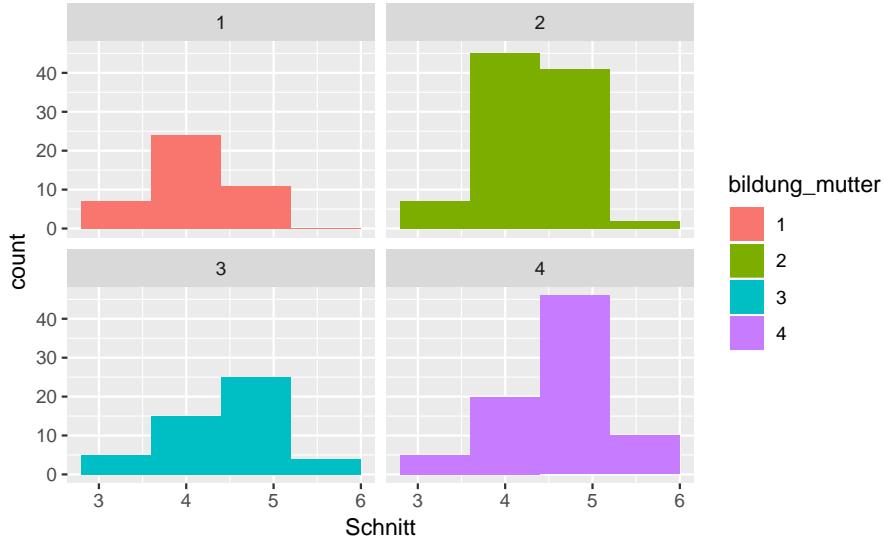
Eine offensichtliche Lösung wäre, die Histogramme für die Bildungsniveaus der Mutter in separaten Grafiken darzustellen.

Genau dies können wir mit den Funktionen `facet_wrap()` und `facet_grid()` machen.

Mit `facet_wrap()` erstellen wir so eine Grafik für jede Kategorie der Gruppierungsvariable:

```
p <- westost %>%
 select(Schnitt, bildung_mutter) %>%
 drop_na() %>%
 ggplot(mapping = aes(x = Schnitt,
 fill = bildung_mutter)) +
 facet_wrap(~ bildung_mutter)

p + geom_histogram(binwidth = 0.8)
```

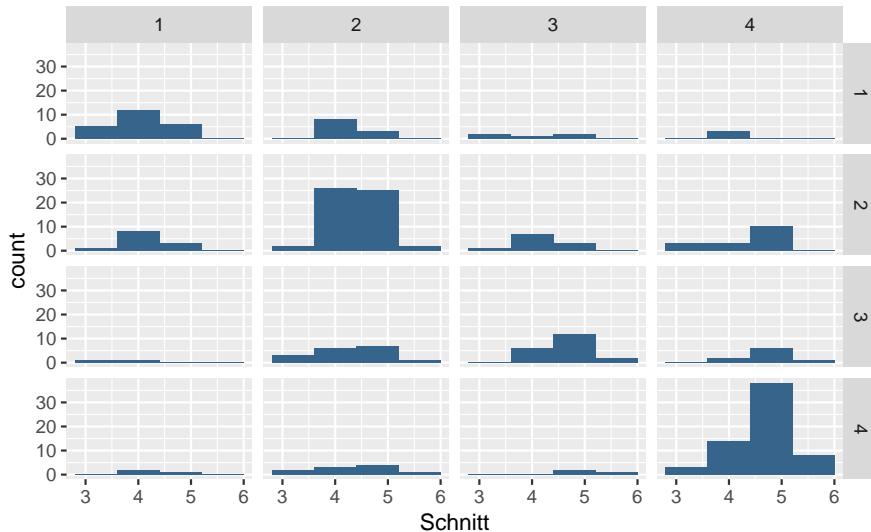


- Das ~ (Tilde) Zeichen bedeutet hier ungefähr: in Abhängigkeit oder als “Funktion” von.

Wenn wir zwei Gruppierungsvariablen haben, können wir mit `facet_grid()` ein Raster erzeugen.

```
p <- westost %>%
 select(Schnitt, bildung_mutter, bildung_vater) %>%
 drop_na() %>%
 ggplot(mapping = aes(x = Schnitt)) +
 facet_grid(bildung_mutter ~ bildung_vater)

p + geom_histogram(binwidth = 0.8,
 fill = 'steelblue4')
```



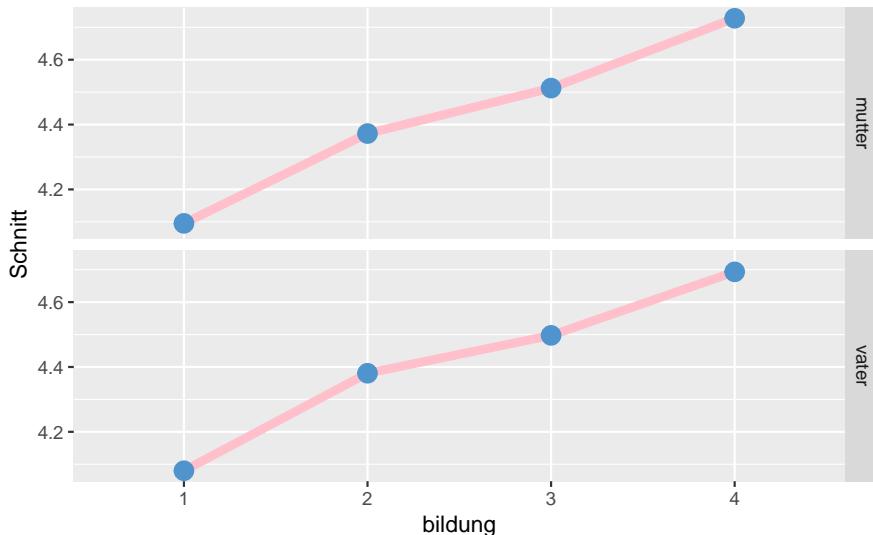
Hier werden die Stufen von `bildung_mutter` in den Zeilen dargestellt, die Stufen von `bildung_vater` in den Spalten.

Als zweites Beispiel können wir den Notenschnitt in Abhängigkeit der Bildung der Eltern als Liniendiagramm darstellen, und zwar in separaten Plots für Väter und Mütter getrennt. Anhand dieses Beispiels sehen wir, dass wir `facet_grid()` auch dann verwenden können, wenn wir nur eine Gruppierungsvariable haben, und zwar um die Anzahl Zeilen, bzw. Spalten festzulegen.

Wenn wir die Gruppierung in den Zeilen haben wollen, schreiben wir `facet_grid(Gruppierungsvariable ~ .)`, wenn sie in den Spalten wollen, schreiben wir `'facet_grid(. ~ Gruppierungsvariable)`. Der Punkt `.` bedeutet hier, dass wir für die Zeilen/Spalten keine Gruppierungsvariable verwenden.

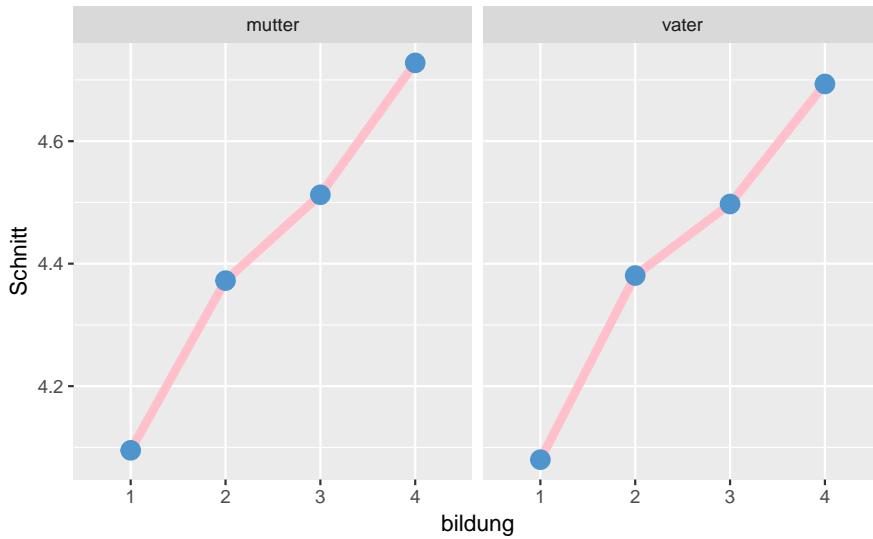
```
p <- bildung %>%
 ggplot(aes(x = bildung,
 y = Schnitt,
 group = eltern))

p + geom_line(size = 2, color = 'pink') +
 geom_point(size = 4, color = 'steelblue3') +
 # Stufen von 'eltern' in die Zeilen
 facet_grid(eltern ~ .)
```



```
p <- bildung %>%
 ggplot(aes(x = bildung,
 y = Schnitt,
 group = eltern))

p + geom_line(size = 2, color = 'pink') +
 geom_point(size = 4, color = 'steelblue3') +
 # Stufen von 'eltern' in die Spalten
 facet_grid(. ~ eltern)
```



## 5.6 Farben und Themes

Bisher hat `ggplot2` automatisch für uns Farben gewählt, wenn wir Farben für eine Gruppierung verlangt haben. Die Standard Farbpalette ist jedoch für Farbenblinde äusserst schlecht geeignet. Es gibt viele Farbpaletten, welche wir verwenden könnten.

Wir definieren hier jedoch eine eigene, für Farbenblinde geeignete Farbpalette.

```
palette <- c("#000000", "#E69F00",
 "#56B4E9", "#009E73",
 "#FOE442", "#0072B2",
 "#D55E00", "#CC79A7")
```

Wir definieren hier also einen Vektor von acht Hex Codes. Folglich dürfte unsere Gruppierungsvariable nicht mehr als acht Kategorien haben.

- Sie können hier natürlich eine eigene Farbpalette erstellen, indem Sie entweder Hex Codes oder Farbnamen angeben. ■

Die Palette verwenden wir so:

- um Formen auszufüllen, verwenden wir

```
scale_fill_manual(values = palette)
```

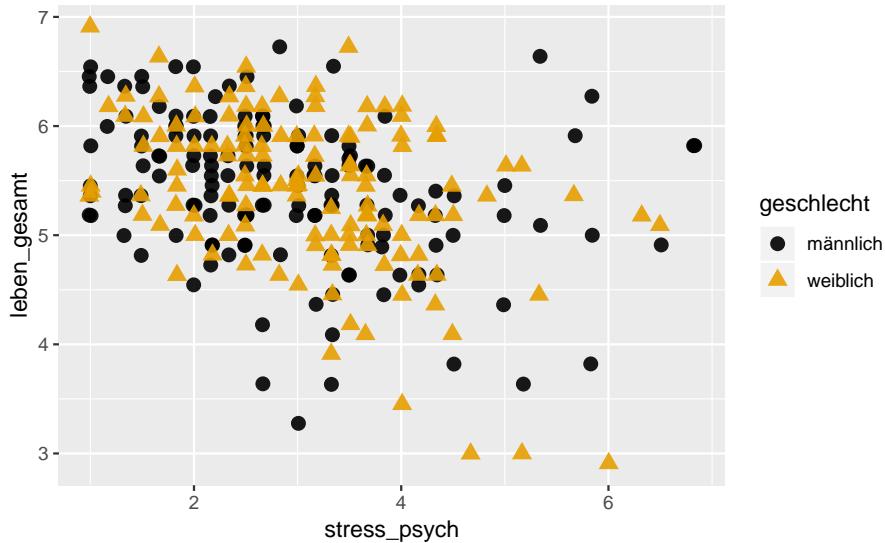
- für Linien und Punkte verwenden wir:

```
scale_colour_manual(values = palette)
```

Als Beispiel plotten wir nochmals den Zusammenhang zwischen `stress_psych` und `leben_gesamt`, diesmal mir unserer Farbpalette.

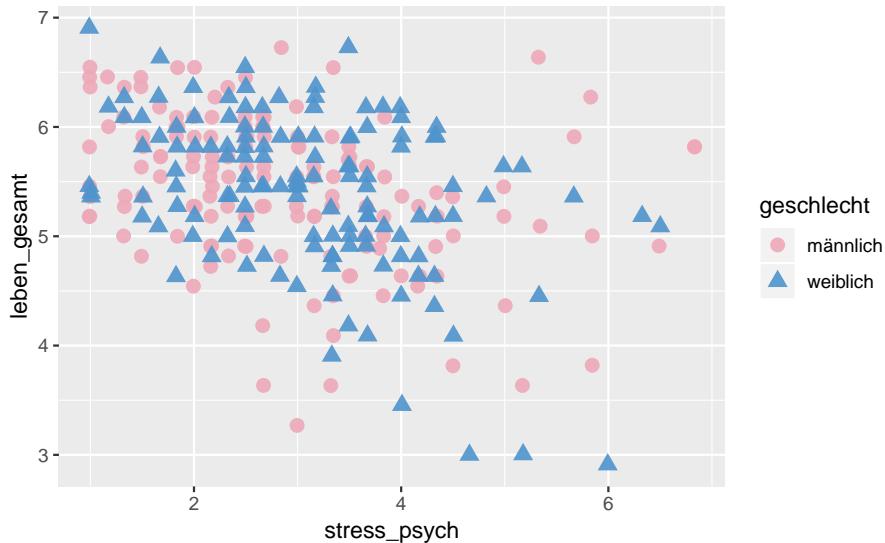
```
p <- westost %>%
 select(stress_psych, leben_gesamt, geschlecht) %>%
 drop_na() %>%
 ggplot(mapping = aes(x = stress_psych,
 y = leben_gesamt,
 color = geschlecht,
 shape = geschlecht))
```

```
p + geom_jitter(size = 3, alpha = 0.9) +
 scale_colour_manual(values = palette)
```



Wir könnten die Farben auch so ‘von Hand’ bestimmen:

```
p + geom_jitter(size = 3, alpha = 0.9) +
 scale_colour_manual(values = c("pink2", "steelblue3"))
```

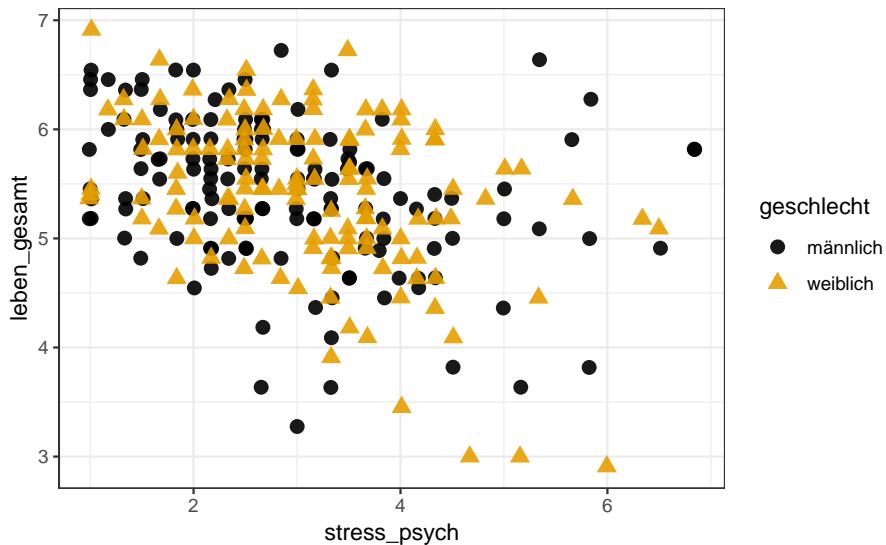


Ein weiterer heikler Punkt ist der graue Hintergrund, den ggplot2 automatisch wählt. Diesen können wir am einfachsten ändern, indem wir ein `theme` definieren. Es gibt zwei solcher `themes`, welche einen weissen Hintergrund haben: `theme_bw()` und `theme_classic()`. Diese unterscheiden sich darin, dass `theme_classic()` keine ‘grid lines’ zeichnet, und nur die linke und untere Achse.

```
p <- westost %>%
 select(stress_psych, leben_gesamt, geschlecht) %>%
 drop_na() %>%
 ggplot(mapping = aes(x = stress_psych,
 y = leben_gesamt,
 color = geschlecht,
```

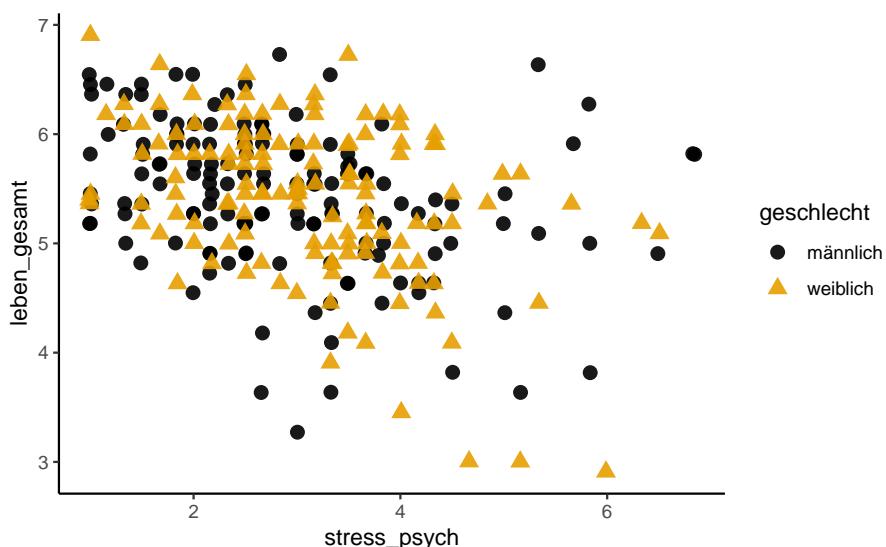
```
shape = geschlecht))

p + geom_jitter(size = 3, alpha = 0.9) +
 scale_colour_manual(values = palette) +
 theme_bw()
```



```
p <- westost %>%
 select(stress_psych, leben_gesamt, geschlecht) %>%
 drop_na() %>%
 ggplot(mapping = aes(x = stress_psych,
 y = leben_gesamt,
 color = geschlecht,
 shape = geschlecht))

p + geom_jitter(size = 3, alpha = 0.9) +
 scale_colour_manual(values = palette) +
 theme_classic()
```



## 5.7 Beschriftungen

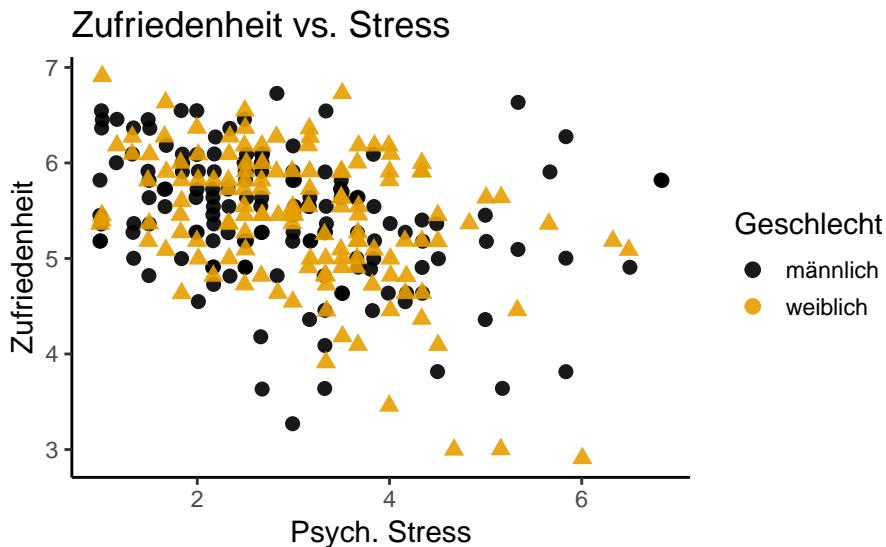
Wir können nun auch noch mit `xlab()` und `ylab()` die Beschriftungen der X/Y-Achsen ändern, und mit der Funktion `ggtitle()` dem Plot einen Titel geben. Mit der Funktion `labs()` können wir zusätzlich noch den Titel der Legende ändern.

- Da wir sowohl `color` als auch `shape` als Gruppierung verwendet haben, erhalten wir zwei Legenden. Eine davon, z.B. diejenige für `shape` können wir mit `guides(shape = FALSE)` weglassen. ■

Zuletzt wollen wir auch die Schriftgrösse ändern, da die Standardgrösse oft zu klein erscheint. Dies erreichen wir mit dem Argument `base_size = SCHRIFTGRÖSSE` der `theme_*` Funktionen.

```
p <- westost %>%
 select(stress_psych, leben_gesamt, geschlecht) %>%
 drop_na() %>%
 ggplot(mapping = aes(x = stress_psych,
 y = leben_gesamt,
 color = geschlecht,
 shape = geschlecht))

p + geom_jitter(size = 3, alpha = 0.9) +
 scale_colour_manual(values = palette) +
 theme_classic(base_size = 14) +
 ggtitle("Zufriedenheit vs. Stress") +
 xlab("Psych. Stress") +
 ylab("Zufriedenheit") +
 # Titel der color-Legende ist "Geschlecht"
 labs(color = "Geschlecht") +
 # Legende für shape weglassen
 guides(shape = FALSE)
```



## 5.8 Grafiken speichern

Wenn wir eine schöne Grafik erstellt haben, wollen wir sie natürlich speichern. Dies können wir mit der Funktion `ggsave()` machen. Die Funktion nimmt als Argumente den Dateinamen, den Namen des Plot Objekts und weitere Eigenschaften, wie die gewünschte Höhe und Breite des Plots. Diese können z.B. in

“cm” angegebenen werden, mit dem Argument `units = "cm"`. Um die Grafik zu speichern, müssen wir also unser fertiges `ggplot2` Objekt einer Variablen zugewiesen haben:

```
p <- westost %>%
 select(stress_psych, leben_gesamt, geschlecht) %>%
 drop_na() %>%
 ggplot(mapping = aes(x = stress_psych,
 y = leben_gesamt,
 color = geschlecht,
 shape = geschlecht))

Wir nennen die Grafik 'my_plot'
my_plot <- p + geom_jitter(size = 3, alpha = 0.9) +
 scale_colour_manual(values = palette) +
 theme_classic(base_size = 14) +
 ggtitle("Zufriedenheit vs. Stress") +
 xlab("Psych. Stress") +
 ylab("Zufriedenheit") +
 # Titel der color-Legende ist "Geschlecht"
 labs(color = "Geschlecht") +
 # Legende für shape weglassen
 guides(shape = FALSE)
```

`my_plot` kann nun gespeichert werden:

```
ggsave(filename = "my_plot.png",
 plot = my_plot)
#> Saving 6 x 3.71 in image
```

Es gibt auch weitere Möglichkeiten, Grafiken zu speichern. Diese werden wir in einem separaten Screencast erklären.

## 5.9 Übungsaufgabe

In dieser Übungsaufgabe wollen wir die sechs Selbstwirksamkeitsskalen im West/Ost Datensatz untersuchen. Wir gehen davon aus, dass diese in einem Zusammenhang mit dem Gesamtnotenschnitt stehen. Bevor wir eine multiple Regression rechnen, wollen wir versuchen, Zusammenhänge zwischen diesen Variable grafisch darstellen.

```
selbstwirksamkeit_wide <- westost %>%
 select(ID, Schnitt, starts_with("swk_")) %>%
 drop_na()

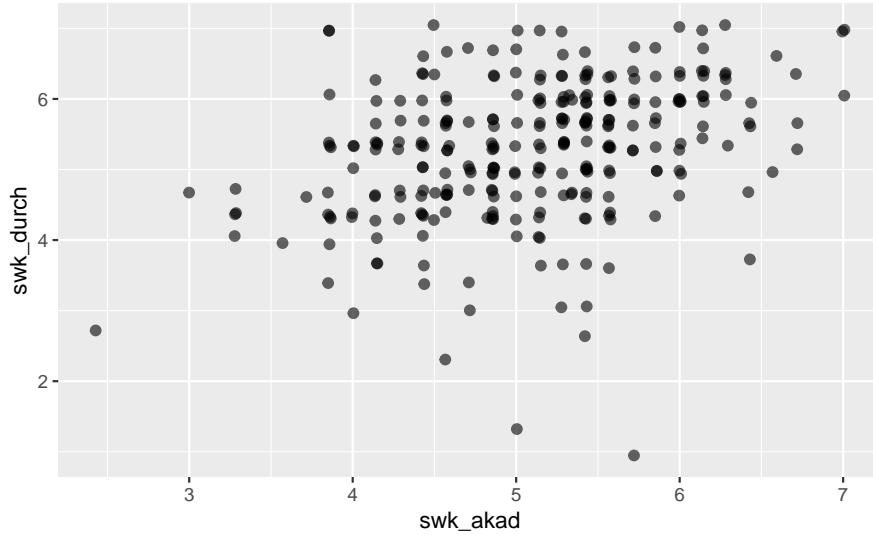
selbstwirksamkeit_wide
#> # A tibble: 279 x 8
#> ID Schnitt swk_akad swk_selbstreg swk_durch swk_motselbst swk_sozharm
#> <fct> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1 2 4 5 4 5 4.6 5.57
#> 2 14 4 4.86 4.5 5.33 4.6 4.71
#> 3 15 4 4 4.38 4.33 5.6 4.43
#> 4 17 4 6.14 5.62 6 6 6
#> 5 18 4 5.14 4.62 4.33 4.4 4.14
#> 6 19 4 5.43 4.62 6.33 5.2 6.29
#> # ... with 273 more rows, and 1 more variable: swk_bez <dbl>
```

Versuchen Sie, Zusammenhänge zwischen den sechs Selbstwirksamkeitsskalen zu entdecken, indem Sie diese

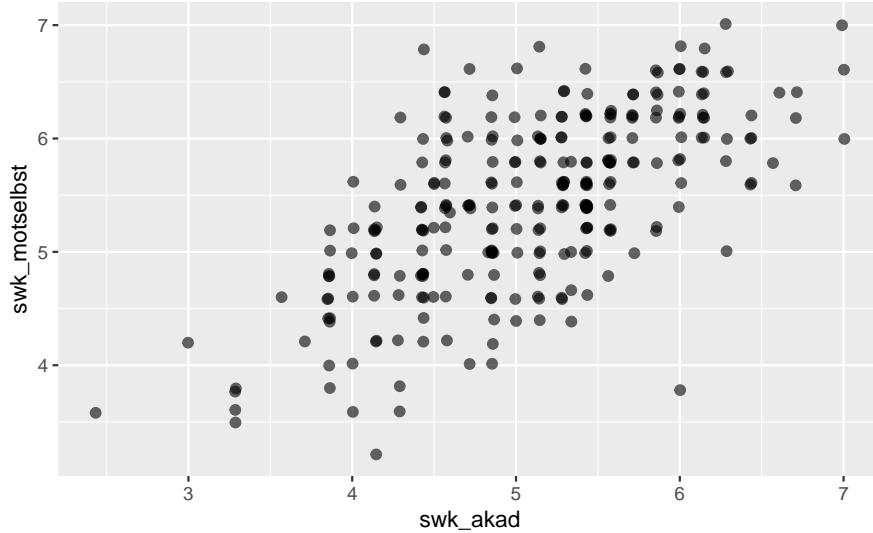
grafisch darstellen. Versuchen Sie zum Beispiel, für jedes Variablenpaar einen Scatterplot (Punktdiagramm) zu erstellen.

```
p <- selbstwirksamkeit_wide %>%
 ggplot()
```

```
p + geom_jitter(aes(x = swk_akad, y = swk_durch), alpha = 0.6, size = 2)
```



```
p + geom_jitter(aes(x = swk_akad, y = swk_motselbst), alpha = 0.6, size = 2)
```



# Chapter 6

## Statistische Verteilungen

In der Statistikausbildung haben wir bisher häufig für einen bestimmten Test eine empirische Prüfgrösse berechnet (z.B. einen  $t$ -Wert) und für den Signifikanztest diesen mit einem kritischen Wert verglichen, den wir in einer Tabelle nachgesehen haben. Manchmal war das nicht möglich, weil die benötigte Verteilung mit bestimmten Freiheitsgraden (z.B. bei der  $t$ - oder *Chi-Quadrat*-Verteilung mit mehr als 30 Freiheitsgraden) oder das benötigte Quantil (z.B. aufgrund eines “krummen” as nach Adjustierung für mehrfache Tests) nicht zur Verfügung stand.

Mit R benötigt man keine Tabellen mehr. Das Programm bietet eine sehr einfache und effiziente Möglichkeit der Berechnung von Verteilungskennwerten für eine Reihe von Verteilungen (wir benötigen nur einen kleinen Teil davon).

- Eine Übersicht über alle zur Verfügung stehenden Verteilungen erhalten Sie über `help("Distributions")`. ■

Kritische Werte werden über die Quantilfunktion (auch “inverse kumulative Verteilungsfunktion” genannt) berechnet und  $p$ -Werte über die (kumulative) Verteilungsfunktion. Daneben gibt es die Möglichkeit, die Wahrscheinlichkeitsdichten (bei kontinuierlichen Wahrscheinlichkeitsverteilungen) bzw. die Punktwahrscheinlichkeiten (bei diskreten Wahrscheinlichkeitsverteilungen) zu berechnen. Ausserdem ist es möglich, für jede der betrachteten Funktionen mittels (Pseudo-)Zufallsgenerator eine Stichprobe von Zufallszahlen aus dieser Verteilung zu ziehen. Wir haben diese Funktion bereits für die Generierung normalverteilter Beispieldaten in den vorangegangenen Kapiteln benutzt.

### 6.1 Normalverteilung

Wir werden die verschiedenen statistischen Funktionen anhand der Normalverteilung einführen und dann anschliessend weitere Verteilungen betrachten. Mit `help("Normal")` bekommen wir eine Übersicht der statistischen Funktionen für die Normalverteilung:

```
The Normal Distribution

Description
Density, distribution function, quantile function and random generation for
the normal distribution with mean equal to mean and standard deviation equal to sd.

Usage
dnorm(x, mean = 0, sd = 1, # Dichtefunktion
 log = FALSE)
```

```

pnorm(q, mean = 0, sd = 1, # Verteilungsfunktion (für p-Werte)
 lower.tail = TRUE,
 log.p = FALSE)
qnorm(p, mean = 0, sd = 1, # Quantilfunktion (für kritische Werte)
 lower.tail = TRUE,
 log.p = FALSE)
rnorm(n, mean = 0, sd = 1) # (Pseudo-)Zufallsgenerator normalverteilter Werte

Arguments
x, q vector of quantiles. # Wert(e) der Normalverteilung, für den die Dichte (dnorm)
 # bzw. die Verteilungsfunktion (pnorm) berechnet werden soll.
p vector of probabilities. # Kumulative Wahrscheinlichkeit(en), für die das Quantil der
 # Normalverteilung berechnet werden soll
n number of observations. # Anzahl der Zufallszahlen, die generiert werden sollen
mean vector of means. # Mittelwert der Normalverteilung, aus der gezogen wird (auch
 # ein Vektor von Mittelwerten ist möglich, dann wird der Reihe
 # nach aus verschiedenen Normalverteilungen gezogen)
sd vector of SDs. # Standardabweichung der Normalverteilung, aus der gezogen wird
 # (normalerweise nur eine SD, aber siehe Mittelwerte)
log, log.p logical; # Es ist auch möglich, die Dichten, Quantile und Verteilungs-
 # funktionen für logarithmierte Normalverteilungen zu berechnen.
 # if TRUE, probabilities p
 # are given as log(p).
lower.tail logical; # wenn FALSE: 1) bei pnorm() wird statt der Verteilungsfunktion
 # P[X <= x] die komplementäre Funktion P[X > x] berechnet;
 # 2) bei der Berechnung von Quantilen über qnorm() wird das
 # Argument p als 1-p interpretiert

```

## Verteilungsfunktion

Die Verteilungsfunktion `pnorm()` benötigen wir hauptsächlich zur Berechnung von  $p$ -Werten. Im Englischen wird die Verteilungsfunktion als **cumulative distribution function (CDF)** bezeichnet. Der einfachste Fall betrifft den linksseitigen (einseitigen)  $p$ -Wert für einen bestimmten standardnormalverteilten ( $z$ -verteilten) empirischen  $z$ -Wert. Dann entspricht der  $p$ -Wert direkt der Verteilungsfunktion der Standardnormalverteilung.

- Wenn wir die Argumente `mean` und `sd` nicht angeben, werden die default-Werte `mean = 0` und `sd = 1` benutzt und wir erhalten die **Standardnormalverteilung**. ■

Die Funktion `pnorm()` ersetzt die Tabelle für die Standardnormalverteilung, wenn wir für einen bestimmten  $z$ -Wert die Fläche links von diesem Wert benötigen.

```

pnorm(q = -1.73)
#> [1] 0.04181514

meist lässt man das q weg, denn R interpretiert das erste Argument auch ohne
Spezifikation als q-Quantil

pnorm(-1.73)
#> [1] 0.04181514

```

Der  $p$ -Wert eines einseitigen *linksseitigen* Signifikanztests für  $z = -1,73$  wäre also  $p = 0,042$  und damit bei  $\alpha = 0,05$  signifikant.

Falls es sich um einen einseitigen *rechtsseitigen* Signifikanztest handelt, benötigen wir die Fläche der Standard-

normalverteilung *rechts* von einem bestimmten empirischen *z*-Wert. Diese erhalten wir mit dem Argument `lower.tail = FALSE`.

```
pnorm(1.645)
#> [1] 0.9500151
pnorm(1.645, lower.tail = FALSE)
#> [1] 0.04998491
```

Wie Sie wissen wird der Wert 1,645 der Standardnormalverteilung in Verteilungstabellen oft als kritischer Wert für den einseitigen rechtsseitigen Signifikanztest angegeben. Links von diesem Wert sollten also 95% der Verteilung liegen und rechts davon 5% der Verteilung. Hier sehen wir, dass das nicht ganz exakt ist.

Falls es sich um einen zweiseitigen Signifikanztest handelt, müssen wir den einseitigen *p*-Wert verdoppeln:

```
#2-seitiger p-Wert bei negativem z-Wert
2*pnorm(-1.73)
#> [1] 0.08363028

#2-seitiger p-Wert bei positivem z-Wert
2*pnorm(1.73, lower.tail = FALSE) #oder
#> [1] 0.08363028
2*(1-pnorm(1.73))
#> [1] 0.08363028
```

Ein zweiseitiger Test wäre also für  $z = -1,73$  (und  $z = +1,73$ ) nicht mehr signifikant.

Wir können `pnorm()` aber auch benutzen, um die Perzentile einer Reihe normalverteilter Werte zu erhalten. Zum Beispiel möchten wir wissen, welchen Prozenträngen eine Reihe von Intelligenztestwerten entsprechen:

```
IQ_Werte <- c(74, 87, 100, 110, 128, 142)
pnorm(q = IQ_Werte, mean = 100, sd = 15)
#> [1] 0.04151822 0.19306234 0.50000000 0.74750746 0.96902592 0.99744487
```

Die IQ-Werte 74, 87, 100, 110, 128 und 142 entsprechen also den Perzentilen 4,2%, 19,3%, 50,0%, 74,8%, 96,9% und 99,7%.

## Quantilfunktion

Die Quantilfunktion `qnorm()` ist das Komplement zur Verteilungsfunktion. Im Englischen wird sie als **inverse cumulative distribution function (ICDF)** bezeichnet. Mit `qnorm()` erhalten wir einen *z*-Wert (d.h. ein Quantil der Standardnormalverteilung) für eine gegebene Fläche *p*, die das erste Argument der Funktion darstellt. Die Funktion `qnorm()` (mit den Default-Argumenten) ersetzt die Tabelle für die Standardnormalverteilung, wenn wir für eine bestimmte Fläche (Verteilungsfunktion) der Standardnormalverteilung das dazugehörige *z*-Quantil ermitteln wollen. In der Praxis benötigen wir diese Funktion zur Berechnung von kritischen Werten für ein bestimmtes  $\alpha$ -Niveau bei ein- und zweiseitigen Signifikanztests.

Hier die am häufigsten benutzen *z*-Quantile für  $\alpha = 0,01$  und  $\alpha = 0,05$  für ein- und zweiseitige Tests:

```
Alpha = 0,01
qnorm(p = 0.005) # zweiseitiger Test, unterer kritischer Wert
#> [1] -2.575829
qnorm(p = 0.995) # zweiseitiger Test, oberer kritischer Wert
#> [1] 2.575829
qnorm(p = 0.01) # einseitiger linksseitiger kritischer Wert
#> [1] -2.326348
qnorm(p = 0.99) # einseitiger rechtsseitiger kritischer Wert
#> [1] 2.326348
```

```
Alpha = 0,05
qnorm(p = 0.025) # zweiseitiger Test, unterer kritischer Wert
#> [1] -1.959964
qnorm(p = 0.975) # zweiseitiger Test, oberer kritischer Wert
#> [1] 1.959964
qnorm(p = 0.05) # einseitiger linksseitiger kritischer Wert
#> [1] -1.644854
qnorm(p = 0.95) # einseitiger rechtsseitiger kritischer Wert
#> [1] 1.644854
```

Diese  $z$ -Werte sind Ihnen in ihrer gerundeten Form, also als 1,645, 1,96, 2,33 oder 2,58 bereits häufiger begegnet.

- Alternativ können wir die oberen bzw. rechtsseitigen Werte auch mit dem unteren Quantil und dem Argument `lower.tail = FALSE` berechnen: `qnorm (p = 0.975)` ist also äquivalent zu `qnorm (p = 0.025, lower.tail = FALSE)`

Von der Berechnung kritischer Werte einmal abgesehen können wir auch beliebige Quantile z.B. der IQ-Verteilung berechnen:

```
Dezile der IQ-Verteilung
dezile <- seq(0, 1, by = 0.1)
dezile
#> [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
qnorm(p = dezile, mean = 100, sd = 15)
#> [1] -Inf 80.77673 87.37568 92.13399 96.19979 100.00000 103.80021
#> [8] 107.86601 112.62432 119.22327 Inf
```

Mit einem IQ von knapp 120 gehört man also zu den intelligentesten 10% einer Population.

## Dichtefunktion

Die Dichtefunktion (**probability density function, PDF**) eines bestimmten Wertes einer normalverteilten Variablen wird relativ selten benötigt. Zur Erinnerung: die Wahrscheinlichkeit einzelner Werte einer stetigen Zufallsvariablen ist gleich 0, nur **Intervalle** von Werten haben eine Wahrscheinlichkeit grösser als 0 (diese entspricht dann der **Fläche** unter der Kurve, d. h. dem Integral). Einzelne Werte besitzen aber eine Wahrscheinlichkeits**dichte**, diese entspricht der Höhe der Kurve der Wahrscheinlichkeitsverteilung an der Stelle  $X = x$ . Die Dichte benötigen wir z.B. wenn wir eine Normalverteilung plotten möchten.

Beispiel:

```
dnorm(x = 0)
#> [1] 0.3989423
```

Die Dichtefunktion der Standardnormalverteilung ist  $\phi(z) = \frac{1}{\sqrt{2\cdot\pi\cdot e^{z^2}}}$ . Die Dichte für den Mittelwert der Standardnormalverteilung ( $\mu = 0$ ) erhalten wir daher als  $\phi(z) = \frac{1}{\sqrt{2\cdot\pi\cdot e^{0^2}}} = \frac{1}{\sqrt{2\cdot\pi}}$ :

```
1/(sqrt(2*pi))
#> [1] 0.3989423
```

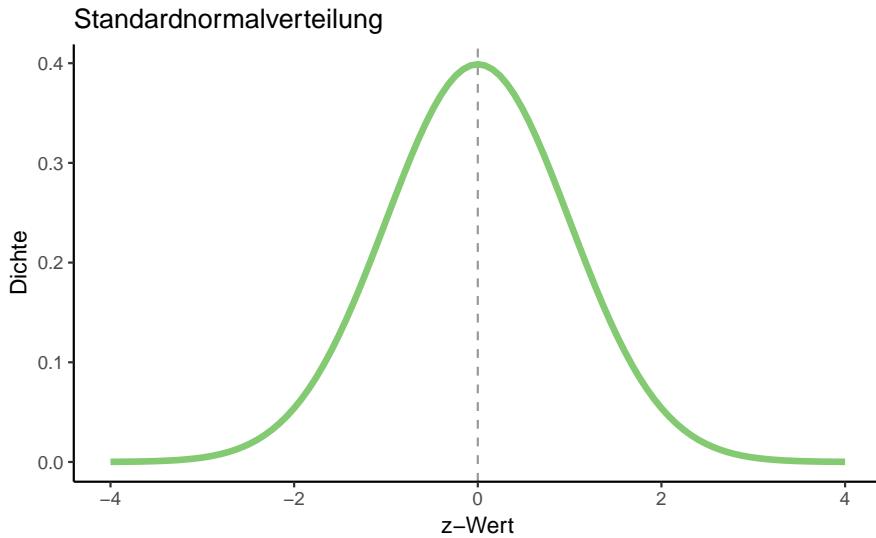
Die Dichte an den Stellen  $+/- 1$  SD ergibt sich dann als:

```
dnorm(1)
#> [1] 0.2419707
1/(sqrt(2*pi*exp(1)))
#> [1] 0.2419707
```

Das ist natürlich Spielerei. Richtig gut gebrauchen können wir `dnorm()`, wenn wir die Normalverteilung plotten möchten:

```
library(tidyverse)
p <- data_frame(x = c(-4,4)) %>%
 ggplot(aes(x = x))

p + geom_vline(xintercept = 0, linetype = "dashed", alpha = 0.4) +
 stat_function(fun = dnorm, color = "#84CA72", size = 1.5) +
 ggtitle("Standardnormalverteilung") +
 xlab("z-Wert") +
 ylab("Dichte") +
 theme_classic()
```

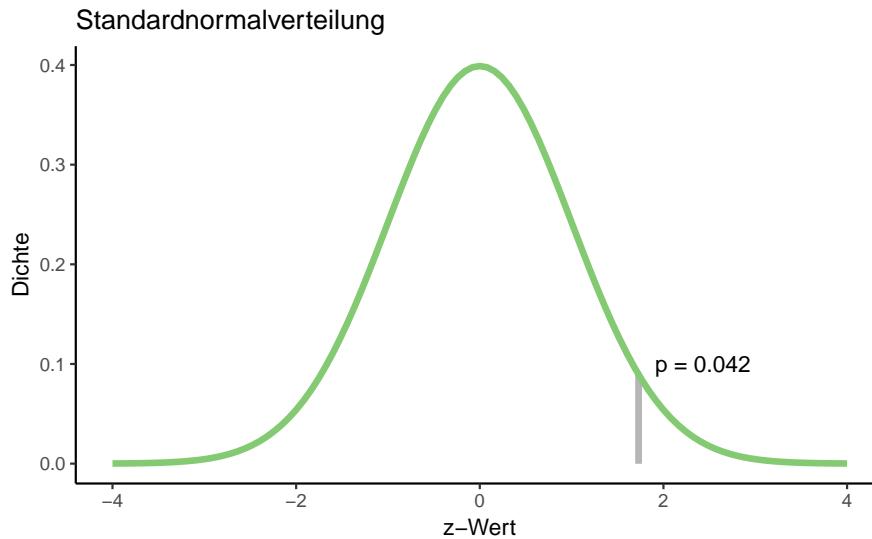


Wir können auch einen bestimmten  $z$ -Wert in das Diagramm einzeichnen und uns für diesen im Anschluss mit `pnorm()` einen  $p$ -Wert berechnen lassen:

```
mein_z <- 1.73

p_wert <- round(pnorm(mein_z, lower.tail = FALSE), 3)

p + geom_segment(aes(x = mein_z,
 y = 0,
 xend = mein_z,
 yend = (dnorm(mein_z))),
 color = "grey70",
 size = 1.5,
 alpha = 0.8) +
 stat_function(fun = dnorm, color = "#84CA72", size = 1.5) +
 ggtitle("Standardnormalverteilung") +
 xlab("z-Wert") +
 ylab("Dichte") +
 annotate("text", label = paste0("p = ", p_wert),
 x = mein_z + 0.7,
 y = 0.1,
 color = "black") +
 theme_classic()
```



```
pnorm(mein_z, lower.tail = FALSE)
#> [1] 0.04181514
```

## Simulation von Daten

Mit `rnorm()` können wir (Pseudo-)Zufallszahlen erzeugen. "Pseudo" deshalb, weil es sich bei der Funktion nicht um einen wahren Zufallsgenerator handelt, sondern um einen Algorithmus, der Zufallsziehungen simuliert. Ein tatsächlicher Zufallsgenerator ist etwas sehr kompliziertes und nicht so einfach in einer Software in einem deterministischen Computer implementierbar. Der Vorteil des Pseudo-Zufallsgenerators (z.B. für dieses Skript) ist, dass wir mit Hilfe der Startwert-Funktion `set.seed()` eine reproduzierbare "Zufalls"-Ziehung erhalten können. `set.seed()` lässt den Ziehungsalgorithmus an einer bestimmten Stelle beginnen, so dass sich mit demselben Startwert immer die gleichen Zahlen ergeben. Es spielt keine Rolle, welchen Startwert wir mit `set.seed()` vorgeben.

Generieren wir zur Abwechslung also 22 IQ-Werte:

```
set.seed(534)
r_IQ_Werte <- rnorm(n = 22, mean = 100, sd = 15)
r_IQ_Werte
#> [1] 91.85366 112.20598 93.59749 95.60904 89.71878 93.37631 87.61408
#> [8] 92.83598 93.02380 88.35326 132.49322 87.30990 107.95051 95.45687
#> [15] 92.59292 84.40998 135.42350 67.04528 91.38498 130.92657 81.36571
#> [22] 102.44571
mean(r_IQ_Werte)
#> [1] 97.59062
sd(r_IQ_Werte)
#> [1] 16.91699
```

## 6.2 t-Verteilung

Mit `help("TDist")` bekommen wir eine Übersicht der statistischen Funktionen für die *t*-Verteilung:

```
The Student t Distribution

Description
Density, distribution function, quantile function and random generation for the t
distribution with df degrees of freedom (and optional non-centrality parameter ncp).
```

```
Usage
dt(x, df, ncp # Dichtefunktion
log = FALSE)
pt(q, df, ncp, # Verteilungsfunktion (für p-Werte)
lower.tail = TRUE,
log.p = FALSE)
qt(p, df, ncp, # Quantilfunktion (für kritische Werte)
lower.tail = TRUE,
log.p = FALSE)
rt(n, df, ncp) # (Pseudo-)Zufallsgenerator t-verteilter Werte
```

Die Argumente sind die gleichen wie bei der Normalverteilung mit folgenden Unterschieden: Es müssen die Freiheitsgrade (`df`) der *t*-Verteilung angegeben werden. Die Argumente `mean` und `sd` entfallen, dafür gibt es den Nonzentralitätsparameter `ncp`. Dieser wird (wie der Name schon sagt) für nonzentrale *t*-Verteilungen benötigt, die nicht den Mittelwert 0 haben. Wir haben bisher nur die zentrale *t*-Verteilung (mit  $\mu = 0$ ) benutzt, denn diese entspricht der für die *t*-Tests benötigten H<sub>0</sub>-Verteilung. Nonzentrale *t*-Verteilungen (H<sub>1</sub>-Verteilungen) sind im Gegensatz zur zentralen *t*-Verteilung asymmetrisch und werden für die Poweranalyse benötigt. Wir werden diese Verteilungen hier nicht betrachten und daher das Argument `ncp` auch nicht verwenden.

## Verteilungsfunktion

Die Verteilungsfunktion `pt()` benötigen wir zur Berechnung von *p*-Werten. Sie ersetzt die Tabelle für die zentralen *t*-Verteilungen (mit verschiedenen *dfs*), wenn wir für einen bestimmten *t*-Wert die Fläche links von diesem Wert erhalten wollen.

```
pt(q = -1.73, df = 6)
#> [1] 0.06717745
pt(-1.73, 6)
#> [1] 0.06717745
```

Der *p*-Wert eines einseitigen *linksseitigen* Signifikanztests für  $t = -1,73$  ( $df = 6$ ) wäre also  $p = 0,067$  und damit bei  $\alpha = 0,05$  nicht signifikant.

Falls es sich um einen einseitigen *rechtsseitigen* Signifikanztest handelt, benötigen wir die Fläche der *t*-Verteilung *rechts* von einem bestimmten empirischen *t*-Wert. Diese erhalten wir mit dem Argument `lower.tail = FALSE`.

```
pt(1.73, df = 6)
#> [1] 0.9328225
pt(1.73, df = 6, lower.tail = FALSE)
#> [1] 0.06717745
```

Falls es sich um einen zweiseitigen Signifikanztest handelt, müssen wir den einseitigen *p*-Wert verdoppeln:

```
2-seitiger p-Wert bei negativem t-Wert
2 * pt(-1.73, df = 6)
#> [1] 0.1343549

2-seitiger p-Wert bei positivem t-Wert
2 * pt(1.73, df = 6, lower.tail = FALSE)
#> [1] 0.1343549
```

## Quantilfunktion

Mit `qt()` erhalten wir einen  $t$ -Wert für eine gegebene Fläche  $p$ . In den meisten  $t$ -Tabellen sind nur bestimmte  $t$ -Quantile tabelliert. Im Lehrbuch von Eid, Gollwitzer & Schmitt (2015) z.B. die  $t$ -Werte für die Verteilungsfunktionen  $p$  von 0,6, 0,8, 0,9, 0,95, 0,975, 0,99 und 0,995. Diese können wir uns z.B. für die  $t$ -Verteilung mit  $df = 6$  ausgeben lassen:

```
meine_p <- c(0.6, 0.8, 0.9, 0.95, 0.975, 0.99, 0.995)
qt(meine_p, df = 6)
#> [1] 0.2648345 0.9057033 1.4397557 1.9431803 2.4469119 3.1426684 3.7074280
```

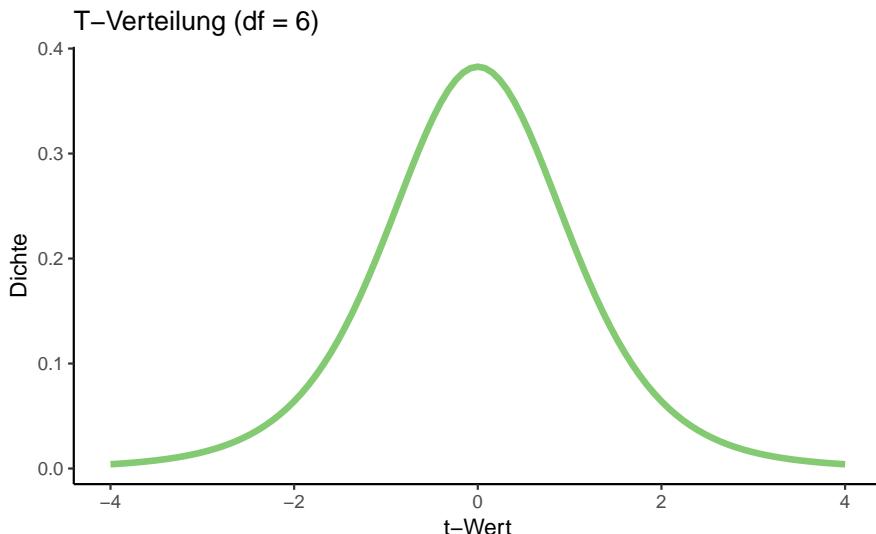
- Vergleichen Sie die Werte mit denjenigen aus der  $t$ -Tabelle! Berechnen Sie weitere Quantile für  $t$ -Verteilungen mit  $df > 30$ , die Sie nicht in der Tabelle finden. ■

## Dichtefunktion

Die Dichtefunktion benötigen wir wieder für den Plot:

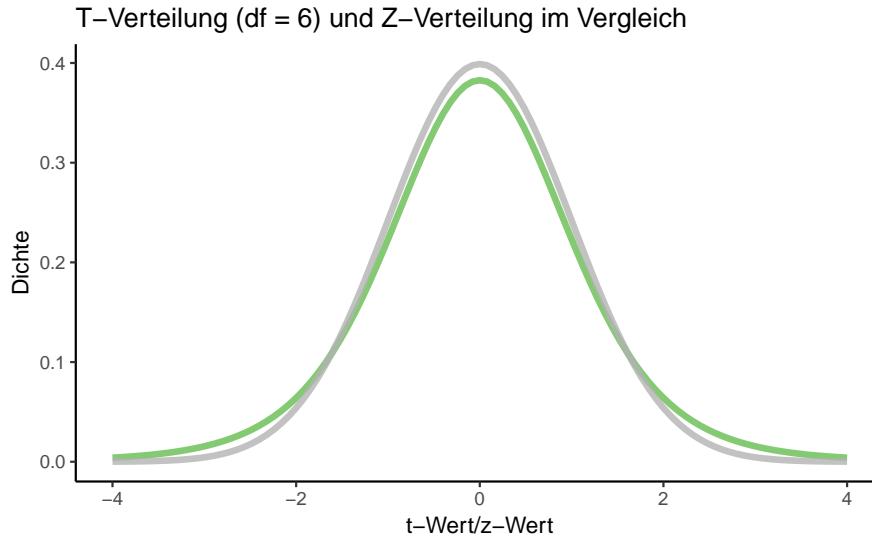
```
library(tidyverse)
p <- data_frame(x = c(-4,4)) %>%
 ggplot(aes(x = x))

p + stat_function(fun = dt, args = list(df = 6), color = "#84CA72", size = 1.5) +
 ggtitle("T-Verteilung (df = 6)") +
 xlab("t-Wert") +
 ylab("Dichte") +
 theme_classic()
```



```
library(tidyverse)
p <- data_frame(x = c(-4,4)) %>%
 ggplot(aes(x = x))

p + stat_function(fun = dt, args = list(df = 6), color = "#84CA72", size = 1.5) +
 stat_function(fun = dnorm, color = "grey70", size = 1.5, alpha = 0.8) +
 ggtitle("T-Verteilung (df = 6) und Z-Verteilung im Vergleich") +
 xlab("t-Wert/z-Wert") +
 ylab("Dichte") +
 theme_classic()
```



### Simulation von Daten

Mit `rt` lassen sich  $t$ -Verteilte Zufallszahlen generieren. Die Funktionalität ist genau gleich wie bei der Normalverteilung. Da die Simulation  $t$ -verteilter Werte selten benötigt wird, werden wir hier nicht näher darauf eingehen.

## 6.3 Chi-Quadrat-Verteilung

Für die Chi-Quadrat-Verteilung (`help("Chisquare")`) ist die Funktionalität genau gleich wie für die  $t$ -Verteilung. Auch hier müssen die jeweiligen Freiheitsgrade angegeben werden und auch hier existieren nonzentrale Verteilungen, die wir aber nicht weiter betrachten. Die Funktionen heissen demnach `pchisq` (Verteilungsfunktion), `qchisq` (Quantilfunktion), `dchisq` (Dichtefunktion) und `rchisq` (Zufallsgenerator).

### Verteilungsfunktion

Bei der Berechnung von  $p$ -Werten für die Chi-Quadrat-Verteilung wird in den meisten Fällen nur das obere (rechte) Ende der Verteilung benötigt, da die meisten Tests so konstruiert sind, dass alle Abweichungen von  $H_0$  (egal in welche Richtung) zu einem grösseren Chi-Quadrat-Wert führen. Wir verwenden daher für die Berechnung eines  $p$ -Wertes immer das Argument `lower.tail = FALSE`.

- Eine Ausnahme stellt der Varianztest für *eine* Stichprobe dar, bei dem auch das untere (linke) Ende der Chi-Quadrat-Verteilung zur Anwendung kommt. ■

Beispielberechnung eines  $p$ -Wertes für einen empirischen Chi-Quadrat-Wert von 4,56 ( $df = 2$ ):

```
pchisq(q = 4.86, df = 2, lower.tail = FALSE)
#> [1] 0.08803683
```

### Quantilfunktion

Mit `qchisq()` erhalten wir die Quantile der Chi-Quadrat-Verteilung, also z.B. die kritischen Werte für  $\alpha = 0,05$ . Statistiker sind sehr stolz, wenn Sie die kritischen Werte für die Chi-Quadrat-Verteilungen mit 1, 2, 3, 4 und 5 Freiheitsgraden auswendig können, da diese sehr häufig für Modellvergleiche (Likelihood-Ratio-Tests) benötigt werden (vgl. z.B. Statistik III: Logistische Regressionsanalyse).

```
meine_dfs <- 1:5
qchisq(0.95, df = meine_dfs)
#> [1] 3.841459 5.991465 7.814728 9.487729 11.070498
```

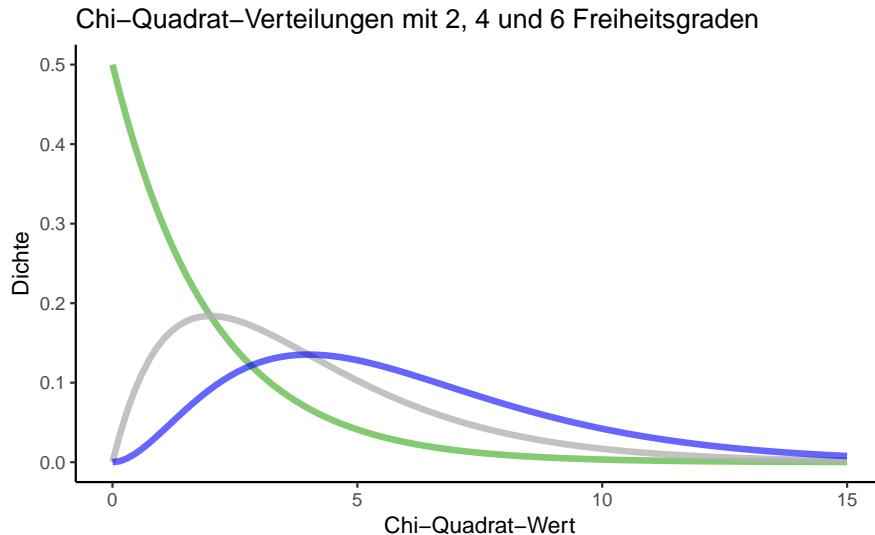
- Vergleichen Sie die Werte mit denjenigen aus der Chi-Quadrat-Tabelle und berechnen Sie andere - nicht tabellierte - Quantile sowie Quantile für Chi-Quadrat-Verteilungen mit  $df > 30$  (nicht tabelliert). ■

## Dichtefunktion

Die Dichtefunktion `dchisq()` benötigen wir auch hier wieder für den Plot:

```
library(tidyverse)
p <- data_frame(x = c(0, 15)) %>%
 ggplot(aes(x = x))

p + stat_function(fun = dchisq, args = list(df = 2), color = "#84CA72", size = 1.5) +
 stat_function(fun = dchisq, args = list(df = 4), color = "grey70", size = 1.5, alpha = 0.8) +
 stat_function(fun = dchisq, args = list(df = 6), color = "blue", size = 1.5, alpha = 0.6) +
 ggtitle("Chi-Quadrat-Verteilungen mit 2, 4 und 6 Freiheitsgraden") +
 xlab("Chi-Quadrat-Wert") +
 ylab("Dichte") +
 theme_classic()
```



## Simulation von Daten

Mit `rchisq()` lassen sich Chi-Quadrat-verteilte Zufallszahlen generieren. Dies wird aber selten benötigt und daher hier nicht behandelt.

## 6.4 F-Verteilung

Auch für die statistischen Funktionen der *F*-Verteilung (`help("FDist")`) gelten die gleichen Regeln wie für die Chi-Quadrat- und die *t*-Verteilung. Dazu kommt, dass hier jeweils zwei Arten von Freiheitsgraden angegeben werden müssen: Zählerfreiheitsgrade ( $df_1$ ) und Nennerfreiheitsgrade ( $df_2$ ). Die Funktionen heißen demnach `pf` (Verteilungsfunktion), `qf` (Quantilfunktion), `df` (Dichtefunktion) und `rf` (Zufallsgenerator).

## Verteilungsfunktion

Bei der Berechnung von  $p$ -Werten für die  $F$ -Verteilung wird in den meisten Fällen genau wie bei der Chi-Quadrat-Verteilung nur das obere (rechte) Ende der Verteilung benötigt. Wir verwenden daher für die Berechnung eines  $p$ -Wertes hier immer das Argument `lower.tail = FALSE`. Die wohl typischste Anwendung für den  $F$ -Test ist die einfaktorielle Varianzanalyse. Dort testen wir mit einem Omnibustest, ob sich signifikante Mittelwertsunterschiede zwischen mehreren Gruppen zeigen. Alle Abweichungen von  $H_0$  führen zu einer grösseren  $QS_{zwischen}$  und damit zu einem höheren empirischen  $F$ -Wert. Damit muss auch der gesamte  $\alpha$ -Fehler am oberen Ende der Verteilung berücksichtigt werden.

- Eine Ausnahme stellt der  $F$ -Test für den Vergleich zweier Varianzen aus zwei unabhängigen Stichproben dar, bei dem (theoretisch) auch das untere (linke) Ende der  $F$ -Verteilung zur Anwendung kommen kann. In der Praxis wird das meist vermieden, indem die grössere der beiden Varianzen in den Zähler genommen wird (vgl. Statistik II). ■

Beispielberechnung eines  $p$ -Wertes für einen empirischen  $F$ -Wert von 3,89 ( $df_1 = 2$ ,  $df_2 = 40$ ):

```
pf(q = 3.89, df1 = 2, df2 = 40, lower.tail = FALSE)
#> [1] 0.02859413
```

Dieser  $F$ -Test wäre also signifikant, da  $p < 0,05$ .

## Quantilfunktion

Mit `qf()` erhalten wir die Quantile der  $F$ -Verteilung, also z.B. die kritischen Werte für  $\alpha = 0,05$ . In den Tabellen sind meistens nur wenige  $p$ -Quantile tabelliert, da bereits für die vielen Kombinationen von Freiheitsgraden sehr viel Platz benötigt wird (bei Eid, Gollwitzer & Schmitt (2015) z.B. nur für die Verteilungsfunktionen  $p$  von 0,9, 0,95, 0,975 und 0,99). Ausserdem fehlen viele “krumme” Freiheitsgradkombinationen ganz in den Tabellen.

Beispiel: Verschiedene kritische Werte (für  $\alpha = 0,10$ ,  $\alpha = 0,05$ ,  $\alpha = 0,025$  und  $\alpha = 0,01$ ) für eine  $F$ -Verteilung mit  $df_1 = 2$ ,  $df_2 = 40$ :

```
meine_p2 <- c(0.9, 0.95, 0.975, 0.99)
qf(meine_p2, df1 = 2, df2 = 40)
#> [1] 2.440369 3.231727 4.050992 5.178508
```

- Vergleichen Sie die Werte mit denjenigen aus der  $F$ -Tabelle! Berechnen Sie außerdem die kritischen Werte für eine  $F$ -Verteilung mit  $df_1 = 16$  und  $df_2 = 45$ . Diese werden Sie nicht in der Tabelle finden! ■

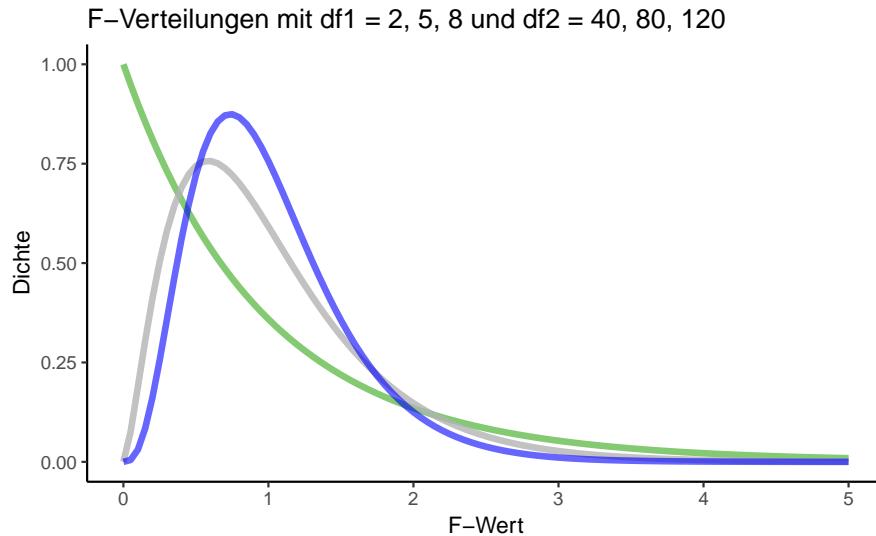
## Dichtefunktion

Die Dichtefunktion `df()` benötigen wir wieder für den Plot:

```
library(tidyverse)
p <- data_frame(x = c(0, 5)) %>%
 ggplot(aes(x = x))

p + stat_function(fun = df, args = list(df1 = 2, df2 = 40),
 color = "#84CA72", size = 1.5) +
 stat_function(fun = df, args = list(df1 = 5, df2 = 80),
 color = "grey70", size = 1.5, alpha = 0.8) +
 stat_function(fun = df, args = list(df1 = 8, df2 = 120),
 color = "blue", size = 1.5, alpha = 0.6) +
 ggttitle("F-Verteilungen mit df1 = 2, 5, 8 und df2 = 40, 80, 120") +
 xlab("F-Wert") +
```

```
ylab("Dichte") +
theme_classic()
```



## Simulation von Daten

Mit `rf()` lassen sich  $F$ -verteilte Zufallszahlen generieren. Dies wird jedoch selten benötigt und daher hier nicht vertieft.

## 6.5 Übungsaufgaben

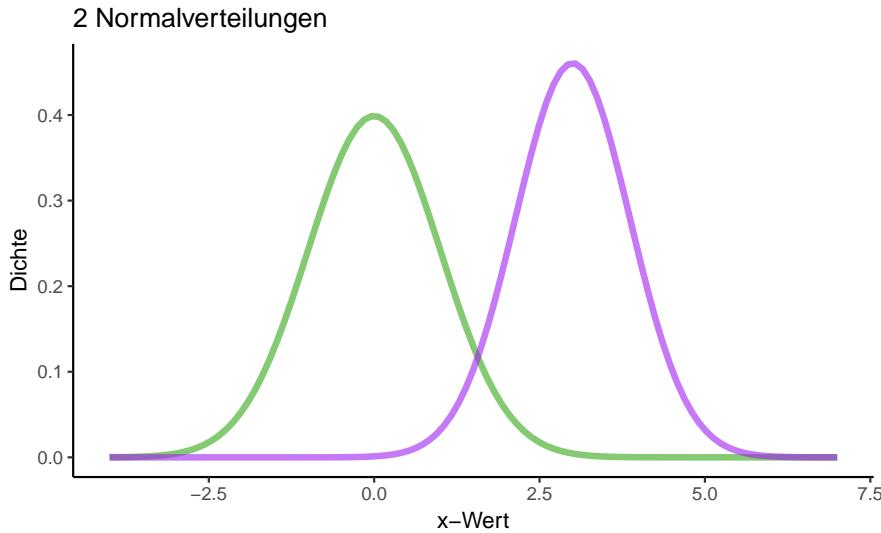
### Dichten und Plots

Berechnen Sie die Dichten der folgenden IQ-Werte: 70, 80, 90, 100, 110, 120, 130

Plotten Sie die beiden folgenden Normalverteilungen in einen gemeinsamen Plot:  $N(\mu = 0, \sigma^2 = 1)$  und  $N(\mu = 3, \sigma^2 = 0.75)$ . Benutzen Sie verschiedene Farben für die beiden Verteilungen. Tipp: Varianz  $\neq$  SD!

```
library(tidyverse)
p <- data_frame(x = c(-4, 7)) %>%
 ggplot(aes(x = x))

p + stat_function(fun = dnorm, color = "#84CA72", size = 1.5) +
 stat_function(fun = dnorm, args = list(3, sqrt(0.75)), color = "purple", size = 1.5, alpha = 0.6) +
 ggtitle("2 Normalverteilungen") +
 xlab("x-Wert") +
 ylab("Dichte") +
 theme_classic()
```



### Verteilungsfunktionen und $p$ -Werte

Berechnen Sie die Verteilungsfunktion für  $z = 1$ .

Welcher Flächenanteil einer beliebigen Normalverteilung liegt im Bereich +/- 1 SD vom MW?

Welcher Prozentsatz der Bevölkerung hat einen IQ zwischen 105 und 115?

Berechnen Sie den einseitigen rechtsseitigen  $p$ -Wert für die empirische Prüfgrösse  $t = 2,045$  ( $df = 8$ ).

Berechnen Sie den zweiseitigen  $p$ -Wert für die empirische Prüfgrösse  $t = 0,73$  ( $df = 14$ ).

Berechnen Sie den zweiseitigen  $p$ -Wert für die empirische Prüfgrösse  $t = -0,73$  ( $df = 14$ ).

Ein Likelihood-Ratio-Test zum Vergleich zweier logistischer Regressionsmodelle (unrestringiertes Modell mit drei Prädiktorvariablen, restringiertes Modell mit einer Prädiktorvariablen) ergibt den Wert der empirischen Prüfgrösse Chi-Quadrat = 5,75. Sind die Effekte der beiden zusätzlichen Prädiktorvariablen im unrestringierten (vollständigen) Modell gemeinsam (als Set) signifikant? Berechnen Sie den  $p$ -Wert.

Sie erhalten einen  $F$ -Wert von 4,36. Ist der Test signifikant ( $df_1 = 2$ ,  $df_2 = 16$ )? Berechnen Sie den  $p$ -Wert.

### Quantile und kritische Werte

Berechnen Sie das 60%-Quantil der IQ-Verteilung. (Welches ist der IQ-Wert, den 60% der Bevölkerung nicht überschreiten?)

Welches ist der IQ-Wert, den 5% der Bevölkerung erreichen oder überschreiten? (IQ-Wert, der die oberen 5% der Verteilung nach unten abschneidet?)

Welches sind die IQ-Werte, zwischen denen die mittleren 82% der Bevölkerung liegen? (Was ist das 82% Zentrale Schwankungsintervall der IQ-Verteilung?)

Berechnen Sie den kritischen  $t$ -Wert ( $df = 28$ ) für den einseitigen rechtsseitigen Test mit  $\alpha = 0.05$ .

Berechnen Sie die kritischen  $t$ -Werte ( $df = 28$ ) für einen zweiseitigen Test mit  $\alpha = 0.05$ .

Berechnen Sie den kritischen  $F$ -Wert einer  $F$ -Verteilung mit  $df_1 = 5$  und  $df_2 = 77$  ( $\alpha = 0.01$ ).



# Chapter 7

## Funktionen

Bisher haben wir uns einige Kenntnisse der R Sprache angeeignet. Wir haben gelernt, R als Taschenrechner zu benutzen, Datensätze zu transformieren, und Grafiken zu erstellen. Etwas wesentliches fehlt jedoch: Wenn wir eine Programmiersprache wie R benutzen, haben wir die Möglichkeit, repetitive Arbeiten zu automatisieren. Dies bedeutet, dass wir versuchen, einen grossen Teil der Arbeit einem Computer zu übergeben. Ein sehr wichtiges Werkzeug dabei ist das Schreiben eigener Funktionen. Dies ist in R erstaunlich einfach. Wir zeigen dies anhand eines Beispiels, in welchem wir zuerst eine Formel in R Code übersetzen, und dann mit dem R Code eine eigene Funktion definieren.

### 7.1 Schiefe (skewness)

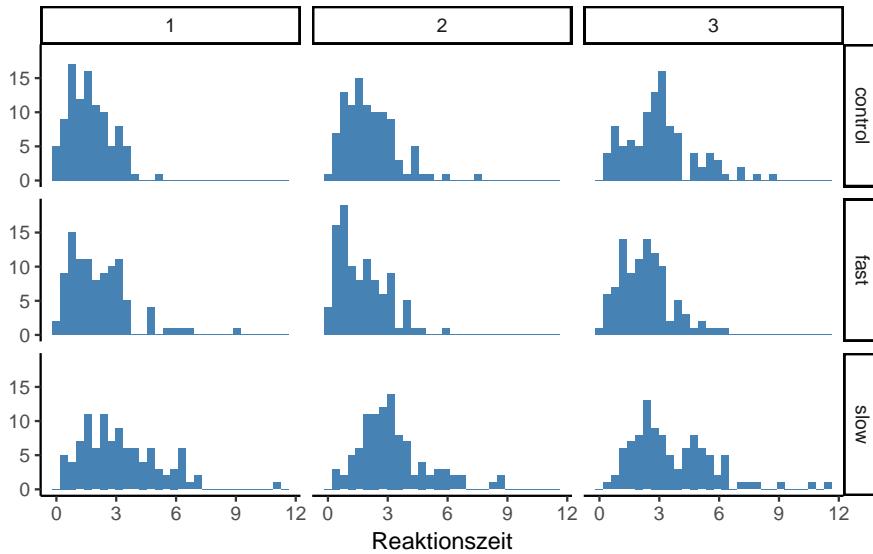
In der Psychologie haben wir es oft mit Reaktionszeiten zu tun. Einen solchen Datensatz haben wir bereits in das *long* Format transformiert und grafisch dargestellt.

```
library(tidyverse)

rt_wide <- read_csv("data/RTdata.csv")
#> Parsed with column specification:
#> cols(
#> ID = col_double(),
#> control = col_double(),
#> slow = col_double(),
#> fast = col_double()
#>)

rt_long <- rt_wide %>%
 gather(key = condition, value = rt,
 control, slow, fast, -ID) %>%
 mutate(ID = factor(ID),
 condition = factor(condition))

rt_long %>%
 ggplot(aes(x = rt)) +
 geom_histogram(bins = 30, fill = "steelblue") +
 facet_grid(condition ~ ID) +
 xlab("Reaktionszeit") +
 ylab("") +
 theme_classic()
```



In dieser Grafik lässt sich gut erkennen, dass Reaktionszeiten oft eine schiefe Verteilung haben. Diese Schiefe (skewness) lässt sich berechnen:

$$\text{Schiefe} = \frac{\sum_{m=1}^n (x_m - \bar{x})^3}{n \cdot s_X^3}$$

$x$  ist ein Vektor von Beobachtungen der Länge  $n$ ,  $\bar{x}$  ist dessen Mittelwert und  $s_X$  die empirische Standardabweichung  $s_X = \sqrt{\frac{1}{n} \sum_{m=1}^n (x_m - \bar{x})^2}$ . Wenn die Verteilung symmetrisch ist, beträgt die Schiefe 0. Für linksschiefe Verteilungen ist die Schiefe negativ und für rechtsschiefe Verteilung wird die Schiefe positiv. Reaktionszeiten sind oft rechtsschief, und deshalb würden wir erwarten, dass wir für solche Daten eine Schiefe  $> 0$  erhalten.

Leider gibt es in R keine Funktion, mit der wir die Schiefe berechnen können, d.h. diese Funktion ist in R nicht standardmäßig implementiert (zumindest nicht in den Basispaketen von R). Dies stellt aber kein Problem dar, denn wir können eine solche Funktion ohne Weiteres selber definieren.

Zuerst wollen wir die Formel für die Schiefe in R Code aufschreiben. Wir stellen fest, dass sich die Formel vereinfachen lässt:

$$\text{Schiefe} = \frac{1}{n} \sum_{m=1}^n z_m^3$$

mit

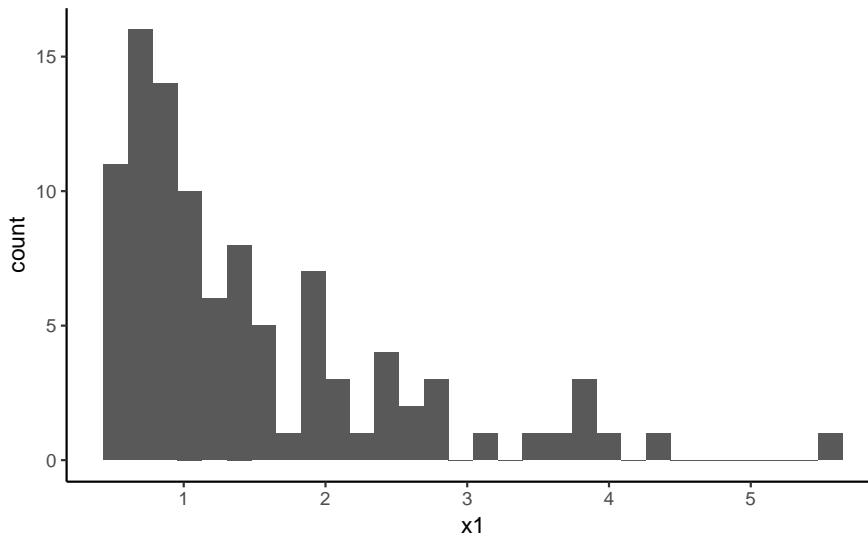
$$z_m = \frac{x_m - \bar{x}}{s_X}$$

Wenn wir die Schiefe eines Vektors berechnen wollen, zentrieren wir diesen zuerst, indem wir den Mittelwert subtrahieren und durch die Standardabweichung teilen. Dann bilden wir die dritte Potenz, summieren alle Elemente und dividieren durch  $n$ .

Um die Berechnung zu vereinfachen, generieren wir einen Vektor von Zufallszahlen einer Gammaverteilung. Diese Verteilung wird oft benutzt, um Reaktionszeiten zu modellieren, da sie nur für positive reelle Zahlen definiert ist und außerdem rechtsschief ist.

```
set.seed(43772)
x1 <- 0.5 + rgamma(100, shape = 1.0, scale = .9)

ggplot(data = NULL, aes(x = x1)) +
 geom_histogram(bins = 30) + theme_classic()
```



Wir brauchen nun die Länge dieses Vektors:

```
n <- length(x1)
```

und den Mittelwert und die Standardabweichung:

```
mean_x1 <- mean(x1)
sd_x1 <- sqrt(sum((x1 - mean_x1)^2) / (n))
```

um die  $z$ -transformierten Werte zu berechnen:

```
z1 <- (x1 - mean_x1) / sd_x1
```

- Warum verwenden wir nicht einfach die `sd()` Funktion? Der Grund dafür ist, dass diese Funktion in R die Stichprobenstandardabweichung berechnet, d.h. durch  $n - 1$  dividiert. Wir wollen jedoch die Schiefe lediglich als Kennzahl einer Stichprobe verwenden und nicht um die Schiefe einer Population zu schätzen, d.h. wir dividieren durch  $n$ . ■

Die Schiefe kann nun berechnet werden:

```
schiefe <- sum(z1^3) / n
schiefe
#> [1] 1.599983
```

Wir erwartet ist dieser Wert  $> 0$ . Dies bedeutet, dass die Verteilung rechtschief ist und nicht symmetrisch.

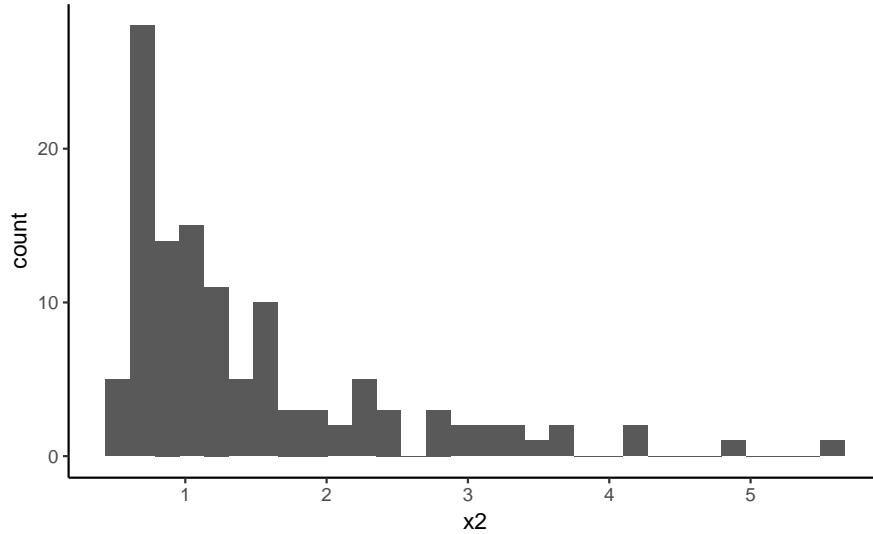
Wir fassen die Schritte nochmals zusammen:

```
n <- length(x1)
mean_x1 <- mean(x1)
sd_x1 <- sqrt(sum((x1 - mean_x1)^2) / (n))
z1 <- (x1 - mean_x1) / sd_x1
schiefe <- sum(z1^3) / n
schiefe
#> [1] 1.599983
```

Bisher haben wir eine mathematische Formel in R Code übersetzt. Dies ist (hoffentlich) schon ein Erfolgserlebnis. Aber was machen wir nun, wenn wir die Schiefe eines zweiten Vektors berechnen wollen?

```
set.seed(43772)
x2 <- 0.6 + rgamma(120, shape = 0.8, scale = .95)

ggplot(data = NULL, aes(x = x2)) +
 geom_histogram(bins = 30) + theme_classic()
```



Eine offensichtliche Lösung bietet sich an: wir kopieren den Code, den wir für `x1` benutzt haben und ersetzen `x1` durch `x2`:

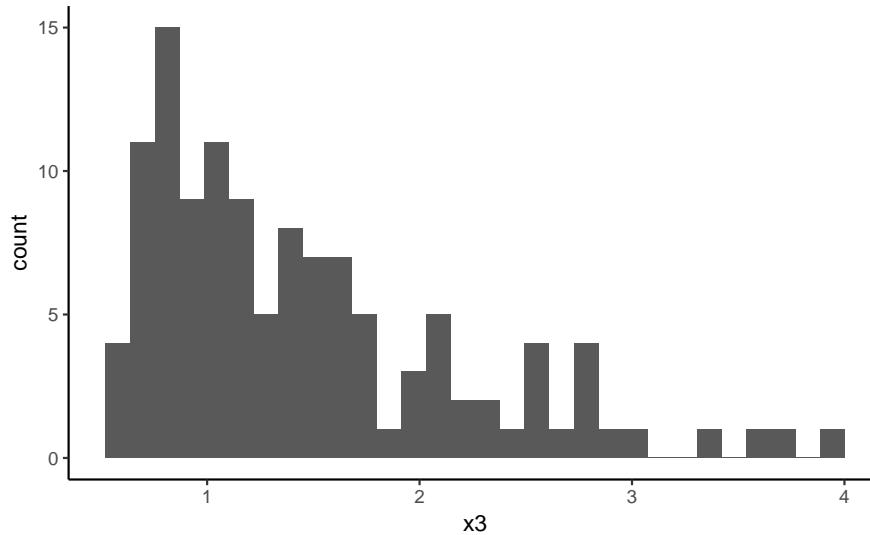
```
n <- length(x2)
mean_x2 <- mean(x2)
sd_x2 <- sqrt(sum((x2 - mean_x2)^2) / (n))
z2 <- (x2 - mean_x2) / sd_x2
schiefe <- sum(z2^3) / n
schiefe
#> [1] 1.748647
```

- Diese Lösung ist nicht zu empfehlen: Code mit copy/paste zu duplizieren ist sehr fehleranfällig. ■

Wir berechnen die Schiefe ein drittes Mal:

```
set.seed(43772)
x3 <- 0.6 + rgamma(120, shape = 1.3, scale = 0.7)

ggplot(data = NULL, aes(x = x3)) +
 geom_histogram(bins = 30) + theme_classic()
```



```
n <- length(x3)
mean_x3 <- mean(x3)
sd_x3 <- sqrt(sum((x2 - mean_x3)^2) / (n))
z3 <- (x3 - mean_x3) / sd_x3
schiefe <- sum(z2^3) / n
schiefe
#> [1] 1.748647
```

■ Im letzten Code Chunk sind zwei Fehler passiert. Finden Sie Sie? ■

Es wäre deutlich besser, wenn wir nur einmal eine Funktion definieren müssen, welche die Schiefe berechnet, und diese dann beliebig oft anwenden können. Abgesehen davon, dass dieses Vorgehen weniger anfällig für Fehler ist, hat dies zusätzlich den Vorteil, dass wir uns nur einmal Gedanken darüber machen, wie wir die Formeln in R Code übersetzen. Danach (wenn wir sicher sind, dass wir keine Fehler gemacht haben) können wir die Funktion einfach verwenden.

## 7.2 Eigene Funktionen definieren

Wir erinnern uns daran, dass eine Funktion einen Namen und Argumente hat:

```
function_name(arg1, arg2 = val2)
```

In diesem Fall hat die Funktion mit Namen `function_name` zwei Argumente, `arg1` und `arg2`. Nur `arg2` hat hier einen Defaultwert. Dies bedeutet, dass das Argument `arg1` unbedingt angegeben werden muss. `arg2` hingegen muss nicht unbedingt angegeben werden. Falls die Funktion mit nur einem Argument aufgerufen wird, also als `function_name(arg1)`, wird für `arg2` der Defaultwert `val2` eingesetzt.

Die Funktion, welche wir nun definieren wollen, braucht einen Namen. Wir nennen sie `skewness`. Als Argument wird diese Funktion einen Vektor erhalten, dessen Schiefe berechnet werden soll. Dieses Argument wollen wir ganz allgemein als `x` bezeichnen. Es macht keinen Sinn, hier einen Defaultwert für das Argument `x` zu wählen.

Die Funktion wird so aufgerufen:

```
skewness(x)
```

Eine Funktion wird in R wie eine Variable mit dem Zuweisungspfeil definiert. Außerdem brauchen wir die Funktion `function()`, welche eine neue Funktion kreiert.

```
skewness <- function(x) {
 # Code
}
```

Mit `function(x)` sagen wir einfach, dass unsere neue Funktion ein Argument namens `x` erhalten wird.

Als letztes brauchen wir nun die geschweiften Klammern `{}`. Zwischen diesen schreiben wir den R Code, den die Funktion ausführen wird. Dies wird `function body` genannt. Wir müssen nur noch wissen, was die Funktion uns zurückgeben soll. Die letzte Befehlszeile im `function body` (die natürlich auf vorherigen Befehlszeilen aufbaut) ist üblicherweise genau das, was als Output bzw. `return value` von der Funktion ausgegeben wird.

■ Im Gegensatz zu manchen anderen Programmiersprachen braucht es hier kein explizites `return` statement. Es genügt, wenn wir in der letzten Zeile eine Funktion aufrufen oder lediglich den Wert einer Variablen ausgeben lassen. Ein häufiger gemachter Fehler ist jedoch, eine Variable mit dem Zuweisungspfeil zu definieren, und zu erwarten, dass die Variable von der Funktion ausgegeben wird, denn das Resultat einer Zuweisung wird nicht als `return value` der Funktion übergeben. Die so definierte Variable muss daher noch einmal hingeschrieben werden (siehe Beispiel). ■

### 7.3 Skewness Funktion

Nun sind wir bereit, unsere `skewness` Funktion zu definieren.

```
skewness <- function(x) {
 # n ist die Anzahl Elemente in x
 n <- length(x)

 # Mittelwert und SD berechnen und speichern
 mean_x <- mean(x)
 sd_x <- sqrt(sum((x - mean_x)^2) / (n))

 # z-transformierte Variable berechnen und speichern
 z <- (x - mean_x) / sd_x

 # Schiefe berechnen (das Resultat einer Zuweisung wird nicht
 # zurückgegeben)
 schiefe <- sum(z^3) / n

 # schiefe wird (implizit) evaluiert. Wir könnten auch explizit
 # print(schiefe) schreiben, oder noch expliziter
 # return(schiefe)
 schiefe
}
```

Wir können die definierte Funktion an die Konsole schicken, indem wir den Cursor irgendwo in der Funktionsdefinition platzieren und aus dem Menu `Code > Run Region > Run Function Definition` auswählen. Es gibt dafür auch eine Tastaturabkürzung, Alt+ CMD + F (MacOS) oder Alt+ Control + F (Windows/Linux).

Sobald wir die Funktionsdefinition an die Konsole geschickt haben, können wir die Funktion ganz gewöhnlich aufrufen. Selber definierte Funktionen werden übrigens im *Environment*-Bereich unter *Functions* angezeigt.

```
skewness(x1)
#> [1] 1.599983
skewness(x2)
#> [1] 1.748647
```

Selbstverständlich kann der `return value` auch einer neuen Variablen zugewiesen werden:

```
skew_x3 <- skewness(x3)
skew_x3
#> [1] 1.241044
```

- Wir haben innerhalb der Funktion `skewness()` verschiedene Variablen als Zwischenschritte definiert, z.B. `n`, `mean_x`, `sd_x` und `z`. Diese existieren aber nur innerhalb der Funktion, sind also “von aussen” nicht sichtbar. Lediglich die Variable, welche in der letzten Zeile evaluiert wird, wird an das *Global Environment* übergeben.

Die Variable `x`, welche als Argument der Funktion übergeben wird, existiert unter diesem Namen auch nur innerhalb der Funktion. Es kann im *Global Environment* ggf. eine Variable geben, welche ebenfalls `x` heisst. Dies ist aber egal, weil R für den Funktionsaufruf ein neues *Environment* definiert. ■

## 7.4 Weitere Beispiele

Wir sehen uns nun weitere einfache Beispiele an, um den Umgang mit Funktionen zu üben.

### 7.4.1 Vervielfachen

Als erstes Beispiel definieren wir eine Funktion, welche eine Zahl, die als erstes Argument übergeben wird, mit dem zweiten Argument multipliziert. Als Defaultwert wählen wir für dieses Argument den Wert 1. Dies bedeutet, dass die Funktion das erste Argument selber wieder retourniert, falls wir für das zweite Argument nichts anderes definieren.

```
vervielfachen <- function(x, k = 1) {
 k * x
}

mit default value für k
wert <- vervielfachen(2)
wert
#> [1] 2

vervielfachen(2, 3)
#> [1] 6
vervielfachen(333, 43)
#> [1] 14319
```

Wir können die Argumente auch benennen:

```
vervielfachen(x = 4, k = 2)
#> [1] 8
```

### 7.4.2 Potenzfunktion

Wir definieren eine Funktion, welche die Potenzfunktion  $a \cdot x^b$  ausführt. Für diese Funktion brauchen wir drei Argumente: `a`, `b` und `x`. Wir entscheiden uns, für `a` und `b` Defaultwerte von 1 zu verwenden.

```
potenz <- function(x, a = 1, b = 1) {
 a * x^b
}

potenz(2)
#> [1] 2
potenz(2, 1, 3)
```

```
#> [1] 8
potenz(x = 2, a = 2, b = 3)
#> [1] 16
```

### 7.4.3 Standardabweichung

Als letztes Beispiel wollen wir unsere eigene Funktion zur Berechnung der empirischen Standardabweichung definieren. Dabei achten wir nun darauf, dass fehlende Werte entfernt werden, bevor wir versuchen, den Mittelwert zu berechnen.

```
standard_abweichung <- function(x) {
 # zuerst NA's entfernen
 x <- na.omit(x)
 # x hat nun sicher keine fehlenden Werte
 sqrt(sum((x - mean(x))^2) / (length(x)))
}

standard_abweichung(x1)
#> [1] 1.003002

x <- rnorm(12)
Das erste Element ist ein fehlender Wert
x[1] <- NA
x
#> [1] NA -0.2933334 -1.1924669 3.4490925 -0.6391923 0.4956852
#> [7] -1.5414534 0.9317232 -2.2546778 -0.4523381 0.5655404 -1.2481360

standard_abweichung(x)
#> [1] 1.478227
```

## 7.5 Übungsbeispiele

### 7.5.1 Skewness-Funktion anpassen

Versuchen Sie, die oben definierte `skewness()` Funktion anzupassen, damit diese auch mit Vektoren umgehen kann, welche fehlende Werte enthalten.

### 7.5.2 Gerade Zahlen

Schreiben Sie eine Funktion, welche aus einem Vektor nur die geraden Zahlen auswählt (wir haben dies in Kapitel 2 schon gemacht).

```
x <- seq(1, 20, by = 1)
x
#> [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Hinweis:
index <- x %% 2 == 0
x[index]
```

### 7.5.3 Schiefe für RT Datensatz berechnen

Berechnen Sie mit `group_by()` und `summarise()` die Schiefe für jede Person in jeder Bedingung in dem `rt_long` Datensatz. Verwenden Sie die selber definierte `skewness()` Funktion.

# Chapter 8

## Deskriptive Statistik

### 8.1 Zusammenfassung von Verteilungskennwerten aller Variablen in der Datenmatrix

Die Funktion `summary(Datenmatrix)` gibt deskriptive Statistiken für alle Variablen in einer Datenmatrix aus. Für numerische Variablen werden Minimum, Maximum, Quartile, Median, und Mittelwert ausgegeben, für Faktoren die Häufigkeiten der Faktorstufen (Levels). Ausserdem für beide Variablentypen die Anzahl der fehlenden Werte.

Wir importieren zunächst den Beispieldatensatz und weisen den Variablen ihr Skalenniveau zu. Danach löschen wir noch alle Einzelitems aus dem Datensatz, damit es etwas übersichtlicher wird. Diesen reduzierten Datensatz, den wir `westost_skalen` nennen, speichern wir dann noch als `.Rdata`-File in unserem Datenordner, um ihn beim nächsten Mal einfach laden und uns die ganze Importarbeit sparen zu können.

```
library(tidyverse)
library(haven)
westost <- read_sav("data/Beispieldatensatz.sav")
westost$ID <- as.factor(westost$ID)
westost$westost <- as_factor(westost$westost)
westost$geschlecht <- as_factor(westost$geschlecht)
westost$SES <- as_factor(westost$SES, ordered = TRUE)
westost$Deutsch <- as_factor(westost$Deutsch, ordered = TRUE)
westost$Mathe <- as_factor(westost$Mathe, ordered = TRUE)
westost$Fremdspr <- as_factor(westost$Fremdspr, ordered = TRUE)
westost$bildung_vater <- as_factor(westost$bildung_vater)
westost$bildung_mutter <- as_factor(westost$bildung_mutter)
westost$bildung_vater_b <- as_factor(westost$bildung_vater_b)
westost$bildung_mutter_b <- as_factor(westost$bildung_mutter_b)
save(westost, file = "data/westost.Rdata")

westost_skalen <- westost %>% select (-c(T1_1 : T7_12))

save(westost_skalen, file = "data/westost_skalen.Rdata")
load(file = "data/westost_skalen.Rdata")
westost_skalen
#> # A tibble: 286 x 33
#> ID westost geschlecht alter SES Deutsch Mathe Fremdspr Schnitt
#> <fct> <fct> <fct> <dbl> <ord> <ord> <ord> <ord> <dbl>
#> 1 2 West männlich 14 unte~ befrie~ befr~ ausreic~ 4
```

```
#> 2 14 West männlich 14 durc~ befrie~ befr~ ausreic~ 4
#> 3 15 West männlich 14 durc~ befrie~ befr~ befried~ 4
#> 4 17 West männlich 15 durc~ befrie~ befr~ ausreic~ 4
#> 5 18 West männlich 14 durc~ gut gut befried~ 4
#> 6 19 West männlich 15 über~ befrie~ befr~ ausreic~ 4
#> # ... with 280 more rows, and 24 more variables: bildung_vater <fct>,
#> # bildung_mutter <fct>, bildung_vater_b <fct>, bildung_mutter_b <fct>,
#> # swk_akad <dbl>, swk_selbstreg <dbl>, swk_durch <dbl>,
#> # swk_motselbst <dbl>, swk_sozharm <dbl>, swk_bez <dbl>,
#> # unt_eltern <dbl>, unt_freunde <dbl>, sch_leistung <dbl>,
#> # sch_bez <dbl>, sch_regeln <dbl>, ges_gerecht <dbl>, ges_nepot <dbl>,
#> # leben_selbst <dbl>, leben_fam <dbl>, leben_schule <dbl>,
#> # leben_freunde <dbl>, leben_gesamt <dbl>, stress_somat <dbl>,
#> # stress_psych <dbl>
```

Jetzt verschaffen uns mit `summary()` einen deskriptiven Überblick über den Datensatz.

```
summary(westost_skalen)
#> ID westost geschlecht alter
#> 1 : 1 West:143 männlich:152 Min. :13.0
#> 2 : 1 Ost :143 weiblich:134 1st Qu.:14.0
#> 10 : 1 Median :15.0
#> 11 : 1 Mean :14.7
#> 12 : 1 3rd Qu.:15.0
#> 14 : 1 Max. :17.0
#> (Other):280
#>
#> SES Deutsch Mathe
#> sehr niedrig : 1 ungenügend : 0 ungenügend : 0
#> unterdurchschnittlich: 4 mangelhaft : 3 mangelhaft : 11
#> durchschnittlich :186 ausreichend :52 ausreichend :55
#> überdurchschnittlich : 69 befriedigend:112 befriedigend:111
#> sehr hoch : 24 gut :100 gut : 88
#> NA's : 2 sehr gut : 16 sehr gut : 19
#> NA's : 3 NA's : 2
#>
#> Fremdspr Schnitt
#> ungenügend : 0 Min. :3.000
#> mangelhaft :10 1st Qu.:4.000
#> ausreichend :39 Median :4.500
#> befriedigend:103 Mean :4.462
#> gut :102 3rd Qu.:5.000
#> sehr gut : 21 Max. :6.000
#> NA's : 11 NA's : 6
#>
#> bildung_vater
#> Hauptschulabschluss oder niedriger :41
#> Realschulabschluss (mittlere Reife) :94
#> Fachabitur, Abitur :41
#> Fachhochschulabschluss, Universitätsabschluss:93
#> NA's :17
#>
#>
#> bildung_mutter bildung_vater_b
#> Hauptschulabschluss oder niedriger :43 niedrig:135
#> Realschulabschluss (mittlere Reife) :95 hoch :134
#> Fachabitur, Abitur :50 NA's : 17
```

## 8.1. ZUSAMMENFASSUNG VON VERTEILUNGSKENNWERTEN ALLER VARIABLEN IN DER DATENMATRIX147

```
#> Fachhochschulabschluss, Universitätsabschluss:84
#> NA's :14
#>
#>
#> bildung_mutter_b swk_akad swk_selbstreg swk_durch
#> niedrig:138 Min. :2.429 Min. :2.875 Min. :1.000
#> hoch :134 1st Qu.:4.571 1st Qu.:4.750 1st Qu.:4.667
#> NA's :14 Median :5.143 Median :5.250 Median :5.333
#> Mean :5.098 Mean :5.246 Mean :5.285
#> 3rd Qu.:5.571 3rd Qu.:5.875 3rd Qu.:6.000
#> Max. :7.000 Max. :7.000 Max. :7.000
#> NA's :1
#>
#> swk_motselbst swk_sozharm swk_bez unt_eltern
#> Min. :3.200 Min. :1.571 Min. :3.333 Min. :1.500
#> 1st Qu.:5.000 1st Qu.:4.857 1st Qu.:5.167 1st Qu.:5.333
#> Median :5.400 Median :5.381 Median :5.667 Median :6.167
#> Mean :5.447 Mean :5.280 Mean :5.616 Mean :5.872
#> 3rd Qu.:6.000 3rd Qu.:5.857 3rd Qu.:6.167 3rd Qu.:6.667
#> Max. :7.000 Max. :7.000 Max. :7.000 Max. :7.000
#> NA's :1
#>
#> unt_freunde sch_leistung sch_bez sch_regeln
#> Min. :2.000 Min. :1.667 Min. :1.000 Min. :1.750
#> 1st Qu.:5.500 1st Qu.:3.667 1st Qu.:3.163 1st Qu.:4.000
#> Median :6.000 Median :4.333 Median :3.750 Median :4.925
#> Mean :5.928 Mean :4.326 Mean :3.820 Mean :4.875
#> 3rd Qu.:6.667 3rd Qu.:5.000 3rd Qu.:4.500 3rd Qu.:5.750
#> Max. :7.000 Max. :7.000 Max. :7.000 Max. :7.000
#> NA's :1
#> NA's :2
#>
#> ges_gerecht ges_nepot leben_selbst leben_fam
#> Min. :1.000 Min. :1.000 Min. :1.000 Min. :1.000
#> 1st Qu.:3.500 1st Qu.:3.600 1st Qu.:5.000 1st Qu.:5.333
#> Median :4.167 Median :4.400 Median :5.500 Median :6.000
#> Mean :4.160 Mean :4.423 Mean :5.433 Mean :5.760
#> 3rd Qu.:5.000 3rd Qu.:5.200 3rd Qu.:6.000 3rd Qu.:6.667
#> Max. :6.667 Max. :7.000 Max. :7.000 Max. :7.000
#> NA's :1
#> NA's :3
#> NA's :1
#> NA's :2
#>
#> leben_schule leben_freunde leben_gesamt stress_somat
#> Min. :1.333 Min. :2.000 Min. :2.909 Min. :1.000
#> 1st Qu.:4.000 1st Qu.:5.500 1st Qu.:5.000 1st Qu.:2.167
#> Median :4.667 Median :6.000 Median :5.455 Median :3.000
#> Mean :4.670 Mean :5.898 Mean :5.417 Mean :3.020
#> 3rd Qu.:5.333 3rd Qu.:6.500 3rd Qu.:5.909 3rd Qu.:3.667
#> Max. :7.000 Max. :7.000 Max. :6.909 Max. :7.000
#> NA's :1
#> NA's :1
#> NA's :1
#> NA's :3
#>
#> stress_psych
#> Min. :1.000
#> 1st Qu.:2.167
#> Median :2.917
#> Mean :2.989
#> 3rd Qu.:3.667
#> Max. :6.833
#> NA's :2
```

## 8.2 Deskriptivstatistik für nominalskalierte Variablen

### 8.2.1 Häufigkeiten mit der Funktion `table`

Mit der Funktion `table` können wir uns die Häufigkeiten der Kategorien (Faktorstufen) eines Faktors ausgeben lassen. Besonders nützlich ist das, wenn die Häufigkeiten kombinierter Faktorstufen mehrerer Faktorstufen von Interesse sind (Kreuztabelle). Wir haben die Funktion schon im Kapitel 5 (“Grafiken”) kennengelernt.

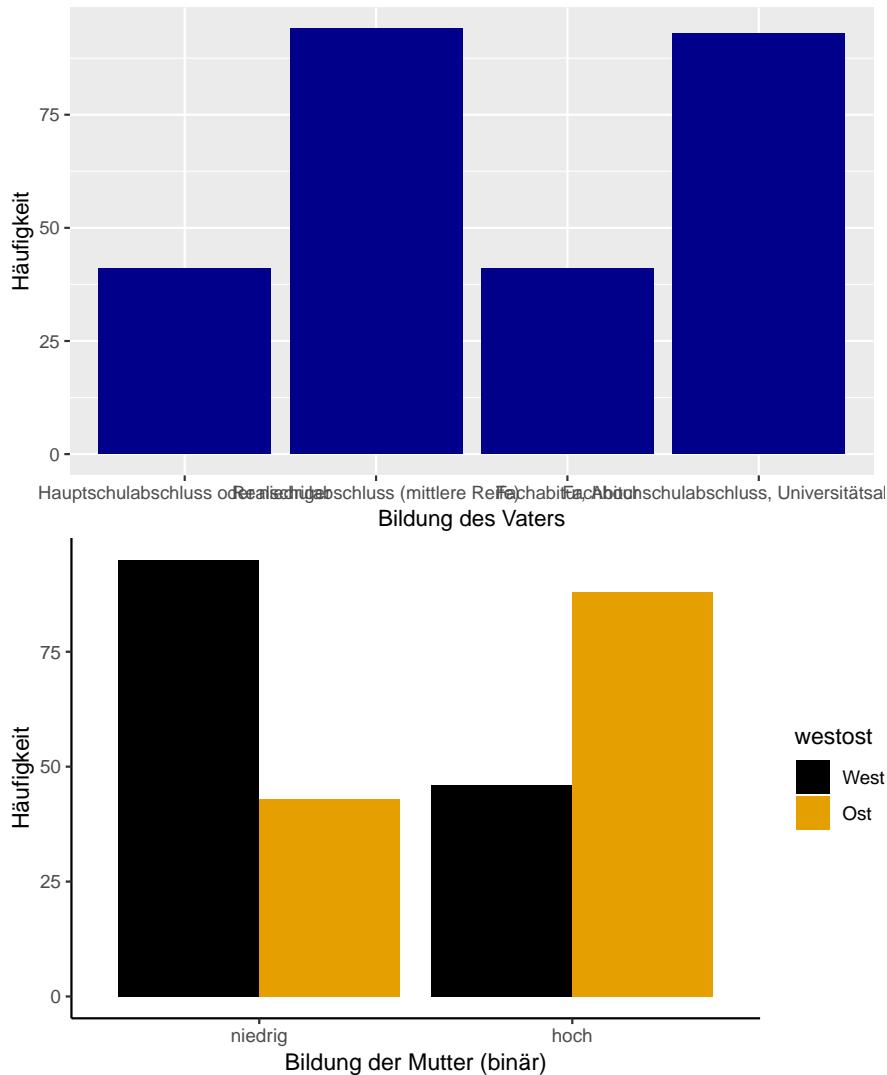
```
table(westost_skalen$bildung_vater)
#>
#> Hauptschulabschluss oder niedriger 41
#> Realschulabschluss (mittlere Reife) 94
#> Fachabitur, Abitur 41
#> Fachhochschulabschluss, Universitätsabschluss 93
#>
table(westost_skalen$bildung_vater_b, westost_skalen$bildung_mutter_b)
#>
#> niedrig hoch
#> niedrig 102 32
#> hoch 35 99
table(westost_skalen$westost, westost_skalen$bildung_mutter)
#>
#> Hauptschulabschluss oder niedriger
#> West 43
#> Ost 0
#>
#> Realschulabschluss (mittlere Reife) Fachabitur, Abitur
#> West 52 20
#> Ost 43 30
#>
#> Fachhochschulabschluss, Universitätsabschluss
#> West 26
#> Ost 58
```

Wir können uns auch nochmal mit `ggplot2` Häufigkeitsdiagramme ansehen:

```
p <- westost_skalen %>%
 select(bildung_vater) %>%
 drop_na() %>%
 ggplot(aes(x = bildung_vater))
p + geom_bar(fill = "darkblue") +
 xlab("Bildung des Vaters") +
 ylab("Häufigkeit")

p1 <- westost_skalen %>%
 select(bildung_mutter_b, westost) %>%
 drop_na() %>%
 ggplot(aes(x = bildung_mutter_b, fill = westost))
p1 + geom_bar(position = "dodge") +
 xlab("Bildung der Mutter (binär)") +
 ylab("Häufigkeit") +
```

```
theme_classic() +
scale_fill_manual(values = c("#000000", "#E69F00"))
```



Mit der Funktion `prop.table()` ist es ausserdem möglich, sich relative Häufigkeiten (Anteile) ausgeben zu lassen:

```
table(westost_skalen$bildung_vater_b, westost_skalen$bildung_mutter_b) %>% prop.table() %>% round(2)
#>
#> niedrig hoch
#> niedrig 0.38 0.12
#> hoch 0.13 0.37

oder in Prozent

100*table(westost_skalen$bildung_vater_b, westost_skalen$bildung_mutter_b) %>% prop.table() %>% round(4)
#>
#> niedrig hoch
#> niedrig 38.06 11.94
#> hoch 13.06 36.94
```

Meistens interessieren uns aber nicht die relativen *Gesamthäufigkeiten* (die sich über die gesamte Kreuztabelle zu 1 bzw. zu 100% aufaddieren), sondern die zeilen- und spaltenbedingten relativen Häufigkeiten (Anteile). Erstere erhalten wir mit `margin = 1` der `prop.table()`-Funktion, letztere mit `margin = 2`. Da es kein weiteres Argument außer `margin` gibt, können wir auch direkt den Wert des Arguments angeben, also `prop.table(1)` für zeilenbedingte Anteile und `prop.table(2)` für spaltenbedingte Anteile.

- `prop.table()` hat als erstes Argument eigentlich eine Kreuztabelle mit Häufigkeiten, wie sie `table()` liefert. Die von uns benutzte Schreibweise mit dem Wert des Arguments `margin` für zeilen-(1) bzw. spalten(2)-bedingte Anteile (oder für gemeinsame Anteile = DEFAULT/NONE) als erstem Argument der Funktion ist nur zusammen mit dem `%>%`-Operator möglich. ■

```
table(westost_skalen$geschlecht, westost_skalen$westost)
#>
#> West Ost
#> männlich 83 69
#> weiblich 60 74

Relative Häufigkeiten bezogen auf die gesamte Tabelle
table(westost_skalen$geschlecht, westost_skalen$westost) %>% prop.table() %>% round(2)
#>
#> West Ost
#> männlich 0.29 0.24
#> weiblich 0.21 0.26

Zeilenbedingte relative Häufigkeiten
table(westost_skalen$geschlecht, westost_skalen$westost) %>% prop.table(1) %>% round(2)
#>
#> West Ost
#> männlich 0.55 0.45
#> weiblich 0.45 0.55

Spaltenbedingte relative Häufigkeiten
table(westost_skalen$geschlecht, westost_skalen$westost) %>% prop.table(2) %>% round(2)
#>
#> West Ost
#> männlich 0.58 0.48
#> weiblich 0.42 0.52
```

### 8.2.2 Modus

Für die Berechnung des Modus einer nominalskalierten Variablen gibt es keine eigene Funktion. Entweder man liest den Modus (die Kategorie mit der grössten Häufigkeit) aus der Häufigkeitstabelle (oder dem Häufigkeitsdiagramm) ab, oder man hilft sich mit:

```
names(which.max(table(westost_skalen$bildung_vater)))
#> [1] "Realschulabschluss (mittlere Reife)"

oder

table(westost_skalen$bildung_vater) %>% which.max() %>% names()
```

Wir können das ganze auch in eine Funktion einbetten:

```
mein_modus <- function(x) {
 names(which.max(table(x)))
}

mein_modus(westost_skalen$bildung_vater)
#> [1] "Realschulabschluss (mittlere Reife)"
```

### 8.2.3 Relativer Informationsgehalt $H$

Der Relative Informationsgehalt (auch relative Entropie genannt) ist ein Dispersionsmass für nominalskalierte Variablen, kann aber auch bei höherem Skalenniveau berechnet werden. Wenn alle Kategorien gleich häufig vorkommen, ist  $H$  maximal (= 1). Kommt nur eine Kategorie vor (d.h. die Häufigkeiten für alle anderen Kategorien sind 0), ist  $H$  minimal (= 0).

$$H = -\frac{1}{\ln(k)} \cdot \sum_{j=1}^k h_j \cdot \ln(h_j)$$

Leider gibt es für den relativen Informationsgehalt keine in R implementierte Funktion (unseres Wissens auch nicht in Packages, aber s.u.). Da wir jetzt eigene Funktionen programmieren können, versuchen wir, die obige Formel in eine Funktion umzusetzen.

```
mein_H <- function(x) {
 k <- length(levels(x)) # Anzahl Kategorien k des Faktors
 const <- 1/log(k) # Konstante 1/ln(k)
 freqs <- table(x) # Häufigkeiten der Kategorien des Faktors
 props <- freqs/sum(freqs) # Relative Häufigkeiten h
 entrop <- -sum(props*log(props)) # Entropie: [Summe über alle h*ln(h)] * (-1)
 H <- const * entrop # Zuweisung H = Relativer Informationsgehalt
 H # Function call
}

table(westost_skalen$bildung_vater)
#>
#> Hauptschulabschluss oder niedriger 41
#> Realschulabschluss (mittlere Reife) 94
#> Fachabitur, Abitur 41
#> Fachhochschulabschluss, Universitätsabschluss 93
mein_H(westost_skalen$bildung_vater)
#> [1] 0.943552

table(westost_skalen$bildung_mutter)
#>
#> Hauptschulabschluss oder niedriger 43
#> Realschulabschluss (mittlere Reife) 95
#> Fachabitur, Abitur 50
#> Fachhochschulabschluss, Universitätsabschluss 84
```

```

mein_H(westost_skalen$bildung_mutter)
#> [1] 0.961722

table(westost_skalen$SES)
#>
#> sehr niedrig unterdurchschnittlich durchschnittlich
#> 1 4 186
#> überdurchschnittlich sehr hoch
#> 69 24
mein_H(westost_skalen$SES)
#> [1] 0.5652135

table(westost_skalen$Deutsch)
#>
#> ungenügend mangelhaft ausreichend befriedigend gut
#> 0 3 52 112 100
#> sehr gut
#> 16
mein_H(westost_skalen$Deutsch)
#> [1] NaN

table(westost_skalen$Mathe)
#>
#> ungenügend mangelhaft ausreichend befriedigend gut
#> 0 11 55 111 88
#> sehr gut
#> 19
mein_H(westost_skalen$Mathe)
#> [1] NaN

table(westost_skalen$Fremdspr)
#>
#> ungenügend mangelhaft ausreichend befriedigend gut
#> 0 10 39 103 102
#> sehr gut
#> 21
mein_H(westost_skalen$Fremdspr)
#> [1] NaN

table(westost_skalen$westost)
#>
#> West Ost
#> 143 143
mein_H(westost_skalen$westost)
#> [1] 1

```

■ Offensichtlich ist unsere Funktion noch nicht ganz perfekt. Sie gibt als Ergebnis `NaN` (Not a Number), sobald mindestens eine der Kategorien unbesetzt ist (`freqs = 0`). Dann ergibt sich nämlich  $\log(0) = -\text{Inf}$ , was noch kein Problem ist, aber  $0 * \log(0)$  geht leider nicht (=`NaN`). Wir müssten die Funktion also erweitern und eine Bedingung einfügen, die dafür sorgt, dass in diesem Fall  $0 * \log(0)$  auf den Wert 0 gesetzt wird. ■

Es gibt ein Packagen namens `entropy`, dass uns nicht die *relative* Entropie, sondern die unstandardisierte En-

tropie gibt. Auch bei diesem Mass gilt, dass es umso grösser wird, je gleichmässiger verteilt die Häufigkeiten auf die verschiedenen Kategorien sind. Seine Ausprägung ist aber auch von der Anzahl der Kategorien abhängig, so dass es nicht unabhängig von der Kategorienzahl interpretiert werden kann.

$$\text{Entropie} = - \sum_{j=1}^k h_j \cdot \ln(h_j)$$

```
install.packages("entropy")
```

Als Input benötigt die Funktion `entropy()` aus diesem Package einen Vektor mit Häufigkeiten, wir müssen also die Häufigkeiten unserer Kategorien mit `table()` vorbereiten.

```
library(entropy)
freqs <- table(westost_skalen$bildung_vater)
entropy(freqs)
#> [1] 1.308041
```

Nun können wir die `entropy()`-Funktion auch in unsere vorhin programmierte Funktion einbauen und somit den relativen Informationsgehalt  $H$  berechnen:

```
mein_H2 <- function(x) {
 k <- length(levels(x)) # Anzahl Kategorien k des Faktors
 const <- 1/log(k) # Konstante 1/ln(k)
 freqs <- table(x) # Häufigkeiten der Kategorien des Faktors
 entrop <- entropy::entropy(freqs) # Entropie mit der Funktion entropy()
 # aus dem package entropy
 H <- const * entrop # Zuweisung H = Relativer Informationsgehalt
 H # Function call
}
```

```
table(westost_skalen$bildung_vater)
#>
#> Hauptschulabschluss oder niedriger
#> 41
#> Realschulabschluss (mittlere Reife)
#> 94
#> Fachabitur, Abitur
#> 41
#> Fachhochschulabschluss, Universitätsabschluss
#> 93
mein_H2(westost_skalen$bildung_vater)
#> [1] 0.943552
```

```
table(westost_skalen$bildung_mutter)
#>
#> Hauptschulabschluss oder niedriger
#> 43
#> Realschulabschluss (mittlere Reife)
#> 95
#> Fachabitur, Abitur
#> 50
#> Fachhochschulabschluss, Universitätsabschluss
#> 84
mein_H2(westost_skalen$bildung_mutter)
#> [1] 0.961722
```

```



```

## 8.3 Deskriptivstatistik für ordinale und metrische Variablen

Wir betrachten die Deskriptivstatistik für ordinale und metrische Variablen hier gemeinsam, da sie sich die meisten Funktionen teilen. Auf ordinale Daten sind nur Quantile (z.B. Median, Quartile) und darauf aufbauende Masse anwendbar, auf metrische Daten auch Masse wie arithmetisches Mittel und Varianz, die mindestens Intervallskalenniveau voraussetzen.

Die meisten Masse haben wir bereits in Kapitel 2.1.5 als “Statistische Funktionen” kennengelernt.

```
mean(x, na.rm = FALSE) Mittelwert
```

<code>sd(x)</code>	(Stichproben-)Standardabweichung
<code>var(x)</code>	(Stichproben-)Varianz
<code>median(x)</code>	Median
<code>quantile(x, probs, type)</code>	Quantile von x. probs: Vektor mit Wahrscheinlichkeiten
<code>min(x)</code>	Minimalwert x_min
<code>max(x)</code>	Maximalwert x_max
<code>range(x)</code>	x_min und x_max

Beispiele und Vertiefung:

Wir bilden zunächst einen Vektor mit 16 Zahlen zwischen 1 und 9 und sortieren ihn anschliessend.

```
x <- c(4, 7, 9, 2, 3, 6, 6, 4, 1, 2, 8, 5, 5, 7, 3, 2)
n <- length(x)
x <- sort(x)
x
#> [1] 1 2 2 2 3 3 4 4 5 5 6 6 7 7 8 9
```

```
median(x)
#> [1] 4.5
quantile(x, c(.25, .75))
#> 25% 75%
#> 2.75 6.25
```

Der Median = 4.5 entspricht unseren Erwartungen, da es sich dabei um das Mittel des 8. und 9. Messwertes (sortierte Werte) handelt. So hatten wir den Median auch in Statistik I definiert, sofern  $n$  eine gerade Zahl ist (bei  $n$  ungerade ist der Median der Messwert, der auf die Stelle  $(\frac{n+1}{2})$  der Verteilung folgt). Ganz ähnlich hatten wir auch die Quartile (und allgemeiner: alle möglichen Quantile) definiert: Falls  $n \cdot p$  keine ganze Zahl ist, ist das entsprechende Quantil der jeweilige Messwert an der Stelle  $(n \cdot p) + 1$ . Falls  $n \cdot p$  dagegen eine ganze Zahl ist, ist das entsprechende Quantil das Mittel der Messwerte an den Stellen  $n \cdot p$  und  $(n \cdot p) + 1$ . Da 16 durch 4 teilbar ist und daher sowohl  $n \cdot 0.25$  als auch  $n \cdot 0.75$  ganze Zahlen ergeben, müsste das erste Quartil nach dieser Logik also  $Q_1 = \frac{x_{n \cdot 0.25} + x_{n \cdot 0.25 + 1}}{2} = \frac{x_4 + x_5}{2} = \frac{2+3}{2} = 2.5$  und das dritte Quartil  $Q_3 = \frac{x_{n \cdot 0.75} + x_{n \cdot 0.75 + 1}}{2} = \frac{x_{12} + x_{13}}{2} = \frac{6+7}{2} = 6.5$  ergeben. Wir erhalten jedoch  $Q_1 = 2.75$  und  $Q_3 = 6.25$ .

Die Antwort ist, dass `quantile()` über 9(!) verschiedene Methoden (`type = 1-9`) zur Berechnung von Quantilen verfügt (`help('quantile')`). Die Standardmethode (Defaultwert) ist `type = 7`, die von uns bevorzugte Methode ist aber `type = 2`. Daher:

```
quantile(x, c(.25, .75), type = 2)
#> 25% 75%
#> 2.5 6.5
```

Welche Methode benutzt `summary()`?

```
summary(x)
#> Min. 1st Qu. Median Mean 3rd Qu. Max.
#> 1.000 2.750 4.500 4.625 6.250 9.000
```

Methode 7!

Bei den Funktionen für Varianz und Standardabweichung ist zu beachten, dass R hier die Stichprobenvarianz und -standardabweichung berechnet. Falls wir die empirische Varianz möchten, müssen wir das Ergebnis von `var(x)` mit  $(\frac{n-1}{n})$  multiplizieren. Um die empirische Standardabweichung zu erhalten müssen wir auch den Umweg über die Varianz nehmen.

```
var(x) # Stichprobenvarianz
#> [1] 5.716667
var(x)*(n-1)/n # Empirische Varianz
```

```
#> [1] 5.359375
sd(x) # Stichprobenstandardabweichung
#> [1] 2.390955
sqrt(var(x)*(n-1)/n)) # Empirische Standardabweichung
#> [1] 2.315032
```

Im letzten Kapitel haben wir unsere eigene Funktion für die empirische Standardabweichung geschrieben:

```
standard_abweichung <- function(x) {
 x <- na.omit(x)
 sqrt(sum((x - mean(x))^2) / (length(x)))
```

Wir können statt der direkten Berechnung auch den gerade gezeigten Weg über die Stichprobenvarianz gehen:

```
emp_sd <- function(x) {
 x <- na.omit(x)
 sqrt(var(x) * ((length(x)-1)/length(x)))
}

standard_abweichung(x)
#> [1] 2.315032
emp_sd(x)
#> [1] 2.315032
Ist es auch wirklich genau dasselbe?
all.equal(standard_abweichung(x), emp_sd(x))
#> [1] TRUE
```

### 8.3.1 Deskriptive Zusammenfassung ausgewählter numerischer Variablen

```
install.packages("RcmdrMisc")
```

Die Funktion `numSummary()` aus dem Package `RcmdrMisc` gibt deskriptive Statistiken für ausgewählte Variablen aus.

Die generische Form mit Defaultwerten ist: `numSummary(Datenmatrix[, c("var1", "var2")], groups = Datenmatrix$faktor, statistics = c("mean", "sd", "quantiles"), quantiles = c(0, .25, .5, .75, 1))`.

Zusätzliche `statistics`-Optionen sind: `se(mean)` (Standardfehler des Mittelwerts =  $\frac{sd}{\sqrt{n}}$ ), `cv` (Variationskoeffizient = SD/Mittelwert), `skewness` (Schiefe) und `kurtosis` (Kurtosis).

Beispiel: Berechnung deskriptiver Statistiken für die Variablen "Emotionale Unterstützung durch Eltern" (`unt_eltern`) und "Emotionale Unterstützung durch Freunde" (`unt_freunde`) mit Mittelwert, Standardabweichung, Standardfehler des Mittelwerts, Schiefe, Kurtosis sowie den 10%- und 90%-Quantilen.

Das Ganze mit Hilfe des `groups`-Befehls getrennt für die Stufen des Faktors Geschlecht (`geschlecht`):

```
library(RcmdrMisc)
numSummary(westost_skalen[,c("unt_eltern", "unt_freunde")],
 groups = westost_skalen$geschlecht,
 statistics = c("mean", "sd", "se(mean)", "skewness",
 "kurtosis", "quantiles"),
 quantiles = c(.1,.9))
#>
#> Variable: unt_eltern
#> mean sd se(mean) skewness kurtosis 10% 90%
```

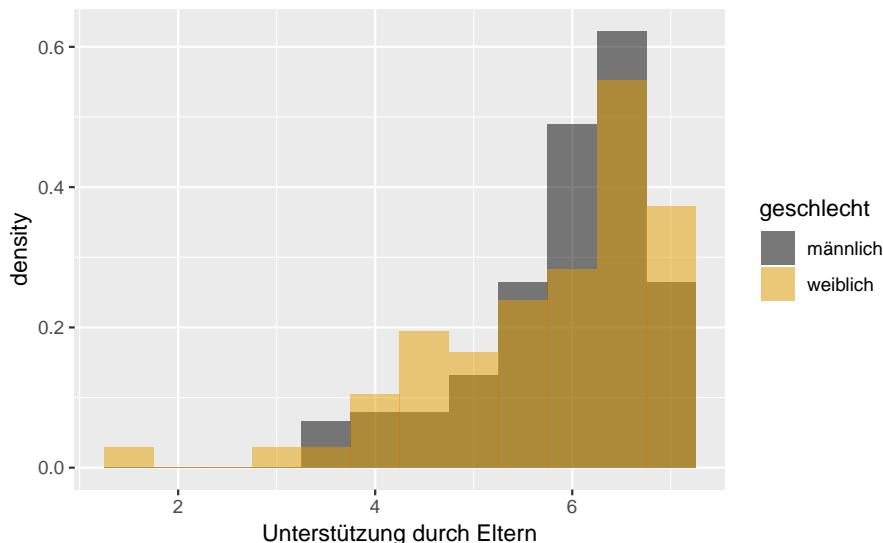
```
#> männlich 5.94415 0.8687628 0.07069892 -1.204500 1.088983 4.666667 6.833333
#> weiblich 5.79005 1.1232414 0.09703328 -1.332349 2.016779 4.333333 6.833333
#> n NA
#> männlich 151 1
#> weiblich 134 0
#>
#> Variable: unt_freunde
#> mean sd se(mean) skewness kurtosis 10%
#> männlich 5.584868 1.0055152 0.08155806 -0.8628965 0.8376967 4.333333
#> weiblich 6.319549 0.7416894 0.06431263 -2.2457703 8.4513611 5.500000
#> 90% n NA
#> männlich 6.833333 152 0
#> weiblich 7.000000 133 1

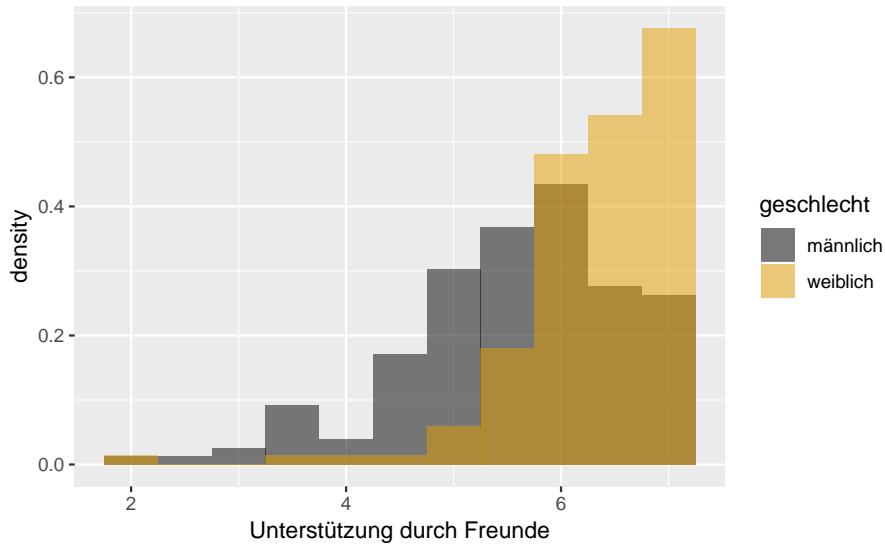
p1 <- westost_skalen %>%
 select(unt_eltern, geschlecht) %>%
 drop_na() %>%
 ggplot(aes(x = unt_eltern, fill = geschlecht, y = ..density..))

p1 + geom_histogram(binwidth = 0.5,
 position = "identity",
 alpha = 0.5) +
 scale_fill_manual(values = c("#000000", "#E69F00")) +
 xlab("Unterstützung durch Eltern")

p2 <- westost_skalen %>%
 select(unt_freunde, geschlecht) %>%
 drop_na() %>%
 ggplot(aes(x = unt_freunde, fill = geschlecht, y = ..density..))

p2 + geom_histogram(binwidth = 0.5,
 position = "identity",
 alpha = 0.5) +
 scale_fill_manual(values = c("#000000", "#E69F00")) +
 xlab("Unterstützung durch Freunde")
```





### 8.3.2 Tabelle mit Statistiken (tapply)

Mit der Funktion `tapply()` lassen sich ausgewählte deskriptive Statistiken für die kombinierten Stufen mehrerer Faktoren darstellen. Geeignet z.B. für die Darstellung von Mittelwertstabellen zur Veranschaulichung der Mittelwertstruktur bei mehrfaktoriellen Varianzanalysen. Die generische Form ist: `tapply(Datenmatrix$var1, list(Faktor1 = Datenmatrix$Faktor1, Faktor2 = Datenmatrix$Faktor2), statistic, na.rm = TRUE)`.

An die Stelle von `statistic` muss die gewünschte Statistik (nur jeweils eine) eingefügt werden. Zur Auswahl stehen z.B. `mean, median, sum, sd`.

Zunächst wählen wir mit der `select`-Funktion aus dem `dplyr`-Package die für das Beispiel benötigten Variablen aus (nicht unbedingt nötig).

```
support <- westost_skalen %>%
 select(ID,
 starts_with("unt"),
 westost, geschlecht,
 ends_with("_b"))

names(support)
#> [1] "ID" "unt_eltern" "unt_freunde"
#> [4] "westost" "geschlecht" "bildung_vater_b"
#> [7] "bildung_mutter_b"
```

Beispiel: Mittelwerte der Variable “Emotionale Unterstützung durch Eltern” für die kombinierten Faktorstufen von `geschlecht` und `westost`:

```
tapply(support$unt_eltern,
 list(Geschlecht = support$geschlecht, Region = support$westost),
 mean, na.rm = TRUE) %>%
 round(2)

#> Region
#> Geschlecht West Ost
#> männlich 5.79 6.13
#> weiblich 5.74 5.83
```

Wir haben das gleiche bereits mit `dplyr` gemacht. Hier mit zwei Varianten für den Umgang mit fehlenden

Werten:

```
1) Fehlende Werte werden nur für unt_eltern entfernt
support %>%
 group_by(geschlecht, westost) %>%
 summarise(mean = mean(unt_eltern, na.rm = TRUE))
#> # A tibble: 4 x 3
#> geschlecht westost mean
#> <fct> <fct> <dbl>
#> 1 männlich West 5.79
#> 2 männlich Ost 6.13
#> 3 weiblich West 5.74
#> 4 weiblich Ost 5.83

2) Hier werden alle Personen entfernt, die auf irgendeiner der Variablen
im Datensatz support fehlende Werte haben
support %>%
 group_by(geschlecht, westost) %>%
 drop_na() %>%
 summarise(mean = mean(unt_eltern))
#> # A tibble: 4 x 3
#> geschlecht westost mean
#> <fct> <fct> <dbl>
#> 1 männlich West 5.81
#> 2 männlich Ost 6.14
#> 3 weiblich West 5.73
#> 4 weiblich Ost 5.78
```

■ Man kann in `tapply()` (und ganz allgemein in R) auch so genannte *anonyme Funktionen* definieren. Wir benutzen dazu `function(x)` "ad-hoc" innerhalb der `tapply()`-Funktion und ohne geschweifte Klammern. Wenn wir zum Beispiel eine Tabelle mit dem Standardfehler des Mittelwerts erhalten wollen ( $SE = \frac{sd}{\sqrt{n}}$ ) können wir das so in `tapply` einfügen:

```
tapply(support$unt_eltern,
 list(Geschlecht = support$geschlecht, Region = support$westost),
 function(x) sd(x, na.rm = TRUE)/sqrt(length(x))) %>%
 round(3)
#> Region
#> Geschlecht West Ost
#> männlich 0.107 0.081
#> weiblich 0.152 0.126
```

■ Als Argument `x` der anonymen Funktion wird das Argument `x` der übergeordneten Funktion benutzt, hier also die Variable `unt_eltern`.

Mit `dplyr` würden wir wahrscheinlich zuerst eine explizite Funktion `Standardfehler` definieren und diese dann aus `dplyr` aufrufen:

```
standard_fehler <- function(x) {
 sd(x, na.rm = TRUE)/sqrt(length(x))
}

support %>%
```

```
group_by(geschlecht, westost) %>%
 summarise(Standardfehler = standard_fe Fehler(unt_eltern))

#> # A tibble: 4 x 3
#> geschlecht westost Standardfehler
#> <fct> <fct> <dbl>
#> 1 männlich West 0.107
#> 2 männlich Ost 0.0814
#> 3 weiblich West 0.152
#> 4 weiblich Ost 0.126
```

- Es ist auch möglich, eine so vordefinierte Funktion mit `tapply` zu verwenden (nur den Funktionsnamen einfügen!):

```
tapply(support$unt_eltern,
 list(Geschlecht = support$geschlecht, Region = support$westost),
 standard_fe Fehler)
round(3)

#> Region
#> Geschlecht West Ost
#> männlich 0.107 0.081
#> weiblich 0.152 0.126
```

### 8.3.3 Deskriptive Statistiken mit dem psych-Package

```
install.packages("psych")
```

Das für Datenanalysen in der Psychologie häufig verwendete Package `psych` besitzt eine nützliche Funktion zur deskriptiven Statistik für numerische Variablen: `describe`. Die generische Form ist: `describe(x, na.rm = TRUE, interp = FALSE, skew = TRUE, ranges = TRUE, trim = .1, type = 3, check = TRUE)`. Für `x` wird der zu analysierende Dataframe oder eine Variable (`dataframe$variable`) eingegeben. Default-Werte sind: `interp = FALSE` bezieht sich auf die Berechnung des Medians (wenn = TRUE wird bei geradem  $n$  zwischen den beiden mittleren Werten interpoliert -> unsere Methode), `skew = TRUE` bezieht sich auf die Ausgabe von Schiefe und Kurtosis sowie des getrimmten Mittels, `ranges = TRUE` auf die Ausgabe der Spannweite und `trim = .1` auf den Anteil der Verteilung, der für den getrimmten Mittelwert am oberen und am unteren Verteilung abgeschnitten wird (per default werden also oben und unten je 10% abgeschnitten, d.h. das getrimmte Mittel wird aus den mittleren 80% der Daten ermittelt), `type = 3` bezieht sich auf die Methode der Berechnung von Schiefe und Kurtosis (mehr dazu siehe unter `?psych::describe`), und `check = TRUE` bezieht sich darauf, dass überprüft werden soll, ob ggf. non-numerische Variablen vorhanden sind (für die die Funktion `describe` keine Verwendung hat). Ist `check = FALSE` angegeben und existieren non-numerische Variablen, wird eine Fehlermeldung ausgegeben.

Lassen wir uns für den Support-Datensatz deskriptive Statistiken ausgeben:

```
library(psych)
describe(support)

#> vars n mean sd median trimmed mad min max
#> ID* 1 286 143.50 82.71 143.50 143.50 106.01 1.0 286
#> unt_eltern 2 285 5.87 1.00 6.17 6.01 0.74 1.5 7
#> unt_freunde 3 285 5.93 0.96 6.00 6.06 0.99 2.0 7
#> westost* 4 286 1.50 0.50 1.50 1.50 0.74 1.0 2
#> geschlecht* 5 286 1.47 0.50 1.00 1.46 0.00 1.0 2
#> bildung_vater_b* 6 269 1.50 0.50 1.00 1.50 0.00 1.0 2
```

```
#> bildung_mutter_b* 7 272 1.49 0.50 1.00 1.49 0.00 1.0 2
#> range skew kurtosis se
#> ID* 285.0 0.00 -1.21 4.89
#> unt_eltern 5.5 -1.35 2.05 0.06
#> unt_freunde 5.0 -1.25 1.85 0.06
#> westost* 1.0 0.00 -2.01 0.03
#> geschlecht* 1.0 0.13 -1.99 0.03
#> bildung_vater_b* 1.0 0.01 -2.01 0.03
#> bildung_mutter_b* 1.0 0.03 -2.01 0.03
describe(support[, c("unt_eltern", "unt_freunde")])
#> vars n mean sd median trimmed mad min max range skew
#> unt_eltern 1 285 5.87 1.00 6.17 6.01 0.74 1.5 7 5.5 -1.35
#> unt_freunde 2 285 5.93 0.96 6.00 6.06 0.99 2.0 7 5.0 -1.25
#> kurtosis se
#> unt_eltern 2.05 0.06
#> unt_freunde 1.85 0.06
describe(support[, c("unt_eltern", "unt_freunde")], trim = 0.2)
#> vars n mean sd median trimmed mad min max range skew
#> unt_eltern 1 285 5.87 1.00 6.17 6.08 0.74 1.5 7 5.5 -1.35
#> unt_freunde 2 285 5.93 0.96 6.00 6.09 0.99 2.0 7 5.0 -1.25
#> kurtosis se
#> unt_eltern 2.05 0.06
#> unt_freunde 1.85 0.06
describe(support[, c("unt_eltern", "unt_freunde")], skew = FALSE,
 range = FALSE)
#> vars n mean sd se
#> unt_eltern 1 285 5.87 1.00 0.06
#> unt_freunde 2 285 5.93 0.96 0.06
```

*mad* steht für *median average deviation* (Mittlere Absolutabweichung vom Median = alternatives Streuungsmaß). Die als Faktoren erkannten Variablen erhalten einen \*. Da sie aber (auch) numerisch kodiert sind, werden trotzdem (in diesem Fall unbrauchbare) Statistiken ausgegeben. Aus diesem Grund beschränken wir uns in den folgenden Befehlen auf die beiden interessierenden metrischen Variablen *unt\_eltern* und *unt\_freunde*. Mit *trim* = 0.2 fordern wir ein getrimmtest Mittel an, bei dem 20% der Werte am unteren sowie 20% der Werte am oberen Ende der Verteilung entfernt werden. Mit *skew* = FALSE, *range* = FALSE fordern wir eine deskriptive Statistik ohne Schiefe und Kurtosis sowie ohne *median*, *trimmed*, *mad*, *min*, *max*, *range* an, erhalten also nur die grundlegenden Statistiken zu Mittelwert, Standardabweichung und Standardfehler des Mittelwerts.

## 8.4 Kovarianz- und Korrelationsanalyse

### 8.4.1 Kovarianzen

Die generische Form zur Berechnung einer Kovarianzmatrix lautet: `cov(Datenmatrix[,c("var1", "var2", "var3")], use = "complete.obs")`. In der Diagonale befinden sich die Varianzen der Variablen, in den oberen und unteren Dreiecken jeweils die Kovarianzen.

Für das Beispiel basteln wir uns einen Datensatz mit allen Selbstwirksamkeitsskalen:

```
swk <- westost_skalen %>%
 select(starts_with("swk"))
names(swk)
#> [1] "swk_akad" "swk_selbstreg" "swk_durch" "swk_motselbst"
#> [5] "swk_sozharm" "swk_bez"
```

```

cov(swk, use = "complete.obs") %>%
 round(3)
#> swk_akad swk_selbstreg swk_durch swk_motselbst swk_sozharm
#> swk_akad 0.595 0.348 0.266 0.380 0.220
#> swk_selbstreg 0.348 0.647 0.252 0.448 0.216
#> swk_durch 0.266 0.252 0.970 0.314 0.458
#> swk_motselbst 0.380 0.448 0.314 0.591 0.246
#> swk_sozharm 0.220 0.216 0.458 0.246 0.617
#> swk_bez 0.227 0.282 0.243 0.240 0.291
#> swk_bez
#> swk_akad 0.227
#> swk_selbstreg 0.282
#> swk_durch 0.243
#> swk_motselbst 0.240
#> swk_sozharm 0.291
#> swk_bez 0.566

auch hier können wir den pipe-Operator verwenden
und das Ergebnis in einem Objekt abspeichern:

swk_cov <- swk %>%
 cov(use = "complete.obs") %>%
 round(3)
swk_cov
#> swk_akad swk_selbstreg swk_durch swk_motselbst swk_sozharm
#> swk_akad 0.595 0.348 0.266 0.380 0.220
#> swk_selbstreg 0.348 0.647 0.252 0.448 0.216
#> swk_durch 0.266 0.252 0.970 0.314 0.458
#> swk_motselbst 0.380 0.448 0.314 0.591 0.246
#> swk_sozharm 0.220 0.216 0.458 0.246 0.617
#> swk_bez 0.227 0.282 0.243 0.240 0.291
#> swk_bez
#> swk_akad 0.227
#> swk_selbstreg 0.282
#> swk_durch 0.243
#> swk_motselbst 0.240
#> swk_sozharm 0.291
#> swk_bez 0.566

```

### 8.4.2 Korrelationen

Die generische Form zur Berechnung einer (Pearson-)Korrelationmatrix lautet: `cor(Datenmatrix[,c("var1", "var2", "var3")], method = c("pearson", "kendall", "spearman"), use = "complete.obs")`. Wir können uns also entweder Pearson-Korrelationen (default) oder (für ordinale Daten) Kendalls tau oder Spearman-Rangkorrelationen ausgeben lassen.

```

swk_cor <- swk %>%
 cor(method = "pearson", use = "complete.obs") %>%
 round(3)
swk_cor
#> swk_akad swk_selbstreg swk_durch swk_motselbst swk_sozharm
#> swk_akad 1.000 0.562 0.351 0.640 0.363
#> swk_selbstreg 0.562 1.000 0.318 0.724 0.342
#> swk_durch 0.351 0.318 1.000 0.415 0.592

```

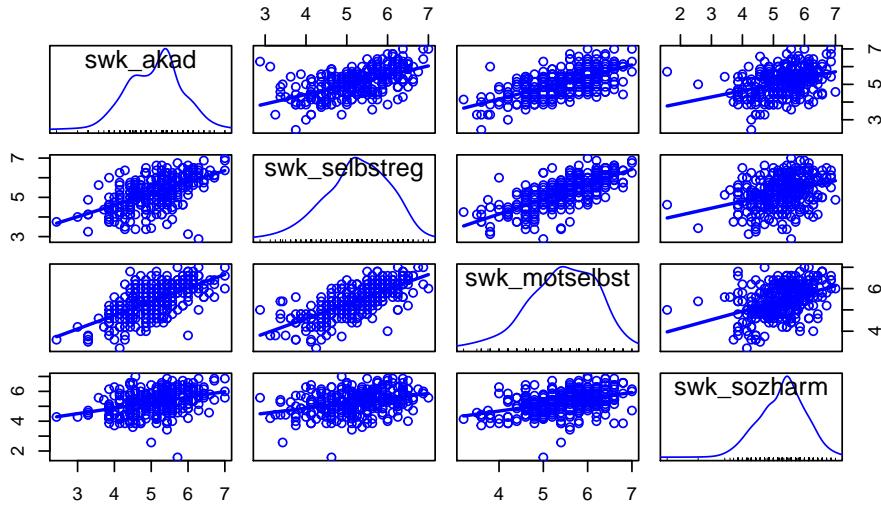
```
#> swk_motselbst 0.640 0.724 0.415 1.000 0.407
#> swk_sozharm 0.363 0.342 0.592 0.407 1.000
#> swk_bez 0.392 0.466 0.327 0.415 0.493
#> swk_bez
#> swk_akad 0.392
#> swk_selbstreg 0.466
#> swk_durch 0.327
#> swk_motselbst 0.415
#> swk_sozharm 0.493
#> swk_bez 1.000
```

Manchmal möchte man eine Matrix berichten, in deren *unterer* Dreiecksmatrix sich die Kovarianzen befinden und in deren *oberer* Dreiecksmatrix sich die Pearson-Korrelationen befinden. Außerdem sollen sich in der Diagonalen die Varianzen der einzelnen Variablen befinden. Eine solche erhalten wir, wenn wir die obere Dreiecksmatrix der Varianz-Kovarianz-Matrix (mit `cov()` berechnet) durch die obere Dreiecksmatrix der Korrelationsmatrix ersetzen:

```
swk_cov[upper.tri(swk_cov)] <- swk_cor[upper.tri(swk_cor)]
swk_mixed <- swk_cov
swk_mixed # Matrix mit Kovarianzen im unteren Dreieck, Varianzen
#> swk_akad swk_selbstreg swk_durch swk_motselbst swk_sozharm
#> swk_akad 0.595 0.562 0.351 0.640 0.363
#> swk_selbstreg 0.348 0.647 0.318 0.724 0.342
#> swk_durch 0.266 0.252 0.970 0.415 0.592
#> swk_motselbst 0.380 0.448 0.314 0.591 0.407
#> swk_sozharm 0.220 0.216 0.458 0.246 0.617
#> swk_bez 0.227 0.282 0.243 0.240 0.291
#> swk_bez
#> swk_akad 0.392
#> swk_selbstreg 0.466
#> swk_durch 0.327
#> swk_motselbst 0.415
#> swk_sozharm 0.493
#> swk_bez 0.566
in der Diagonalen und Korrelationen im oberen Dreieck
```

Eine gute Möglichkeit der Darstellung einer Scatterplot-Matrix bietet die Funktion `scatterplotMatrix` aus dem `car`-Package. Interessant ist insbesondere, dass man in der Diagonale verschiedene univariate Statistiken anzeigen lassen kann: `diagonal = c("density", "boxplot", "histogram", "oned", "qqplot", "none")`. Neben der linearen Regressionslinie kann man sich mit `smooth = TRUE` auch eine nichtlineare sogenannte LOWESS-Linie (LOCally WEighted Scatterplot Smoother) anzeigen lassen, um die (Nicht-)Linearität des Zusammenhangs beurteilen.

```
library(car)
scatterplotMatrix(~ swk_akad + swk_selbstreg + swk_motselbst + swk_sozharm,
 smooth = FALSE,
 diagonal = "density",
 data = swk)
```



- Probieren Sie `scatterplotMatrix()` mit verschiedenen Argumenten/Optionen aus! ■

Mit der Funktion `cor.test` testet man Korrelationen auf Signifikanz, allerdings ist (leider) nur jeweils ein Korrelationstest möglich. Es können also nicht alle Korrelationen einer Korrelationsmatrix gleichzeitig getestet werden. Die generische Form ist: `cor.test(x, y, alternative = c("two.sided", "less", "greater"), method = c("pearson", "kendall", "spearman") , conf.level = 0.95)`. Man kann mit dieser Funktion also neben der ungerichteten Alternativhypothese (`alternative = "two.sided"`) auch gerichtete Alternativhypthesen testen: (`alternative = "less"`) für die Alternativhypothese einer negativen Korrelation ("less than 0") und (`alternative = "greater"`) für die Alternativhypothese einer positiven Korrelation ("greater than 0").

Wir wählen einen neuen Beispieldatensatz mit den Lebenszufriedenheits- und den Stressvariablen aus.

```
lebstress <- westost_skalen %>%
 select(starts_with("leben"),
 stress_psych, stress_somat)
names(lebstress)
#> [1] "leben_selbst" "leben_fam" "leben_schule" "leben_freunde"
#> [5] "leben_gesamt" "stress_psych" "stress_somat"
```

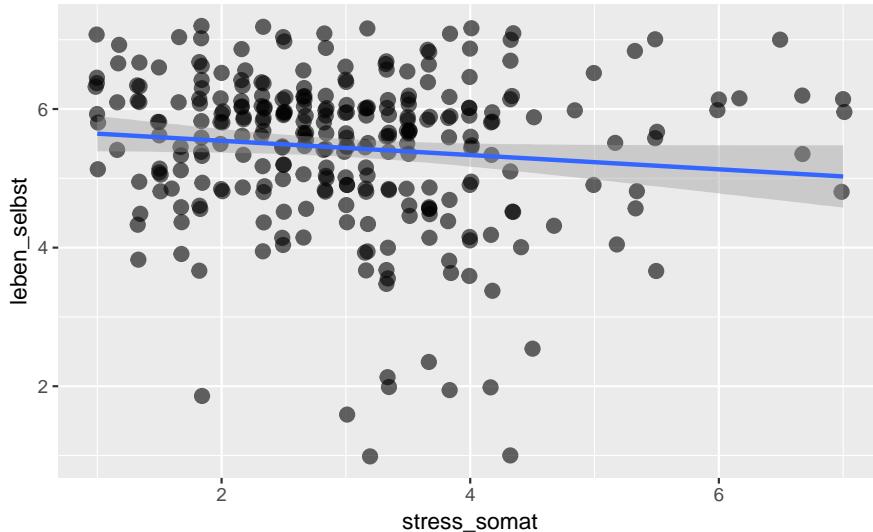
Wir interessieren uns für den Test der Korrelation zwischen `leben_selbst` und `stress_somat`. Wir haben Grund zu der Annahme, dass Personen, die eine höhere Zufriedenheit mit dem eigenen Selbst berichten auch ein geringeres somatisches Stresserleben berichten. Unsere Alternativhypothese ist also die einer negativen Korrelation. Dementsprechend setzen wir das Argument `alternative = "less"` und erhalten somit den einseitigen linksseitigen *p*-Wert.

```
cor.test(lebstress$leben_selbst, lebstress$stress_somat,
 alternative = "less", method = "pearson")
#>
#> Pearson's product-moment correlation
#>
#> data: lebstress$leben_selbst and lebstress$stress_somat
#> t = -1.871, df = 280, p-value = 0.03119
#> alternative hypothesis: true correlation is less than 0
#> 95 percent confidence interval:
#> -1.00000000 -0.01310724
#> sample estimates:
#> cor
```

```
#> -0.111122
```

```
p <- lebstress %>%
 select(leben_selbst, stress_somat) %>%
 drop_na() %>%
 ggplot(mapping = aes(x = stress_somat,
 y = leben_selbst))

p + geom_jitter(size = 3, alpha = 0.6) +
 stat_smooth(method = "lm", se = TRUE)
```



- Wählen Sie zwei andere Variablen aus und testen Sie die Korrelation. Überlegen Sie (aufgrund psychologischer Theorie und/oder common sense), welche Alternativhypothese Ihnen sinnvoll erscheint! ■

Im psych-Package gibt es die Funktion `corr.test()`, die uns die  $p$ -Werte auch für eine ganze Korrelationsmatrix liefert. Allerdings sind hier keine gerichteten Alternativhypthesen möglich. Es werden Matrizen der Korrelationen, der Stichprobengrößen und der  $p$ -Werte ausgegeben. Bei der  $p$ -Werte-matrix ist zu beachten, dass in der unteren Dreiecksmatrix die unadjustierten (zweiseitigen)  $p$ -Werte stehen und in der oberen Dreiecksmatrix die nach Holm adjustierten  $p$ -Werte (in der Diagonalen stehen außerdem die  $p$ -Werte für die Varianzen der Variablen, die uns hier nicht interessieren). Die Holm-Prozedur entspricht einer sequentiellen Alpha-Korrektur, die weniger stark als die Bonferroni-Prozedur korrigiert, den family-wise-error aber dennoch kontrolliert (vgl. Statistik II).

- Bei der Holm-Prozedur (Bonferroni-Holm-Methode) wird der kleinste  $p$ -Wert mit der Anzahl der Tests ( $(p(p - 1)/2;$   $p$  = Anzahl Variablen in der Korrelationsmatrix) multipliziert, der zweitkleinste mit der Anzahl Tests minus 1, usw. bis zum grössten  $p$ -Wert, der nicht mehr korrigiert wird (d.h. mit  $p - p + 1 = 1$  multipliziert wird). Allerdings wird der  $p$ -Wert nur *dann* mit der entsprechenden Anzahl Tests multipliziert, wenn der resultierende adjustierte  $p$ -Wert nicht kleiner wird als der vorher adjustierte (kleinere)  $p$ -Wert. Sollte dies der Fall sein, wird dem Test derselbe  $p$ -Wert wie dem Vorgänger zugewiesen. Damit wird verhindert, dass ein kleinerer Effekt nach der Adjustierung einen kleineren  $p$ -Wert als ein grösßerer Effekt bekommt (was z.B. auch dazu führen könnte, dass letzterer nicht-signifikant aber gleichzeitig ersterer signifikant werden würde).
- 

```
library(psych)
lebstress %>% corr.test()
#> Call:corr.test(x = .)
```

```

#> Correlation matrix
#>
#> leben_selbst leben_fam leben_schule leben_freunde
#> leben_selbst 1.00 0.32 0.18 0.35
#> leben_fam 0.32 1.00 0.36 0.28
#> leben_schule 0.18 0.36 1.00 0.15
#> leben_freunde 0.35 0.28 0.15 1.00
#> leben_gesamt 0.63 0.77 0.70 0.55
#> stress_psych -0.31 -0.25 -0.19 -0.24
#> stress_somat -0.11 -0.13 -0.16 -0.06
#>
#> leben_gesamt stress_psych stress_somat
#> leben_selbst 0.63 -0.31 -0.11
#> leben_fam 0.77 -0.25 -0.13
#> leben_schule 0.70 -0.19 -0.16
#> leben_freunde 0.55 -0.24 -0.06
#> leben_gesamt 1.00 -0.36 -0.19
#> stress_psych -0.36 1.00 0.64
#> stress_somat -0.19 0.64 1.00
#> Sample Size
#>
#> leben_selbst leben_fam leben_schule leben_freunde
#> leben_selbst 285 284 285 285
#> leben_fam 284 284 284 284
#> leben_schule 285 284 285 285
#> leben_freunde 285 284 285 285
#> leben_gesamt 285 284 285 285
#> stress_psych 283 282 283 283
#> stress_somat 282 281 282 282
#>
#> leben_gesamt stress_psych stress_somat
#> leben_selbst 285 283 282
#> leben_fam 284 282 281
#> leben_schule 285 283 282
#> leben_freunde 285 283 282
#> leben_gesamt 285 283 282
#> stress_psych 283 284 283
#> stress_somat 282 283 283
#> Probability values (Entries above the diagonal are adjusted for multiple tests.)
#>
#> leben_selbst leben_fam leben_schule leben_freunde
#> leben_selbst 0.00 0.00 0.01 0.00
#> leben_fam 0.00 0.00 0.00 0.00
#> leben_schule 0.00 0.00 0.00 0.05
#> leben_freunde 0.00 0.00 0.01 0.00
#> leben_gesamt 0.00 0.00 0.00 0.00
#> stress_psych 0.00 0.00 0.00 0.00
#> stress_somat 0.06 0.03 0.01 0.31
#>
#> leben_gesamt stress_psych stress_somat
#> leben_selbst 0 0.00 0.12
#> leben_fam 0 0.00 0.09
#> leben_schule 0 0.01 0.04
#> leben_freunde 0 0.00 0.31
#> leben_gesamt 0 0.00 0.01
#> stress_psych 0 0.00 0.00
#> stress_somat 0 0.00 0.00
#>
#> To see confidence intervals of the correlations, print with the short=FALSE option

```

■ Es ist auch möglich, andere Adjustierungen der  $p$ -Werte vorzunehmen. Das ist allerdings bisher nicht wirklich gut in R implementiert. Es existiert in **base R** eine Funktion `p.adjust`, die es ermöglicht, einen Vektor von  $p$ -Werten nach verschiedenen Methoden zu adjustieren, z.B. `method = bonferroni` oder auch `method = holm`. Dazu muss man den entsprechenden Vektor von unadjustierten  $p$ -Werten aus der durch die `corr.test()`-Funktion ausgegebenen Matrix extrahieren (also nur die Elemente der unteren Dreiecksmatrix). ■

```
library(psych)
Ergebnis in Objekt abspeichern, um die p-Werte extrahieren zu können
cor_lebstress <- lebstress %>%
 corr.test()

Auswahl der unteren Dreiecksmatrix
p_values <- cor_lebstress$p[lower.tri(cor_lebstress$p)]
```

# Anwenden der `p.adjust`-Funktion mit unterschiedlichen Methoden auf die # unadjustierten  $p$ -Werte

```
p.adjust(p_values, method = "bonferroni") %>%
 round(2)
#> [1] 0.00 0.05 0.00 0.00 0.00 1.00 0.00 0.00 0.00 0.00 0.63 0.24 0.00 0.02
#> [15] 0.15 0.00 0.00 1.00 0.00 0.04 0.00
p.adjust(p_values, method = "holm") %>%
 round(2)
#> [1] 0.00 0.01 0.00 0.00 0.00 0.12 0.00 0.00 0.00 0.00 0.09 0.05 0.00 0.01
#> [15] 0.04 0.00 0.00 0.31 0.00 0.01 0.00
```

### 8.4.3 Partialkorrelation

Die generische Form zur Berechnung einer Partialkorrelationsmatrix lautet: `partial.cor(Datenmatrix[, c("var1", "var2", "var3")], use = "complete.obs")`. Wichtig: Die Auspartialisierung für die entsprechende bivariate Korrelation erfolgt für alle restlichen Variablen!

```
Normale Korrelationsmatrix zum Vergleich
cor(westost_skalen[,c("swk_akad", "swk_selbstreg", "Schnitt")],
 use = "complete.obs")
#> swk_akad swk_selbstreg Schnitt
#> swk_akad 1.0000000 0.5624954 0.4184297
#> swk_selbstreg 0.5624954 1.0000000 0.3847193
#> Schnitt 0.4184297 0.3847193 1.0000000

Partialkorrelationsmatrix mit drei Variablen (nur jeweils eine Variable
auspartialisiert -> Partialkorrelationen erster Ordnung)
partial.cor(westost_skalen[,c("swk_akad", "swk_selbstreg", "Schnitt")],
 use = "complete.obs")
#>
#> Partial correlations:
#> swk_akad swk_selbstreg Schnitt
#> swk_akad 0.00000 0.47894 0.26472
#> swk_selbstreg 0.47894 0.00000 0.19889
#> Schnitt 0.26472 0.19889 0.00000
#>
#> Number of observations: 280
```

```
Partialkorrelationsmatrix mit vier Variablen (jeweils zwei Variablen
werden auspartialisiert -> Partialkorrelationen zweiter Ordnung)
partial.cor(westost_skalen[,c("swk_akad", "swk_selbstreg",
 "Schnitt", "leben_schule")],
 use = "complete.obs")
#>
#> Partial correlations:
#> swk_akad swk_selbstreg Schnitt leben_schule
#> swk_akad 0.00000 0.43015 0.22486 0.07308
#> swk_selbstreg 0.43015 0.00000 0.07488 0.30949
#> Schnitt 0.22486 0.07488 0.00000 0.33116
#> leben_schule 0.07308 0.30949 0.33116 0.00000
#>
#> Number of observations: 279
```

## 8.5 Reliabilitätsanalyse mit dem psych-Package

### 8.5.1 Subdatensätze für die Reliabilitätsanalyse erstellen

Wir benötigen für die Reliabilitätsanalyse denjenigen Beispieldatensatz, in dem noch alle Einzelitems enthalten sind (`westost`). Dann bilden wir zwei Subdatensätze: Einen für die Skala “Schulleistungsselbstwirksamkeit”, die wir mit `swk_akad` abkürzen. Aus dem Info-Dokument zum Beispieldatensatz können wir entnehmen, dass die Items T1\_1, T1\_7, T1\_14, T1\_16, T1\_26, T1\_27 und T1\_31 zu dieser Subskala gehören, deren Reliabilität (interne Konsistenz) wir bestimmen wollen. Der zweite Subdatensatz ist für die Skala “Soziale Selbstwirksamkeit: Harmonie”, die wir als `swk_sozharm` bezeichnen. Zu dieser Skala gehören die Items T1\_2, T1\_8, T1\_15, T1\_23, T1\_29, T1\_32, T1\_35.

```
library(psych)
load(file = "data/westost.Rdata")
names(westost)
#> [1] "ID" "westost" "geschlecht"
#> [4] "alter" "T1_1" "T1_2"
#> [7] "T1_3" "T1_4" "T1_5"
#> [10] "T1_6" "T1_7" "T1_8"
#> [13] "T1_9" "T1_10" "T1_11"
#> [16] "T1_12" "T1_13" "T1_14"
#> [19] "T1_15" "T1_16" "T1_17"
#> [22] "T1_18" "T1_19" "T1_20"
#> [25] "T1_21" "T1_22" "T1_23"
#> [28] "T1_24" "T1_25" "T1_26"
#> [31] "T1_27" "T1_28" "T1_29"
#> [34] "T1_30" "T1_31" "T1_32"
#> [37] "T1_33" "T1_34" "T1_35"
#> [40] "T1_36" "T2_1" "T2_2"
#> [43] "T2_3" "T2_4" "T2_5"
#> [46] "T2_6" "T3_1" "T3_2"
#> [49] "T3_3" "T3_4" "T3_5"
#> [52] "T3_6" "T4_1" "T4_2"
#> [55] "T4_3" "T4_4" "T4_5"
#> [58] "T4_6" "T4_7" "T4_8"
#> [61] "T4_9" "T4_10" "T4_11"
#> [64] "T5_1" "T5_2" "T5_3"
#> [67] "T5_4" "T5_5" "T5_6"
```

```

#> [70] "T5_7" "T5_8" "T5_9"
#> [73] "T5_10" "T5_11" "T6_1"
#> [76] "T6_2" "T6_3" "T6_4"
#> [79] "T6_5" "T6_6" "T6_7"
#> [82] "T6_8" "T6_9" "T6_10"
#> [85] "T6_11" "T7_1" "T7_2"
#> [88] "T7_3" "T7_4" "T7_5"
#> [91] "T7_6" "T7_7" "T7_8"
#> [94] "T7_9" "T7_10" "T7_11"
#> [97] "T7_12" "SES" "Deutsch"
#> [100] "Mathe" "Fremdspr" "Schnitt"
#> [103] "bildung_vater" "bildung_mutter" "bildung_vater_b"
#> [106] "bildung_mutter_b" "swk_akad" "swk_selbstreg"
#> [109] "swk_durch" "swk_motselbst" "swk_sozharm"
#> [112] "swk_bez" "unt_eltern" "unt_freunde"
#> [115] "sch_leistung" "sch_bez" "sch_regeln"
#> [118] "ges_gerecht" "ges_nepot" "leben_selbst"
#> [121] "leben_fam" "leben_schule" "leben_freunde"
#> [124] "leben_gesamt" "stress_somat" "stress_psych"

swk_akad <- westost %>%
 select(one_of(c("T1_1", "T1_7", "T1_14", "T1_16",
 "T1_26", "T1_27", "T1_31")))

swk_sozharm <- westost %>%
 select(one_of(c("T1_2", "T1_8", "T1_15", "T1_23",
 "T1_29", "T1_32", "T1_35")))

```

### 8.5.2 Cronbachs Alpha berechnen

Die interne Konsistenz (Cronbachs Alpha) berechnen wir jetzt mit der Funktion `alpha()`:

```

alpha(swk_akad)
#>
#> Reliability analysis
#> Call: alpha(x = swk_akad)
#>
#> raw_alpha std.alpha G6(smc) average_r S/N ase mean sd median_r
#> 0.63 0.63 0.65 0.19 1.7 0.033 5.1 0.77 0.18
#>
#> lower alpha upper 95% confidence boundaries
#> 0.56 0.63 0.7
#>
#> Reliability if an item is dropped:
#> raw_alpha std.alpha G6(smc) average_r S/N alpha se var.r med.r
#> T1_1 0.59 0.59 0.60 0.19 1.4 0.037 0.022 0.168
#> T1_7 0.58 0.58 0.57 0.19 1.4 0.039 0.013 0.192
#> T1_14 0.57 0.56 0.57 0.18 1.3 0.039 0.022 0.097
#> T1_16 0.64 0.64 0.65 0.23 1.8 0.032 0.020 0.192
#> T1_26 0.61 0.61 0.63 0.21 1.6 0.035 0.024 0.180
#> T1_27 0.58 0.58 0.59 0.19 1.4 0.039 0.021 0.168
#> T1_31 0.56 0.57 0.56 0.18 1.3 0.040 0.015 0.180
#>
```

```
#> Item statistics
#> n raw.r std.r r.cor r.drop mean sd
#> T1_1 286 0.53 0.56 0.45 0.34 5.0 1.2
#> T1_7 284 0.61 0.58 0.52 0.39 5.0 1.5
#> T1_14 283 0.61 0.62 0.54 0.41 4.9 1.4
#> T1_16 285 0.43 0.43 0.24 0.18 5.8 1.4
#> T1_26 285 0.47 0.50 0.34 0.27 5.3 1.2
#> T1_27 281 0.61 0.58 0.48 0.38 4.9 1.5
#> T1_31 283 0.63 0.61 0.56 0.43 4.7 1.4
#>
#> Non missing response frequency for each item
#> 1 2 3 4 5 6 7 miss
#> T1_1 0.01 0.01 0.07 0.20 0.33 0.30 0.08 0.00
#> T1_7 0.02 0.04 0.12 0.13 0.29 0.24 0.16 0.01
#> T1_14 0.03 0.05 0.07 0.19 0.26 0.32 0.10 0.01
#> T1_16 0.02 0.02 0.03 0.09 0.15 0.32 0.36 0.00
#> T1_26 0.01 0.01 0.05 0.14 0.31 0.33 0.15 0.00
#> T1_27 0.04 0.03 0.09 0.21 0.23 0.26 0.15 0.02
#> T1_31 0.03 0.03 0.13 0.22 0.26 0.23 0.10 0.01
```

Optional kann man einen “key-Vektor” einbauen. Das ist besonders nützlich, wenn man einige Items hat, die inhaltlich in die entgegengesetzte Richtung formuliert sind und daher eigentlich vor der Reliabilitätsanalyse rekodiert werden müssten. Der “key-Vektor” zeigt an, welche Items rekodiert berücksichtigt werden müssen (hier: keine!)

```
alpha(swk_akad, keys = c(1, 1, 1, 1, 1, 1))
#>
#> Reliability analysis
#> Call: alpha(x = swk_akad, keys = c(1, 1, 1, 1, 1, 1))
#>
#> raw_alpha std.alpha G6(smc) average_r S/N ase mean sd median_r
#> 0.63 0.63 0.65 0.19 1.7 0.033 5.1 0.77 0.18
#>
#> lower alpha upper 95% confidence boundaries
#> 0.56 0.63 0.7
#>
#> Reliability if an item is dropped:
#> raw_alpha std.alpha G6(smc) average_r S/N alpha se var.r med.r
#> T1_1 0.59 0.59 0.60 0.19 1.4 0.037 0.022 0.168
#> T1_7 0.58 0.58 0.57 0.19 1.4 0.039 0.013 0.192
#> T1_14 0.57 0.56 0.57 0.18 1.3 0.039 0.022 0.097
#> T1_16 0.64 0.64 0.65 0.23 1.8 0.032 0.020 0.192
#> T1_26 0.61 0.61 0.63 0.21 1.6 0.035 0.024 0.180
#> T1_27 0.58 0.58 0.59 0.19 1.4 0.039 0.021 0.168
#> T1_31 0.56 0.57 0.56 0.18 1.3 0.040 0.015 0.180
#>
#> Item statistics
#> n raw.r std.r r.cor r.drop mean sd
#> T1_1 286 0.53 0.56 0.45 0.34 5.0 1.2
#> T1_7 284 0.61 0.58 0.52 0.39 5.0 1.5
#> T1_14 283 0.61 0.62 0.54 0.41 4.9 1.4
#> T1_16 285 0.43 0.43 0.24 0.18 5.8 1.4
#> T1_26 285 0.47 0.50 0.34 0.27 5.3 1.2
#> T1_27 281 0.61 0.58 0.48 0.38 4.9 1.5
```

```
#> T1_31 283 0.63 0.61 0.56 0.43 4.7 1.4
#>
#> Non missing response frequency for each item
#> 1 2 3 4 5 6 7 miss
#> T1_1 0.01 0.01 0.07 0.20 0.33 0.30 0.08 0.00
#> T1_7 0.02 0.04 0.12 0.13 0.29 0.24 0.16 0.01
#> T1_14 0.03 0.05 0.07 0.19 0.26 0.32 0.10 0.01
#> T1_16 0.02 0.02 0.03 0.09 0.15 0.32 0.36 0.00
#> T1_26 0.01 0.01 0.05 0.14 0.31 0.33 0.15 0.00
#> T1_27 0.04 0.03 0.09 0.21 0.23 0.26 0.15 0.02
#> T1_31 0.03 0.03 0.13 0.22 0.26 0.23 0.10 0.01
```

Dagegen ergibt folgender Befehl mit (fälschlicherweise) spezifizierten Rekodierungen auf den Variablen 2 und 5:

```
alpha(swk_akad, keys = c(1, -1, 1, 1, -1, 1, 1))
#>
#> Reliability analysis
#> Call: alpha(x = swk_akad, keys = c(1, -1, 1, 1, -1, 1, 1))
#>
#> raw_alpha std.alpha G6(smc) average_r S/N ase mean sd median_r
#> 0.63 0.63 0.65 0.19 1.7 0.033 5.1 0.77 0.18
#>
#> lower alpha upper 95% confidence boundaries
#> 0.56 0.63 0.7
#>
#> Reliability if an item is dropped:
#> raw_alpha std.alpha G6(smc) average_r S/N alpha se var.r med.r
#> T1_1 0.59 0.59 0.60 0.19 1.4 0.037 0.022 0.168
#> T1_7 0.58 0.58 0.57 0.19 1.4 0.039 0.013 0.192
#> T1_14 0.57 0.56 0.57 0.18 1.3 0.039 0.022 0.097
#> T1_16 0.64 0.64 0.65 0.23 1.8 0.032 0.020 0.192
#> T1_26 0.61 0.61 0.63 0.21 1.6 0.035 0.024 0.180
#> T1_27 0.58 0.58 0.59 0.19 1.4 0.039 0.021 0.168
#> T1_31 0.56 0.57 0.56 0.18 1.3 0.040 0.015 0.180
#>
#> Item statistics
#> n raw.r std.r r.cor r.drop mean sd
#> T1_1 286 0.53 0.56 0.45 0.34 5.0 1.2
#> T1_7 284 0.61 0.58 0.52 0.39 5.0 1.5
#> T1_14 283 0.61 0.62 0.54 0.41 4.9 1.4
#> T1_16 285 0.43 0.43 0.24 0.18 5.8 1.4
#> T1_26 285 0.47 0.50 0.34 0.27 5.3 1.2
#> T1_27 281 0.61 0.58 0.48 0.38 4.9 1.5
#> T1_31 283 0.63 0.61 0.56 0.43 4.7 1.4
#>
#> Non missing response frequency for each item
#> 1 2 3 4 5 6 7 miss
#> T1_1 0.01 0.01 0.07 0.20 0.33 0.30 0.08 0.00
#> T1_7 0.02 0.04 0.12 0.13 0.29 0.24 0.16 0.01
#> T1_14 0.03 0.05 0.07 0.19 0.26 0.32 0.10 0.01
#> T1_16 0.02 0.02 0.03 0.09 0.15 0.32 0.36 0.00
#> T1_26 0.01 0.01 0.05 0.14 0.31 0.33 0.15 0.00
#> T1_27 0.04 0.03 0.09 0.21 0.23 0.26 0.15 0.02
```

```
#> T1_31 0.03 0.03 0.13 0.22 0.26 0.23 0.10 0.01
```

Erklärung des Outputs:

```
raw_alpha # Auf den Kovarianzen basierendes Alpha
std.alpha # Auf den Korrelationen basierendes Alpha
G6(smc) # Guttman's Lambda 6 (alternatives Reliabilitätsmaß)
average_r # Durchschnittliche Interitem-Korrelation
mean # Skalenmittelwert (über alle Items)
sd # Standardabweichung des Gesamtscores (der Skala,
 # über alle Items)
alpha.drop # Obige Statistiken für den Fall, dass jeweils ein Item
 # weggelassen wird
item.stats # verschiedene Item-Total-Korrelationen (für Rohwerte
 # sowie korrigiert)

alpha(swk_sozharm)
#>
#> Reliability analysis
#> Call: alpha(x = swk_sozharm)
#>
#> raw_alpha std.alpha G6(smc) average_r S/N ase mean sd median_r
#> 0.8 0.82 0.81 0.39 4.5 0.018 5.3 0.78 0.36
#>
#> lower alpha upper 95% confidence boundaries
#> 0.77 0.8 0.84
#>
#> Reliability if an item is dropped:
#> raw_alpha std.alpha G6(smc) average_r S/N alpha se var.r med.r
#> T1_2 0.76 0.78 0.76 0.37 3.5 0.022 0.011 0.32
#> T1_8 0.76 0.78 0.76 0.37 3.5 0.022 0.012 0.32
#> T1_15 0.79 0.80 0.79 0.41 4.1 0.020 0.017 0.37
#> T1_23 0.82 0.82 0.81 0.44 4.6 0.016 0.011 0.39
#> T1_29 0.78 0.80 0.79 0.40 4.1 0.020 0.018 0.35
#> T1_32 0.77 0.78 0.77 0.37 3.5 0.022 0.013 0.35
#> T1_35 0.77 0.78 0.76 0.37 3.6 0.022 0.011 0.36
#>
#> Item statistics
#> n raw.r std.r r.cor r.drop mean sd
#> T1_2 285 0.75 0.75 0.71 0.61 5.2 1.27
#> T1_8 281 0.75 0.76 0.71 0.63 5.5 1.10
#> T1_15 284 0.64 0.64 0.54 0.49 4.7 1.10
#> T1_23 278 0.60 0.55 0.42 0.37 4.5 1.47
#> T1_29 286 0.65 0.65 0.56 0.51 5.1 1.10
#> T1_32 281 0.73 0.75 0.70 0.62 5.8 0.99
#> T1_35 281 0.72 0.75 0.71 0.62 6.1 0.96
#>
#> Non missing response frequency for each item
#> 1 2 3 4 5 6 7 miss
#> T1_2 0.01 0.01 0.07 0.20 0.27 0.27 0.17 0.00
#> T1_8 0.00 0.01 0.02 0.15 0.26 0.39 0.17 0.02
#> T1_15 0.01 0.01 0.07 0.30 0.38 0.17 0.05 0.01
#> T1_23 0.04 0.06 0.12 0.24 0.27 0.20 0.08 0.03
#> T1_29 0.00 0.01 0.06 0.20 0.36 0.27 0.10 0.00
#> T1_32 0.01 0.00 0.01 0.05 0.25 0.44 0.23 0.02
```

```
#> T1_35 0.00 0.00 0.01 0.04 0.18 0.39 0.38 0.02
```

Man kann sich auch nur Alpha ausgeben lassen über die `summary`-Funktion. Dazu muss das Ergebnis der Analyse aber in einem Objekt abgespeichert werden:

```
sozharm_rel <- alpha(swk_sozharm)
summary(sozharm_rel)
#>
#> Reliability analysis
#> raw_alpha std.alpha G6(smc) average_r S/N ase mean sd median_r
#> 0.8 0.82 0.81 0.39 4.5 0.018 5.3 0.78 0.36

oder
swk_sozharm %>%
 alpha() %>%
 summary()
#>
#> Reliability analysis
#> raw_alpha std.alpha G6(smc) average_r S/N ase mean sd median_r
#> 0.8 0.82 0.81 0.39 4.5 0.018 5.3 0.78 0.36
```



# Chapter 9

## Mittelwertsvergleiche

### 9.1 Vergleich eines Stichprobenmittelwerts mit einem fixen Wert ( $\mu_0$ ) - $t$ -Test für eine Stichprobe

Manchmal möchten wir einen Stichprobenmittelwert mit einem bekannten Populationsmittelwert ( $\mu_0$ ) oder mit einem sonstigen fixen Vergleichswert vergleichen. Z.B. möchten wir wissen, ob sich die von den Jugendlichen angegebene Emotionale Unterstützung durch ihre Freunde `unt_freunde` signifikant von dem Mittelpunkt der 7er-Skala ( $\mu_0 = 4$ ) unterscheidet.

Zuerst bilden wir uns wieder einen Subdatensatz, der die beiden Unterstützungsskalen sowie einige Faktoren (für spätere Analysen) beinhaltet.

```
library(tidyverse)
load(file = "data/westost_skalen.Rdata")
support <- westost_skalen %>%
 select(ID,
 starts_with("unt"),
 westost, geschlecht,
 ends_with("_b"))

names(support)
#> [1] "ID" "unt_eltern" "unt_freunde"
#> [4] "westost" "geschlecht" "bildung_vater_b"
#> [7] "bildung_mutter_b"
```

Die benötigte Funktion ist `t.test()`. Für den Einstichproben- $t$ -Test benötigen wir nur die folgenden Argumente (mit default values): `alternative = c("two.sided", "less", "greater")`, `mu = 0`, und `conf.level = .95`.

```
TTest <- support$unt_freunde %>%
 t.test(alternative = 'two.sided',
 mu = 4,
 conf.level = .95)

TTest
#>
#> One Sample t-test
#>
#> data: .
#> t = 33.781, df = 284, p-value < 2.2e-16
```

```
#> alternative hypothesis: true mean is not equal to 4
#> 95 percent confidence interval:
#> 5.815396 6.040043
#> sample estimates:
#> mean of x
#> 5.927719
```

Der Output der `t.test()` Funktion ist eine Liste mit der Objektklasse "htest" (hypothesis test).

```
typeof(TTest)
#> [1] "list"
attributes(TTest)
#> $names
#> [1] "statistic" "parameter" "p.value" "conf.int" "estimate"
#> [6] "null.value" "alternative" "method" "data.name"
#>
#> $class
#> [1] "htest"
```

Die Elemente dieser Liste können einzeln ausgewählt werden. Den (geschätzten) Mittelwert erhalten wir mit:

```
TTest$estimate
#> mean of x
#> 5.927719
```

Den  $p$ -Wert bekommen wir mit:

```
TTest$p.value
#> [1] 1.75588e-101
```

Die Freiheitsgrade mit:

```
TTest$parameter
#> df
#> 284
```

Und das Konfidenzintervall mit:

```
TTest$conf.int
#> [1] 5.815396 6.040043
#> attr("conf.level")
#> [1] 0.95
```

Das Ergebnis zeigt, dass sich der Stichprobenmittelwert = 5.93 vom Skalenmittelpunkt = 4 hochsignifikant unterscheidet.

Im Beispiel wird die ungerichtete Alternativhypothese ( $H_1 : \mu \neq \mu_0$ ) überprüft. Bei gerichteten Alternativhypotesen ( $H_1 : \mu > \mu_0$  oder  $H_1 : \mu < \mu_0$ ) gibt man `alternative = "greater"` bzw. `alternative = "less"` an, um den Test einseitig durchzuführen.

## 9.2 Vergleich zweier Stichprobenmittelwerte

### 9.2.1 Zwei unabhängige Stichproben

#### *t*-Test für unabhängige Stichproben

Den *t*-Test für zwei unabhängige Stichproben bekommen wir auch mit der Funktion `t.test()`. In diesem Fall erfolgt die Angabe der Testvariablen und des Faktors aber in der für R typischen Formelschreibweise:

Variable ~ Faktor ( $\Rightarrow$  “Variable by Faktor” = “Testvariable wird von Faktor vorhergesagt” = “Mittelwerteunterschied der Testvariablen zwischen den beiden Stufen des Faktors”).

- Wir verwenden hier die Formelschreibweise, es gibt aber auch die Möglichkeit, zwei Vektoren/Variablen anzugeben, die die Werte der beiden Gruppen beinhalten (s.u.). ■

Die Datenmatrix muss extra angegeben werden (`data =`) und es muss eine Angabe zur Homoskedastizität der Testvariablen gemacht werden: `var.equal = FALSE/TRUE` (`FALSE` ist default). Bei `var.equal = FALSE` wird der Welch-Two-Sample-*t*-test (für Heteroskedastizität korrigierter *t*-Test) durchgeführt.

Auch hier kann die Art der Alternativhypothese entsprechend spezifiziert werden. Wird eine gerichtete Alternativhypothese spezifiziert, muss die Kodierreihenfolge der Faktorstufen berücksichtigt werden. Wir nehmen z.B. an, dass Mädchen eine höhere Emotionale Unterstützung durch die Freunde (`unt_freunde`) berichten als Jungen. Da bei `geschlecht` die Reihenfolge der Faktorstufen = "maennlich" "weiblich" ist, muss die Alternativhypothese als  $H_1 : \mu_{maennlich} < \mu_{weiblich}$  formuliert werden (und nicht als  $\mu_{weiblich} > \mu_{maennlich}$ ).

Man bezeichnetet die Form  $H_1 : \mu_{maennlich} < \mu_{weiblich}$  auch als **Gleichheitsform** und die Form  $H_1 : \mu_{maennlich} - \mu_{weiblich} < 0$  als **Nullform** der Alternativhypothese.

Das benötigte Argument `alternative = "less"` bezieht sich also auf die Nullform der Alternativhypothese, im vorliegenden Fall auf  $H_1 : \mu_{maennlich} - \mu_{weiblich} < 0$ . Es wird getestet, ob die Differenz der Mittelwerte  $< 0$  ist.

Zu Übungszwecken gehen wir zunächst einmal davon aus, dass in der Population die Varianzen von `unt_freunde` bei Jungen und Mädchen gleich gross sind und führen daher den normalen *t*-Test durch:

```
levels(support$geschlecht)
#> [1] "maennlich" "weiblich"
t.test(unt_freunde ~ geschlecht, alternative = "less",
 conf.level = .95,
 var.equal = TRUE,
 data = support)

#>
#> Two Sample t-test
#>
#> data: unt_freunde by geschlecht
#> t = -6.9351, df = 283, p-value = 1.379e-11
#> alternative hypothesis: true difference in means is less than 0
#> 95 percent confidence interval:
#> -Inf -0.5598577
#> sample estimates:
#> mean in group maennlich mean in group weiblich
#> 5.584868 6.319549
```

Um die Homoskedastizitätsannahme zu überprüfen, führen wir nun noch den Levene-Test mit der Funktion `leveneTest()` aus dem Package `car` durch. Zusätzlich zur Testvariablen und zum Faktor müssen wir angeben, welches Lagemass (Mittelwert oder Median) für die Berechnung der Abweichungen gelten soll: bei `center = mean` wird der (eigentliche) Levene-Test durchgeführt, bei `center = median` (default) der Brown-Forsythe-Test, eine gegenüber verschiedenen Formen von Nicht-Normalität robustere Variante des Levene-Tests, der gleichzeitig eine gute Teststärke aufweist.

```
library(car)
leveneTest(support$unt_freunde, support$geschlecht,
 center = median)

#> Levene's Test for Homogeneity of Variance (center = median)
#> Df F value Pr(>F)
```

```
#> group 1 11.987 0.0006185 ***
#> 283
#> ---
#> Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

■ Da die Nullhypothese beim Levene-Test die Wunschhypothese ist (wir “wollen” ja, dass die Voraussetzung der Varianzgleichheit erfüllt ist), wird oft ein grösserer Fehler 1. Art als sonst üblich verwendet, z.B.  $\alpha = .10$  oder sogar  $\alpha = .20$ . Dieses Vorgehen ist in diesem Fall *konservativer*, da die Wunsch-Nullhypothese der Homoskedastizität auf diese Weise bereits früher verworfen wird, d.h. dass die Wahrscheinlichkeit einer fälschlichen Verwerfung der Nullhypothese höher als üblich ( $\alpha = .05$ ) ist und damit das Risiko einer fälschlichen Beibehaltung derselben ( $\beta$ -Fehler) kleiner wird. ■

■ Dieses Vorgehen ist allerdings nur ein ungefähres. Eigentlich wäre es hier notwendig, den  $\beta$ -Fehler auf einen bestimmten Wert festzulegen (z.B. auf  $\beta = 0.05$ ). Dies würde allerdings eine a-priori-Power-Analyse erfordern, für die auch eine Effektstärke, also das unter der Alternativhypothese erwartbare Ausmass der Unterschiedlichkeit der Varianzen festgelegt werden müsste, was in den meisten Fällen eher schwierig sein dürfte. Ausserdem müsste man dann eine bestimmte optimale Stichprobengrösse für die Untersuchung realisieren. ■

Der Levene-Test ist signifikant, die Nullhypothese der Homoskedastizität muss also abgelehnt werden. Daher müssen wir den *t*-Test nochmals als Welch-Test durchführen.

```
t.test(unt_freunde ~ geschlecht, alternative = "less",
 conf.level = .95, var.equal = FALSE,
 data = support)

#>
#> Welch Two Sample t-test
#>
#> data: unt_freunde by geschlecht
#> t = -7.0735, df = 275.37, p-value = 6.251e-12
#> alternative hypothesis: true difference in means is less than 0
#> 95 percent confidence interval:
#> -Inf -0.5632619
#> sample estimates:
#> mean in group männlich mean in group weiblich
#> 5.584868 6.319549
```

### Wilcoxon-Rangsummentest (für zwei unabhängige Stichproben)

Eine nichtparametrische Alternative für den *t*-Test für unabhängige Stichproben ist der Wilcoxon-Rangsummentest. Er testet, ob sich die gemittelte Summe der über die Gesamtstichprobe ermittelten Ränge (und damit die Mediane) der beiden (Teil-)Stichproben signifikant voneinander unterscheiden.

■ Obwohl der Test auf die Rangwerte angewendet wird, muss die zugrunde liegende Variable intervallskaliert sein. Der Wilcoxon-Rangsummentest ist äquivalent zum Mann-Whitney-U-Test. Da (auch) der Median ein erwartungstreuer Schätzer des Populationsmittelwertes ist, prüft der Test im Endeffekt, ob sich die Mittelwerte signifikant voneinander unterscheiden. ■

Wir verwenden die Funktion `wilcox.exact()` aus dem Package `exactRankTests`, da diese im Gegensatz zu anderen Funktionen auch bei Vorliegen von Rangbindungen (gleiche Messwerte) exakte *p*-Werte berechnen kann (eine äquivalente Funktion ist `independence_test()` aus dem Package `coin`).

Als Beispiel vergleichen wir jetzt die *Mediane* von Jungen und Mädchen bezüglich der Variable “Emotionale Unterstützung durch Eltern” `unt_eltern`. Da wir die ungerichtete Alternativhypothese testen wollen und

`alternative = "two.sided"` default ist, geben wir das Argument gar nicht an. Wir geben außerdem `exact = FALSE` an, da bei einer Stichprobe von  $n = 286$  ein exakter Test einen hohen Rechenaufwand bedeuten würde und zudem nicht notwendig ist (bei einem solch grossen  $n$  unterscheiden sich der exakte und der asymptotische  $p$ -Wert nur minimal).

Zunächst installieren wir das benötigte Package: `install.packages("exactRankTests")`

Durchführung des Wilcoxon-Rangsummentests (Teststatistik  $W$  entspricht dem Mann-Whitney- $U$ ):

```
library(exactRankTests)
#> Package 'exactRankTests' is no longer under development.
#> Please consider using package 'coin' instead.
wilcox.exact(unt_eltern ~ geschlecht,
 data = support,
 exact = FALSE)

#>
#> Asymptotic Wilcoxon rank sum test
#>
#> data: unt_eltern by geschlecht
#> W = 10380, p-value = 0.7036
#> alternative hypothesis: true mu is not equal to 0
```

Solange wir wie hier keinen exakten Test (mit Rangbindungen) benötigen, können wir auch auf die Funktion `wilcox.test()` aus dem `stats`-Package zurückgreifen. Hier müssen wir allerdings noch `correct = FALSE` angeben, da sonst ein approximativer Test mit Kontinuitätskorrektur ausgegeben wird:

```
wilcox.test(unt_eltern ~ geschlecht,
 data = support,
 exact = FALSE,
 correct = FALSE)

#>
#> Wilcoxon rank sum test
#>
#> data: unt_eltern by geschlecht
#> W = 10380, p-value = 0.7036
#> alternative hypothesis: true location shift is not equal to 0
```

Zum Vergleich der  $t$ -Test:

```
t.test(unt_eltern ~ geschlecht, alternative = 'two.sided',
 conf.level = .95, var.equal = TRUE,
 data = support)

#>
#> Two Sample t-test
#>
#> data: unt_eltern by geschlecht
#> t = 1.303, df = 283, p-value = 0.1936
#> alternative hypothesis: true difference in means is not equal to 0
#> 95 percent confidence interval:
#> -0.07868849 0.38688920
#> sample estimates:
#> mean in group männlich mean in group weiblich
#> 5.94415 5.79005
```

Der exakte Wilcoxon-Rangsummentest ist insbesondere bei sehr kleinen Stichproben nützlich:

```

x <- c(8.7, 4.5, 6.4, 8.7, 5.6, 6.2)
y <- c(8.6, 8.7, 9.8, 9.5, 8.3)
wilcox.exact(x, y, exact = TRUE)
#>
#> Exact Wilcoxon rank sum test
#>
#> data: x and y
#> W = 5, p-value = 0.08225
#> alternative hypothesis: true mu is not equal to 0

wilcox.exact(x, y, exact = FALSE)
#>
#> Asymptotic Wilcoxon rank sum test
#>
#> data: x and y
#> W = 5, p-value = 0.06539
#> alternative hypothesis: true mu is not equal to 0

t.test(x, y, alternative = "two.sided",
 conf.level = .95,
 var.equal = TRUE)
#>
#> Two Sample t-test
#>
#> data: x and y
#> t = -2.8432, df = 9, p-value = 0.0193
#> alternative hypothesis: true difference in means is not equal to 0
#> 95 percent confidence interval:
#> -4.1240042 -0.4693291
#> sample estimates:
#> mean of x mean of y
#> 6.683333 8.980000

```

Bei diesem Beispiel wird der exakte Wilcoxon-Rangsummentest *nicht* signifikant ( $p = 0.082$ ), beim asymptotischen nähert sich der  $p$ -Wert dem Signifikanzkriterium ( $p = 0.065$ ), und beim (hier wegen der kleinen Stichprobengröße und nicht gegebener Normalverteilung nicht zulässigen)  $t$ -Test ergibt sich ein signifikantes Ergebnis ( $p = 0.019$ ).

- Die Art und Weise der Datenzuweisung mit zwei Vektoren ( $x$  und  $y$ ), die die Daten für die beiden Gruppen repräsentieren, ist eine Alternative zur Formelangabe (sowohl beim Wilcoxon-Rangsummentest als auch beim  $t$ -Test). ■

## 9.2.2 Zwei abhängige Stichproben

### $t$ -Test für abhängige Stichproben

Den  $t$ -Test für abhängige Stichproben erhält man wiederum mit der Funktion `t.test()`.

Als Beispiel möchten wir überprüfen, ob sich das Ausmass der von den Jugendlichen berichteten emotionalen Unterstützung durch die Eltern `unt_eltern` im Mittel vom Ausmass der emotionalen Unterstützung durch die Freunde `unt_freunde` unterscheidet. Es gibt nun zwei Möglichkeiten: Entweder wir belassen den Datensatz im wide Format, wo wir `unt_eltern` und `unt_freunde` als zwei Variablen (Spalten) im Datensatz nebeneinander stehen haben. In diesem Fall erfolgt die Datenzuweisung für die Funktion `t.test()` nicht über die Formelschreibweise, sondern über die Angabe der zu vergleichenden Variablen. Der einzige Unterschied zum  $t$ -Test für unabhängige Stichproben (mit dieser Art der Datenzuweisung) ist, dass wir die Option `paired = TRUE` angeben (default: `paired = FALSE`) und damit auf die Abhängigkeit der Daten in

den beiden Stichproben/Variablen verweisen.

- Angaben zur Homoskedastizität entfallen: beim *t*-Test für abhängige Stichproben wird de facto der Mittelwert der Differenzvariable (für jede Person wird die Differenz zwischen den beiden Variablen gebildet) gegen den Wert 0 getestet, daher gibt es nur noch eine Varianz (diejenige der Differenzvariable). ■

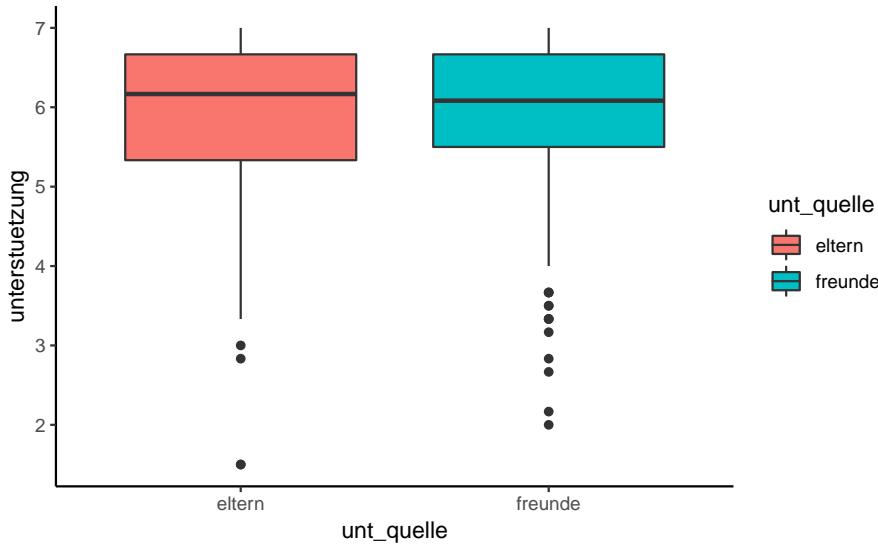
```
t.test(support$unt_eltern, support$unt_freunde,
 alternative = "two.sided",
 conf.level = .95,
 paired = TRUE)
#>
#> Paired t-test
#>
#> data: support$unt_eltern and support$unt_freunde
#> t = -0.73804, df = 283, p-value = 0.4611
#> alternative hypothesis: true difference in means is not equal to 0
#> 95 percent confidence interval:
#> -0.21046776 0.09567902
#> sample estimates:
#> mean of the differences
#> -0.05739437
```

Auch hier haben wir wieder die Möglichkeit, gerichtete Alternativhypotesen ( $H_1 : \mu_1 > \mu_2$  bzw.  $\mu_1 - \mu_2 > 0$  oder  $H_1 : \mu_1 < \mu_2$  bzw.  $\mu_1 - \mu_2 < 0$ ) einseitig zu testen. Für den ersten Fall gibt man `alternative = "greater"`, für den zweiten `alternative = "less"` an.

Die zweite Möglichkeit besteht darin, den Datensatz von *wide* zu *long* zu konvertieren und damit zu berücksichtigen, dass es sich bei *Eltern* und *Freunden* eigentlich um zwei Stufen eines Faktors handelt, den wir als *Unterstützungsquelle* (`unt_quelle`) bezeichnen können. Die gemessene Variable ist (für beide Faktorstufen) *unterstuetzung*. (Die Items der Unterstützungsskala, die `unt_eltern` und `unt_freunde` zugrunde liegen, sind jeweils die gleichen, nur die Bezugnahme auf Eltern bzw. Freunde unterscheidet sich.)

```
library(stringr)
sup_long <- support %>%
 drop_na(unt_eltern, unt_freunde) %>%
 select(-ID, -unt_eltern, -unt_freunde) %>%
 gather(key = unt_quelle, value = unterstuetzung, -ID) %>%
 # mit str_replace unt_ entfernen
 mutate(unt_quelle = str_replace(unt_quelle, ".*_", ""))
 mutate(unt_quelle = factor(unt_quelle))

library(ggplot2)
sup_long %>%
 ggplot(aes(x = unt_quelle, y = unterstuetzung, fill = unt_quelle)) +
 geom_boxplot() +
 theme_classic()
```



```

sup_long %>%
 group_by(unt_quelle) %>%
 summarize(mean = mean(unterstuetzung, na.rm = TRUE),
 N = length(unterstuetzung))
#> # A tibble: 2 x 3
#> unt_quelle mean N
#> <fct> <dbl> <int>
#> 1 eltern 5.87 284
#> 2 freunde 5.93 284

t.test(unterstuetzung ~ unt_quelle,
 alternative = "two.sided",
 conf.level = .95,
 paired = TRUE,
 data = sup_long)
#>
#> Paired t-test
#>
#> data: unterstuetzung by unt_quelle
#> t = -0.73804, df = 283, p-value = 0.4611
#> alternative hypothesis: true difference in means is not equal to 0
#> 95 percent confidence interval:
#> -0.21046776 0.09567902
#> sample estimates:
#> mean of the differences
#> -0.05739437

```

- Übrigens kann ein *t*-Test für zwei abhängige Stichproben auch als Einstichproben-*t*-Test einer Differenzvariablen (gegen  $\mu = 0$ ) formuliert werden. Das funktioniert am besten mit dem wide Datensatz. Die Differenzvariable kann direkt als Datenargument in `t.test()` gebildet werden:

```

t.test(support$unt_eltern - support$unt_freunde, mu = 0)
#>
#> One Sample t-test
#>
#> data: support$unt_eltern - support$unt_freunde
#> t = -0.73804, df = 283, p-value = 0.4611

```

```
#> alternative hypothesis: true mean is not equal to 0
#> 95 percent confidence interval:
#> -0.21046776 0.09567902
#> sample estimates:
#> mean of x
#> -0.05739437
```

### Wilcoxon-Vorzeichen-Rangtest (für zwei abhängige Stichproben)

Eine nichtparametrische Alternative für den *t*-Test für abhängige Stichproben ist der Wilcoxon-Vorzeichen-Rangtest. Er testet, ob sich die mittleren Rangsummen der negativen und der positiven Differenzen der beiden Variablen signifikant voneinander unterscheiden. Es wird also wie beim *t*-Test für abhängige Stichproben zuerst eine Differenzvariable gebildet, dann werden Ränge für die Beträge der Differenzen vergeben und anschliessend nach Vorzeichen getrennt die mittleren Rangsummen ermittelt. Auch für diesen Test gibt es einen exakten und einen approximativen Test.

Wir verwenden wieder die Funktion `wilcox.exact()` aus dem Package `exactRankTests`, da diese im Gegensatz zu anderen Funktionen auch bei Vorliegen von Rangbindungen (gleiche Differenzbeträge) exakte *p*-Werte berechnen kann. Der einzige Unterschied zum Wilcoxon-Rangsummentest (für zwei **unabhängige** Stichproben) ist, dass wir das Argument `paired = TRUE` verwenden müssen. Auf den exakten Test verzichten wir hier wieder aufgrund der grossen Stichprobe. Die Teststatistik *V* spiegelt die Rangsumme der positiven Differenzen wider.

```
wilcox.exact(support$unt_eltern, support$unt_freunde,
 alternative = "two.sided",
 paired = TRUE,
 exact = FALSE)
#>
#> Asymptotic Wilcoxon signed rank test
#>
#> data: support$unt_eltern and support$unt_freunde
#> V = 16477, p-value = 0.9335
#> alternative hypothesis: true mu is not equal to 0
```

Auch hier können wir alternativ mit dem long Datensatz und der Formelschreibweise arbeiten:

```
wilcox.exact(unterstuetzung ~ unt_quelle,
 alternative = "two.sided",
 paired = TRUE,
 exact = FALSE,
 data = sup_long)
#>
#> Asymptotic Wilcoxon signed rank test
#>
#> data: unterstuetzung by unt_quelle
#> V = 16477, p-value = 0.9335
#> alternative hypothesis: true mu is not equal to 0
```

■ Machen Sie sich mit den oben beschriebenen Verfahren zum Vergleich zweier Mittelwerte vertraut, indem Sie für dieselben Daten verschiedene Argumente ausprobieren (z.B. `var.equal = c(TRUE, FALSE)`, `alternative = c("two.sided", "greater", "lesser")`, `conf.level = c(.90, .95)`, `exact = c(TRUE, FALSE)`). Beachten Sie, dass das Argument `paired = c(TRUE, FALSE)` nur Sinn ergibt, wenn Sie

es wirklich mit abhängigen (TRUE) bzw. unabhängigen (FALSE) Stichproben zu tun haben. (Hinweis: Das Argument `exact = c(TRUE, FALSE)` existiert nur beim `wilcox.exact()`-Test.) ■

## 9.3 Vergleich mehrerer Stichprobenmittelwerte

### 9.3.1 Mehrere unabhängige Stichproben

#### Einfaktorielle Varianzanalyse (ANOVA)

Wir führen eine Oneway-ANOVA mit dem Gesamtnotenschnitt (Schnitt) als abhängiger Variable und der Bildung des Vaters (bildung\_vater) als Faktor (UV) durch. Die Nullhypothese ist, dass sich der Gesamtnotenschnitt zwischen Gruppen von Jugendlichen, deren Väter sich durch unterschiedliche Bildungsniveaus auszeichnen, nicht unterscheidet.

Dazu bilden wir zuerst einen Datensatz, der nur die Variablen ID, Schnitt, bildung\_vater und geschlecht (für spätere Analyse) enthält und entfernen alle Personen mit fehlenden Werten.

```
noten <- westost_skalen %>%
 select(ID, Schnitt, bildung_vater, geschlecht) %>%
 drop_na()
noten
#> # A tibble: 264 x 4
#> ID Schnitt bildung_vater geschlecht
#> <fct> <dbl> <fct> <fct>
#> 1 2 4 Fachabitur, Abitur männlich
#> 2 14 4 Realschulabschluss (mittlere Reife) männlich
#> 3 15 4 Realschulabschluss (mittlere Reife) männlich
#> 4 17 4 Realschulabschluss (mittlere Reife) männlich
#> 5 18 4 Fachabitur, Abitur männlich
#> 6 19 4 Hauptschulabschluss oder niedriger männlich
#> # ... with 258 more rows
```

Da die Bezeichnungen (Labels) der Faktorstufen sehr lang sind und das bei der Darstellung unangenehme Folgen hat, führen wir noch ein Relabelling der Faktorstufen durch:

```
table(noten$bildung_vater)
#>
#> Hauptschulabschluss oder niedriger 41
#> Realschulabschluss (mittlere Reife) 93
#> Fachabitur, Abitur 39
#> Fachhochschulabschluss, Universitätsabschluss 91
levels(noten$bildung_vater) <- c("Hauptschule", "Realschule", "Abitur", "Hochschule")
da es für das Online-Skript ein Problem mit den Umlauten gab, wird auch das noch korrigiert:
levels(noten$geschlecht) <- c("maennlich", "weiblich")
table(noten$bildung_vater)
#>
#> Hauptschule Realschule Abitur Hochschule
#> 41 93 39 91
```

Dann berechnen wir die ANOVA über die `aov`-Funktion (`analysis of variance`) und lassen uns anschliessend die Summary zu dem Ergebnisobjekt ausgeben.

Zusätzlich wollen wir über die Funktion `TukeyHSD` alle paarweisen Vergleiche (adjustierte Konfidenzintervalle

und  $p$ -Werte nach Tukey) erhalten. Bevor wir das tun, sollten wir noch sicher stellen, dass Varianzhomogenität gegeben ist. Dies können wir wie oben mit der Funktion `leveneTest` aus dem Package `car` machen. Deskriptive Statistiken zu den Mittelwerten, Standardabweichungen und Varianzen lassen wir uns außerdem zuvor mit der bekannten `dplyr`-Funktion ausgeben:

```
noten %>%
 group_by(bildung_vater) %>%
 summarize(mean = round(mean(Schnitt, na.rm = TRUE), 2),
 sd = round(sd(Schnitt, na.rm = TRUE), 3),
 var = round(var(Schnitt, na.rm = TRUE), 3),
 N = length(Schnitt))

#> # A tibble: 4 x 5
#> bildung_vater mean sd var N
#> <fct> <dbl> <dbl> <dbl> <int>
#> 1 Hauptschule 4.05 0.669 0.448 41
#> 2 Realschule 4.38 0.655 0.429 93
#> 3 Abitur 4.5 0.653 0.427 39
#> 4 Hochschule 4.69 0.683 0.466 91
```

Rein deskriptiv sehen wir, dass höhere Bildungsabschlüsse des Vaters mit einem höheren (und damit besseren, da ins Schweizer Notensystem umkodiert) Gesamtnotenschnitt der Jugendlichen einhergehen. Die Varianzen und Standardabweichungen sehen dagegen relativ homogen aus.

```
library(car)
leveneTest(noten$Schnitt, noten$bildung_vater,
 center = median)

#> Levene's Test for Homogeneity of Variance (center = median)
#> Df F value Pr(>F)
#> group 3 0.4906 0.6891
#> 260
```

Das Ergebnis des Levene-Tests bestätigt, dass sich keine signifikanten Unterschiede der Varianzen des Notenschnitts zwischen den Vaterbildungsgruppen zeigen. Die Voraussetzung der Varianzhomogenität ist also gegeben.

```
Anova.model1 <- aov(Schnitt ~ bildung_vater, data = noten)
summary(Anova.model1)

#> Df Sum Sq Mean Sq F value Pr(>F)
#> bildung_vater 3 12.36 4.120 9.272 7.58e-06 ***
#> Residuals 260 115.53 0.444
#> ---
#> Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
TukeyHSD(Anova.model1, ordered = TRUE)

#> Tukey multiple comparisons of means
#> 95% family-wise confidence level
#> factor levels have been ordered
#>
#> Fit: aov(formula = Schnitt ~ bildung_vater, data = noten)
#>
#> $bildung_vater
#> diff lwr upr p adj
#> Realschule-Hauptschule 0.3269866 0.003864173 0.6501091 0.0461134
#> Abitur-Hauptschule 0.4437774 0.058237843 0.8293169 0.0167437
#> Hochschule-Hauptschule 0.6397481 0.315540988 0.9639551 0.0000039
#> Abitur-Realschule 0.1167907 -0.212031612 0.4456131 0.7950495
#> Hochschule-Realschule 0.3127614 0.058608643 0.5669142 0.0088548
```

```
#> Hochschule-Abitur 0.1959707 -0.133917530 0.5258589 0.4174570
```

Der *F*-Test der Varianzanalyse ist hoch signifikant ( $p < 0.001$ ) und die Post-hoc-Einzelvergleiche mittels Tukey-Test zeigen, dass sich die Mittelwerte des Gesamtnotenschnitts zwischen allen höheren Bildungsabschluss-Gruppen und der niedrigsten Kategorie **Hauptschule** signifikant voneinander unterscheiden ( $p = 0.046$ ,  $p = 0.017$ ,  $p < 0.001$ ), sowie zwischen den Gruppen **Hochschule** und **Realschule** ( $p = 0.009$ ). Die übrigen beiden Mittelwertsvergleiche sind dagegen nicht signifikant.

Wir können die ANOVA alternativ auch mit der Funktion `ezANOVA()` aus dem Package `ez` durchführen:

```
install.packages("ez")
```

Es muss eine Vpnr-Variable angegeben werden (hier: ID) mit dem Argument `wid = ID`. Die Funktion `white.adjust = FALSE` verhindert die Durchführung einer für Heteroskedastizität angepassten ANOVA. Der Levene-Test auf Homoskedastizität wird mit ausgegeben, man kann diese Option bei Nichtvorliegen von Homoskedastizität dann ggf. ändern.

```
library(ez)
ezANOVA(noten, dv = Schnitt, wid = ID,
 between = bildung_vater,
 white.adjust = FALSE,
 detailed = TRUE)
#> Coefficient covariances computed by hccm()
#> $ANOVA
#> Effect DFn DFd SSn SSd F p p<.05
#> 1 bildung_vater 3 260 12.3607 115.5329 9.272342 7.575436e-06 *
#> ges
#> 1 0.0966483
#>
#> $`Levene's Test for Homogeneity of Variance`
#> DFn DFd SSn SSd F p p<.05
#> 1 3 260 0.4063397 71.78362 0.4905869 0.6891123
```

Neben der `TukeyHSD()`-Funktion, die auf das ANOVA-Outcome-Objekt angewendet wird gibt es noch eine andere grundlegende Möglichkeit, Post-hoc Paarvergleiche zu erhalten: über die Funktion `pairwise.t.test()` aus `stats`. Zur Verfügung stehen unter anderem die Methoden `p.adjust.method = c("bonferroni", "holm")`.

```
pairwise.t.test(noten$Schnitt, noten$bildung_vater,
 p.adjust.method = "holm", data = noten)
#>
#> Pairwise comparisons using t tests with pooled SD
#>
#> data: noten$Schnitt and noten$bildung_vater
#>
#> Hauptschule Realschule Abitur
#> Realschule 0.0282 - -
#> Abitur 0.0128 0.3593 -
#> Hochschule 3.9e-06 0.0082 0.2515
#>
#> P value adjustment method: holm
```

Eine weitere (ganz neue) Möglichkeit, eine einfaktorielle Varianzanalyse mit einer Reihe von Optionen durchzuführen, bietet die Funktion `anova()` aus dem Package `jmv` (gehört zum Statistik-Programm **jamovi**). Hier können wir sowohl den Levene-Test, verschiedene Post-hoc-Einzelvergleiche (noch nicht im Skript) sowie Effektgrößen erhalten. Außerdem kann ein Mittelwertdiagramm inkl. Konfidenzintervalle (alternativ: Standardfehler) mit ausgegeben werden.

```

install.packages("jmv")
library(jmv)
ANOVA(noten, dep = "Schnitt", factors = "bildung_vater",
 effectSize = c("eta", "omega"),
 homo = TRUE)
#>
#> ANOVA
#>
#> ANOVA
#>
#> Sum of Squares df Mean Square F p
#>
#> bildung_vater 12.4 3 4.120 9.27 < .001 0.097 0.086
#> Residuals 115.5 260 0.444
#>
#>
#> ASSUMPTION CHECKS
#>
#> Test for Homogeneity of Variances (Levene's)
#>
#> F df1 df2 p
#>
#> 0.603 3 260 0.613
#>

```

### Kruskal-Wallis-Test ( $H$ -Test)

Der  $H$ -Test ist eine nicht-parametrische Alternative zur einfaktoriellen Varianzanalyse. Er kann als Erweiterung des Wilcoxon-Rangsummentest auf mehrere Stichproben verstanden werden. Der Kruskal-Wallis-Test wird meist asymptotisch durchgeführt, da ein exakter Test (bei etwas grösseren Stichproben) viel Rechenleistung erfordert und meist nicht notwendig ist. Getestet wird die Gleichheit der Mediane in den Stichproben.

Zuerst lassen wir uns die Mediane mit der bekannten dplyr-Methode ausgeben:

```

noten %>%
 group_by(bildung_vater) %>%
 summarise(median = median(Schnitt))
#> # A tibble: 4 x 2
#> bildung_vater median
#> <fct> <dbl>
#> 1 Hauptschule 4
#> 2 Realschule 4
#> 3 Abitur 4.6
#> 4 Hochschule 5

```

Zum Vergleich nochmal die traditionelle tapply()-Funktion zur Darstellung der Mediane:

```

tapply(noten$Schnitt, noten$bildung_vater,
 median, na.rm = TRUE)
#> Hauptschule Realschule Abitur Hochschule
#> 4.0 4.0 4.6 5.0

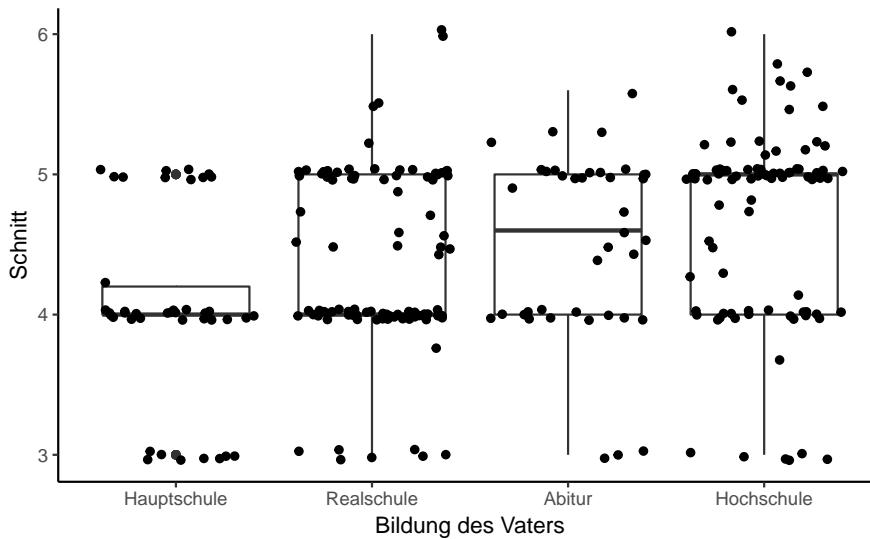
```

Dann führen wir den Kruskal-Wallis-Test durch:

```
kruskal.test(Schnitt ~ bildung_vater, data = noten)
#>
#> Kruskal-Wallis rank sum test
#>
#> data: Schnitt by bildung_vater
#> Kruskal-Wallis chi-squared = 29.008, df = 3, p-value = 2.231e-06
```

Zur Veranschaulichung der Median-Unterschiede eignet sich auch ein Box-Plot:

```
noten %>%
 ggplot(aes(x = bildung_vater, y = Schnitt)) +
 geom_boxplot() +
 geom_jitter() +
 xlab("Bildung des Vaters") +
 theme_classic()
```



Post-hoc Paarvergleiche nach Nemenyi erhält man mit dem Package PMCMR.

```
install.packages("PMCMR")
library(PMCMR)
#> PMCMR is superseded by PMCMRplus and will be no longer maintained. You may wish to install PMCMRplus
posthoc.kruskal.nemenyi.test(x = noten$Schnitt,
 g = noten$bildung_vater,
 dist = "Tukey")
#>
#> Pairwise comparisons using Tukey and Kramer (Nemenyi) test
#> with Tukey-Dist approximation for independent samples
#>
#> data: noten$Schnitt and noten$bildung_vater
#>
#> Hauptschule Realschule Abitur
#> Realschule 0.1663 - -
#> Abitur 0.0375 0.6818 -
#> Hochschule 8.6e-06 0.0024 0.3740
#>
#> P value adjustment method: none
```

Der Kruskal-Wallis-Test lässt sich auch mit der Funktion `anovaNP()` aus dem Package `jmv` durchführen:

```
library(jmv)
anovaNP(noten, deps = "Schnitt", group = "bildung_vater",
 pairs = TRUE)
#>
#> ONE-WAY ANOVA (NON-PARAMETRIC)
#>
#> Kruskal-Wallis
#>
#> ^ df p
#>
#> Schnitt 29.0 3 < .001
#>
#>
#> DWASS-STEEL-CRITCHLOW-FLIGNER PAIRWISE COMPARISONS
#>
#> Pairwise comparisons - Schnitt
#>
#> W p
#>
#> Hauptschule Realschule 3.38 0.017
#> Hauptschule Abitur 4.16 0.003
#> Hauptschule Hochschule 6.76 < .001
#> Realschule Abitur 1.75 0.216
#> Realschule Hochschule 5.31 < .001
#> Abitur Hochschule 2.57 0.069
#>
```

Die  $\alpha$ -Adjustierung nach Dwass-Steel-Critchlow-Fligner ist deutlich weniger konservativ als die nach Nemenyi. Wir kommen aber mit Ausnahme des Vergleichs zwischen Realschule und Hauptschule zur gleichen Signifikanzentscheidung.

### Zweifaktorielle Varianzanalyse

Auch die zweifaktorielle Varianzanalyse lässt sich auf die oben beschriebenen Arten durchführen. Wir wollen nun untersuchen, ob sich der Notendurchschnitt in Abhängigkeit von der Bildung des Vaters sowie des Geschlechts unterscheidet und wie diese beiden Faktoren ggf. interagieren.

Zunächst lassen wir uns wieder deskriptive Statistiken mit `dplyr` ausgeben:

```
noten %>%
 group_by(bildung_vater, geschlecht) %>%
 summarize(mean = round(mean(Schnitt, na.rm = TRUE), 2),
 sd = round(sd(Schnitt, na.rm = TRUE), 3),
 var = round(var(Schnitt, na.rm = TRUE), 3),
 N = length(Schnitt))
#> # A tibble: 8 x 6
#> # Groups: bildung_vater [4]
#> bildung_vater geschlecht mean sd var N
#> <fct> <fct> <dbl> <dbl> <dbl> <int>
#> 1 Hauptschule maennlich 3.86 0.64 0.409 22
#> 2 Hauptschule weiblich 4.27 0.651 0.423 19
#> 3 Realschule maennlich 4.13 0.626 0.392 48
#> 4 Realschule weiblich 4.65 0.578 0.334 45
```

```
#> 5 Abitur maennlich 4.39 0.694 0.481 21
#> 6 Abitur weiblich 4.63 0.595 0.354 18
#> # ... with 2 more rows
noten %>%
 group_by(geschlecht) %>%
 summarize(mean = round(mean(Schnitt, na.rm = TRUE), 2),
 sd = round(sd(Schnitt, na.rm = TRUE), 3),
 var = round(var(Schnitt, na.rm = TRUE), 3),
 N = length(Schnitt))
#> # A tibble: 2 x 5
#> geschlecht mean sd var N
#> <fct> <dbl> <dbl> <dbl> <int>
#> 1 maennlich 4.29 0.723 0.522 145
#> 2 weiblich 4.65 0.612 0.375 119
```

Dann führen wir den Levene-Test auf Varianzhomogenität in den vier Substichproben durch:

```
library(car)
leveneTest(noten$Schnitt ~ noten$bildung_vater * noten$geschlecht,
 center = median)
#> Levene's Test for Homogeneity of Variance (center = median)
#> Df F value Pr(>F)
#> group 7 1.1221 0.3495
#> 256
```

Zunächst führen wir die zweifaktorielle Varianzanalyse mit der traditionellen aov-Methode durch:

```
Anova.model2 <- aov(Schnitt ~ bildung_vater * geschlecht, data = noten)
Anova(Anova.model2, type = 3)
#> Anova Table (Type III tests)
#>
#> Response: Schnitt
#> Sum Sq Df F value Pr(>F)
#> (Intercept) 4315.7 1 10501.8946 < 2.2e-16 ***
#> bildung_vater 12.7 3 10.3151 1.965e-06 ***
#> geschlecht 7.3 1 17.8024 3.405e-05 ***
#> bildung_vater:geschlecht 0.8 3 0.6875 0.5604
#> Residuals 105.2 256
#> ---
#> Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

■ Ein vollständiges Modell (mit allen Haupteffekten und Interaktionen) erhält man in der Formelnotation über das Produkt der beiden Faktoren: FaktorA \* FaktorB. Dies ist gleichbedeutend mit FaktorA + FaktorB + FaktorA:FaktorB. Will man also nur die beiden Haupteffekte ohne den Interaktionseffekt, dann verbindet man die Faktoren mit + statt mit \*. Der Interaktionseffekt wird bei dieser Schreibweise durch den : repräsentiert: FaktorA:FaktorB ■

Es ist wichtig, dass wir die Zusammenfassung des ANOVA-Outcome-Objektes hier mit der Funktion Anova() aus dem car-Package anfordern. Nur so haben wir die Möglichkeit, Typ-II- oder Typ-III-Quadratsummen zu erhalten (mit dem Argument type = 2 bzw. type = 3). Die Voreinstellung und einzige Option bei summary() sind nämlich Typ-I-Quadratsummen, also eine sequentielle Partialisierung aller Effekte (vgl. auch Kapitel “Lineare Modelle”).

■ Kurz gesagt unterscheiden sich Typ-II- und Typ-III-Quadratsummen dadurch, dass sie bei der Berechnung der Haupteffekt-Quadratsummen den Interaktionseffekt auspartialisieren (Typ-III) oder

nicht auspartialisieren (Typ-II). ■

Auch mit `ezANOVA()` erhalten wir dasselbe Ergebnis:

```
library(ez)
ezANOVA(noten, dv = Schnitt, wid = ID,
 between = bildung_vater * geschlecht,
 white.adjust = FALSE,
 type = 3)
#> Coefficient covariances computed by hccm()
#> $ANOVA
#>
#> Effect DFn DFd F p p<.05
#> 2 bildung_vater 3 256 10.3151356 1.965388e-06 *
#> 3 geschlecht 1 256 17.8024414 3.404755e-05 *
#> 4 bildung_vater:geschlecht 3 256 0.6874928 5.604118e-01
#> ges
#> 2 0.107844231
#> 3 0.065019294
#> 4 0.007992167
#>
#> $`Levene's Test for Homogeneity of Variance`
#> DFn DFd SSn SSd F p p<.05
#> 1 7 256 2.070602 67.483 1.122133 0.3494653
```

Und mit `anova()` aus `jmv`:

```
library(jmv)
ANOVA(noten, dep = "Schnitt",
 factors = c("bildung_vater", "geschlecht"),
 effectSize = c("eta", "omega"),
 homo = TRUE,
 ss = 3)
#>
#> ANOVA
#>
#> ANOVA
#>
#>
#> Sum of Squares df Mean Square F p ²
#>
#> bildung_vater 12.717 3 4.239 10.315 < .001 0.101
#> geschlecht 7.316 1 7.316 17.802 < .001 0.058
#> bildung_vater:geschlecht 0.848 3 0.283 0.687 0.560 0.007
#> Residuals 105.202 256 0.411
#>
#>
#> ASSUMPTION CHECKS
#>
#> Test for Homogeneity of Variances (Levene's)
#>
#> F df1 df2 p
#>
#> 1.54 7 256 0.154
#>
```

### 9.3.2 Mehrere abhängige Stichproben

#### Einfaktorielle Varianzanalyse mit Messwiederholung

Als Beispiel betrachten wir hier die aus Statistik II bekannte (fiktive) Untersuchung zu “Honeymoon oder Hangover” (nach einer Aufgabe aus dem Übungsbuch von Budischewski & Kriens, 2012). Dabei wurden in zwei Firmen jeweils 5 Angestellte zu Beginn ihrer Anstellung (**Anfang**) sowie an zwei weiteren Messzeitpunkten (**Drei\_Monate**, **Sechs\_Monate**) nach ihrer Arbeitszufriedenheit befragt. Die Annahme war, dass man zu Beginn sehr zufrieden ist, da man mit Begeisterung an die neue Aufgabe herangeht (“Honeymoon”), dass dann nach einer Zeit aufgrund der Arbeitsbelastung bei einem Neueinstieg eine Phase geringerer Zufriedenheit folgt (“Hangover”), bevor sich das Zufriedenheitsniveau einpendelt (hier nach sechs Monaten). Generell fragen wir hier nach den Unterschieden in den Mittelwerten der Arbeitszufriedenheit über die Zeit, zunächst nur für eine Firma (Firma 1).

Für die Repeated-Measures-ANOVA muss zuerst ein Datenfile im *long-* Format erstellt werden. Wir importieren den SPSS-Datensatz `honeymoon_hangover_2Firmen.sav` und benutzen anschliessend die `gather()`-Funktion aus `tidyverse` für die Konvertierung von *wide* zu *long*.

```
library(haven)
Honeymoon <- read_sav("data/honeymoon_hangover_2Firmen.sav")
Honeymoon$ID <- as.factor(Honeymoon$ID)
Honeymoon$Firma <- as.factor(Honeymoon$Firma)
Honeymoon
#> # A tibble: 10 x 5
#> ID Firma Anfang Drei_Monate Sechs_Monate
#> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1 1 1 9 4 5
#> 2 2 1 9 4 8
#> 3 3 1 9 7 14
#> 4 4 1 10 9 5
#> 5 5 1 8 1 3
#> 6 6 2 10 10 10
#> # ... with 4 more rows

Honeymoon_long <- Honeymoon %>%
 gather(key = Messzeitpunkt, value = Arbeitszufriedenheit,
 -Firma, -ID) %>%
 mutate(Messzeitpunkt = factor(Messzeitpunkt)) %>%
 arrange(Firma, ID)

Honeymoon_long
#> # A tibble: 30 x 4
#> ID Firma Messzeitpunkt Arbeitszufriedenheit
#> <dbl> <dbl> <fct> <dbl>
#> 1 1 1 Anfang 9
#> 2 1 1 Drei_Monate 4
#> 3 1 1 Sechs_Monate 5
#> 4 2 1 Anfang 9
#> 5 2 1 Drei_Monate 4
#> 6 2 1 Sechs_Monate 8
#> # ... with 24 more rows
```

Die Bedingungsmittelwerte erhalten wir mit:

```
Honeymoon_long %>%
 group_by(Firma, Messzeitpunkt) %>%
 summarise(Bedingungsmittelwert = mean(Arbeitszufriedenheit))
```

```
#> # A tibble: 6 x 3
#> # Groups: Firma [2]
#> Firma Messzeitpunkt Bedingungsmittelwert
#> <fct> <fct> <dbl>
#> 1 1 Anfang 9
#> 2 1 Drei_Monate 5
#> 3 1 Sechs_Monate 7
#> 4 2 Anfang 10
#> 5 2 Drei_Monate 10.8
#> 6 2 Sechs_Monate 11
```

Zunächst möchten wir nur die Arbeitszufriedenheit in der ersten Firma untersuchen (d.h. wir betrachten nur *eine* Gruppe in einem Modell ohne Between-Faktor):

```
nur Firma 1 auswählen
Honeymoon_long_F1 <- Honeymoon_long %>%
 filter(Firma == 1)

Bedingungsmittelwerte für Firma 1
Honeymoon_long_F1 %>%
 group_by(Messzeitpunkt) %>%
 summarise(Bedingungsmittelwert = mean(Arbeitszufriedenheit))

#> # A tibble: 3 x 2
#> Messzeitpunkt Bedingungsmittelwert
#> <fct> <dbl>
#> 1 Anfang 9
#> 2 Drei_Monate 5
#> 3 Sechs_Monate 7
```

Die Repeated-Measures-ANOVA bekommen wir, indem wir in der Funktion `ezANOVA()` die Variable `Messzeitpunkt` als within-Faktor definieren:

```
library(ez)
ezANOVA(Honeymoon_long_F1, dv = Arbeitszufriedenheit,
 wid = ID, within = Messzeitpunkt,
 detailed = TRUE)

#> $ANOVA
#> Effect DFn DFd SSn SSd F p p<.05 ges
#> 1 (Intercept) 1 4 735 60 49.000000 0.00219213 * 0.8657244
#> 2 Messzeitpunkt 2 8 40 54 2.962963 0.10890896 0.2597403
#>
#> $`Mauchly's Test for Sphericity`
#> Effect W p p<.05
#> 2 Messzeitpunkt 0.6872428 0.569725
#>
#> $`Sphericity Corrections`
#> Effect GGe p[GG] p[GG]<.05 HFe p[HF] p[HF]<.05
#> 2 Messzeitpunkt 0.7617555 0.1308327 1.134177 0.108909
```

Da der Mauchly-Test nicht signifikant ist, können wir die Nullhypothese einer sphärischen Varianz-Kovarianz-Matrix aufrechterhalten (Sphärizitätsannahme) und müssen daher keine Korrekturen vornehmen. Der Effekt des Faktors `Messzeitpunkt` ist nicht signifikant, ein Unterschied zwischen den Mittelwerten der Arbeitszufriedenheit zwischen den drei Zeitpunkten kann also nicht nachgewiesen werden. Mit  $n = 5$  ist unsere Stichprobe natürlich sehr klein, was mit einer sehr geringen Teststärke verbunden ist. (Wir haben dieses Beispiel in Statistik II vollständig per Hand gerechnet!)

- Auch wenn wir die Korrekturen nach Greenhouse-Geisser- $\epsilon$  (GGe) oder Huynh-Feldt- $\epsilon$  (HFe) anwenden würden - was wir mit dem Argument, dass der Mauchly-Test aufgrund der geringen Stichprobengröße keine ausreichende Teststärke aufweist, aus Vorsicht tun könnten - würde sich nichts ändern: die unter *Sphericity Corrections* aufgeführten p[GGe] und p[HFe] sind beide nicht signifikant. ■

Post-hoc Tests erhalten wir über die Funktion `pairwise.t.test()` aus `stats`, für die wir hier `paired = TRUE` spezifizieren müssen. Hier benutzen wir einmal die Methode `p.adjust.method = "none"` (keine  $\alpha$ -Korrektur) und einmal als `p.adjust.method = "holm"` (Bonferroni-Holm-Korrektur).

```
pairwise.t.test(Honeymoon_long_F1$Arbeitszufriedenheit,
 Honeymoon_long_F1$Messzeitpunkt,
 p.adjust.method = "none",
 paired = TRUE,
 data = Honeymoon_long_F1)

#>
#> Pairwise comparisons using paired t tests
#>
#> data: Honeymoon_long_F1$Arbeitszufriedenheit and Honeymoon_long_F1$Messzeitpunkt
#>
#> Anfang Drei_Monate
#> Drei_Monate 0.022 -
#> Sechs_Monate 0.351 0.333
#>
#> P value adjustment method: none

pairwise.t.test(Honeymoon_long_F1$Arbeitszufriedenheit,
 Honeymoon_long_F1$Messzeitpunkt,
 p.adjust.method = "holm",
 paired = TRUE,
 data = Honeymoon_long_F1)

#>
#> Pairwise comparisons using paired t tests
#>
#> data: Honeymoon_long_F1$Arbeitszufriedenheit and Honeymoon_long_F1$Messzeitpunkt
#>
#> Anfang Drei_Monate
#> Drei_Monate 0.065 -
#> Sechs_Monate 0.665 0.665
#>
#> P value adjustment method: holm
```

Auch die einfaktorielle Varianzanalyse mit Messwiederholung ist bereits in `jmv` implementiert. Eine Auflistung der Optionen finden Sie hier. In `jmv` benötigen wir allerdings wieder einen *wide* Datensatz. Die Philosophie von `jmv` und dem dazugehörigen GUI **jamovi** ist es, die Funktionalität von SPSS möglichst gut nachzubilden, und in SPSS wird für (Standard-)Messwiederholungsverfahren ein *wide* Datensatz verwendet.

Wir wählen also aus dem ursprünglichen *wide* Datensatz `Honeymoon` zunächst nur die erste Firma aus:

```
Honeymoon_wide_F1 <- Honeymoon %>%
 filter(Firma == 1)
```

Der Code (die Syntax) der Funktion `anovaRM()` ist recht umständlich, da es deutlich komplizierter ist, die abhängige Variable und den Messwiederholungsfaktor in einem *wide* Datensatz zu definieren:

```

anovaRM(
 data = Honeymoon_wide_F1,
 rm = list(# Definition RM-Faktor / Faktorstufen
 list(
 label = "Messzeitpunkt",
 levels = c("Anfang", "Drei_Monate", "Sechs_Monate"))),
 rmCells = list(# Zuweisung der "Variablen" (measure)
 list(# zu den Zellen (Faktorstufen)
 measure = "Anfang",
 cell = "Anfang"),
 list(
 measure = "Drei_Monate",
 cell = "Drei_Monate"),
 list(
 measure = "Sechs_Monate",
 cell = "Sechs_Monate"))),
 rmTerms = list(# Aufrufen der RM-Terme, die im Modell
 "Messzeitpunkt"), # berücksichtigt werden sollen
 effectSize = c("eta", "omega"),
 spherTests = TRUE,
 spherCorr = c("none", "GG", "HF"))

#>
#> REPEATED MEASURES ANOVA
#>
#> Within Subjects Effects
#>
#> Sphericity Correction Sum of Squares df Mean Square F p
#>
#> Messzeitpunkt None 40.0 2 20.00 2.96 0.109
#> Messzeitpunkt Greenhouse-Geisser 40.0 1.52 26.26 2.96 0.131
#> Messzeitpunkt Huynh-Feldt 40.0 2.00 20.00 2.96 0.109
#>
#> Residual None 54.0 8 6.75
#> Residual Greenhouse-Geisser 54.0 6.09 8.86
#> Residual Huynh-Feldt 54.0 8.00 6.75
#>
#> Note. Type 3 Sums of Squares
#>
#> Between Subjects Effects
#>
#> Sum of Squares df Mean Square F p
#>
#> Residual 60.0 4 15.0
#>
#> Note. Type 3 Sums of Squares
#>
#> ASSUMPTIONS
#>
#> Tests of Sphericity
#>
#> Mauchly's W p Greenhouse-Geisser Huynh-Feldt

```

```
#>
#> Messzeitpunkt 0.687 0.570 0.762 1.00
#>
#> postHoc = "Messzeitpunkt",
#> postHocCorr = "tukey",
```

### Rangvarianzanalyse für verbundene Stichproben: Friedman-Test

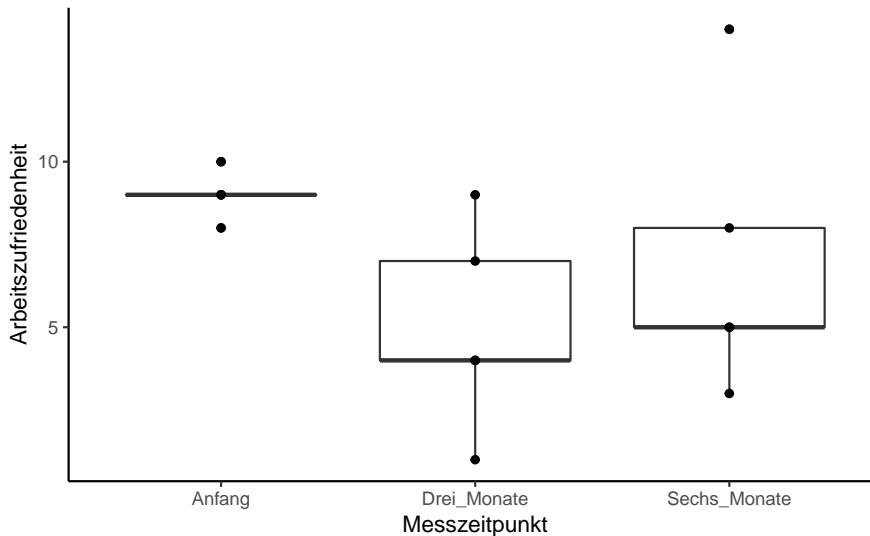
Der Friedman-Test ist eine nicht-parametrische Alternative zur Oneway-Repeated Measures ANOVA. Er kann als Erweiterung des Wilcoxon-Vorzeichen-Rangtest auf mehrere verbundene Stichproben verstanden werden. Der Friedman-Test wird meist asymptotisch durchgeführt, da ein exakter Test (bei etwas grösseren Stichproben) enorme Rechenleistung erfordert. Getestet wird die Gleichheit der Mediane in den verbundenen Stichproben.

Leider funktioniert der Friedman-Test nicht, wenn ID ein Faktor ist, daher muss ID - vorübergehend - in einen numerischen Vektor transformiert werden.

```
Honeymoon_long_F1$ID <- as.numeric(Honeymoon_long_F1$ID)
friedman.test(Arbeitszufriedenheit ~ Messzeitpunkt | ID,
 data = Honeymoon_long_F1)
#>
#> Friedman rank sum test
#>
#> data: Arbeitszufriedenheit and Messzeitpunkt and ID
#> Friedman chi-squared = 6.4, df = 2, p-value = 0.04076
```

Auch hier zur Veranschaulichung der Median-Unterschiede ein Box-Plot:

```
Honeymoon_long_F1 %>%
 ggplot(aes(x = Messzeitpunkt, y = Arbeitszufriedenheit)) +
 geom_boxplot() +
 geom_point() +
 theme_classic()
```



Post-hoc Paarvergleiche nach Nemenyi erhält man wieder mit dem Package PMCMR:

```
library(PMCMR)
posthoc.friedman.nemenyi.test(y = Honeymoon_long_F1$Arbeitszufriedenheit,
 groups = Honeymoon_long_F1$Messzeitpunkt,
 blocks = Honeymoon_long_F1$ID,
 data = Honeymoon_long_F1)

#>
#> Pairwise comparisons using Nemenyi multiple comparison test
#> with q approximation for unreplicated blocked data
#>
#> data: Honeymoon_long_F1$Arbeitszufriedenheit , Honeymoon_long_F1$Messzeitpunkt and Honeymoon_long_F1$ID
#>
#> Anfang Drei_Monate
#> Drei_Monate 0.031 -
#> Sechs_Monate 0.415 0.415
#>
#> P value adjustment method: none
```

Auch den Friedman-Test gibt es in jmv. Für die Funktion `anovaRMNP()` wird aber wieder der *wide* Datensatz benötigt:

```
library(jmv)
anovaRMNP(Honeymoon_wide_F1,
 measures = c("Anfang", "Drei_Monate", "Sechs_Monate"),
 pairs = TRUE)

#>
#> REPEATED MEASURES ANOVA (NON-PARAMETRIC)
#>
#> Friedman
#>
#> ^ df p
#> 6.40 2 0.041
#>
#>
#> Pairwise Comparisons (Durbin-Conover)
#>
#> Statistic p
#>
#> Anfang - Drei_Monate 3.77 0.005
#> Anfang - Sechs_Monate 1.89 0.096
#> Drei_Monate - Sechs_Monate 1.89 0.096
#>
```

Auch hier sind die *p*-Werte der Einzelvergleiche im Vergleich zum Nemenyi-Test deutlich weniger konservativ korrigiert.

### Mixed-Design-Varianzanalyse (mit einem within- und einem between-Faktor)

Jetzt wollen wir die Arbeitszufriedenheitsunterschiede über die drei Messzeitpunkte in zwei verschiedenen Firmen untersuchen. In der ersten Firma (s.o.) konnten wir ja trotz nicht signifikantem Effekt rein deskriptiv das Honeymoon-Hangover-Phänomen beobachten: Arbeitszufriedenheit Anfang: MW = 9 ("Honeymoon"); nach 3 Monaten: MW = 5 ("Hangover"); nach 6 Monaten: MW = 7 ("Einpendelung").

Personalverantwortliche der zweiten Firma behaupten nun aber, dass ihre Firma besonders sei und dort

das Phänomen überhaupt nicht auftrete, weil sie sich zu allen Zeitpunkten sehr um die Zufriedenheit der Mitarbeiter kümmerten.

Man könnte nun die RM-ANOVA einfach separat für die zweite Firma durchführen. Dann würden wir aber eine Nullhypothese testen (die Behauptung ist ja, dass *keine* Unterschiede zwischen den Messzeitpunkten bestehen), was problematisch ist.

Ausserdem ist es immer vorzuziehen, die Daten in *einem* Modell zu analysieren. In diesem Fall können wir die Hypothese formulieren, dass sich die Arbeitszufriedenheitsverläufe zwischen den Firmen **unterscheiden**. In einer Mixed-Design-ANOVA lässt sich diese Hypothese über den Interaktionseffekt **Messzeitpunkt:Firma** [(within-Faktor):(between-Faktor)] untersuchen.

```
Honeymoon_long %>%
 group_by(Firma, Messzeitpunkt) %>%
 summarise(Bedingungsmittelwert = mean(Arbeitszufriedenheit))
#> # A tibble: 6 x 3
#> # Groups: Firma [2]
#> Firma Messzeitpunkt Bedingungsmittelwert
#> <fct> <fct> <dbl>
#> 1 1 Anfang 9
#> 2 1 Drei_Monate 5
#> 3 1 Sechs_Monate 7
#> 4 2 Anfang 10
#> 5 2 Drei_Monate 10.8
#> 6 2 Sechs_Monate 11
```

Die Mixed-Design-ANOVA bekommen wir, indem wir in der Funktion `ezANOVA()` die Variable **Messzeitpunkt** als within-Faktor und **Firma** als between-Faktor definieren:

```
library(ez)
ezANOVA(Honeymoon_long, dv = Arbeitszufriedenheit,
 wid = ID, within = Messzeitpunkt,
 between = Firma, detailed = TRUE)
#> $ANOVA
#> Effect DFn DFd SSn SSd F p p<.05
#> 1 (Intercept) 1 8 2323.2 87.6 212.164384 4.837731e-07 *
#> 2 Firma 1 8 97.2 87.6 8.876712 1.761704e-02 *
#> 3 Messzeitpunkt 2 16 13.4 57.2 1.874126 1.856652e-01
#> 4 Firma:Messzeitpunkt 2 16 29.4 57.2 4.111888 3.622639e-02 *
#> ges
#> 1 0.94132901
#> 2 0.40165289
#> 3 0.08470291
#> 4 0.16877153
#>
#> $`Mauchly's Test for Sphericity`
#> Effect W p p<.05
#> 3 Messzeitpunkt 0.7122598 0.3049541
#> 4 Firma:Messzeitpunkt 0.7122598 0.3049541
#>
#> $`Sphericity Corrections`
#> Effect GGe p[GG] p[GG]<.05 HFe p[HF]
#> 3 Messzeitpunkt 0.7765541 0.19689735 0.9289728 0.18928790
#> 4 Firma:Messzeitpunkt 0.7765541 0.05071043 0.9289728 0.04029652
#> p[HF]<.05
#> 3
```

```
#> 4 *
```

Für dieselbe Analyse in `jmv` benötigen wir wieder den *wide* Datensatz (diesmal den mit beiden Firmen, also *Honeymoon*).

Die Mixed-Design-ANOVA erhalten wir, indem wir zum vorigen Code für die RM-ANOVA die Argumente `bs = "Firma"` (Definition des *between subjects*-Faktors) und `bsTerms = "Firma"` hinzufügen. Letzteres Argument gibt an, welche(r) der unter `bs` angegebenen Faktoren als Term (Effekt) hinzugefügt werden soll. Der Interaktionseffekt zwischen dem `bs`-Faktor und dem `rm`-Faktor wird automatisch eingefügt.

```
anovaRM(
 data = Honeymoon,
 rm = list(# Definition RM-Faktor / Faktorstufen
 list(
 label = "Messzeitpunkt",
 levels = c("Anfang", "Drei_Monate", "Sechs_Monate"))),
 rmCells = list(# Zuweisung der "Variablen" (measure)
 list(# zu den Zellen (Faktorstufen)
 measure = "Anfang",
 cell = "Anfang"),
 list(
 measure = "Drei_Monate",
 cell = "Drei_Monate"),
 list(
 measure = "Sechs_Monate",
 cell = "Sechs_Monate"))),
 rmTerms = list(# Aufrufen der RM-Terme, die im Modell
 "Messzeitpunkt"), # berücksichtigt werden sollen
 bs = "Firma", # Definition Between-Faktor
 bsTerms = list(# Aufrufen der Between-Terme, die im
 "Firma"), # Modell berücksichtigt werden sollen
 effectSize = c("eta", "omega"),
 spherTests = TRUE,
 spherCorr = c("none", "GG", "HF"))

#>
#> REPEATED MEASURES ANOVA
#>
#> Within Subjects Effects
#>
#> Sphericity Correction Sum of Squares df Mean Square F
#>
#> Messzeitpunkt None 13.4 2 6.70 1.87
#> Greenhouse-Geisser 13.4 1.55 8.63 1.87
#> Huynh-Feldt 13.4 1.86 7.21 1.87
#>
#> Messzeitpunkt:Firma None 29.4 2 14.70 4.11
#> Greenhouse-Geisser 29.4 1.55 18.93 4.11
#> Huynh-Feldt 29.4 1.86 15.82 4.11
#>
#> Residual None 57.2 16 3.58
#> Greenhouse-Geisser 57.2 12.42 4.60
#> Huynh-Feldt 57.2 14.86 3.85
#>
#> Note. Type 3 Sums of Squares
#>
```

```
#>
#> Between Subjects Effects
#>
#> Sum of Squares df Mean Square F p ² ²
#>
#> Firma 97.2 1 97.2 8.88 0.018 0.341 0.292
#> Residual 87.6 8 10.9
#>
#> Note. Type 3 Sums of Squares
#>
#>
#> ASSUMPTIONS
#>
#> Tests of Sphericity
#>
#> Mauchly's W p Greenhouse-Geisser Huynh-Feldt
#>
#> Messzeitpunkt 0.712 0.305 0.777 0.929
#>
```

Da der Mauchly-Test wieder nicht signifikant ist, muss keine Sphärizitätskorrektur angewendet werden. Die Ergebnisse zeigen, dass der Interaktionseffekt **Messzeitpunkt:Firma** signifikant ist. Die Verläufe der Arbeitszufriedenheitsmittelwerte unterscheiden sich also signifikant zwischen den beiden Firmen.

Die Interaktion spiegelt wider, dass sich in der zweiten Firma - wie von den Verantwortlichen behauptet - sehr ähnliche Zufriedenheitswerte über die Zeit zeigen, während in der ersten Firma wie oben beschrieben eher ein Honeymoon-Hangover-Prozess am Werk zu sein scheint.

Zusätzlich gibt es einen signifikanten Haupteffekt **Firma**, der anzeigt, dass die zweite Firma auch *insgesamt* (d.h. über die drei Messzeitpunkte hinweg) eine höhere durchschnittliche Arbeitszufriedenheit aufweist. Dieser Haupteffekt wird aber durch den Interaktionseffekt qualifiziert und sollte nicht ohne gleichzeitige Betrachtung des Interaktionseffekts interpretiert werden.

# Chapter 10

## Lineare Modelle

### 10.1 Faktorielle ANOVA

Wir haben schon im Kapitel “Mittelwertsvergleiche” die zweifaktorielle Varianzanalyse mittels `aov()`, `ez::ezANOVA()` und `jmv::anova()` betrachtet. Hier betrachten wir sie nochmals im Rahmen des allgemeinen linearen Modells und der damit verbundenen Funktion `lm()`.

Als Beispiel einer mehrfaktoriellen Varianzanalyse untersuchen wir die Abhängigkeit der “Wahrnehmung der deutschen Gesellschaft als gerecht” (`soc_just`) von den Faktoren “Geschlecht” (`geschl`) und “West- vs. Ostdeutschland” (`ostwest`).

Wir bilden zunächst einen Datensatz `society` mit der Variablen ID und den drei anderen benötigten Variablen:

```
library(tidyverse)
load(file = "data/westost_skalen.Rdata")
society <- westost_skalen %>%
 select(ID, westost, geschlecht, ges_gerecht) %>%
 drop_na()
society
#> # A tibble: 286 x 4
#> ID westost geschlecht ges_gerecht
#> <dbl> <fct> <fct> <dbl>
#> 1 2 West männlich 4.5
#> 2 14 West männlich 4.67
#> 3 15 West männlich 4.67
#> 4 17 West männlich 5.83
#> 5 18 West männlich 4.5
#> 6 19 West männlich 5.5
#> # ... with 280 more rows
```

Der grundlegende Befehl für alle linearen Modelle ist `lm(AV ~ UV1 * UV2 * UV3, data = mydata)`. Die abhängige Variable wird also durch die unabhängigen Variablen vorhergesagt. Die Funktion erkennt, ob es sich bei den UVs um Faktoren oder um kontinuierliche Prädiktoren/Kovariaten handelt und behandelt sie dementsprechend.

Die Verknüpfung der UVs mit `*` bewirkt, dass sowohl alle Haupteffekte, als auch alle Interaktionseffekte im Modell berücksichtigt werden. Will man dagegen nur Haupteffekte, so verknüpft man die UVs mit `+`. Will man nur bestimmte Interaktionseffekte (oder in seltenen Fällen: nur Interaktionseffekte ohne Haupteffekte) verbindet man die UVs, deren Interaktion(en) man betrachten will, mit `:` und fügt den entsprechenden Term mit `+` hinzu.

Z.B. erhält man durch `lm(AV ~ UV1 + UV2 + UV3 + UV1:UV2 + UV2:UV3 + UV1:UV3, data = mydata)` ein Modell mit allen Haupteffekten und allen 2-fach-Interaktionen wohingegen man mit `lm(AV ~ UV1 * UV2 * UV3, data = mydata)` ein full-factorial model erhält, das zusätzlich auch noch die 3-fach-Interaktion `UV1:UV2:UV3` enthält.

Die für ANOVA-Modelle massgeschneiderte Funktion zur Zusammenfassung des Outputs ist `Anova()` aus dem Package `car`. Diese Funktion arbeitet standardmäßig mit Typ-II-Quadratsummen, also Quadratsummen, die nur für Terme auf der gleichen oder einer niedrigeren Ebene auspartialisiert werden, z.B. sind Haupteffekte nur für andere Haupteffekte auspartialisiert (eine niedrigere Ebene gibt es nicht), nicht aber für den oder die Interaktionseffekt(e). Die gegenseitige Auspartialisierung von Haupteffekten spielt nur für nicht-balancierte Designs eine Rolle, also für Modelle, in denen die Zellbesetzungen ungleich bzw. nicht proportional zueinander sind.

In R werden häufig Typ-II-Quadratsummen bevorzugt, da Typ-III-Quadratsummen den Haupteffekt unter Berücksichtigung des Interaktionseffekts darstellen. Die Auspartialisierung von Interaktionseffekten (aus Haupteffekten) ist aber von der Parametrisierung des Modells abhängig. Im Allgemeinen handelt es sich bei Typ-III-Haupteffekten um bedingte Effekte. Wenn gewünscht, können Typ-III-Quadratsummen über die Option `type = 3` statt der Typ-II-Quadratsummen angefordert werden.

Zusätzlich zum ANOVA-Output können wir uns auch die Effektparameter mit der `summary()`-Funktion betrachten.

```
library(car)
AnovaModel_1 <- lm(ges_gerecht ~ geschlecht * westost, data = society)
Anova(AnovaModel_1, type = 2)
#> Anova Table (Type II tests)
#>
#> Response: ges_gerecht
#> Sum Sq Df F value Pr(>F)
#> geschlecht 6.159 1 6.2518 0.01297 *
#> westost 16.869 1 17.1246 4.626e-05 ***
#> geschlecht:westost 0.755 1 0.7661 0.38218
#> Residuals 277.797 282
#> ---
#> Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
Anova(AnovaModel_1, type = 3)
#> Anova Table (Type III tests)
#>
#> Response: ges_gerecht
#> Sum Sq Df F value Pr(>F)
#> (Intercept) 1745.87 1 1772.2809 < 2.2e-16 ***
#> geschlecht 5.58 1 5.6630 0.0179920 *
#> westost 12.89 1 13.0846 0.0003527 ***
#> geschlecht:westost 0.75 1 0.7661 0.3821806
#> Residuals 277.80 282
#> ---
#> Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
summary(AnovaModel_1)
#>
#> Call:
#> lm(formula = ges_gerecht ~ geschlecht * westost, data = society)
#>
#> Residuals:
#> Min 1Q Median 3Q Max
#> -3.1861 -0.6614 -0.0014 0.6852 2.8586
#>
```

```
#> Coefficients:
#>
#> (Intercept) Estimate Std. Error t value Pr(>|t|)
#> geschlechtweiblich -0.4002 0.1682 -2.380 0.017992 *
#> westostOst -0.5849 0.1617 -3.617 0.000353 ***
#> geschlechtweiblich:westostOst 0.2069 0.2364 0.875 0.382181
#> ---
#> Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.9925 on 282 degrees of freedom
#> Multiple R-squared: 0.08566, Adjusted R-squared: 0.07593
#> F-statistic: 8.806 on 3 and 282 DF, p-value: 1.337e-05
```

Für faktorielle ANOVAs existiert auch noch die Funktion `aov()` (vgl. auch Kapitel “Mittelwertsvergleiche”). Sie ist nur bei balancierten Designs zu empfehlen, da sie standardmäßig Typ-I-Quadratsummen benutzt. Bei diesen Quadratsummen werden alle Terme ausschließlich in der Reihenfolge der Eingabe gegeneinander auspartialisiert, d.h. nicht alle Haupteffekte sind füreinander kontrolliert, sondern nur später eingegebene für früher eingebene. Dies spielt allerdings für balancierte Designs keine Rolle, da sich die Auspartialisierung auf Ebene der Haupteffekte nur auf die unterschiedliche Häufigkeitsverteilung der Faktoren bezieht.

Im folgenden wird dies demonstriert, in dem einmal `geschlecht` und einmal `westost` als “erster Faktor” eingegeben werden. Die Typ-I-Quadratsummen erhält man mit `summary()`, mit `Anova()` erhält man auch hier Typ-II-Quadratsummen:

```
AnovaModel_1a <- aov(ges_gerecht ~ geschlecht * westost, data = society)
summary(AnovaModel_1a)
#>
#> Df Sum Sq Mean Sq F value Pr(>F)
#> geschlecht 1 8.40 8.402 8.529 0.00378 **
#> westost 1 16.87 16.869 17.125 4.63e-05 ***
#> geschlecht:westost 1 0.75 0.755 0.766 0.38218
#> Residuals 282 277.80 0.985
#> ---
#> Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
AnovaModel_1b <- aov(ges_gerecht ~ westost * geschlecht, data = society)
summary(AnovaModel_1b)
#>
#> Df Sum Sq Mean Sq F value Pr(>F)
#> westost 1 19.11 19.112 19.402 1.51e-05 ***
#> geschlecht 1 6.16 6.159 6.252 0.013 *
#> westost:geschlecht 1 0.75 0.755 0.766 0.382
#> Residuals 282 277.80 0.985
#> ---
#> Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
Anova(AnovaModel_1a, type = 2)
#> Anova Table (Type II tests)
#>
#> Response: ges_gerecht
#>
#> Sum Sq Df F value Pr(>F)
#> geschlecht 6.159 1 6.2518 0.01297 *
#> westost 16.869 1 17.1246 4.626e-05 ***
#> geschlecht:westost 0.755 1 0.7661 0.38218
#> Residuals 277.797 282
#> ---
#> Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Nun wissen wir also, dass sowohl `geschlecht` als auch `westost` einen signifikanten Effekt auf `ges_gerecht`

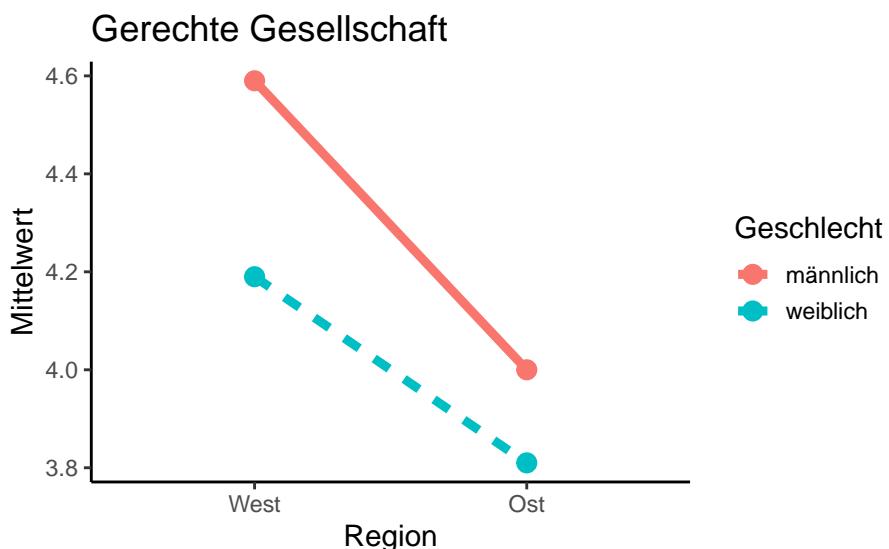
haben, und dass der Interaktionseffekt nicht signifikant ist. Aber wie stellen sich die Mittelwertsunterschiede dar?

```
soc_means <- society %>%
 group_by(westost, geschlecht) %>%
 summarize(mean = round(mean(ges_gerecht, na.rm = TRUE), 2),
 sd = round(sd(ges_gerecht, na.rm = TRUE), 3),
 N = length(ges_gerecht))

soc_means
#> # A tibble: 4 x 5
#> # Groups: westost [2]
#> westost geschlecht mean sd N
#> <fct> <fct> <dbl> <dbl> <int>
#> 1 West männlich 4.59 0.871 83
#> 2 West weiblich 4.19 1.12 60
#> 3 Ost männlich 4 0.968 69
#> 4 Ost weiblich 3.81 1.03 74

p <- soc_means %>%
 ggplot(aes(x = westost,
 y = mean,
 colour = geschlecht,
 linetype = geschlecht,
 group = geschlecht))

p + geom_line(size = 2) +
 geom_point(size = 4) +
 theme_classic(base_size = 14) +
 ggtitle("Gerechte Gesellschaft") +
 xlab("Region") +
 ylab("Mittelwert") +
 labs(color = "Geschlecht") +
 guides(linetype = FALSE)
```



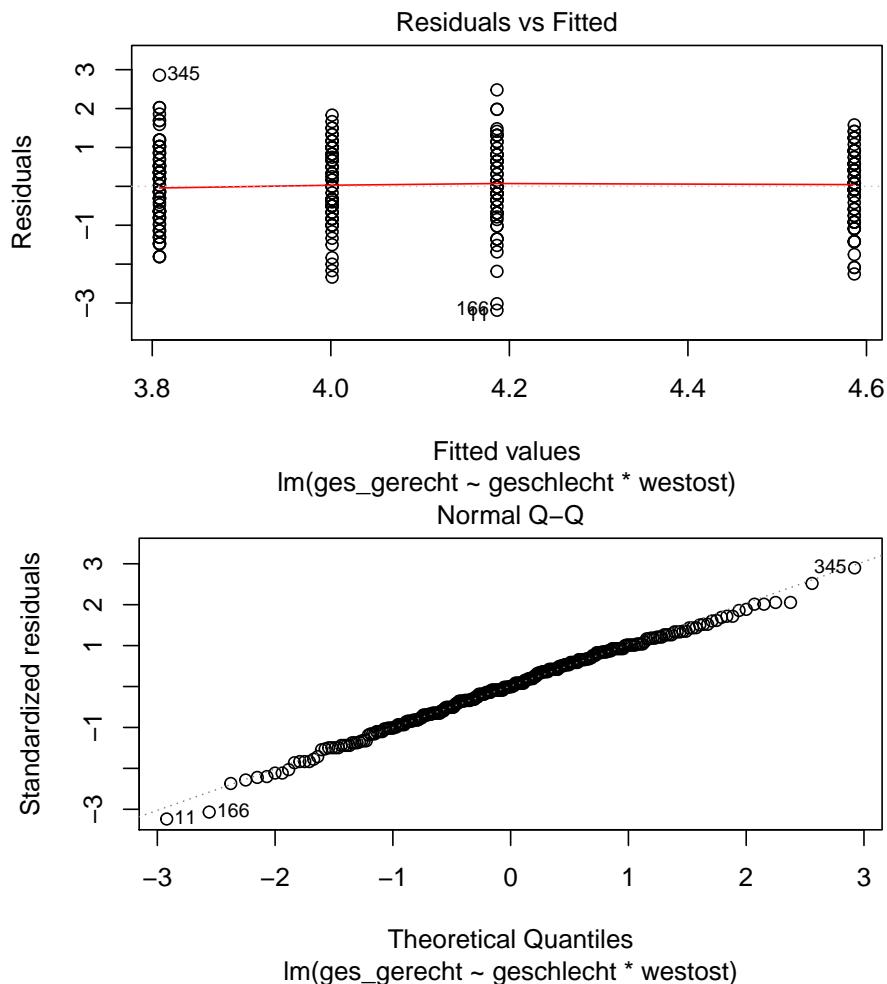
### 10.1.1 Modelldiagnostik über Diagnostische Grafiken (-)

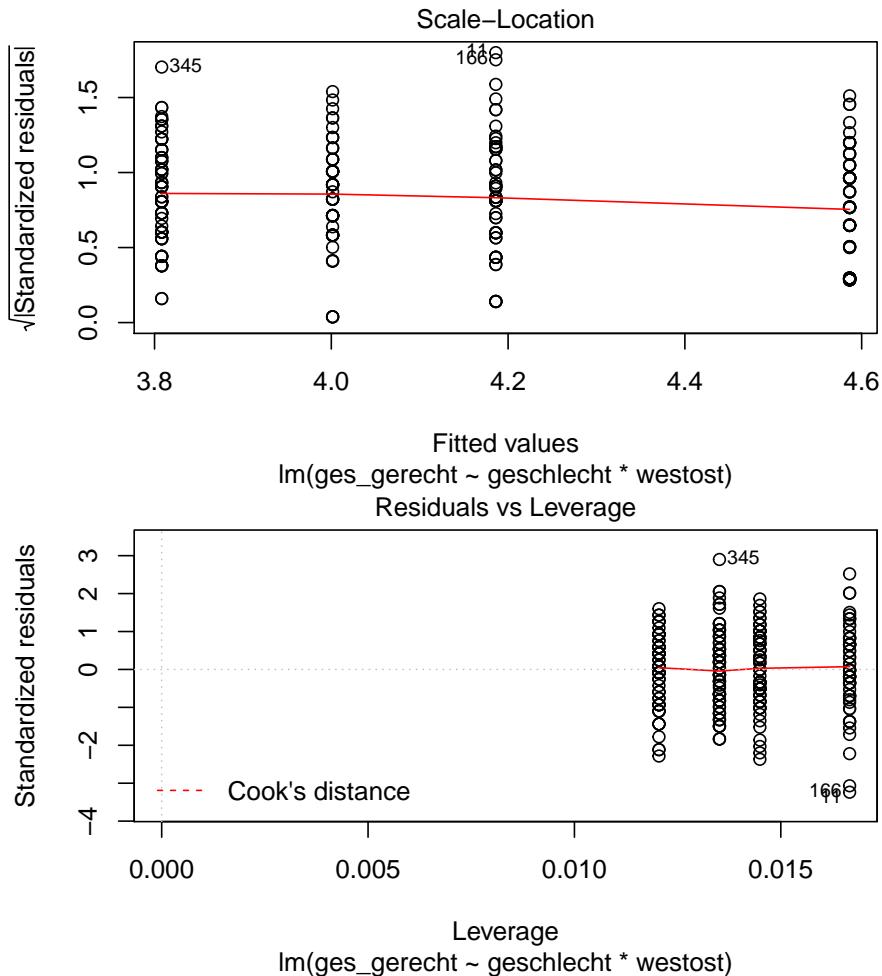
Neben der Veranschaulichung des Modells lassen sich Grafiken auch zur Diagnostik/Prüfung der Voraussetzungen eines linearen Modells benutzen. Dazu eignet sich der Befehl `plot(myModel)` aus dem `RcmdrMisc`-Package.

Es werden vier Grafiken ausgegeben:

1. Residuals vs. Fitted: Prüft korrekte Spezifikation des Modells (Residuen Mittelwert von 0 über gesamten Bereich der prädizierten Werte) -> bei Regressionsmodellen (lineare Restriktion) relevant
2. Normal Q-Q: Prüft die Normalverteilung der Residuen (Ersatz für die Prüfung der NV der AV)
3. Scale-Location: Prüft die Homoskedastizitäts-Annahme. Varianzgleichheit bei möglichst unsystematischer Verteilung der Wurzeln der standardisierten Residualbeträge (keine grossen Unterschiede über den Range der Fitted Values, d.h. in diesem Fall der 4 Gruppen)
4. Residuals vs. Leverage: Diagnose von einflussreichen Ausreisern: Werte mit grossem Hebelwert (Leverage) sind Ausreisser auf den unabhängigen Variablen. Problematisch sind sie aber nur, wenn auch ihr Residuum gross ist. Dann nämlich haben sie einen starken Einfluss auf die Modellparameter. In der Grafik werden Linien für die Cook's Distanz eingezeichnet, jenseits derer kritische Ausreisser mit grossem Einfluss liegen. Die Grenzwerte sind Cook's Distanzen von 0.5 und 1.0, sie werden aber nur eigenzeichnet, falls der Bereich mit diesen Distanzwerten im Darstellungsbereich des Diagramms liegt.

```
library(RcmdrMisc)
plot(AnovaModel_1, labels.id = society$ID)
```





Da es keine bedeutenden Abweichungen gibt, können wir davon ausgehen, dass die Voraussetzungen zur Berechnung dieses ANOVA-Modells erfüllt sind.

## 10.2 Multiple Regression

Als Beispiel dient eine hierarchische Regressionsanalyse zur Vorhersage der Zufriedenheit mit der Schule (`leben_schule`) durch sechs Selbstwirksamkeits-Subskalen, die entweder dem sozialen oder dem akademischen Bereich zuzuordnen sind.

Frage: Welchen Erklärungswert hat die soziale Selbstwirksamkeit über die akademische Selbstwirksamkeit hinaus? -> Hierarchische Regressionsanalyse mit den drei Subskalen zur akademischen Selbstwirksamkeit (Schulleistung `swk_akad`, Lernselbstregulation `swk_selbstreg`, Motivationsselfstregulation `swk_motselbst`) im ersten Modell und zusätzlich den drei Subskalen zur sozialen Selbstwirksamkeit (Durchsetzungsvermögen `swk_durch`, Soziale Harmonie `swk_sozharm`, Beziehungen `swk_bezi`) im zweiten Modell.

Vorbereitung der Daten:

```
school <- westost_skalen %>%
 select(ID, leben_schule, swk_akad, swk_selbstreg, swk_motselbst,
 swk_durch, swk_sozharm, swk_bezi) %>%
 drop_na()
school
#> # A tibble: 284 x 8
#> ID leben_schule swk_akad swk_selbstreg swk_motselbst swk_durch
#> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
```

```
#> <fct> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1 2 4.67 5 4 4.6 5
#> 2 14 5 4.86 4.5 4.6 5.33
#> 3 15 4 4 4.38 5.6 4.33
#> 4 17 5 6.14 5.62 6 6
#> 5 18 4.33 5.14 4.62 4.4 4.33
#> 6 19 4 5.43 4.62 5.2 6.33
#> # ... with 278 more rows, and 2 more variables: swk_sozharm <dbl>,
#> # swk_bez <dbl>
```

### 10.2.1 Hierarchische Regressionsanalyse: Modell 1 (-)

Nur Variablen der akademischen Selbstwirksamkeit (Block 1)

```
RegModel_1 <- lm(leben_schule ~ swk_akad + swk_selbstreg + swk_motselbst,
 data = school)
summary(RegModel_1)
#>
#> Call:
#> lm(formula = leben_schule ~ swk_akad + swk_selbstreg + swk_motselbst,
#> data = school)
#>
#> Residuals:
#> Min 1Q Median 3Q Max
#> -2.39187 -0.57693 0.08034 0.61309 2.19339
#>
#> Coefficients:
#> Estimate Std. Error t value Pr(>|t|)
#> (Intercept) 0.82169 0.42629 1.928 0.0549 .
#> swk_akad 0.27398 0.09360 2.927 0.0037 **
#> swk_selbstreg 0.56290 0.09996 5.631 4.35e-08 ***
#> swk_motselbst -0.09078 0.11261 -0.806 0.4208
#> ---
#> Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.9181 on 280 degrees of freedom
#> Multiple R-squared: 0.262, Adjusted R-squared: 0.2541
#> F-statistic: 33.14 on 3 and 280 DF, p-value: < 2.2e-16
```

### 10.2.2 Hierarchische Regressionsanalyse: Modell 2 (-)

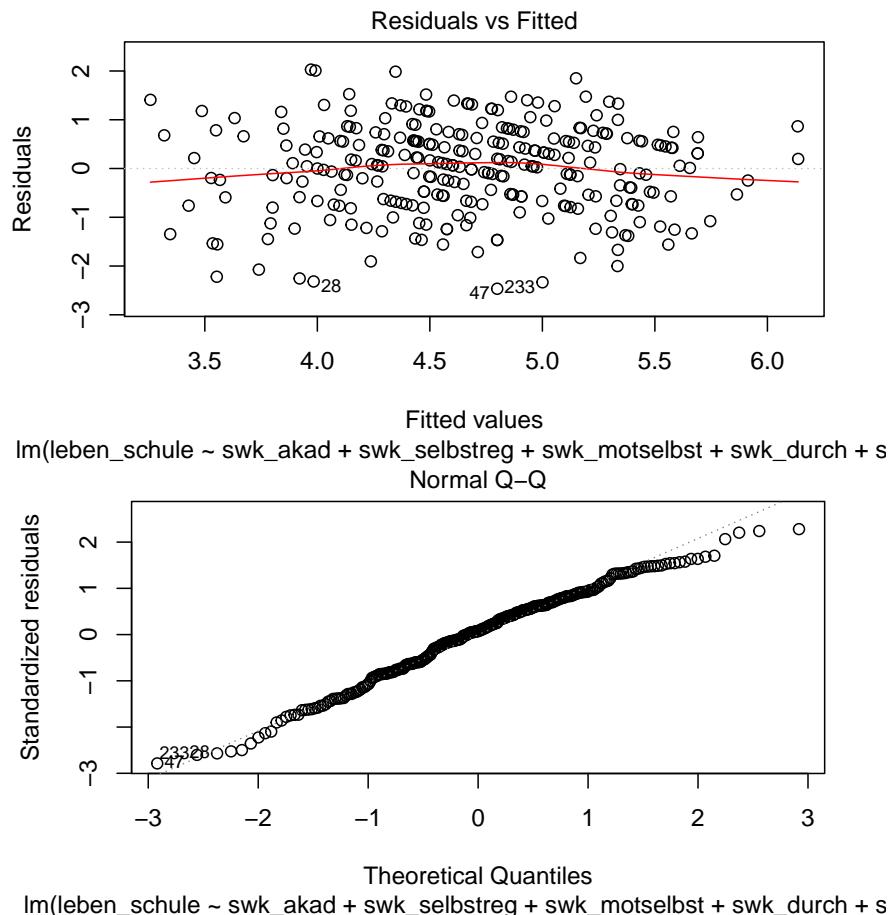
Zusätzlich Variablen der sozialen Selbstwirksamkeit (Block 2)

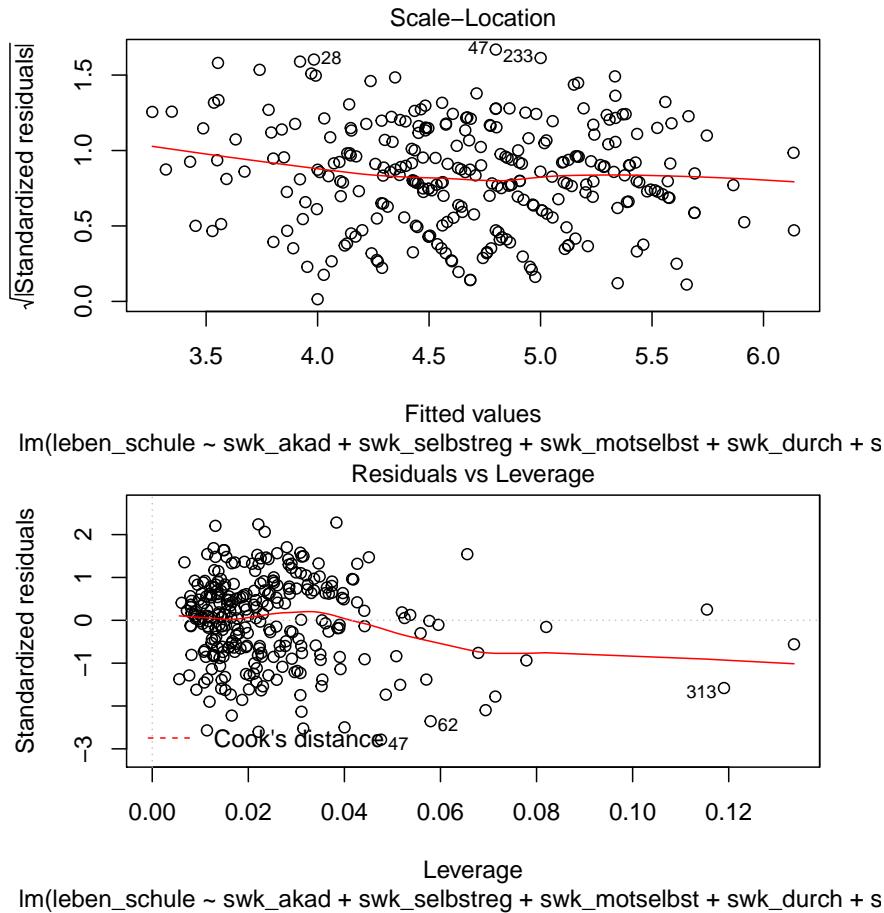
```
RegModel_2 <- lm(leben_schule ~ swk_akad + swk_selbstreg + swk_motselbst +
 swk_durch + swk_sozharm + swk_bez, data = school)
summary(RegModel_2)
#>
#> Call:
#> lm(formula = leben_schule ~ swk_akad + swk_selbstreg + swk_motselbst +
#> swk_durch + swk_sozharm + swk_bez, data = school)
#>
#> Residuals:
#> Min 1Q Median 3Q Max
#> -2.46599 -0.63510 0.06727 0.62685 2.02832
```

```
#>
#> Coefficients:
#>
#> Estimate Std. Error t value Pr(>|t|)
#> (Intercept) 0.17832 0.49775 0.358 0.72042
#> swk_akad 0.23679 0.09390 2.522 0.01224 *
#> swk_selbstreg 0.48641 0.10186 4.775 2.91e-06 ***
#> swk_motselbst -0.10857 0.11429 -0.950 0.34300
#> swk_durch 0.03541 0.06993 0.506 0.61303
#> swk_sozharm -0.07261 0.09319 -0.779 0.43656
#> swk_bez 0.27179 0.08880 3.061 0.00243 **
#> ---
#> Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.907 on 277 degrees of freedom
#> Multiple R-squared: 0.2875, Adjusted R-squared: 0.272
#> F-statistic: 18.63 on 6 and 277 DF, p-value: < 2.2e-16
```

## 10.2.2.1 Die Regressionsdiagnose für Modell 2

```
plot(RegModel_2, labels.id = school$ID)
```





### 10.2.3 Modellvergleich Modell 1 vs. Modell 2 (-)

Nun wollen wir die beiden Modelle mit einem Modellvergleichs- $F$ -Test vergleichen (Signifikanztest für ein Set von Prädiktorvariablen bzw. für  $\Delta R^2$ ). Frage: Ist die Hinzunahme der drei Variablen der sozialen Selbstwirksamkeit insgesamt signifikant?

```
anova(RegModel_1, RegModel_2)
```

### 10.2.4 Modellauswahl bei der multiplen Regressionsanalyse (-)

Wir wollen nun eine schrittweise Modellselektion in Bezug auf das gerade berechnete Regressionsmodell mit `leben_schule` als AV und den sechs Selbstwirksamkeitssubskalen als UVs durchführen. Dabei werden wir Informationskriterien (AIC, BIC) als Auswahlkriterien für den Einschluss eines Prädiktors nutzen. Wir nutzen dazu die Funktion `stepwise()` aus dem `RcmdrMisc`-package, die zudem Funktionen aus dem Package `MASS` benötigt. Wir beginnen mit einer Vorwärtsselektion und variieren die Informationskriterien AIC und BIC:

Schrittweise Modellselektion mit der Methode “Vorwärts (Forward)”:

```
library(RcmdrMisc, MASS)
stepwise(RegModel_2, direction = 'forward', criterion = 'AIC')
#>
#> Direction: forward
#> Criterion: AIC
#>
#> Start: AIC=35.71
```

```

#> leben_schule ~ 1
#>
#> Df Sum of Sq RSS AIC
#> + swk_selbstreg 1 76.413 243.38 -39.831
#> + swk_akad 1 49.802 270.00 -10.361
#> + swk_bez 1 46.921 272.88 -7.348
#> + swk_motselbst 1 43.318 276.48 -3.622
#> + swk_sozharm 1 14.924 304.87 24.142
#> + swk_durch 1 13.890 305.91 25.104
#> <none> 319.80 35.715
#>
#> Step: AIC=-39.83
#> leben_schule ~ swk_selbstreg
#>
#> Df Sum of Sq RSS AIC
#> + swk_bez 1 9.8068 233.58 -49.511
#> + swk_akad 1 6.8302 236.55 -45.915
#> <none> 243.38 -39.831
#> + swk_durch 1 1.0017 242.38 -39.002
#> + swk_sozharm 1 0.8890 242.50 -38.870
#> + swk_motselbst 1 0.1565 243.23 -38.013
#>
#> Step: AIC=-49.51
#> leben_schule ~ swk_selbstreg + swk_bez
#>
#> Df Sum of Sq RSS AIC
#> + swk_akad 1 4.3712 229.21 -52.876
#> <none> 233.58 -49.511
#> + swk_sozharm 1 0.1185 233.46 -47.655
#> + swk_durch 1 0.1162 233.46 -47.652
#> + swk_motselbst 1 0.0000 233.58 -47.511
#>
#> Step: AIC=-52.88
#> leben_schule ~ swk_selbstreg + swk_bez + swk_akad
#>
#> Df Sum of Sq RSS AIC
#> <none> 229.21 -52.876
#> + swk_motselbst 1 0.82215 228.38 -51.897
#> + swk_sozharm 1 0.48634 228.72 -51.479
#> + swk_durch 1 0.00297 229.20 -50.880
#>
#> Call:
#> lm(formula = leben_schule ~ swk_selbstreg + swk_bez + swk_akad,
#> data = school)
#>
#> Coefficients:
#> (Intercept) swk_selbstreg swk_bez swk_akad
#> 0.01421 0.43363 0.24569 0.19758
stepwise(RegModel_2, direction = 'forward', criterion = 'BIC')
#>
#> Direction: forward
#> Criterion: BIC
#>

```

```

#> Start: AIC=39.36
#> leben_schule ~ 1
#>
#> Df Sum of Sq RSS AIC
#> + swk_selbstreg 1 76.413 243.38 -32.533
#> + swk_akad 1 49.802 270.00 -3.063
#> + swk_bez 1 46.921 272.88 -0.050
#> + swk_motselbst 1 43.318 276.48 3.676
#> + swk_sozharm 1 14.924 304.87 31.440
#> + swk_durch 1 13.890 305.91 32.402
#> <none> 319.80 39.364
#>
#> Step: AIC=-32.53
#> leben_schule ~ swk_selbstreg
#>
#> Df Sum of Sq RSS AIC
#> + swk_bez 1 9.8068 233.58 -38.564
#> + swk_akad 1 6.8302 236.55 -34.968
#> <none> 243.38 -32.533
#> + swk_durch 1 1.0017 242.38 -28.055
#> + swk_sozharm 1 0.8890 242.50 -27.923
#> + swk_motselbst 1 0.1565 243.23 -27.066
#>
#> Step: AIC=-38.56
#> leben_schule ~ swk_selbstreg + swk_bez
#>
#> Df Sum of Sq RSS AIC
#> <none> 233.58 -38.564
#> + swk_akad 1 4.3712 229.21 -38.280
#> + swk_sozharm 1 0.1185 233.46 -33.059
#> + swk_durch 1 0.1162 233.46 -33.056
#> + swk_motselbst 1 0.0000 233.58 -32.915
#>
#> Call:
#> lm(formula = leben_schule ~ swk_selbstreg + swk_bez, data = school)
#>
#> Coefficients:
#> (Intercept) swk_selbstreg swk_bez
#> 0.3526 0.5251 0.2793

```

Nach dem AIC, das eher komplexere Modelle bevorzugt, sollte der Prädiktor `se_akad` ins Modell eingeschlossen werden, nach dem BIC nicht. Dasselbe Ergebnis ergibt sich bei der Methode "Rückwärts (Backward)":

```

stepwise(RegModel_2, direction = 'backward', criterion = 'AIC')
#>
#> Direction: backward
#> Criterion: AIC
#>
#> Start: AIC=-48.54
#> leben_schule ~ swk_akad + swk_selbstreg + swk_motselbst + swk_durch +
#> swk_sozharm + swk_bez
#>
#> Df Sum of Sq RSS AIC

```

```

#> - swk_durch 1 0.2109 228.08 -50.279
#> - swk_sozharm 1 0.4994 228.37 -49.920
#> - swk_motselbst 1 0.7422 228.61 -49.619
#> <none> 227.87 -48.542
#> - swk_akad 1 5.2307 233.10 -44.097
#> - swk_bez 1 7.7056 235.57 -41.097
#> - swk_selbstreg 1 18.7590 246.62 -28.074
#>
#> Step: AIC=-50.28
#> leben_schule ~ swk_akad + swk_selbstreg + swk_motselbst + swk_sozharm +
#> swk_bez
#>
#> Df Sum of Sq RSS AIC
#> - swk_sozharm 1 0.3077 228.38 -51.897
#> - swk_motselbst 1 0.6435 228.72 -51.479
#> <none> 228.08 -50.279
#> - swk_akad 1 5.3773 233.45 -45.661
#> - swk_bez 1 7.6624 235.74 -42.895
#> - swk_selbstreg 1 18.7015 246.78 -29.898
#>
#> Step: AIC=-51.9
#> leben_schule ~ swk_akad + swk_selbstreg + swk_motselbst + swk_bez
#>
#> Df Sum of Sq RSS AIC
#> - swk_motselbst 1 0.8221 229.21 -52.876
#> <none> 228.38 -51.897
#> - swk_akad 1 5.1933 233.58 -47.511
#> - swk_bez 1 7.6221 236.01 -44.573
#> - swk_selbstreg 1 18.9576 247.34 -31.250
#>
#> Step: AIC=-52.88
#> leben_schule ~ swk_akad + swk_selbstreg + swk_bez
#>
#> Df Sum of Sq RSS AIC
#> <none> 229.21 -52.876
#> - swk_akad 1 4.3712 233.58 -49.511
#> - swk_bez 1 7.3477 236.55 -45.915
#> - swk_selbstreg 1 21.0515 250.26 -29.921
#>
#> Call:
#> lm(formula = leben_schule ~ swk_akad + swk_selbstreg + swk_bez,
#> data = school)
#>
#> Coefficients:
#> (Intercept) swk_akad swk_selbstreg swk_bez
#> 0.01421 0.19758 0.43363 0.24569
stepwise(RegModel_2, direction = 'backward', criterion = 'BIC')
#>
#> Direction: backward
#> Criterion: BIC
#>
#> Start: AIC=-23
#> leben_schule ~ swk_akad + swk_selbstreg + swk_motselbst + swk_durch +

```

```

#> swk_sozharm + swk_bez
#>
#> Df Sum of Sq RSS AIC
#> - swk_durch 1 0.2109 228.08 -28.3855
#> - swk_sozharm 1 0.4994 228.37 -28.0265
#> - swk_motselbst 1 0.7422 228.61 -27.7247
#> <none> 227.87 -22.9993
#> - swk_akad 1 5.2307 233.10 -22.2027
#> - swk_bez 1 7.7056 235.57 -19.2032
#> - swk_selbstreg 1 18.7590 246.62 -6.1806
#>
#> Step: AIC=-28.39
#> leben_schule ~ swk_akad + swk_selbstreg + swk_motselbst + swk_sozharm +
#> swk_bez
#>
#> Df Sum of Sq RSS AIC
#> - swk_sozharm 1 0.3077 228.38 -33.652
#> - swk_motselbst 1 0.6435 228.72 -33.234
#> <none> 228.08 -28.386
#> - swk_akad 1 5.3773 233.45 -27.416
#> - swk_bez 1 7.6624 235.74 -24.650
#> - swk_selbstreg 1 18.7015 246.78 -11.653
#>
#> Step: AIC=-33.65
#> leben_schule ~ swk_akad + swk_selbstreg + swk_motselbst + swk_bez
#>
#> Df Sum of Sq RSS AIC
#> - swk_motselbst 1 0.8221 229.21 -38.280
#> <none> 228.38 -33.652
#> - swk_akad 1 5.1933 233.58 -32.915
#> - swk_bez 1 7.6221 236.01 -29.977
#> - swk_selbstreg 1 18.9576 247.34 -16.654
#>
#> Step: AIC=-38.28
#> leben_schule ~ swk_akad + swk_selbstreg + swk_bez
#>
#> Df Sum of Sq RSS AIC
#> - swk_akad 1 4.3712 233.58 -38.564
#> <none> 229.21 -38.280
#> - swk_bez 1 7.3477 236.55 -34.968
#> - swk_selbstreg 1 21.0515 250.26 -18.974
#>
#> Step: AIC=-38.56
#> leben_schule ~ swk_selbstreg + swk_bez
#>
#> Df Sum of Sq RSS AIC
#> <none> 233.58 -38.564
#> - swk_bez 1 9.807 243.38 -32.533
#> - swk_selbstreg 1 39.299 272.88 -0.050
#>
#> Call:
#> lm(formula = leben_schule ~ swk_selbstreg + swk_bez, data = school)
#>

```

```
#> Coefficients:
#> (Intercept) swk_selbstreg swk_bez
#> 0.3526 0.5251 0.2793
```

Auch bei der Methode “Rückwärts/Vorwärts” bekommen wir jeweils dasselbe abschliessende Modell:

```
stepwise(RegModel_2, direction = 'backward/forward', criterion = 'AIC')
#>
#> Direction: backward/forward
#> Criterion: AIC
#>
#> Start: AIC=-48.54
#> leben_schule ~ swk_akad + swk_selbstreg + swk_motselbst + swk_durch +
#> swk_sozharm + swk_bez
#>
#> Df Sum of Sq RSS AIC
#> - swk_durch 1 0.2109 228.08 -50.279
#> - swk_sozharm 1 0.4994 228.37 -49.920
#> - swk_motselbst 1 0.7422 228.61 -49.619
#> <none> 227.87 -48.542
#> - swk_akad 1 5.2307 233.10 -44.097
#> - swk_bez 1 7.7056 235.57 -41.097
#> - swk_selbstreg 1 18.7590 246.62 -28.074
#>
#> Step: AIC=-50.28
#> leben_schule ~ swk_akad + swk_selbstreg + swk_motselbst + swk_sozharm +
#> swk_bez
#>
#> Df Sum of Sq RSS AIC
#> - swk_sozharm 1 0.3077 228.38 -51.897
#> - swk_motselbst 1 0.6435 228.72 -51.479
#> <none> 228.08 -50.279
#> + swk_durch 1 0.2109 227.87 -48.542
#> - swk_akad 1 5.3773 233.45 -45.661
#> - swk_bez 1 7.6624 235.74 -42.895
#> - swk_selbstreg 1 18.7015 246.78 -29.898
#>
#> Step: AIC=-51.9
#> leben_schule ~ swk_akad + swk_selbstreg + swk_motselbst + swk_bez
#>
#> Df Sum of Sq RSS AIC
#> - swk_motselbst 1 0.8221 229.21 -52.876
#> <none> 228.38 -51.897
#> + swk_sozharm 1 0.3077 228.08 -50.279
#> + swk_durch 1 0.0192 228.37 -49.920
#> - swk_akad 1 5.1933 233.58 -47.511
#> - swk_bez 1 7.6221 236.01 -44.573
#> - swk_selbstreg 1 18.9576 247.34 -31.250
#>
#> Step: AIC=-52.88
#> leben_schule ~ swk_akad + swk_selbstreg + swk_bez
#>
#> Df Sum of Sq RSS AIC
#> <none> 229.21 -52.876
```

```

#> + swk_motselbst 1 0.8221 228.38 -51.897
#> + swk_sozharm 1 0.4863 228.72 -51.479
#> + swk_durch 1 0.0030 229.20 -50.880
#> - swk_akad 1 4.3712 233.58 -49.511
#> - swk_bez 1 7.3477 236.55 -45.915
#> - swk_selbstreg 1 21.0515 250.26 -29.921
#>
#> Call:
#> lm(formula = leben_schule ~ swk_akad + swk_selbstreg + swk_bez,
#> data = school)
#>
#> Coefficients:
#> (Intercept) swk_akad swk_selbstreg swk_bez
#> 0.01421 0.19758 0.43363 0.24569
stepwise(RegModel_2, direction = 'backward/forward', criterion = 'BIC')
#>
#> Direction: backward/forward
#> Criterion: BIC
#>
#> Start: AIC=-23
#> leben_schule ~ swk_akad + swk_selbstreg + swk_motselbst + swk_durch +
#> swk_sozharm + swk_bez
#>
#> Df Sum of Sq RSS AIC
#> - swk_durch 1 0.2109 228.08 -28.3855
#> - swk_sozharm 1 0.4994 228.37 -28.0265
#> - swk_motselbst 1 0.7422 228.61 -27.7247
#> <none> 227.87 -22.9993
#> - swk_akad 1 5.2307 233.10 -22.2027
#> - swk_bez 1 7.7056 235.57 -19.2032
#> - swk_selbstreg 1 18.7590 246.62 -6.1806
#>
#> Step: AIC=-28.39
#> leben_schule ~ swk_akad + swk_selbstreg + swk_motselbst + swk_sozharm +
#> swk_bez
#>
#> Df Sum of Sq RSS AIC
#> - swk_sozharm 1 0.3077 228.38 -33.652
#> - swk_motselbst 1 0.6435 228.72 -33.234
#> <none> 228.08 -28.386
#> - swk_akad 1 5.3773 233.45 -27.416
#> - swk_bez 1 7.6624 235.74 -24.650
#> + swk_durch 1 0.2109 227.87 -22.999
#> - swk_selbstreg 1 18.7015 246.78 -11.653
#>
#> Step: AIC=-33.65
#> leben_schule ~ swk_akad + swk_selbstreg + swk_motselbst + swk_bez
#>
#> Df Sum of Sq RSS AIC
#> - swk_motselbst 1 0.8221 229.21 -38.280
#> <none> 228.38 -33.652
#> - swk_akad 1 5.1933 233.58 -32.915
#> - swk_bez 1 7.6221 236.01 -29.977

```

```

#> + swk_sozharm 1 0.3077 228.08 -28.386
#> + swk_durch 1 0.0192 228.37 -28.027
#> - swk_selbstreg 1 18.9576 247.34 -16.654
#>
#> Step: AIC=-38.28
#> leben_schule ~ swk_akad + swk_selbstreg + swk_bez
#>
#> Df Sum of Sq RSS AIC
#> - swk_akad 1 4.3712 233.58 -38.564
#> <none> 229.21 -38.280
#> - swk_bez 1 7.3477 236.55 -34.968
#> + swk_motselbst 1 0.8221 228.38 -33.652
#> + swk_sozharm 1 0.4863 228.72 -33.234
#> + swk_durch 1 0.0030 229.20 -32.635
#> - swk_selbstreg 1 21.0515 250.26 -18.974
#>
#> Step: AIC=-38.56
#> leben_schule ~ swk_selbstreg + swk_bez
#>
#> Df Sum of Sq RSS AIC
#> <none> 233.58 -38.564
#> + swk_akad 1 4.371 229.21 -38.280
#> + swk_sozharm 1 0.119 233.46 -33.059
#> + swk_durch 1 0.116 233.46 -33.056
#> + swk_motselbst 1 0.000 233.58 -32.915
#> - swk_bez 1 9.807 243.38 -32.533
#> - swk_selbstreg 1 39.299 272.88 -0.050
#>
#> Call:
#> lm(formula = leben_schule ~ swk_selbstreg + swk_bez, data = school)
#>
#> Coefficients:
#> (Intercept) swk_selbstreg swk_bez
#> 0.3526 0.5251 0.2793

```

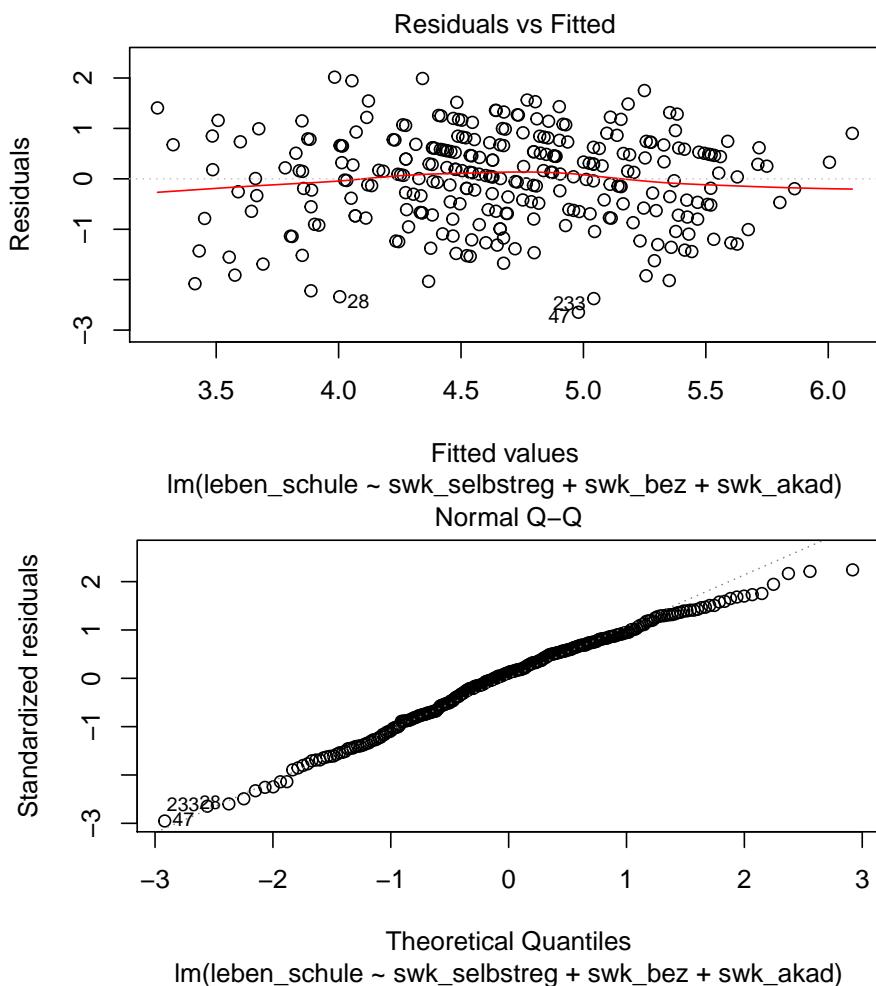
Wir berechnen nun abschliessend noch das Regressionsmodell mit den drei nach dem AIC-Kriterium auszuwählenden Prädiktoren. Zusätzlich lassen wir uns mit `confint()` die Konfidenzintervalle der Regressionskoeffizienten ausgeben. Anschliessend betrachten wir noch die diagnostischen Grafiken:

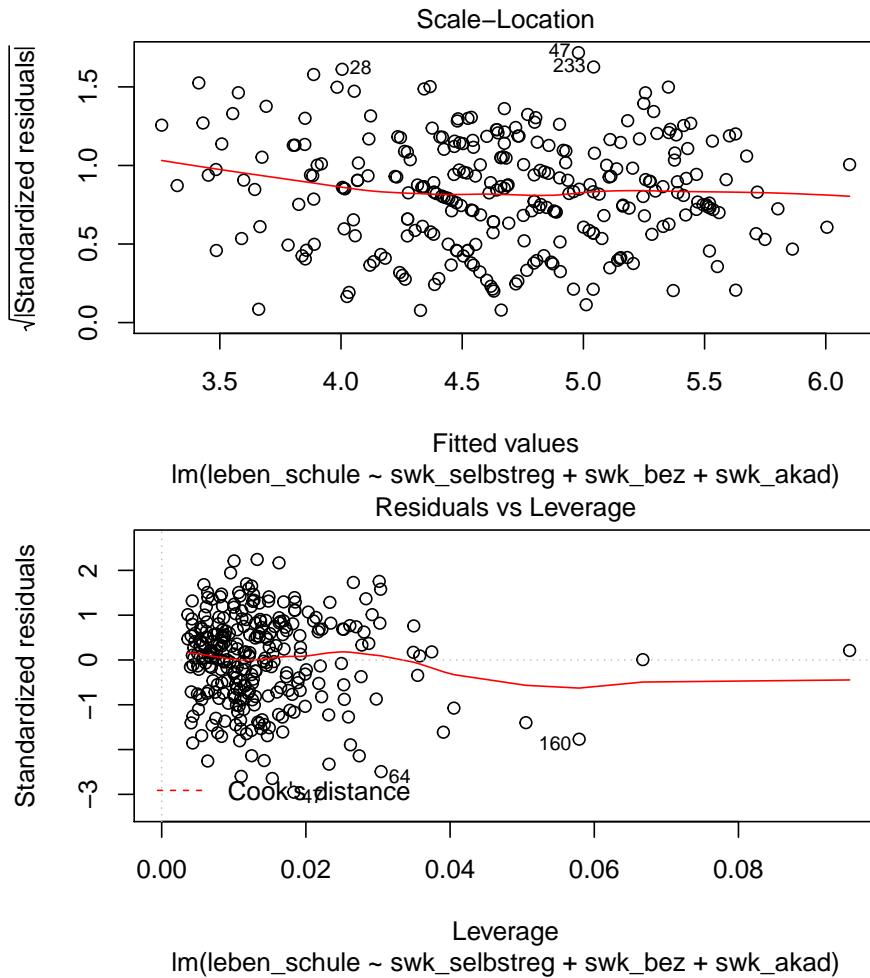
```

RegModel_3 <- lm(leben_schule ~ swk_selbstreg + swk_bez + swk_akad,
 data = school)
summary(RegModel_3)
#>
#> Call:
#> lm(formula = leben_schule ~ swk_selbstreg + swk_bez + swk_akad,
#> data = school)
#>
#> Residuals:
#> Min 1Q Median 3Q Max
#> -2.6462 -0.6430 0.1045 0.6520 2.0162
#>
#> Coefficients:
#> Estimate Std. Error t value Pr(>|t|)
#> (Intercept) 0.01421 0.46836 0.030 0.97582

```

```
#> swk_selbstreg 0.43363 0.08551 5.071 7.2e-07 ***
#> swk_bez 0.24569 0.08200 2.996 0.00298 **
#> swk_akad 0.19758 0.08550 2.311 0.02157 *
#> ---
#> Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.9048 on 280 degrees of freedom
#> Multiple R-squared: 0.2833, Adjusted R-squared: 0.2756
#> F-statistic: 36.89 on 3 and 280 DF, p-value: < 2.2e-16
confint(RegModel_3)
#> 2.5 % 97.5 %
#> (Intercept) -0.90774569 0.9361688
#> swk_selbstreg 0.26530514 0.6019467
#> swk_bez 0.08426199 0.4071084
#> swk_akad 0.02927203 0.3658976
plot(RegModel_3, labels.id = school$ID)
```



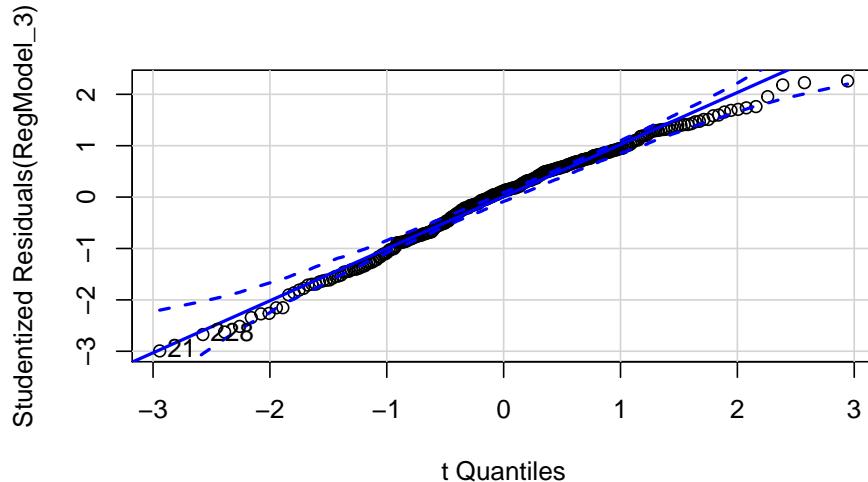


Die Ergebnisse der vier bereits bekannten diagnostischen Grafiken zeigen keine Verletzung der Voraussetzungen des Regressionsmodells an (Lineare Spezifikation des Modells, Normalverteilung der Residuen, Homoskedastizität). Der Residuals vs. Leverage-Plot zeigt auch keine bedeutsamen Cook's Distanz-Werte zur Identifikation einflussreicher Datenpunkte an. Die für die Grenze der Cook's Distanz vorgesehene gestrichelte Linie erscheint nicht in der Grafik, d.h. keiner der Werte kommt auch nur annähernd in die Nähe der vorgeschlagenen Cook's Distanz-Grenzwerte von 0,5 bzw. 1,0 heran.

#### 10.2.4.1 Zusätzliche diagnostische Grafiken ausgeben (-)

Mit `qqPlot()` betrachten wir noch einen weiteren Quantil-Quantil-Plot zur Prüfung der Normalverteilung der Residuen. Der Vorteil dieses Plots ist, dass ein 95%-Konfidenzband (`evaluate = 0.95`) erzeugt wird, ausserhalb dessen Beobachtungen liegen, deren Residuum eine "signifikante" Abweichung von der Normalverteilung aufweist. Mit der Option `simulate = TRUE` (und `reps = [Anzahl Bootstrap-Stichproben]`) wird dieses Konfidenzband über einen parametrischen Bootstrap definiert und ist damit selbst robuster gegenüber einer Abweichung von der Normalverteilung der Residuen. Da in diesem Plot die studentisierten Residuen betrachtet werden, handelt es sich bei den theoretischen Quantilen um die einer  $t$ -Verteilung. Man kann aber mit `distribution = "norm"` auch die Quantile der Standardnormalverteilung auf der x-Achse ausgeben lassen. Mit `labels` können wir wie gewohnt eine Variable angeben, die zur Bezeichnung der extremsten Datenpunkte inverwendet werden soll (ID). Mit `id.n` können wir die Anzahl der so gekennzeichneten Datenpunkte steuern.

```
qqPlot(RegModel_3, envelope = 0.95, distribution = "t", simulate = TRUE, reps = 1000, labels = school$ID)
#> [1] 21 228
```



Mit `influencePlot()` betrachten wir einen weiteren Plot zur Diagnostik einflussreicher Datenpunkte. Es ist eigentlich der gleiche Plot wie der oben mit `plot()` bereits ausgegebene Residuals vs. Leverage Plot (einiger Unterschied in Bezug auf die Daten ist die Verwendung der studentisierten statt der standardisierten Residuen). Im `influencePlot()` werden die Hebelwerte statt mit "Leverage" mit ihrem alternativen Namen "Hat"-Values bezeichnet. Der Vorteil dieses Plots ist, dass er um die Datenpunkte herum eine Kreisfläche ausgibt, die die relative Grösse der Cook's-Distanz dieser Beobachtung repräsentiert (zur Erinnerung: Cook's Distanz repräsentiert eine Kombination aus Hebelwert und standardisierten/studentisierten Residuen). Hier werden außerdem zwei vertikale gestrichelte Linien beim 2- und 3-fachen der durchschnittlichen Hat-Values eingezeichnet. Die horizontalen gestrichelten Linien kennzeichnen eine mögliche Grenzen für grosse Beträge von Residualwerten (2 Standardabweichungen).

```
influencePlot(RegModel_3, labels = school$ID, id.method = "noteworthy", id.n = 5)
```

Bei den Funktionen `qqPlot()` und `influencePlot()` wird sofern

Ausserdem überprüfen wir noch die Prädiktoren auf Multikollinearität mit Hilfe des Variance Inflation Factors (`vif()`). Der VIF ist der Kehrwert der Toleranz. Ein Wert 1 des VIF zeigt an, dass die interessierende Prädiktorvariable mit allen anderen UVs unkorreliert ist. Je grösser der VIF, desto grösser die Multikollinearität. Ein Wert des  $VIF > 10$  wird in der Literatur häufig als auffällig bewertet.

```
vif(RegModel_3)
#> swk_selbstreg swk_bez swk_akad
#> 1.627580 1.320210 1.503083
```

Alle drei VIFs liegen zwischen 1 und 2 und sind damit unauffällig.

## 10.3 Übungsaufgaben:

### 10.3.1 1) Zweifaktorielle ANOVA: Note (Schnitt oder Bereich) ~ Dichotome Bildung Vater \* Dichotome Bildung Mutter (-)

Diese Aufgabe kann sowohl mit dem Gesamtnotenschnitt als AV (Schnitt) oder mit den Fachnoten (Deutsch Mathe Fremdspr Schnitt) durchgeführt werden. Wählen Sie eine aus! Berechnen Sie die zweifaktorielle Varianzanalyse mittels `lm()` und `Anova()`. Lassen Sie sich einmal Typ-II und einmal Typ-III-Quadratsummen ausgeben. Lassen Sie sich außerdem die Parameterschätzer mit `summary()` ausgeben. Vergleichen Sie die unter `summary()` erhaltenen Signifikanztests mit jenen der Typ-II und Typ-III-Quadratsummen `Anovas()`. (Optional: Berechnen Sie Typ-I-Quadratsummen-Modelle mit `aov()` mit den beiden Faktoren in unterschiedlicher Reihenfolge). Stellen Sie die Mittelwerte in einem gruppierten Liniendiagramm dar und interpretieren Sie die Ergebnisse.

```

load("data/westost_skalen.Rdata")
education <- westost_skalen %>%
 select(ID, westost, geschlecht,
 Deutsch, Mathe, Fremdspr, Schnitt,
 ends_with("_b")) %>%
 drop_na()
education$Deutsch <- as.double(education$Deutsch)
education$Mathe <- as.double(education$Mathe)
education$Fremdspr <- as.double(education$Fremdspr)
education
#> # A tibble: 254 x 9
#> ID westost geschlecht Deutsch Mathe Fremdspr Schnitt bildung_vater_b
#> <fct> <fct> <fct> <dbl> <dbl> <dbl> <dbl> <fct>
#> 1 2 West männlich 4 4 3 4 hoch
#> 2 14 West männlich 4 4 3 4 niedrig
#> 3 15 West männlich 4 4 4 4 niedrig
#> 4 17 West männlich 4 4 3 4 niedrig
#> 5 18 West männlich 5 5 4 4 hoch
#> 6 19 West männlich 4 4 3 4 niedrig
#> # ... with 248 more rows, and 1 more variable: bildung_mutter_b <fct>

```

### 10.3.2 2) Hierarchische Regression: Psychisches Stresserleben ~ 6 Selbstwirksamkeitsskalen (+ Modellauswahl mit AIC & BIC, + Regressionsdiagnostik für final model) (-)

Berechnen Sie eine hierarchische Regressionsanalyse mit `stress_psych` als AV und den 6 SWK-Skalen als UVs. Dabei sollen die Skalen der sozialen SWK in einem ersten Block ins Modell aufgenommen werden (theoretisch proximaler: `swk_durch swk_sozharm swk_bez`) und die der akademischen SWK in einem zweiten Block (theoretisch distaler: `swk_akad swk_selbstreg swk_motselbst`). Außerdem soll eine Modellselektion mittels `stepwise()` mit den Informationskriterien AIC und BIC sowie mit den Selektionsstrategien `forward` `backward` `forward/backward` und `backward/forward` vorgenommen werden. Wählen Sie die Ihnen die aufgrund dieser Analyse am sinnvollsten erscheinende Kombination von Prädiktoren aus und berechnen Sie ein final model, für das Sie dann auch mit `plot()`, `qqPlot()` und `influencePlot()` eine Regressionsdiagnostik durchführen. Schliessen Sie ggf. einflussreiche Datenpunkte aus und berechnen Sie das final model ein letztes Mal ohne diese Personen.

```

load("data/westost_skalen.Rdata")
stress_swk <- westost_skalen %>%
 select(ID, swk_akad, swk_selbstreg,
 swk_motselbst, swk_durch, swk_sozharm,
 swk_bez, stress_psych) %>%
 drop_na()
stress_swk
#> # A tibble: 283 x 8
#> ID swk_akad swk_selbstreg swk_motselbst swk_durch swk_sozharm swk_bez
#> <fct> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1 2 5 4 4.6 5 5.57 4.67
#> 2 14 4.86 4.5 4.6 5.33 4.71 4.67
#> 3 15 4 4.38 5.6 4.33 4.43 5.83
#> 4 17 6.14 5.62 6 6 6 6
#> 5 18 5.14 4.62 4.4 4.33 4.14 4.67
#> 6 19 5.43 4.62 5.2 6.33 6.29 6.5
#> # ... with 277 more rows, and 1 more variable: stress_psych <dbl>

```