

Kleine Einführung in R

R¹ ist vom Konzept her kein Statistikprogramm, sondern eine Programmierumgebung. Um das richtig zu genießen, ...

- fragen Sie nicht "Was kann R?", sondern "Was möchte ich?" und dann "Wie kann ich R dazu bringen, das zu tun?",
- arbeiten Sie mit Syntax und
- nutzen Sie die IDE RStudio², um mit R zu arbeiten.

R ist sehr leicht erweiterbar. Zusätzlich zur Basis-Version sind mehrere Tausend Zusatzpakete für spezielle Anwendungen veröffentlicht. Durch die große Gruppe von Anwendern, die ihre Spezialpakete zur Verfügung stellen, werden auch neue Methoden sehr schnell in R implementiert. Falls noch kein Paket für eine bestimmte Prozedur besteht, ist es relativ einfach, die Berechnung selbst zu programmieren.

1 R und RStudio installieren

Installieren Sie die Programme R und RStudio. Beides ist freie (und quelloffene) Software, die auf Windows-, Macintosh- und Linux-Betriebssystemen läuft.

- R wird unter der URL <http://cran.r-project.org> bereit gestellt. Laden Sie die Installationsdateien für Ihr Betriebssystem herunter und folgen Sie den entsprechenden Installationsanweisungen.
- RStudio finden Sie unter <http://www.rstudio.com/ide/download/desktop>. Auch hier wählen Sie die für Ihr Betriebssystem passende Version aus. Der Rest sollte selbsterklärend sein.

1 R Core Team, *R: A Language and Environment for Statistical Computing* (Vienna, Austria: R Foundation for Statistical Computing, 2015), <https://www.R-project.org/>.

2 RStudio Team, *RStudio: Integrated Development Environment for R* (Boston, MA: RStudio, Inc., 2012), <http://www.rstudio.com/>.

2 Erster Rundgang

Wenn Sie RStudio starten sehen Sie einen Bildschirm wie in Abbildung 1, der in vier Fenster unterteilt ist. Im linken unteren Fenster („Console“) läuft das Programm R. Alles andere drum herum sind Hilfsmittel, die RStudio zusätzlich bereit stellt.

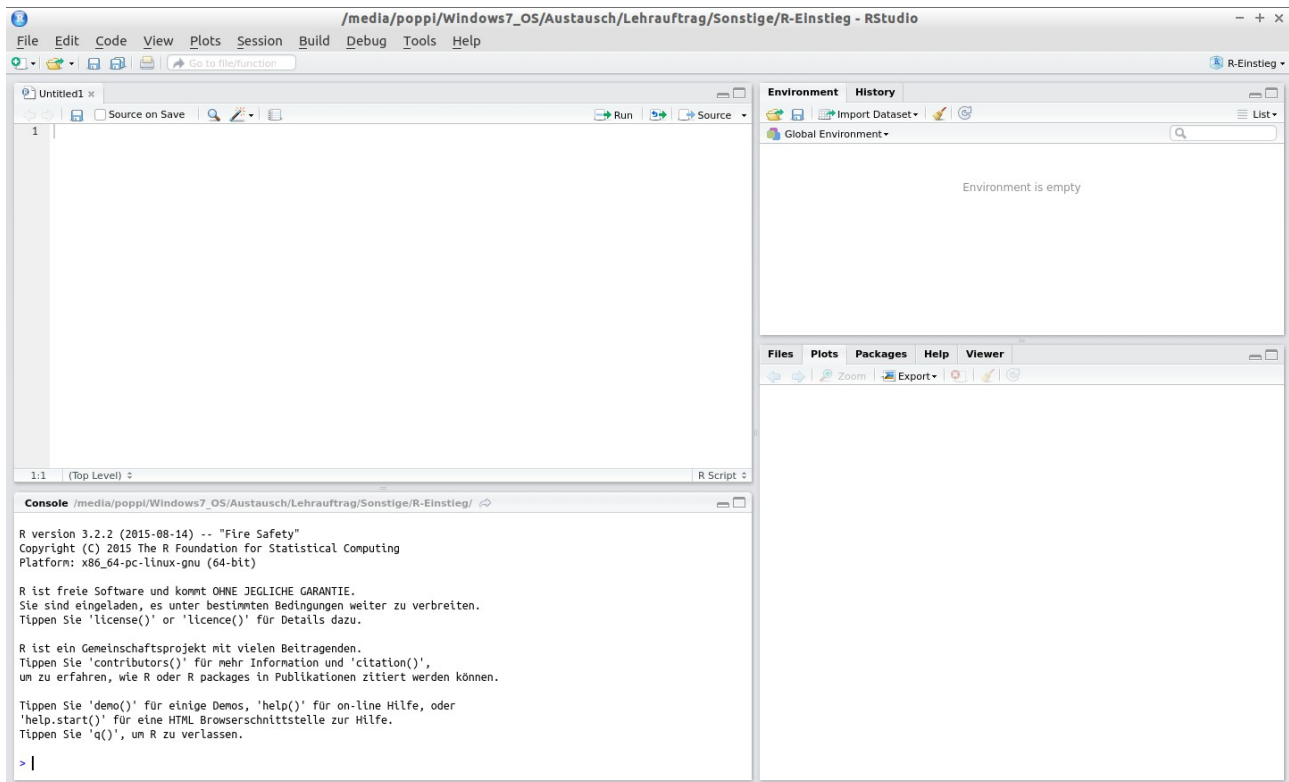


Abbildung 1: RStudio-Bildschirm

Probieren Sie Ihre erste statistische Berechnung aus, in dem Sie den Mittelwert der vier Altersangaben 24, 54, 30 und 18 berechnen. Geben Sie die Rechnung in die R-Konsole (also das linke untere Fenster) ein und drücken abschließend die Eingabe-Taste. Das Ganze sollte so aussehen.

```
> (24 + 54 + 19 + 30) / 4  
[1] 31.75
```

R als Taschenrechner

Die obere Zeile zeigt die Eingabe (geben Sie „>“ nicht mit an), die untere das Ergebnis (31.75). Beachten Sie, dass R immer einen Punkt als Dezimaltrenner verwendet.

Der Dezimaltrenner ist in R ein Punkt!

Die Zahl in eckigen Klammern nummeriert die Ergebnisse. In diesem Fall scheint das Überflüssig, weil es nur eines gibt. Das kann aber auch einmal anders sein.

UM DIE DATEN NICHT JEDES MAL NEU EINGEBEN zu müssen, können Sie sie einem sogenannten *Objekt* zuweisen:

```
> Alter <- c(24, 54, 30, 19)
```

Daten mit `c()` aneinander hängen und mit `<-` (Kleiner, Minus) einem *Objekt* zuweisen

Zunächst passiert nichts weiter. Sie können die Daten aber jederzeit wieder aufrufen, indem Sie „Alter“ eingeben.

```
> Alter
[1] 24 54 30 19
```

Nach dem Muster `Name <- Inhalt` können Sie beliebig viele Objekte im sogenannten *Workspace* ablegen. Um sich anzeigen zu lassen, welche Objekte im *Workspace* sind, geben Sie `ls()` ein.

```
> ls()
[1] "agePlot"      "Alter"        "Bereich"      "daten"
"daten2"       "daten3"       "datum"       "EmpVar"
[9] "fehlend2"     "fehlende"     "frami"        "framiWide"
"geschlecht"   "groesse"      "lowbwt"      "Mittelwert"
[17] "Modus"        "N"            "naPlot"      "navi"
"navi2"        "spider"       "Summe"       "Systemzeit"
[25] "VAS.A"        "VAS.B"        "VK"          "wert1"
"who"          "whoWide"      "x"           "y"
[33] "zahlen"       "ZufallChi"
```

Auch im oberen rechten Fenster werden unter *Environment* die Objekte in Ihrem Workspace angezeigt. In der linken Spalte der Name, rechts daneben weitere Informationen zum Objekt.

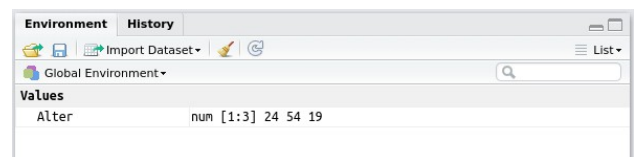


Abbildung 2: *Environment-Fenster in RStudio zeigt Objekte im Workspace an*

BERECHNEN SIE DEN MITTELWERT aus den gespeicherten Daten, indem Sie zunächst die Summe bilden, ...

```
> Summe <- sum(Alter)
```

`sum()`

... dann die Stichprobengröße ermitteln ...

```
> N <- length(Alter)
```

`length()`

... und schließlich beides durch einander teilen.

```
> Summe/N
[1] 31.75
```

WENN SIE DIE GLEICHE PROZEDUR immer wieder mit verschiedenen Daten ausführen wollen, empfiehlt es sich, eine neue Funktion (Befehl) zu erstellen.

```
> Mittelwert <- function(x) {
+   sum(x) / length(x)
+ }
```

Eigene Funktionen mit `function()`

Probieren Sie Ihre neue Funktion mit verschiedenen Daten aus.

```
> Mittelwert(Alter)
[1] 31.75
> Mittelwert(c(2, 5, 38, 1, 1, 23))
[1] 11.66667
```

Natürlich hat R auch eine eigene Funktion für den Mittelwert.

```
> mean(Alter)
[1] 31.75
```

mean()

SIE HABEN JETZT EINIGE BEFEHLE kennen gelernt: `c()`, `ls()`, `sum()`, `length()`, `funktion()` und `mean()`. Diese Befehle werden in R *Funktionen* genannt und setzen sich immer aus einem Namen und einem Teil in Klammern zusammen.

Aufbau von Funktionen

Beim Namen wird Groß- und Kleinschreibung berücksichtigt. „mean()“ wird z.B. als Funktion erkannt, nicht aber „Mean()“.

Innerhalb der Klammern muss in den meisten Fällen zuerst angegeben werden, auf welche Daten die Funktion angewendet werden soll, z.B. „mean(Alter)“. Weiterhin können hier Angaben gemacht werden, auf welche Art die Funktion ausgeführt werden soll (Argumente). Jede installierte Funktion hat eine Hilfe-Seite, in der diese Argumente beschrieben werden.

Hilfe finden

Zu jeder Funktion gibt es einen Hilfstext, den Sie mit einem vorangestellten `?` aufrufen können. Geben Sie z.B. `?mean` ein. Im rechten unteren Fenster wird die entsprechende Hilfeseite aufgerufen. Diese Hilfeseiten sind immer gleich aufgebaut. Als erstes gibt es unter *Description* eine Kurze Beschreibung, was der Funktion macht. Unter *Usage* sehen Sie, welche Eingaben erwartet werden und wie die Standardeinstellungen sind. Im Abschnitt *Arguments* werden alle Argumente beschrieben, die die Funktion enthalten kann. Bei manchen Hilfeseiten werden diese Argumente im Abschnitt *Details* noch genauer beschrieben. Unter *Value* wird beschrieben, welche Ausgaben die Funktion erzeugt. In *References* werden Quellenangaben gemacht. *See Also* verweist auf ähnliche oder verwandte Funktionen. Zu guter Letzt gibt *Examples* oft sehr hilfreiche Beispiele, um eine Funktion zu verstehen.

Hilfe mit ? Aufrufen

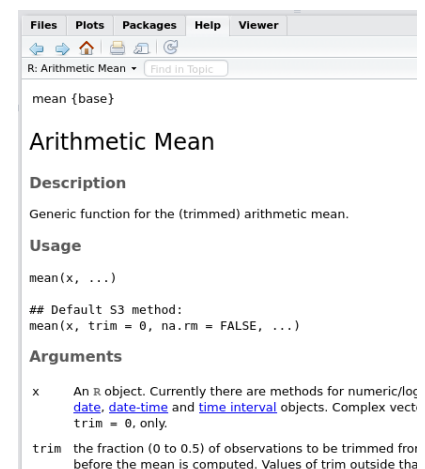


Abbildung 3: Ausschnitt aus der Hilfe-Seite für die Funktion `mean()`

Für die Funktion `mean()` muss man also mindestens angeben, auf welche Daten (`x`) sie angewendet wird. Mit dem Argument `trim = 0` wird festgelegt, dass die Daten nicht getrimmt werden und mit

`na.rm = FALSE` wird festgelegt, dass fehlende Werte (die werden in R mit „NA“ gekennzeichnet) nicht aus der Berechnung ausgeschlossen werden. Probieren Sie aus, was das bedeutet:

```
> mean(c(24, NA, 30, 19))
[1] NA
> mean(c(24, NA, 30, 19), na.rm = TRUE)
[1] 24.33333
```

In der Standardeinstellung werden fehlende Werte nicht entfernt und der Mittelwert kann deshalb nicht berechnet werden. Das Ergebnis ist wieder NA. Mit den Argument `na.rm=TRUE` werden die fehlenden Werte entfernt und der Mittelwert `NA,30,19,na.rm=TRUE` wird berechnet.

WENN SIE NICHT MEHR GENAU wissen, wie eine Funktion geschrieben wird, tippen Sie die ersten Buchstaben und lassen sich von der Autovervollständigungsfunktion von Rstudio helfen.

Autovervollständigung

Wenn sie damit nicht weiter kommen, können Sie gesuchte Inhalte mit zwei vorangestellten Fragezeichen eingeben. Es werden dann alle geladenen Hilfe-Seiten nach diesem Begriff durchsucht. Wenn Sie z.B. Hilfe zum Kolmogorov-Smirnov-Test suchen, geben Sie `??Kolmogor` ein. Es wird Ihnen angezeigt, dass es eine Funktion `ks.test()` gibt, die den gewünschten Test berechnet.

Inhalte mit ?? suchen

Wenn Sie innerhalb von R nicht die benötigte Hilfe bekommen, finden Sie sie bestimmt im Internet. Geben sie Ihre Frage nach dem Muster „R Inhalt“ in Ihre Suchmaschine ein. So finden Sie eigentlich immer jemanden, der die gleiche Frage schon einmal gestellt und kompetente Antworten bekommen hat.

Internet:

„R Suchbegriff“ in die Suchmaschine eingeben

Pakete

Es kann sein, dass Sie für eine gewünschte Prozedur zunächst keine Funktion finden. In den meisten Fällen gibt es dann ein Zusatzpaket mit so einer Funktion, dass Sie nutzen können. Zur Zeit stehen etwas über 7000 solcher Pakete auf den offiziellen R-Servern zur Verfügung. Sie können sie sich unter <https://cran.r-project.org/> ansehen.

Pakete auf <https://cran.r-project.org/>

Zuerst müssen Sie so ein Paket mit den Befehl

```
> install.packages('Paketname')
```

Installieren mit `install.packages()`

installieren. Der Befehl lädt das Paket automatisch vom CRAN-Server herunter und installiert es. Die Pakete sind als Spiegelabbilder weltweit auf verschiedenen Servern abgelegt. Beim ersten mal müssen sie einen (geografisch möglichst nahen) Server auswählen,


von dem Sie die Dateien herunterladen.

Dann müssen Sie das Paket mit

```
> library('Paketname')
```

laden. Das Installieren ist eine einmalige Angelegenheit. Das laden müssen Sie jedes mal erneut ausführen, wenn sie R neu starten.

RSTUDIO BIETET IHNEN im rechten unteren Fenster unter *Packages* grafische Hilfsmittel zur Paketverwaltung an. Es werden alle installierten Pakete angezeigt. Die geladenen Pakete sind mit einem Häkchen markiert. Wenn Sie auf den Namen eines Paketes klicken, kommen Sie zu den zugehörigen Hilfe-Seiten.

Mit  Install können sie neue Pakete installieren.

Laden mit `library()`

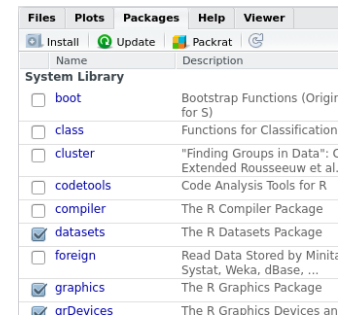


Abbildung 4: RStudio-Fenster Packages

Grafiken

R hat wunderbare Grafikfunktionen. Probieren Sie mal eine ganz einfache Variante aus.

```
> plot(Alter)
```

Verschaffen Sie sich einen Eindruck, was für Grafiken Sie mit R erstellen können, indem Sie

```
> demo(graphics)
```

oder

```
> library(lattice)
```

```
> demo(lattice)
```

eingeben.

Mit `demo()` bekommen Sie angezeigt, welche Beispiele Sie sich noch ansehen können.

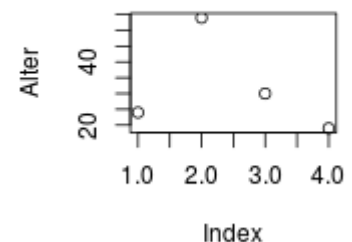


Abbildung 5 : Sehr einfache grafische Darstellung von Alter

History und Skript

Oft müssen geben Ihre Funktionen beim ersten Versuch noch nicht das gewünschte Ergebnis und Sie müssen noch etwas daran feilen, bis Sie zufrieden sind. Sie müssen dann nicht alles neu eintippen, sondern können mit der Pfeil-nach-oben-Taste die vorherigen Funktionen aufrufen.

UNTER *HISTORY* SEHEN SIE DIE VORHERIGEN FUNKTIONEN auch im oberen rechten RStudio-Fenster. Hier können Sie einzelne Funktionen

Mit ↑-Taste vorherige Befehle aufrufen.

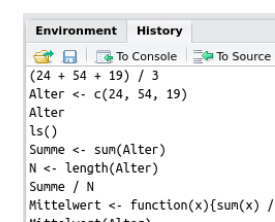
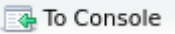
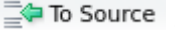




Abbildung 6: History-Fenster in RStudio

anwählen und mit der Schaltfläche  in die R-Konsole übertragen, dort bearbeiten und ausführen.

SIE KÖNNEN FUNKTIONEN MIT  auch in das linke obere Fenster kopieren. Hier wird dann eine Skript-Datei erstellt, in der Sie Ihre Syntax überarbeiten, kommentieren und als ganzes speichern können. Sie können Befehle, die Sie in das Skript geschrieben haben ausführen, indem Sie die Schaltfläche  klicken, oder mit der Tastenkombination *Strg-Einfügen*. Das gesamte Skript führen Sie mit der Schaltfläche  aus. Gespeicherte Skripte können Sie auch aus der R-Syntax heraus abrufen.

Syntax schreiben

```
> source('Dateiname')
```

Kommentieren Sie Ihre Syntax mit einem vorangestellten #, damit Sie und womöglich auch andere Personen später nachvollziehen können, was Sie gemacht haben.

Mit # kommentieren

Im Sinne der Nachvollziehbarkeit und Reproduzierbarkeit sollten Sie möglichst alle Arbeitsschritte vom Einlesen der Daten bis hin zur Darstellung der Ergebnisse als Skript programmieren.

Abbildung 6 zeigt ein Beispiel für so ein kommentiertes Skript.

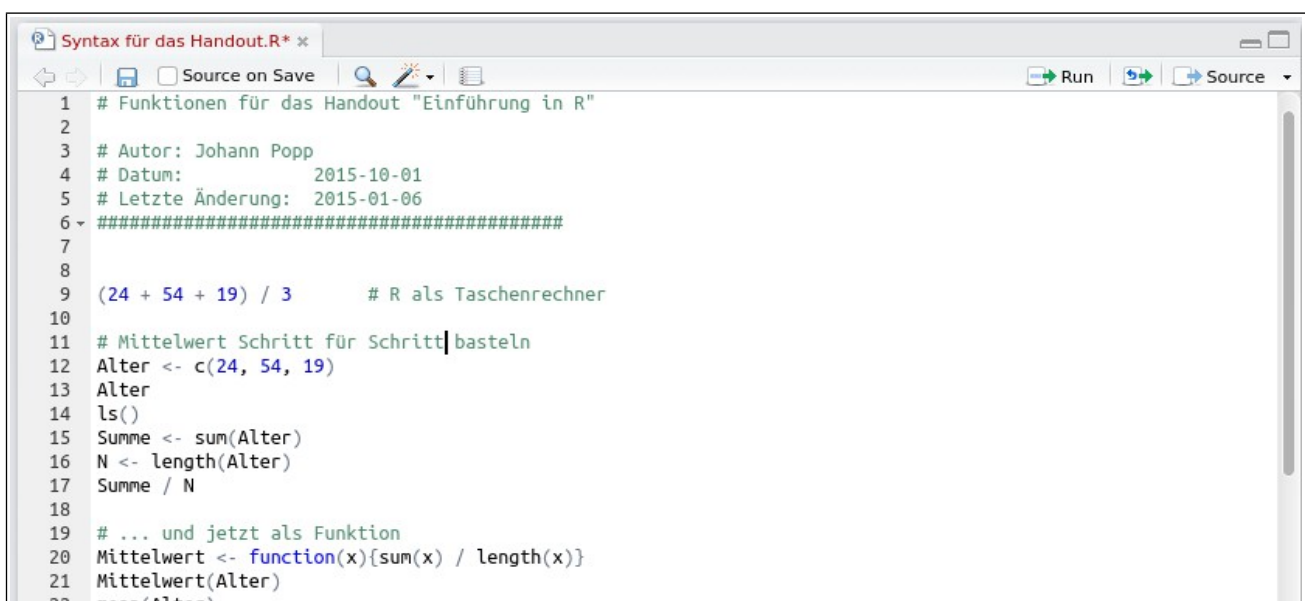


Abbildung 7: Beispiel eines kommentierten R-Skriptes

Ergebnisse speichern


Bevor Sie Ergebnisse speichern, sollten Sie den Dateipfad bestimmen, in dem gespeichert wird. R legt alle Ergebnisse in der sogenannten *Working Directory* ab. Um heraus zu bekommen, was die aktuelle Working Directory ist, verwenden Sie

Working Directory

```
> getwd()
```


Ändern können Sie die Working Directory mit


```
> setwd('Dateipfad')
```

Es empfiehlt sich für jedes Projekt einen Ordner anzulegen und ihn als Working Directory zu definieren, um alle Ergebnisse dort zu speichern. Mit RStudio kann man solche Projekte leicht Handhaben. Klicken Sie rechts oben  Project: (None) ▾ und dann *New Project...* (oder im Menü *File – New Project...*). Es folgen ein paar Dialogfelder. Für den Anfang starten Sie mit *New Directory – Empty Project* und geben sie im Dialogfeld *Create New Project* den Dateipfad und einen Ordernamen an.

RStudio setzt nicht nur die Working Directory auf den Projektordner, sondern ermöglicht auch weitere Projekthilfsmittel wie z.B. Versionskontrolle und Synchronisation mit Git und GitHub.

Rechts oben wird Ihnen jetzt der Name des Projektes angezeigt, z.B.

 R-Einstieg ▾. Wenn Sie dort klicken, können Sie leicht zwischen verschiedenen Projekten hin und her springen.

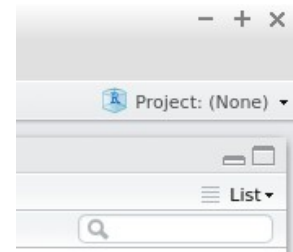
Auf jeden Fall sollten Sie ihr Syntax-Skript speichern. Über das Menü mit *File – Save* oder mit . R-Skripte haben die Dateinamenserweiterung „.R“. Technisch gesehen sind es aber einfache Textdateien.

Den R-Workspace mit allen erstellten Daten- und Ergebnisobjekten können Sie mit

```
> save.image('Dateiname.RData')
```

speichern.

Rstudio-Projekte



Skript speichern

R-Workspace speichern

Übung

In einem fiktiven Experiment wird das Ausmaß der Angst von Personen mit einer visuellen Analogskala von 0-10 einmal gemessen, wenn ihnen eine Vogelspinne gezeigt wird (A) und ein anderes mal, wenn ihnen das Foto einer Vogelspinne gezeigt wird (B)³.

- Geben Sie die Daten in R ein und führen Sie einen T-Test durch.
- Interpretieren Sie das Ergebnis
- Schauen Sie sich das Skript Ihres Sitznachbarn an
 - Ist es Nachvollziehbar?
 - Ist es Korrekt?

VAS A	VAS B
41	46
63	33
31	31
28	20
46	33
24	8
42	30

3

Andy P Field, Jeremy Miles, und Zoë Field, *Discovering Statistics Using R* (London; Thousand Oaks, Calif.: Sage, 2012).

3 Daten einlesen und aufarbeiten

Datenformate

R kann Daten in verschiedenen Formaten verarbeiten:

- `numeric()`: Zahlen (eine Sonderform ist `integer()` für ganze Zahlen)
- `character()`: Schriftzeichen (entspricht *string* in SPSS)
- `logical()`: *TRUE* / *FALSE*
- `factor()`: Kategorien (entspricht Variablen mit Wertelabels in SPSS)
- `POSIXlt()`, `POSIXct()`: Datums- und Zeitangaben.

Datenformate
`numeric()`,
`character()`,
`logical()`,
`factor()`,
`POSIXlt()`, `POSIXct()`

Es gibt kein spezielles Format für ordinale Daten. Man kann sie als Zahlen (`numeric`) oder als sortierte Faktoren speichern:

Ordinale Daten

```
> x <- ordered(c("nie", "oft", "oft", "selten"),
>             levels = c("nie", "selten", "oft"))
> class(x)
[1] "ordered" "factor"
> x
[1] nie    oft    oft    selten
Levels: nie < selten < oft
```

BEIM EINLESEN ORDNET R AUTOMATISCH ein passendes Datenformat zu. Um zu prüfen, welches Datenformat das ist, tippen Sie `class()`.

Datenformat abrufen mit
`class()`

```
> groesse <- c(172, 168, 184, 175)
> class(groesse)
[1] "numeric"
```

Dem Objekt `groesse` wurde hier automatisch das Datenformat *numeric* zugewiesen. Um das Format zu ändern, können Sie `as.character`, `as.numeric`, `as.logical`, `as.factor` benutzen.

Datenformat ändern
`as. ... ()`

```
> as.character(groesse)
[1] "172" "168" "184" "175"
> as.factor(groesse)
[1] 172 168 184 175
Levels: 168 172 175 184
```

Das funktioniert allerdings nicht immer.

```
> geschlecht <- c("w", "m", "m", "w")
> class(geschlecht)
[1] "character"
> as.numeric(geschlecht)
[1] NA NA NA NA
```

DIE KLASSISCHE DATENMATRIX, bei der jede Zeile eine statistische Einheit darstellt und jede Spalte eine Variable ist in R ein Objekt der Klasse `data.frame()`.

Datenmatrix:
`data.frame()`

```
> daten <- data.frame(Alter, groesse, geschlecht, x)
> daten
  Alter groesse geschlecht      x
1    24    172          w    nie
2    54    168          m    oft
3    30    184          m    oft
4    19    175          w selten
```

Das funktioniert natürlich nur, wenn die Variablen *Alter*, *groesse*, *geschlecht* und *x* bereits in Ihrem Workspace vorhanden sind. Falls nicht, können Sie die Daten auch innerhalb des `data.frame()`-Befehls eingeben.

```
> daten2 <- data.frame(Alter = c(24, 54, 30, 19),
>                      groesse = c(172, 168, 184, 175),
>                      geschlecht = c("w", "m", "m", "w"),
>                      x = c("nie", "oft", "oft", "selten"))
  Alter groesse geschlecht      x
1    24    172          w    nie
2    54    168          m    oft
3    30    184          m    oft
4    19    175          w selten
```

ANDERE OBJEKTKLASSEN, die Daten in mehreren Dimensionen enthalten sind

`matrix()`,
`array()`,
`list()`

- `matrix()`: Auch eine zweidimensionale Datenmatrix. Im Gegensatz zu `data.frame()` haben aber alle Spalten das gleiche Datenformat (in der Regel `numeric()`).
- `array()`: Kann Daten (des gleichen Datenformates) nicht nur in zwei, sondern beliebig vielen Dimensionen enthalten.
- `List()`: Hier können Sie Objekte beliebiger Länge und beliebiger Formate in einem Objekt zusammen führen, z.B. einen Datensatz als `data.frame()` und eine Beschreibung des Datensatzes als `character()`.

Daten einlesen

Eine Methode, Daten in R einzulesen haben Sie bereits kennen gelernt, nämlich durch Eingabe als Skript oder direkt in die Konsole. Wenn Sie einmal ein Objekt der Klasse `data.frame()` erstellt haben, können Sie mit `fix()` direkt in der Datenmatrix Werte

`data.frame()`

ändern oder ergänzen.

```
> fix(daten2)
```

```
fix()
```

SOBALD DIE DATENMENGEN aber etwas größer werden, ist es besser, die Daten mit einem Datenbankprogramm (z.B. Libre Office Base, Epi Info, Epidata) zu erfassen und dann in R zu übertragen. Die zuverlässigste Methode der Datenübertragung ist es meist, die eingegebenen Daten als Textdatei (z.B. csv-Datei) zu speichern und aus diesem Format in R einzulesen.

Textdateien einlesen

Textdateien der unterschiedlichsten Formate liest R mit der Funktion `read.table()`. Die wichtigsten Argumente dieser Funktion sind

```
read.table()
```

- `file`: Dateiname (und ggf. Pfad)
- `header` (FALSE/TRUE): Enthält die erste Zeile Variablennamen?
- `sep`: Mit welchem Zeichen werden die Spalten getrennt?
- `dec`: Welches Zeichen ist der Dezimaltrenner?

Versuchen Sie das einmal mit einer Datendatei aus der Framingham-Studie. Kopieren Sie die Datei zunächst in Ihren Arbeitsornder. (Sind Sie unsicher, welcher das ist? → `getwd()`)

```
> frami <- read.table("framingham.csv", header = TRUE,
                      sep = ",", dec = ".")
```

Die Datei heißt in diesem Fall „framingham.csv“. Die erste Zeile enthält die Variablennamen, das Trennzeichen ist ein Komma und der Dezimaltrenner ein Punkt.

Lassen Sie sich die ersten paar Zeilen des Datensatzes anzeigen, um zu prüfen, ob Sie alles richtig gemacht haben.

```
> head(frami)
```

Eine Verkürzung für csv-Dateien ist die Funktion `read.csv()`, bei der genau die oben genannten Einstellungen für `header`, `sep` und `dec` vorgegeben sind. Hier muss man dann nur noch den Dateinamen angeben:

```
read.csv()
```

```
> frami <- read.csv("framingham.csv")
```

Bei Kontinental-Europäischen Ländereinstellungen werden Dezimalstellen mit einem Komma getrennt und csv-Dateien verwenden das Semikolon als Trenner. Für solche Dateien kann man `read.csv2()` verwenden.

```
read.csv2()
```

Tabulatorgetrennte Dateien können mit `read.delim()` und `read.delim2()` eingelesen werden.

```
read.delim()
read.delim2()
```

WENN DIE DATEI IN EINEM ANDEREN ORDNER gespeichert ist, kann es mühsam werden, den ganzen Dateipfad einzutragen. Mit Hilfe von `file.choose()` wird ein Dateimanager geöffnet, in dem man sich bis zur gewünschten Datei voran klicken kann.

```
file.choose()
```

SOGAR DIREKT AUS DEM INTERNET können Dateien eingelesen werden. Versuchen Sie es einmal mit Beispieldaten aus Andy Fields „Discovering Statistics using R“.

Daten aus dem Internet
einlesen

```
>
read.delim("http://studysites.uk.sagepub.com/dsur/study/
DSUR%20Data%20Files/Chapter%209/SpiderWide.dat")
  picture real
1       30   40
2       35   35
3       45   50
4       40   55
5       50   65
6       35   55
7       55   50
8       25   35
9       30   30
10      45   50
11      40   60
12      50   39
```

ANDERS HERUM KÖNNEN SIE DATEN als Textdateien mit den entsprechenden Befehlen `write.table()`, `write.csv()`, `write.delim()` speichern.

Speichern mit
`write.table()`,
`write.csv()`,
`write.delim()`

Spezielle Statistikformate einlesen

Um Daten aus anderen Statistikprogrammen direkt einzulesen, ohne den Umweg über Textdateien zu gehen, können Sie das Paket `foreign` verwenden. Das Paket liest und schreibt Dateien in den Formaten von SPSS, Stata, SAS und anderen.

Paket `foreign`

Laden Sie z.B. Fields Spinnen-Datensatz aus dem SPSS-Format. Kopieren Sie sich dazu zuerst die Datei „SpiderRM.SAV“ in Ihren Arbeitsordner.

Installieren Sie nun das Paket `foreign`. (Das müssen Sie nur beim ersten mal machen.)

```
install.packages(
  "foreign")
```

```
> install.packages("foreign")
```

Jetzt aktivieren Sie das Paket. (Das müssen Sie jedes mal machen, wenn Sie R neu starten.)

```
library(foreign)
```

```
> library(foreign)
```

Lesen Sie die SPSS-Datei

`read.spss()`

```
> daten3 <- read.spss("SpiderRM.SAV")
> daten3

$PICTURE
[1] 30 35 45 40 50 35 55 25 30 45 40 50

$REAL
[1] 40 35 50 55 65 55 50 35 30 50 60 39

attr(,"label.table")
attr(,"label.table")$PICTURE
NULL

attr(,"label.table")$REAL
NULL

attr(,"variable.labels")
          PICTURE          REAL
"Picture of Spider"  "Real Spider"
```

Das sieht noch nicht so schön aus. Es muss noch in einen `data.frame` umgewandelt werden:

```
> spider <- as.data.frame(daten3)
> spider
   PICTURE REAL
1        30   40
2        35   35
3        45   50
4        40   55
5        50   65
6        35   55
7        55   50
8        25   35
9        30   30
10       45   50
11       40   60
12       50   39
```

Daten aufbereiten

Sich einen Überblick verschaffen

Als erstes sollten Sie sich einen Überblick über den geladenen Datensatz verschaffen. Schauen Sie sich den Framingham-Datensatz noch einmal mit `head` an.

`head()`

```
> head(frami)
```

	RANDID	SEX	PERIOD	TIME	TOTCHOL	AGE	SYSBP	DIABP	CURSMOKE	CIGPDAY	BMI	DIABETES	BPMEAS	HEARTRT	GLUCOSE	PREVCHD	PREVAP	PREVMI	PREVSTRK
1	2448	m	1	0	195	39	106.0	70.0	0	0	26.97	0	0	80	77	0	0	0	0
2	2448	m	3	4628	209	52	121.0	66.0	0	0	NA	0	0	69	92	0	0	0	0
3	6238	f	1	0	250	46	121.0	81.0	0	0	28.73	0	0	95	76	0	0	0	0
4	6238	f	2	2156	260	52	105.0	69.5	0	0	29.43	0	0	80	86	0	0	0	0
5	6238	f	3	4344	237	58	108.0	66.0	0	0	28.50	0	0	80	71	0	0	0	0
6	9428	m	1	0	245	48	127.5	80.0	1	20	25.34	0	0	75	70	0	0	0	0
	PREVHYP	HDLC	LDLC	DEATH	ANGINA	HOSPMI	MI_FCHD	ANYCHD	STROKE	CVD	HYPERTEN	TIMEAP	TIMEMI	TIMEMIFC	TIMECHD	TIMESTRK	TIMECVD	TIMEDTH	TIMEHYP
1	0	NA	NA	0	0	1	1	1	0	1	0	8766	6438	6438	6438	8766	6438	8766	8766
2	0	31	178	0	0	1	1	1	0	1	0	8766	6438	6438	6438	8766	6438	8766	8766
3	0	NA	NA	0	0	0	0	0	0	0	0	8766	8766	8766	8766	8766	8766	8766	8766
4	0	NA	NA	0	0	0	0	0	0	0	0	8766	8766	8766	8766	8766	8766	8766	8766
5	0	54	141	0	0	0	0	0	0	0	0	8766	8766	8766	8766	8766	8766	8766	8766
6	0	NA	NA	0	0	0	0	0	0	0	0	8766	8766	8766	8766	8766	8766	8766	8766

Sie bekommen für jede Variable im Datensatz die ersten sechs Einträge angezeigt. (Mit `tail()` können Sie sich auch die letzten Einträge anzeigen lassen.)

`tail()`

DIE FUNKTION `str()` gibt Ihnen detaillierte Informationen über die Struktur des Datensatzes (oder eines anderen Objektes):

`str()`

```
> str(frami)
```

Zunächst erfahren Sie, dass es sich bei dem Objekt um einen `data.frame` mit 11637 statistischen Einheiten und 38 Variablen handelt. Dann wird für jede Variable das Datenformat angegeben (bei Faktoren werden auch die Wertelabels) und die ersten Einträge werden angezeigt.

Abbildung 8: Das Equivalent zu `str()` im RStudio Environment

Die gleichen Angaben bekommen Sie auch rechts oben im Environment-Fenster von RStudio (Abbildung 8)

DIE FUNKTION `summary()` gibt eine inhaltliche Zusammenfassung der Daten.

`summary()`

```
> summary(frami)
```

Für alle Variablen der Klasse `numeric` (bzw. `integer`) werden die Quartilen und der Median angegeben. Für die Variable `SEX` (Datenformat `factor`) werden Häufigkeiten angegeben. Ebenso werden Häufigkeiten für Variablen der Klasse `logical` (`CURSMOKE` etc.) genannt. Zusätzlich wird die Anzahl der Missings (`NA`) angezeigt, falls vorhanden.

Die Funktion `summary()` ist ein typisches Beispiel für die sogenannte objektorientierte Programmierung in R. Ein und die selbe Funktion produziert verschiedene Ausgaben, abhängig von der Objektklasse des Objektes, auf das sie angewandt wird. Besonders `summary()` kann man auf sehr viele verschiedene Objektklassen anwenden und bekommt immer verschiedene inhaltlich sinnvolle Ergebnisse.

Objektorientierte
Programmierung

Mit `View()` wird ein neues Fenster mit der Datenmatrix geöffnet.

Datenmatrix anzeigen lassen:
`View()`

```
> View(frami)
```

Das gleiche können Sie auch erreichen, wenn Sie im RStudio Environment (rechts oben) auf den Namen eines `data.frame` klicken.

Fehlende Werte

Fehlende Werte werden in R mit `NA` kodiert.

`NA`

Einige Funktionen schließen fehlende Werte in der Grundeinstellung aus:

```
> fehlende <- c(3, 2, 2, NA, 4, 1, 6, 6, NA, 1)
> table(fehlende)
fehlende
1 2 3 4 6
2 2 1 1 2
> table(fehlende, useNA = "ifany")
fehlende
  1      2      3      4      6 <NA>
2      2      1      1      2      2
```

Andere Funktionen ergeben `NA`, wenn die Fehlenden nicht explizit ausgeschlossen werden

```
> sum(fehlende)
[1] NA
> sum(fehlende, na.rm = TRUE)
[1] 25
```

Wieder andere Funktionen machen gar keinen Unterschied zwischen fehlenden und nicht fehlenden werten.

```
> length(fehlende)
[1] 10
```

Mit der Funktion `is.na()` besteht aber immer die Möglichkeit fehlende Werte zu finden und ggf. auszuschließen.

`is.na()`
findet Fehlende Werte

```
> is.na(fehlende)
[1] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
TRUE FALSE
> fehlende[is.na(fehlende)]
[1] NA NA
> fehlende[!is.na(fehlende)]
[1] 3 2 2 4 1 6 6 1
```

DIE MÖGLICHKEIT BESONDERE KODIERUNGEN für Fehlende Werte zu definieren (z.B. „999“ = fehlend, „888“ = nicht anwendbar), besteht nicht. Man kann solche Kodierungen bequem in `NA` umwandeln:

Spezielle Kodierungen in `NA` umwandeln

```
> fehlend2 <- data.frame(var1 = c(3,5,1,999, 2, 999),
                        var2 = c(999,6,23,999,999,8))
> fehlend2
  var1 var2
1    3 999
2    5    6
```



```
3    1    23
4  999  999
5    2    999
6  999    8
```

Jetzt kommt die Umwandlung.

```
> fehlend2[fehlend2 == 999] <- NA
> fehlend2
  var1 var2
1    3  NA
2    5    6
3    1   23
4   NA  NA
5    2  NA
6   NA    8
```

BERECHNUNGEN KÖNNEN MANCHMAL nicht definierte numerische Ergebnisse ergeben. Diese werden als NaN (Not a Number) klassifiziert.

NaN

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

HIER SIND EIN PAAR FUNKTIONEN, um sich einen Überblick über Missings zu machen. Sie werden an dieser Stelle nicht im Detail erklärt, sondern nur beispielhaft am Framingham-Datensatz vorgestellt.

Missings pro Variable (Spalte) in absoluten oder relativen Zahlen

Missings pro Variable

```
> apply(frami, 2, function(x) sum(is.na(x)))
> apply(frami, 2,
        function(x) sum(is.na(x)) / length(x))
```

Missings pro statistischer Einheit in absoluten oder relativen Zahlen.

Missings pro Zeile

```
> apply(frami, 1, function(x) sum(is.na(x)))
> apply(frami, 1,
        function(x) sum(is.na(x)) / length(x))
```

Graphische Darstellung der Verteilung von Missings im Datensatz. Diesen Plot halte ich für so hilfreich, dass ich ihn gleich als Funktion programmiert habe. Wenn Sie den Code ausführen, haben Sie die Funktion `naPlot()`, die Sie später auch auf andere Datensätze anwenden können.

Verteilung der Missings im Datensatz

```
> naPlot <- function(x) {
  image(!is.na(t(x)), xlab = "Variablen",
        ylab = "statistische Einheiten",
        main = "Fehlende Werte im Datensatz",
        xaxt = "n", yaxt = "n")
  axis(1, at = seq(0, 1, length.out = ncol(x)),
        labels = names(x), cex.axis = 0.5, las = 3)
}
```

Eigene Funktion:
`naPlot()`

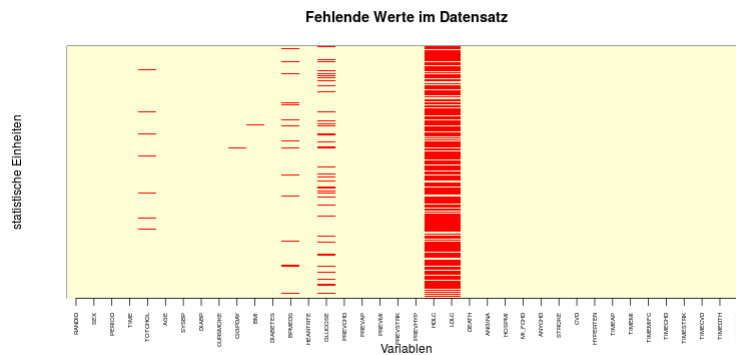


Abbildung 9 : Verteilung der Missings im Framingham-Datensatz

```
> naPlot(frami)
```

```
> plot(apply(frami, 2,
             function(x) sum(is.na(x)) / length(x)),
       ylab = "Anteil Missings pro Variable")
```

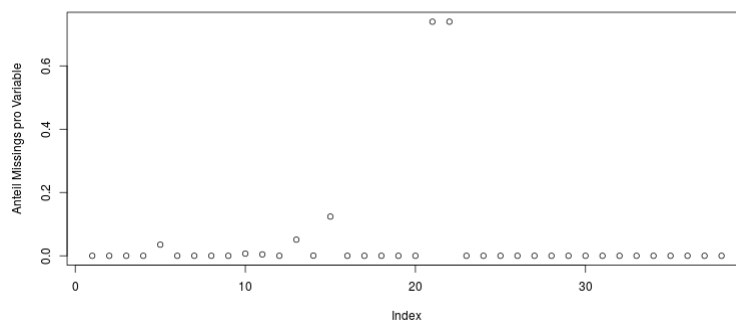


Abbildung 10 : Anteil Missings pro Variable, Framingham-Datensatz

UM MISSINGS MIT HILFE MULTIPLER IMPUTATION aufzufüllen, kann

Multiple Imputation mit
`mice`

man das Paket `mice`⁴ verwenden.

Einzelne Variablen anwählen

Lassen Sie sich das geringste Alter aus dem Framingham-Datensatz anzeigen.

```
< min(AGE)
```

Das funktioniert nicht, weil R das Objekt AGE nicht findet. Um zu sehen, welche Objekte sich im Workspace befinden, tippen Sie

```
ls()
```

```
> ls()
[1] "agePlot"      "Alter"        "Bereich"      "daten"
"daten2"       "daten3"       "datum"       "EmpVar"
[9] "fehlend2"     "fehlende"     "frami"       "framiWide"
"geschlecht"   "groesse"      "lowbwt"      "Mittelwert"
[17] "Modus"        "N"            "naPlot"      "navi"
"navi2"        "spider"       "Summe"       "Systemzeit"
[25] "VAS.A"        "VAS.B"        "VK"          "wert1"
"who"          "whoWide"      "x"           "y"
[33] "zahlen"       "ZufallChi"
```

In der Tat wird kein Objekt AGE angezeigt, weil es sich innerhalb eines anderen Objektes befindet, nämlich innerhalb des data.frame „frami“

Um Variablen innerhalb eines Datensatzes anzuwählen, müssen Sie den Namen des data.frame angeben, gefolgt von einem Dollarzeichen, gefolgt vom Variablennamen (z.B. `frami$AGE`)

```
Daten$Variable
```

```
> min(frami$AGE)
[1] 32
```

Eine Alternative ist die Funktion `with()`:

```
with()
```

```
> with(frami, min(AGE))
[1] 32
```

Um es sich zu ersparen, immer wieder den Datensatz explizit angeben zu müssen, kann man mit `attach()` die Variablen des Datensatzes auch bis auf weiteres dem Suchpfad hinzufügen.

```
attach()
```

```
> attach(frami)
> min(AGE)
[1] 32
```

Ich würde diese Methode jedoch nicht empfehlen, weil man damit leicht den Überblick darüber verliert, aus welchem Datensatz die Variablen stammen, die man verwendet. Auf jeden Fall sollte man die Variablen am Schluss mit `detach()` wieder aus dem Suchpfad

```
detach()
```

⁴ Stef Buuren und Karin Groothuis-Oudshoorn, „mice: Multivariate imputation by chained equations in R“, *Journal of statistical software* 45, Nr. 3 (2011), <http://doc.utwente.nl/78938/>.

entfernen

```
> detach(frami)
```

Im Datensatz navigieren

Um zu bestimmten Daten innerhalb von Datensätzen (und auch anderen Objekten) zu gelangen, können Sie eckige Klammern einsetzen. Probieren Sie z.B einmal

```
> zahlen <- c("eins", "zwei", "drei", "vier", "fünf")
> zahlen[3]
[1] "drei"
```

Mit dem Index 3 innerhalb der eckigen Klammern haben Sie den dritten Eintrag des Objektes *zahlen* ausgewählt.

Da ein Datensatz zweidimensional ist, müssen Sie hier zwei Angaben machen, getrennt durch ein Komma. Die erste Angabe steht für die Zeilen (statistische Einheiten), die zweite für die Spalten (Variablen).

```
> navi <- data.frame(zahlen, buchst = c("a", "b", "c",
"d", "e"))
> navi
  zahlen buchst
1   eins      a
2   zwei      b
3   drei      c
4   vier      d
5   fünf      e
> navi[3, 2]
[1] c
Levels: a b c d e
```

Hier wurde der Eintrag der dritten Zeile in der zweiten Spalte des Datensatzes *navi* ausgewählt, nämlich *c*.

Wenn Sie alle Einträge einer Dimension anwählen wollen, machen Sie keine Angabe; z.B alle Variablen der dritten Zeile:

```
> navi[3, ]
  zahlen buchst
3   drei      c
```

Um alle Einträge außer dem Angegebenen anzuwählen, setzen Sie ein Minuszeichen vor die Angabe:

```
> navi[-3, ]
  zahlen buchst
1   eins      a
2   zwei      b
4   vier      d
5   fünf      e
```

Einen zusammenhängenden Datenbereich wählen Sie nach dem

Mit [] Daten auswählen

Bei data.frame:
[Zeile, Spalte]

Keine Angabe um alle
Einträge einer Dimension
auszuwählen

Alle außer:
Minuszeichen

Datenbereich:
[start:ende]

Muster start:ende aus:

```
> navi[2:4, ]
  zahlen buchst
2   zwei      b
3   drei      c
4   vier      d
```

Mehrere nicht zusammenhängende Einträge wählen Sie aus, indem Sie die Indices mit c() verknüpfen:

```
> navi[c(1, 3), ]
  zahlen buchst
1   eins      a
3   drei      c
```

Mehrere anwählen:
[c ()]

Sie können sie Auswahl auch nutzen, um die Reihenfolge zu ändern:

```
> navi[5:1, 2:1]
  buchst zahlen
5      e   fünf
4      d   vier
3      c   drei
2      b   zwei
1      a   eins
```

Reihenfolge ändern

Daten auswählen, die bestimmte Bedingungen erfüllen

Sie können beliebige logische Bedingungen in die eckigen Klammern schreiben. Wählen Sie z.B. aus dem Datensatz navi alle Zeilen aus, bei denen die Variable zahlen „zwei“ enthält.

```
> navi[zahlen == "zwei", ]
  zahlen buchst
2   zwei      b
```

Beachten Sie hier das doppelte Gleichheitszeichen für „ist gleich“ steht. Sie können natürlich auch andere Bedingungen anwenden:

Bedingung	Symbol
Ist gleich	==
Ungleich	!=
Größer	>
Größer, gleich	>=
Kleiner	<
Kleiner, gleich	<=
Logische Verknüpfung von Bedingungen	
UND	&
ODER	(AltGr <)

Tabelle 1: Symbole für Bedingungen und logische Verknüpfungen

Datensatz sortieren

Um einen Datensatz zu sortieren, fügen Sie die Funktion `order()` in die eckigen Klammern ein:

```
> navi[order(navi$zahlen), ]
```

	zahlen	buchst
3	drei	c
1	eins	a
5	fünf	e
4	vier	d
2	zwei	b

`order()`

In diesem Beispiel wird der Datensatz „navi“ nach der (alphabetischen) Reihenfolge der Variable „zahlen“ sortiert. Um das etwas genauer zu verstehen, schauen Sie sich einmal nur das Ergebnis von `order()` an:

```
> order(navi$zahlen)
[1] 3 1 5 4 2
```

Die `order()`-Funktion gibt Ihnen die Abfolge der Indizes an, wenn nach „zahlen“ sortiert wird. Wenn Sie diese Abfolge in die eckigen Klammern einfügen, wird der gesamte Datensatz danach sortiert. Sie können das Überprüfen, indem Sie die Abfolge selbst eintippen:

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

	zahlen	buchst
3	drei	c
1	eins	a
5	fünf	e
4	vier	d
2	zwei	b

Variablen hinzufügen

Es gibt verschiedene Möglichkeiten, einem Datensatz neue Variablen hinzuzufügen. Sie können z.B. einen Vektor definieren und ihn dann mit `cbind()` an den Datensatz anhängen

```
> wert1 <- c(1, 2, 3, 4, 5)
```

```
> cbind(navi, wert1)
```

	zahlen	buchst	wert1
1	eins	a	1
2	zwei	b	2
3	drei	c	3
4	vier	d	4
5	fünf	e	5

`cbind()`

(`cbind()` steht für *column bind*, also Spalten verbinden. Analog gibt es `rbind()` *row bind*, um Zeilen anzuhängen.)

Im oben gezeigten Beispiel wurde „wert1“ allerdings nur für den Moment an den Datensatz angehängt. Um das dauerhaft zu

speichern, müssen Sie das ganze Konstrukt einem Objekt zuweisen.

```
> navi <- cbind(navi, wert1)
```

Jetzt haben sie das Objekt „navi“ aktualisiert, indem Sie ihm neue Werte zugewiesen haben.

```
> navi
  zahlen buchst wert1
1   eins      a      1
2   zwei      b      2
3   drei      c      3
4   vier      d      4
5   fünf      e      5
```

Eine andere Methode ist es, eine neue Variable zu benennen und ihr Daten zuzuweisen.

```
> navi$wert2 <- c(1, 10, 100, 1000, 10000)
> navi
  zahlen buchst wert1 wert2
1   eins      a      1      1
2   zwei      b      2     10
3   drei      c      3    100
4   vier      d      4   1000
5   fünf      e      5  10000
```

```
datensatz$variable
<- c()
```

Werte berechnen

Sie können neue Werte aus den bestehenden berechnen.

```
> navi$wert1 * navi$wert2
[1]      1     20     300    4000   50000
```

Die Werte werden Zeile für Zeile berechnet. Wenn dabei ein Vektor kürzer ist als der andere, wird der kürzere wiederholt.

```
> navi$wert1 * 10
[1] 10 20 30 40 50
```

Wenn die Länge des kürzeren Vektors kein Teiler der Länge des längeren Vektors ist, erhalten Sie eine Warnmeldung.

```
> navi$wert1 * c(1, 10)
[1] 1 20 3 40 5
Warnmeldung:
In navi$wert1 * c(1, 10) : Länge des längeren Objektes
ist kein Vielfaches der Länge des kürzeren Objektes
```

Um die Berechnung dem Datensatz hinzuzufügen müssen Sie sie wieder einer Variable zuweisen.

```
> navi$wert1mal2 <- navi$wert1 * navi$wert2
```

Wichtige mathematische Funktionen in R:

Berechnung	Symbol ; Funktion
Addition	+ ; <code>sum(x)</code>
Subtraktion	- ; <code>diff(x)</code>
Produkt	* ; <code>prod(x)</code>
Division	/
Quadrieren (x^2)	x^2
x^a	x^a
Wurzel	$x^{(1/2)}$; <code>sqrt(x)</code>
Natürlicher Logarithmus	<code>log(x)</code>
e^x	<code>exp(x)</code>
Log zur Basis a	<code>log(x, a)</code>
Fakultät	$x!$
Sinus	<code>sin(x)</code>
Cosinus	<code>cos(x)</code>
Tangens	<code>tan(x)</code>
Runden	<code>round(x, digits = 0)</code>
Absoluter Wert ohne Vorzeichen	<code>abs(x)</code>

Tabelle 2: Wichtige mathematische Funktionen in R

Variablen umkodieren

Mit Bedingungen innerhalb von eckigen Klammern können Sie Variablen umkodieren.

```
> navi$wertGr[navi$wert1 < 4] <- "< 4"
> navi$wertGr[navi$wert1 >= 4] <- ">= 4"
> navi
  zahlen buchst wert1 wert2 wert1mal2 wertGr
1   eins      a     1     1         1    < 4
2   zwei      b     2    10        20    < 4
3   drei      c     3   100       300    < 4
4   vier      d     4  1000      4000   >= 4
5   fünf      e     5 10000     50000   >= 4
```

Gruppierungen von numerischen Variablen können Sie leicht mit `cut()` vornehmen.

`cut()`

```
> cut(navi$wert1, breaks = 2)
[1] (0.996,3] (0.996,3] (0.996,3] (3,5]      (3,5]
Levels: (0.996,3] (3,5]
```

Die Funktion `cut()` nimmt sich hier die Variable `wert1` und teilt sie in der Mitte zwei Teile auf. Sie können die Schnittpunkte aber auch beliebig setzen. Dazu müssen Sie sie inklusive eines Start- und eines

Endpunktes angeben, z.B. `c(0, 4, 6)`.

```
> cut(navi$wert1, breaks = c(0, 4, 6))
[1] (0,4] (0,4] (0,4] (0,4] (4,6]
Levels: (0,4] (4,6]
```

Sie erhalten jetzt Gruppierete Werte mit dem Schnittpunkt 4. Werte genau am Schnittpunkt (in diesem Falle 4) sind der unteren Gruppe zugeordnet. Die Aufteilung ist also 0-4 und >4-6. Um ihn der höheren Gruppe zuzuordnen, kann man das Argument `right = FALSE` hinzufügen.

```
> cut(navi$wert1, breaks = c(0, 4, 6), right = FALSE)
[1] [0,4) [0,4) [0,4) [4,6) [4,6)
Levels: [0,4) [4,6)
```

Schließlich können Sie den einzelnen Kategorien mit dem Argument `labels` noch hübschere Namen geben:

```
> cut(navi$wert1, c(0, 4, 6), labels = c("0-4", ">4"))
[1] 0-4 0-4 0-4 0-4 >4
Levels: 0-4 >4
```

Besonders elegant können Sie `cut()` mit `quantile()` verknüpfen, um eine Gruppierung in Quartilen (oder andere Perzentilen) zu bewirken.

`quantile()`

```
> cut(navi$wert1, quantile(navi$wert1), include.lowest = TRUE)
[1] [1,2] [1,2] (2,3] (3,4] (4,5]
Levels: [1,2] (2,3] (3,4] (4,5]
```

Den Spezialfall einer Dichotomisierung erreicht man auch sehr einfach mit einer Bedingung.

Dichotomisieren mit einer Bedingung

```
> navi$wert1 > 4
[1] FALSE FALSE FALSE FALSE TRUE
```

Zeilen hinzufügen

Wenn Sie Fälle aus verschiedenen Datenmasken (die allerdings unbedingt die gleiche Struktur haben müssen) zusammenführen wollen, können Sie `rbind()` *row bind* verwenden

`rbind()`

```
> navi2 <- data.frame(zahlen = "sechs", buchst = "a",
                     wert1 = 6, wert2 = 100000,
                     wert1mal2 = 600000,
                     wertGr = ">= 4")
> rbind(navi, navi2)

  zahlen buchst wert1 wert2 wert1mal2 wertGr
1  eins      a     1 1e+00     1e+00    < 4
2  zwei      b     2 1e+01     2e+01    < 4
3  drei      c     3 1e+02     3e+02    < 4
```

4	vier	d	4 1e+03	4e+03	>= 4
5	fünf	e	5 1e+04	5e+04	>= 4
6	sechs	a	6 1e+05	6e+05	>= 4

Beachten Sie, dass Die Variablen *wert2* und *wert1mal2* jetzt in wissenschaftlicher Notation dargestellt werden. Das hat nichts mit der `rbind()`-Funktion zu tun, sondern damit, dass R bei Zahlen einer gewissen Größe automatisch auf diese Notation umschwenkt.

Komplexere Zusammenführungen von Datensätzen, die eben nicht ganz identische Struktur haben kann man mit `merge()` bewerkstelligen. In diesem Beispiel ist das Ergebnis identisch mit `rbind()`.

`merge()`

```
> merge(navi, navi2, all = TRUE, sort = FALSE)
  zahlen buchst wert1 wert2 wert1mal2 wertGr
1   eins      a     1 1e+00     1e+00    < 4
2   zwei      b     2 1e+01     2e+01    < 4
3   drei      c     3 1e+02     3e+02    < 4
4   vier      d     4 1e+03     4e+03   >= 4
5   fünf      e     5 1e+04     5e+04   >= 4
6  sechs      a     6 1e+05     6e+05   >= 4
```

Andere Datensatz-Funktionen

Zeilen und Spalten eines Datensatzes vertauscht man mit `t()`.

`t()`

```
> t(navi)
      [,1] [,2] [,3] [,4] [,5]
zahlen "eins" "zwei" "drei" "vier" "fünf"
buchst  "a"   "b"   "c"   "d"   "e"
wert1   "1"   "2"   "3"   "4"   "5"
wert2   "    " " 1"  " 10" " 100" "1000" "10000"
wert1mal2 "    " " 1"  " 20" " 300" "4000" "50000"
wertGr   "< 4" "< 4" "< 4" ">= 4" ">= 4"
```

MANCHMAL IST ES SINNVOLL, EINE FUNKTION nicht nur auf eine einzelne Variable, sondern auf alle Spalten oder auf alle Zeilen eines Datensatzes anzuwenden. Das ist mit `apply()`. Sie können sich z.B. von allen Variablen im Datensatz *navi* die Standardabweichung ausgeben lassen.

```
> apply(navi, 2, sd)
      zahlen      buchst      wert1      wert2      apply()
wert1mal2      NA      NA      1.581139  4368.044093
21942.744774      NA
```

Als erstes Argument von `apply()` geben Sie den Datensatz an. Dann bestimmen Sie, ob die Berechnung zeilenweise (1) oder spaltenweise (2) erfolgen soll. Das dritte Argument gibt die auszuführende Funktion an (hier `sd`), und zwar ohne Klammern.

Das funktioniert natürlich nur für die numerischen Variablen *wert1*,

wert2 und *wert1mal2*. Die Variablen *zahlen*, *buchst* und *wertGr* haben die Klasse *character* bzw. *factor* und ergeben deshalb NA. Man kann sie hier getrost weg lassen.

```
> apply(navi[, 3:5], 2, sd)
      wert1      wert2      wert1mal2
1.581139  4368.044093 21942.744774
```

Wenn Sie die auszuführende Funktion detaillierter steuern wollen, oder mehrere Funktionen kombinieren wollen, müssen Sie die Anweisung wiederum als Funktion schreiben, z.B. den Variationskoeffizienten $VK = sd_x / \bar{x}$.

Funktionen innerhalb von `apply()`

```
> VK <- function(x) {
  sd(x) / mean(x)
}

> apply(navi[3:5], 2, VK)
      wert1      wert2      wert1mal2
0.5270463  1.9656395  2.0197295
```

In dieser Variante haben Sie mit `function(x)` erst eine Funktion *VK* programmiert, die den Variationskoeffizienten von *x* berechnet und sie dann mit `apply()` auf die Variablen *wert1*, *wert2* und *wert1mal2* angewendet. Sie können die Funktion aber auch innerhalb von `apply()` formulieren.

```
> apply(navi[3:5], 2, function(x) {sd(x) / mean(x)})
      wert1      wert2      wert1mal2
0.5270463  1.9656395  2.0197295
```

Eine andere Variante von `apply()` haben Sie bereits in Bezug auf fehlende Werte kennen gelernt.

```
apply(frami, 1, function(x) sum(is.na(x)))
```

Hier wird beim Datensatz *frami* für jede Zeile (1) die Funktion `sum(is.na(x))` berechnet, also die Summe der Fehlenden pro statistischer Einheit. Das gibt bei 11627 Zeilen einen recht unübersichtliches Ergebnis. Um sich einen Überblick zu verschaffen, können Sie sich das Ergebnis besser als Häufigkeitstabelle ...

```
> table(apply(frami, 1, function(x) sum(is.na(x))))
 0    1    2    3    4    5    6
2306  549 7415 1046  293  17    1
```

Anzeige als
Häufigkeitstabelle

... oder als Grafik anzeigen lassen.

```
plot(apply(frami, 1, function(x){sum(is.na(x))}))
```

Anzeige als Grafik

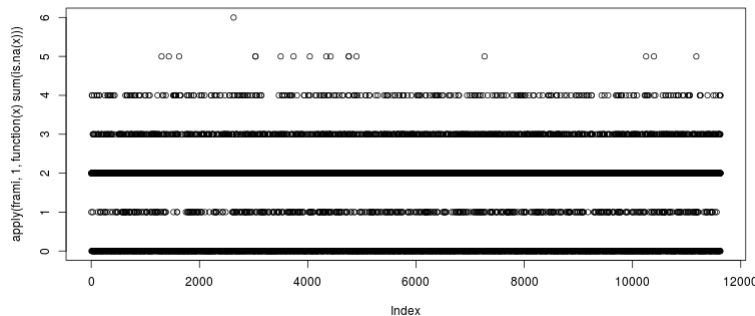


Abbildung 11 : Anzeige als Grafik: Anteil Missings pro statistische Einheit, Framingham-Datensatz

BEI DIESER SCHREIBWEISE WURDEN JETZT schon einige Funktionen in einander verschachtelt. So ein Code ist zwar relativ kurz, kann aber schwer zu verstehen sein. Sie müssen ihn bezüglich der Klammern von Innen nach außen lesen. In diesem Fall beginnen Sie mit `x`. Mit `is.na(x)` prüfen Sie, ob er Inhalt des Objektes `x` ein fehlender Wert ist. Auf der nächsten Stufe summieren die die fehlenden Werte mit `sum(...)` auf. Das Ganze definieren Sie mit `function(x){...}` als Funktion, die sie mit `apply(frami, 1, ...)` auf die Zeilen des Datensatzes *frami* anwenden. Schließlich stellen Sie das Ergebnis dieser Berechnung mit `plot(...)` grafisch dar.

Verschachtelter Code

Gerade für Anfänger ist es oft übersichtlicher, die Funktionen nicht zu verschachteln, sondern hinter einander abzuarbeiten, indem die Zwischenergebnisse als Objekte gespeichert werden. Am Ende kommt das gleiche heraus:

Sequentieller Code

```
> summeDerMissings <- function(x){
  fehlend <- is.na(x)
  sum(fehlend)
}
> fehlendProZeile <- apply(frami, 1, summeDerMissings)
> plot(fehlendProZeile)
```

Der Nachteil der sequentiellen Schreibweise ist, dass sie dabei viele Objekte erstellen, die in ihrer Fülle auch wieder verwirrend sein können. Deshalb ist es sinnvoll Objekte, die Sie nicht mehr benötigen mit `rm()` aus dem Workspace zu entfernen.

Objekte entfernen mit `rm()`

```
rm(fehlendProZeile, summeDerMissings)
```

Wenn Sie einmal den Workspace komplett von Objekten befreien

wollen, geht das mit

```
rm(list = ls()).
```

WENN DIE GLEICHE BEOBACHTUNG für eine statistische Einheit mehrmals ermittelt wird, kann der Datensatz in einer breiten oder einer langen Form strukturiert werden. In der Breiten Form wird jeder der Beobachtungen in einer eigenen Variable gespeichert. Jede statistische Einheit beansprucht eine Datenzeile. In der langen Struktur werden alle Beobachtungen in der gleichen Variable gespeichert und es gibt eine zweite Variable, die den Zeitpunkt der Beobachtung angibt. Jede statistische Einheit hat hier so viele Zeilen wie Beobachtungen. Ein Beispiel für die breite Struktur ist der Beispieldatensatz für Angst vor Spinnen, den Sie Bereits aus der ersten Übung kennen:

Breite und lange Struktur bei Mehrfachbeobachtungen

```
> spider
  PICTURE REAL
1       30   40
2       35   35
3       45   50
4       40   55
5       50   65
6       35   55
7       55   50
8       25   35
9       30   30
10      45   50
11      40   60
12      50   39
```

Mit `reshape()` können Sie zwischen breiter und langer Datensatzstruktur wechseln:

`reshape()`
von breit auf lang

```
> reshape(spider, varying = list(1:2), direction =
"long", v.names = "VAS", timevar = "Type")
  Type VAS id
1.1   1  30  1
2.1   1  35  2
3.1   1  45  3
4.1   1  40  4
5.1   1  50  5
6.1   1  35  6
7.1   1  55  7
8.1   1  25  8
9.1   1  30  9
10.1  1  45 10
11.1  1  40 11
12.1  1  50 12
1.2   2  40  1
2.2   2  35  2
3.2   2  50  3
4.2   2  55  4
5.2   2  65  5
6.2   2  55  6
7.2   2  50  7
8.2   2  35  8
```

```
9.2      2   30   9
10.2     2   50  10
11.2     2   60  11
12.2     2   39  12
```

Als erstes Argument geben Sie dabei den Datensatz an. Mit `varying = list(1:2)` bestimmen Sie, dass es die Spalten 1 bis 2 sind, die die wiederholten Messungen enthalten. Mit `direction = "long"` legen Sie fest, dass in ein langes Format umgewandelt werden soll. Schließlich bestimmen `v.names` und `timevar` die Variablennamen.

Als Ergebnis bekommen Sie einen *langen* Datensatz, der jetzt alle Angstmessungen in der Variable `VAS` zusammenfasst und 24 statt 12 Zeilen hat. Die Variable `Type` enthält die Informationen dazu, um welche Messung es sich handelt und zusätzlich wurde eine Variable `id` erstellt, die die ID-Nummer (hier die Zeilennummer) der statistischen Einheit des Ursprungsdatensatzes enthält.

Im Gegensatz weist der Framingham-Datensatz eine lange Struktur auf. Das bedeutet, dass die Werte von drei Beobachtungszeitpunkten jeweils in den gleichen Variablen gespeichert sind. Die Variable `PERIOD` gibt an, um welchen dieser Zeitpunkte es sich handelt. Die Variable `RANDID` ist die ID-Nummer jeweils einer Person.

`reshape()`
von lang auf breit

Sie können diesen Datensatz mit `reshape()` in ein breites Format überführen:

```
> framiWide <- reshape(frami, idvar = "RANDID",
                        timevar = "PERIOD",
                        direction = "wide")
```

Übung:

Wandeln Sie den Framingham-Datensatz in das breite Format um.

- Wie verändern sich Länge und Inhalt der Zeilen und Spalten?
- Was fällt Ihnen in Bezug auf die Verteilung von Missings auf? (Nutzen Sie die Beispiele für `apply()` und die selbst erstellte Funktion `naPlot()`.)

Datumsfunktionen

R verarbeitet Datums- und Zeitangaben nach dem POSIX-Standard. Gespeichert werden sie entweder als Objekt der Klasse `POSIXct` (Sekunden seit dem 1.1.1970) oder der Klasse `POSIXlt` (Liste mit Jahr, Monat, Tag, Stunde, Minute, Sekunde, Zeitzone ...). Gleich

POSIX-Standard

welches der beiden Formate Sie nutzen, die Anzeige ist immer als Datum, Zeit und Zeitzone.

Lassen Sie sich zur Illustration einmal die Systemzeit Ihres Computers anzeigen.

`Sys.time()`

```
> Systemzeit <- Sys.time()
> Systemzeit
[1] "2016-02-05 18:24:31 CET"
```

UM EINZELNE DATUMS- ODER ZEITANGABEN zu extrahieren, muss der Eintrag allerdings im POSIXlt-Format vorliegen.

Einzelne Datums- oder Zeitangaben extrahieren

```
> Systemzeit <- as.POSIXlt(Systemzeit)
```

Jetzt können Sie sich z.B. die Stunde oder den Tag anzeigen lassen.

```
> Systemzeit$hour
[1] 18
> Systemzeit$mday
[1] 5
```

Extraktion von einzelnen Zeit- Datumsdetails aus Objekten der Klasse POSIXlt

Tabelle 3: Extraktion von Zeit- und Datumsdetails

<code>\$sec</code>	Sekunden
<code>\$min</code>	Minuten
<code>\$hour</code>	Stunden
<code>\$mday</code>	Tag des Monats
<code>\$mon</code>	Monat (0-11 nach dem 1. Monat)
<code>\$year</code>	Jahre seit 1900
<code>\$wday</code>	Wochentag (0-6 nach Sonntag)
<code>\$yday</code>	Tag des Jahres (0-365)
<code>\$zone</code>	Zeitzone

Einige Details können auch direkt mit den Funktionen `weekdays()`, `month()` und `quarters()` extrahiert werden.

`weekdays()`

`months()`

`quarters()`

```
> weekdays(Systemzeit)
[1] "Freitag"
> months(Systemzeit)
[1] "Februar"
> quarters(Systemzeit)
[1] "Q1"
```

DIFFERENZEN ZWISCHEN ZWEI DATUMS- ZEITANGABEN können mit `difftime()` berechnet werden, z.B. die Zeit, seit Sie zuerst die Systemzeit gespeichert haben.

`difftime()`

```
> difftime(Sys.time(), Systemzeit)
Time difference of 0.7158325 secs
```

Die Zeiteinheiten, in denen die Differenz berechnet werden soll, können sie mit dem Argument `units =` bestimmen.

```
> difftime(Sys.time(), Systemzeit, units = "days")
Time difference of 8.346549e-06 days
```

ALPHANUMERISCHE DATEN JEDER ART kann man mit `strptime()` in das POSIX-Format überführen.

`strptime()`

```
Zeit06>>=
datum <- c("23.3.2004", "2.10.2015")
strptime(datum, format = "%d.%m.%Y")
@
```

Mit dem Argument `format =` können Sie beliebige Muster bestimmen, in denen die Datums-Zeit-Angaben notiert sind. Mit dem Vorzeichen `%` wird dabei jeweils angegeben, was die Einträge bedeuten.

%S	Sekunde (0-59)
%M	Minute (0-59)
%H	Stunde (0-23)
%I	Stunde (01-12)
%p	AM/PM
%Z	Zeitzone
%T	Zeit im Format %H:%M:%S
%d	Tag des Monats (1-31)
%m	Monat (1-12)
%y	Jahr (0-99) ohne Jahrhundert
%Y	Jahr, vierstellig
%C	Jahrhundert (Jahr geteilt durch 100)
%F	Datum im ISO 8601 Format %Y-%m-%d

Tabelle 4: Die wichtigsten Kodierungen, um Zeit- und Datumsangaben in alphanumerischen Formaten zu Kennzeichnen

Text-Funktionen

Hier geht es um Funktionen, mit denen man Objekte der Klasse `character` bearbeitet. Um bestimmte Textmuster zu finden, verwendet man `grep()`. Sie können z.B. suchen, welche Einträge der Variable `zahlen` im Datensatz `navi` ein „e“ enthält.

Textmuster suchen


```
> grep("e", navi$zahlen)
[1] 1 2 3 4
```

grep()

Es wird hier zunächst das zu suchende Muster (in Anführungszeichen) angegeben, dann das Objekt, in dem gesucht werden soll.

Als Ergebnis erhalten Sie die Indexnummern der Einträge, die das Muster enthalten. Um die entsprechenden Einträge selbst anzuzeigen, ergänzen Sie `value = TRUE`.

```
> grep("e", navi$zahlen, value = TRUE)
[1] "eins" "zwei" "drei" "vier"
```

Mit `grepl()` können Sie sich die Bedingung, ob das Objekt das Textmuster enthält als logischen Wert anzeigen lassen.

grepl()

```
> grepl("e", navi$zahlen)
[1] TRUE TRUE TRUE TRUE FALSE
```

Mit dem System der Regular Expressions können Sie bis ins Detail bestimmen, welche Muster an welcher Stelle gesucht werden sollen. Sie können z.B. festlegen, dass der Buchstabe „e“ nur am Beginn der Zeichenkette gesucht werden soll.

Regular Expressions

```
> grep("^e", navi$zahlen, value = TRUE)
[1] "eins"
```

Für detaillierte Informationen zur Verwendung von Regular Expressions, schauen Sie sich die R-Hilfe unter

`?regex`

an.

Mit Hilfe von `agrep()` können Sie auch eine unscharfe Suche realisieren.

Unschärfe Suche mit
agrep()

```
> agrep("zwo", navi$zahlen, value = TRUE)
[1] "zwei"
```

UM TEXTMUSTER ZU ERSETZEN, nutzen Sie `sub()`.

Textmuster ersetzen mit
sub()

```
> sub("f", "-GNRZ-", navi$zahlen)
[1] "eins"      "zwei"      "drei"      "vier"      "-GNRZ-üf"
```

Hier wird zuerst das zu suchende Textmuster angegeben, dann die Ersetzung und dann das Datenobjekt. `sub()` ersetzt das Textmuster nur beim ersten Auftreten. Um es bei jedem Auftreten zu ersetzen, nutzen Sie `gsub()`.

```
> gsub("f", "-GNRZ-", navi$zahlen)
[1] "eins"      "zwei"      "drei"
```

gsub()

```
"vier"           "-GNRZ-ün-GNRZ-"
```

UM TEXTTEILE ZUSAMMENZUFÜGEN, nutzen Sie `paste()`.

Textelemente
zusammenführen mit
`paste()`

```
> paste("Die Zahl", navi$wert1, "ist", navi$wertGr)
[1] "Die Zahl 1 ist < 4" "Die Zahl 2 ist < 4" "Die
Zahl 3 ist < 4" "Die Zahl 4 ist >= 4" "Die Zahl 5 ist
>= 4"
```

Das ist besonders hilfreich, wenn Sie später Texte in eine Grafik einfügen. Die Zusammenführung wird dabei für jeden Eintrag der Datenobjekte vorgenommen.

Wie die verschiedenen Textelemente getrennt werden, können Sie mit `sep =` festlegen. (Die Vorgabe ist ein Leerzeichen " ".)

```
> paste("Die Zahl", navi$wert1, "ist", navi$wertGr,
sep = "-")
[1] "Die Zahl-1-ist-< 4" "Die Zahl-2-ist-< 4" "Die
Zahl-3-ist-< 4" "Die Zahl-4-ist->= 4" "Die Zahl-5-
ist->= 4"
```

Mit `collapse =` bestimmen Sie wie die verschiedenen Einträge der Datenobjekte getrennt werden. (Die Vorgabe ist `NULL`, jedes als einzelner Texteintrag).

```
> paste("Die Zahl", navi$wert1, "ist", navi$wertGr,
collapse = " & ")
[1] "Die Zahl 1 ist < 4 & Die Zahl 2 ist < 4 & Die
Zahl 3 ist < 4 & Die Zahl 4 ist >= 4 & Die Zahl 5 ist
>= 4"
```

Weitere Textfunktionen können Sie unter anderem mit dem Paket *stringr*⁵ aufrufen.

Weiterführende Ressourcen zur Datenaufbereitung

- Kommentierte R-Syntax zu Schenderas „Datenqualität mit SPSS“⁶ finden Sie auf <https://github.com/JohannPopp/Datenaufbereitung>.
- Ein Online-Kurs „Getting and Cleaning Data“ der Johns Hopkins University steht unter <https://www.coursera.org/learn/data-cleaning> zur Verfügung.
- Handling and processing strings in R⁷ gibt einen umfassenden Überblick über Textfunktionen in R (<http://lib.psylab.info/files/Sanchez2013.pdf>).

5 Hadley Wickham, *stringr: Simple, Consistent Wrappers for Common String Operations*, 2015, <https://CRAN.R-project.org/package=stringr>.

6 Christian F. G. Schendera, *Datenqualität mit SPSS* (München u.a.: Oldenbourg, 2007).

7 Gaston Sanchez, *Handling and processing strings in R* (Berkley: Trowchez Editions, 2013), <http://lib.psylab.info/files/Sanchez2013.pdf>.

4 Explorative Datenanalyse

Deskriptive Statistiken für einzelne Variablen

Viele deskriptive Statistiken sind in den vorherigen Kapiteln schon angerissen worden. Hier sollen sie noch einmal systematisch beschrieben werden.

Deskriptive Statistiken für den schnellen Überblick bietet die Funktion `summary()`. Sie ist ein typisches Beispiel für objektorientierte Programmierung, weil sie je nach Objektklasse, auf die sie angewendet wird, unterschiedliche Ergebnisse liefert.

`summary()`

Auf einzelne Variablen angewendet, werden bei der Klasse `numeric` Quartilen, der Mittelwert und die Anzahl der Missings ausgegeben.

```
> summary(frami$TOTCHOL)
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
NA's
 107.0   210.0   238.0   241.2   268.0   696.0
409
```

Bei Variablen der Klasse `logical` und `factor` werden die Häufigkeiten und die Anzahl der Missings angegeben.

```
> summary(frami$SEX)
  f      m
6605 5022
> summary(frami$CURSMOKE)
  Mode  FALSE    TRUE  NA's
logical  6598    5029     0
```

Auf ein Objekt der Klasse `data.frame` angewendet, werden für jede der enthaltenen Variablen die entsprechenden Statistiken ausgegeben.

```
> summary(frami)
```

Häufigkeiten

Häufigkeiten können Sie mit der Funktion `table()` aufrufen.

`table()`

```
> table(frami$BPMEDS)

FALSE  TRUE
10090   944
```

In der Standardeinstellung werden Missings von dieser Funktion nicht angezeigt. Um das zu ändern ergänzen Sie das Argument `useNA = "always"` oder `useNA = "ifany"`.

```
> table(frami$BPMEDS, useNA = "always")
```

```
FALSE TRUE <NA>
10090  944  593
```

Sie können `table()` auch für Kreuztabellen verwenden, indem Sie eine zweite Variable hinzufügen.

Kreuztabellen mit
`table()`

```
> table(frami$SEX, frami$BPMEDS)
```

```
      FALSE TRUE
f    5597  663
m    4493  281
```

Um den Überblick zu behalten, welche Variable in den Zeilen und welche in den Spalten steht, können Sie sie innerhalb der Funktion benennen

```
< table(Geschlecht = frami$SEX,
        Blutdruckmedikamente = frami$BPMEDS)
```

```
      Blutdruckmedikamente
Geschlecht FALSE TRUE
f    5597  663
m    4493  281
```

Wenn Sie eine dritte (vierte ...) Variable hinzufügen, wird die Kreuztabelle entsprechend stratifiziert.

```
< table(Geschlecht = frami$SEX,
        Blutdruckmedikamente = frami$BPMEDS,
        Raucher = frami$CURSMOKE)
```

```
, , Raucher = FALSE
```

```
      Blutdruckmedikamente
Geschlecht FALSE TRUE
f    3456  486
m    2100  178
```

```
, , Raucher = TRUE
```

```
      Blutdruckmedikamente
Geschlecht FALSE TRUE
f    2141  177
m    2393  103
```

RELATIVE HÄUFIGKEITEN KÖNNEN SIE BERECHNEN, indem Sie `prop.table()` hinzufügen.

Relative Häufigkeiten mit
`prop.table()`

```
> prop.table(table(frami$BPMEDS))
```

```
      FALSE      TRUE
0.91444626 0.08555374
```

Wenn die Tabelle mehrere Dimensionen hat, können Sie mit dem Argument `margin =` bestimmen, auf welche Dimensionen sich

die Anteile beziehen. Bei einer Kreuztabelle steht 1 für die Zeilen, 2 für die Spalten. Wenn kein Wert für `margin` angegeben wird, beziehen sich die Anteile auf die Gesamtsumme.

```
> prop.table(table(frami$SEX, frami$BPMEDS), margin = 1)
```

```
      FALSE      TRUE
f 0.89408946 0.10591054
m 0.94113951 0.05886049
```

Das Ganze wird noch deutlicher, wenn mit `addmargins()` die Summen der Tabelle ergänzt werden.

Summen hinzufügen mit `addmargins()`

```
> addmargins(prop.table(table(frami$SEX,
frami$BPMEDS), margin = 1))
```

```
      FALSE      TRUE      Sum
f 0.89408946 0.10591054 1.00000000
m 0.94113951 0.05886049 1.00000000
Sum 1.83522896 0.16477104 2.00000000
```

KUMULIERTE HÄUFIGKEITEN KÖNNEN SIE MIT `cumsum()` berechnen. Im folgenden Beispiel werden erst die absoluten und dann die relativen kumulativen Häufigkeiten der Angstmessungen aus dem Spinnen-Beispieldatensatz von Field⁸ berechnet.

Kumulierte Häufigkeiten mit `cumsum()`

```
> cumsum(table(spider$REAL))
30 35 39 40 50 55 60 65
1  3  4  5  8 10 11 12
> cumsum(prop.table(table(spider$REAL)))
      30      35      39      40      50      55
0.08333333 0.25000000 0.33333333 0.41666667 0.66666667
0.83333333 0.91666667 1.00000000
```

Lage- und Streuungsmaße

Die Funktion für den arithmetischen Mittelwert ist `mean()` und die Funktion für den Median ist `median()`. Beachten Sie dass beide Funktionen in der Standardeinstellung `NA` ergeben, wenn es fehlende Werte gibt. Um diese fehlenden Werte auszuschließen, ergänzen Sie das Argument `na.rm = TRUE`.

`mean()`, `median()`

```
> mean(frami$TOTCHOL)
[1] NA
> mean(frami$TOTCHOL, na.rm = TRUE)
[1] 241.1624
> median(frami$TOTCHOL, na.rm = TRUE)
[1] 238
```

8 Field, Miles, und Field, *Discovering Statistics Using R*.

ES GIBT KEINE EIGENE FUNKTION FÜR DEN MODALWERT. So können Sie eine erstellen:

Modalwert

```
> Modus <- function(x) {
  names(table(x))[table(x) == max(table(x))]
}

> Modus(frami$TOTCHOL)

[1] "240"
```

DIE WICHTIGSTEN STREUUNGSMASSE können Sie mit `min()`, `max()`, `var()`, `sd()` und `quantile()` aufrufen.

`min()`, `max()`,
`var()`, `sd()`,
`quantile()`

```
> min(frami$TOTCHOL, na.rm = TRUE)
[1] 107
> max(frami$TOTCHOL, na.rm = TRUE)
[1] 696
> var(frami$TOTCHOL, na.rm = TRUE)
[1] 2058.258
> sd(frami$TOTCHOL, na.rm = TRUE)
[1] 45.36803
> quantile(frami$TOTCHOL, na.rm = TRUE)
 0%  25%  50%  75% 100%
107  210  238  268  696
```

In der Standardeinstellung berechnet die Funktion `quantile()` Quartilen. Mit dem Argument `probs` = kann man aber auch Perzentilen jeder Art berechnen, z.B. 10%-Perzentilen.

```
> quantile(frami$TOTCHOL,
  probs = c(0, 0.1, 0.2, 0.3, 0.4, 0.5,
            0.6, 0.7, 0.8, 0.9, 1),
  na.rm = TRUE)
 0%  10%  20%  30%  40%  50%  60%  70%  80%  90% 100%
107  187  203  216  227  238  249  261  276  298  696
```

Das optionale Argument `type` = bestimmt, welcher von neun möglichen Algorithmen zur Berechnung verwendet wird.

ABSTECHER: ANSTATT DIE PERZENTILEN EINZELN anzugeben, kann man sie auch mit der Funktion `seq()` erstellen, z.B. so:

Sequenzen erzeugen mit
`seq()`

```
> seq(0, 1, 0.1)
[1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

Das erste Argument (`from` =) legt den Startpunkt der Sequenz fest, das zweite Argument (`to` =) den Endpunkt und das dritte Argument (`by` =) die Schrittweite, in der die Sequenz ansteigt. Alternativ zu `by` = kann man auch mit `length.out` = die Länge der Sequenz angeben.

```
> seq(0, 1, length.out = 11)
[1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

Übung:

- Berechnen Sie die 5%-Perzentilen der Cholesterinwerte aus dem Framingham-Datensatz.
- Die Funktion `var()` schätzt die Varianz der Grundgesamtheit, indem die quadrierten Abweichungen vom Mittelwert durch $n-1$ geteilt werden. Erstellen Sie eine Funktion für die empirische Varianz, bei der die quadrierten Abweichungen durch n geteilt werden.

$$Emp. Var. = \frac{\sum (x - \bar{x})^2}{n}$$

Maße für verschiedene Subgruppen berechnen:

Mit `tapply()` können gleichzeitig Maße für verschiedene Subgruppen berechnen, z.B. das mittlere Alter für Frauen und Männer.

`tapply()`

```
> tapply(frami$AGE, frami$SEX, mean)
      f      m 
55.02029 54.49363
```

Die Funktion `tapply()` arbeitet nach dem gleichen Prinzip wie `apply()`. Als erstes geben Sie hier die Variable an, für die Sie einen Wert berechnen wollen, als zweites die Variable, die die zu unterscheidenden Gruppen definiert und als drittes die Funktion, die Sie ausführen wollen.

Übung:

- Berechnen Sie die mittleren Cholesterinwerte des Framingham-Datensatzes getrennt nach Geschlecht.

R-Grafik

In R sind verschiedene Grafiksysteme implementiert. In der Grundausstattung stehen *base* und *grid* zur Verfügung. Das *base*-System ist das ursprüngliche Grafik-System von R. Später wurde *grid* entwickelt, um insbesondere die Steuerung von verschiedenen Einzelgrafiken innerhalb einer Abbildung zu verbessern.

Der Preis, den man für die erweiterten Funktionen von *grid* zahlt, ist ein aufwändigerer Code, bei dem die zusätzlichen Einzelheiten definiert werden müssen. Zusatzpakete wie *lattice* und *ggplot2* bauen auf *grid* auf, verfügen aber über sogenannte high-level-Funktionen, die viele der Details automatisch regeln.

Eine ganz neue Weiterentwicklung ist das Paket *ggvis*, das wunderbare Möglichkeiten für interaktive Grafiken zur Verfügung stellt.

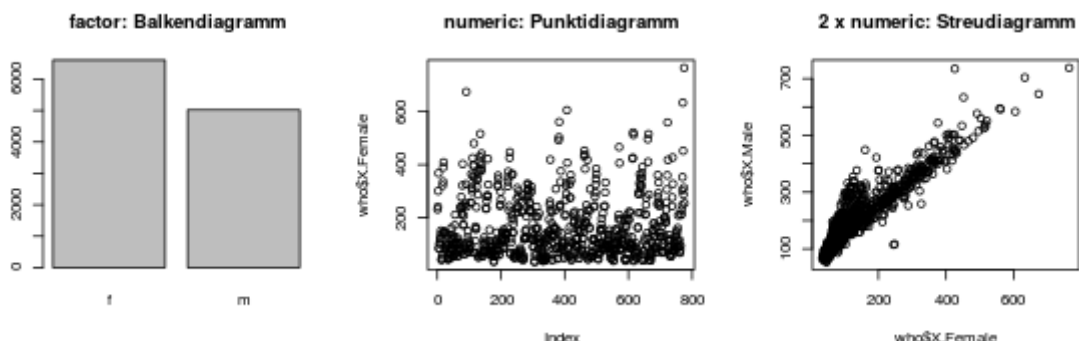
Unter <http://www.r-graph-gallery.com/> kann man sich eine Vielzahl von Beispielgrafiken und den dazu gehörigen Code ansehen. Hier werden *base* und *ggplot2* besprochen.

Basissystem

Analog zu `summary()` gibt es für Grafiken die Funktion `plot()`. Sie erstellt je nach Objektklasse, auf die sie angewendet wird unterschiedliche sinnvolle Grafiken.

`plot()`

```
> plot(frami$SEX)
> plot(who$X.Female)
> plot(who$X.Female, who$X.Male)
> plot(who)
```



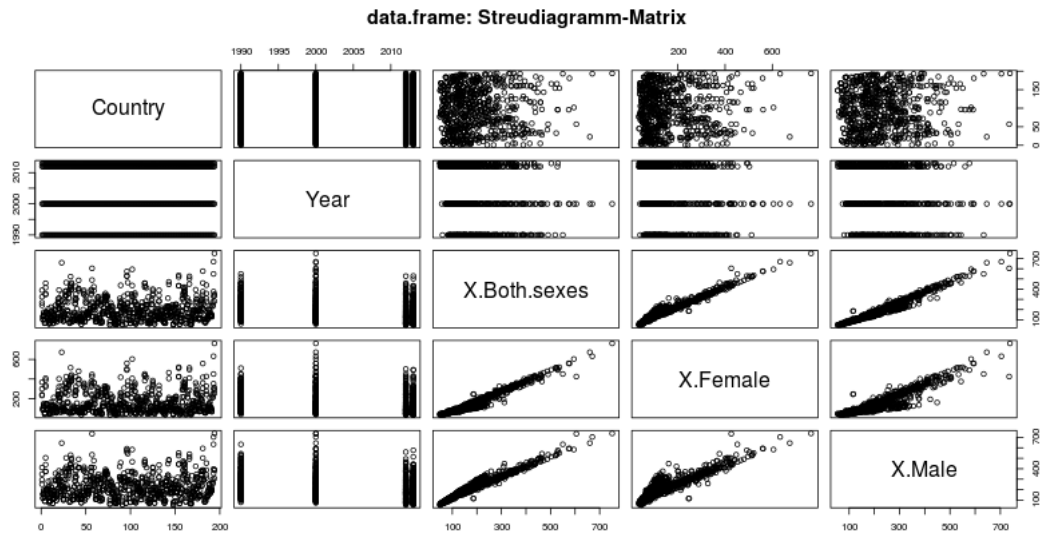


Abbildung 12 : Anwendung von `plot()` auf verschiedene Objektklassen

Das *base*-Grafiksystem arbeitet nach dem Prinzip von Papier und Bleistift. Eine erstellte Grafik können Sie nicht ändern, aber Sie können sie mit weiteren Elementen übermalen. Grundsätzlich kann man zwischen high-level-Funktionen unterscheiden, die eine ganze Grafik erstellen und low-level-Funktionen, mit denen man einzelne Elemente einer Grafik erstellt. Tabelle 5 zeigt wichtige high-level-Funktionen.

Funktion	Grafik
<code>barplot()</code>	Balkendiagramm
<code>boxplot()</code>	Boxplot
<code>hist()</code>	Histogramm
<code>pie()</code>	Tortendiagramm
<code>qqnorm()</code>	Q-Q-Normalverteilungsdiagramm
<code>scatter.smooth()</code>	Streudiagramm mit loess-Anpassungslinie
<code>image()</code>	Farbabbild einer Matrix
<code>stem()</code>	Stängel-Blatt-Diagramm

Tabelle 5: Wichtige high-level-Funktionen des base-Grafiksystems

DAS ERSCHEINUNGSBILD DIESER GRAFIKEN lässt sich mit einer Reihe von Argumenten steuern. Hier sind die wichtigsten:

Tabelle 6: Wichtige Steuerungsargumente für base-Grafiken

Argument	Wirkung
<code>main = ""</code>	Titel
<code>sub = ""</code>	Untertitel
<code>xlab = ""</code>	Beschriftung der x-Achse
<code>ylab = ""</code>	Beschriftung der y-Achse
<code>cex =,</code> <code>cex.main,</code> <code>cex.sub,</code> <code>cex.axis,</code> <code>cex.lab</code>	Schrift vergrößern (>1) oder verkleinern (<1)
<code>las =</code>	Ausrichtung der Achsenbeschriftung: 0 = parallel zur Achse 1 = horizontal 2 = rechtwinklig zur Achse 3 = vertikal
<code>srt =</code>	Schriftrotation
<code>pch =</code>	Darstellungssymbol von Punkten
<code>col =</code>	Füllfarbe
<code>bg =</code>	Hintergrundfarbe
<code>fg =</code>	Vordergrundfarbe
<code>lwd =</code>	Linienbreite
<code>lty =</code>	Linientyp
<code>lwd =</code>	Linienbreite
<code>xlim =,</code> <code>ylim</code>	Darstellungsbereich der Achse nach dem Muster <code>c(minimum, maximum)</code>
<code>xaxt =,</code> <code>yaxt</code>	Darstellungsart der Achse. "n" unterdrückt die Anzeige
<code>xpd =</code>	Begrenzung der dargestellten Daten auf die Grafik (TRUE), die Abbildung (FALSE) oder keine (NA)
<code>print =</code>	Soll die Grafik angezeigt werden? TRUE/FALSE

Eine vollständige Beschreibung der base-Grafikparameter lässt sich unter `?par` abrufen

`?par`

DAS DARSTELLUNGSSYMBOL VON PUNKTEN wird im Argument `pch =` mit Zahlen von 1-20 angegeben. Die Bedeutung können Sie der folgenden Abbildung entnehmen.

Steuerung der Punktsymbole mit `pch =`

```
> plot(1:20, rep(1, 20), pch = 1:20,
      ylim = c(0.9, 1.2), xaxt = "n", xlab = "",
      yaxt = "n", ylab = "", main = "pch =: Welche
```

```
Zahl erzeugt welches Symbol?)
> text(1:20, rep(1, 20) + 0.1, labels = 1:20,
      adj = c(0.5, 0), cex = 0.8)
```

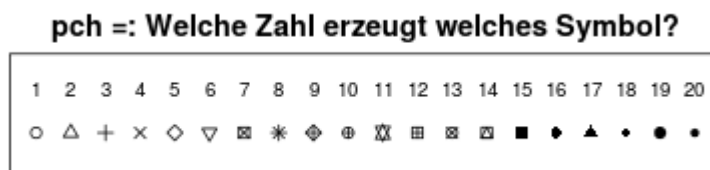


Abbildung 13 : Steuerung der Punktsymbole mit pch =

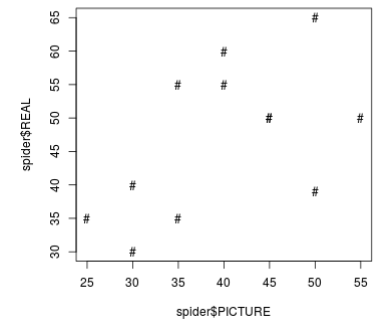


Abbildung 14 : Punktsymbol in Anführungszeichen

Sie können aber auch ein beliebiges Zeichen in Anführungsstrichen angeben.

```
> plot(spider$PICTURE, spider$REAL, pch = "#")
```

FARBEN KÖNNEN SIE AUF VERSCHIEDENE ART bestimmen, als Wörter ("red", "blue", "darkblue" etc.), Zahlen (1-8) oder im hexadezimalen RGB-Format.

```
> plot(1:8, rep(1, 8), col = 1:8,
      pch = c("1","2","3","4","5","6","7", "8"),
      xlab = "", xaxt = "n", ylab = "", yaxt = "n",
      xlim = c(-2,8.5), main = "Farbkodes als Wörter,
      Zahlen und im RGB-Format")

> text(1:8, rep(1.2, 8),
      col = c("black", "red", "green", "blue",
              "cyan", "magenta", "yellow", "grey"),
      labels = c("black", "red", "green", "blue",
                 "cyan", "magenta", "yellow", "grey"),
      srt = 70)

> text(1:8, rep(0.8, 8), col = c("#000000", "#FF0000",
                                  "#00FF00", "#0000FF", "#00FFFF", "#FF00FF",
                                  "#FFFF00", "#D0D0D0"),
      labels = c("#000000", "#FF0000", "#00FF00",
                 "#0000FF", "#00FFFF", "#FF00FF", "#FFFF00",
                 "#D0D0D0"), srt = 70)

> text(rep(-2,3), c(1.2, 1, 0.8),
      labels = c("Wort-Kode:", "Zahlenkode:",
                 "RGB-Kode:"), adj = c(0, 0.5))
```

Farbkodes als Wörter, Zahlen und im RGB-Format

Wort-Kode:	black	red	green	blue	cyan	magenta	yellow	grey
Zahlenkode:	1	2	3	4	5	6	7	8
RGB-Kode:	#000000	#FF0000	#00FF00	#0000FF	#00FFFF	#FF00FF	#FFFF00	#D0D0D0

Abbildung 15 : Steuerung von Farben im base-Grafiksystem

Es gibt insgesamt 657 Farben, die Sie durch Wörter auswählen können. Eine vollständige liste gibt

Farbauswahl mit Worten

```
> colours()
```

Die Auswahl durch Zahlen ist an verschiedene Farbpaletten gebunden. Besonders ausgewogene Farbpaletten sind im Paket *RcolorBrewer* enthalten.

Farbpaletten

```
> plot(1:8, rep(0.9, 8), col = rainbow(8), pch = 19,
      cex = 2, xlim = c(-3,8), ylim = c(-0.2, 1),
      xaxt = "n", xlab = "", yaxt = "n", ylab = "")
> points(1:8, rep(0.8, 8), col = terrain.colors(8),
      pch = 19, cex = 2)
> points(1:8, rep(0.7, 8), col = heat.colors(8), pch =
      19, cex = 2)
> points(1:8, rep(0.6, 8), col = topo.colors(8), pch =
      19, cex = 2)
> points(1:8, rep(0.5, 8), col = cm.colors(8), pch =
      19, cex = 2)
> points(1:8, rep(0.3, 8), col = grey(1:8 / 8), pch =
      19, cex = 2)
> library(RColorBrewer)
> points(1:8, rep(0.1, 8), col = brewer.pal(8,
      "Purples"), pch = 19, cex = 2)
> points(1:8, rep(0, 8), col = brewer.pal(8, "RdBu"),
      pch = 19, cex = 2)
> points(1:8, rep(-0.1, 8), col = brewer.pal(8,
      "Dark2"), pch = 19, cex = 2)
> text(-3, seq(0.9, -0.1, -0.1),
      labels = c("rainbow(8)", "terrain.colors(8)",
      "heat.colors(8)", "topo.colors(8)",
      "cm.colors(8)", "", "grey(1:8 / 8)", "",
      "RColorBrewer:", "brewer.pal(8, '...')", ""),
      adj = c(0, 0.5 ))
```

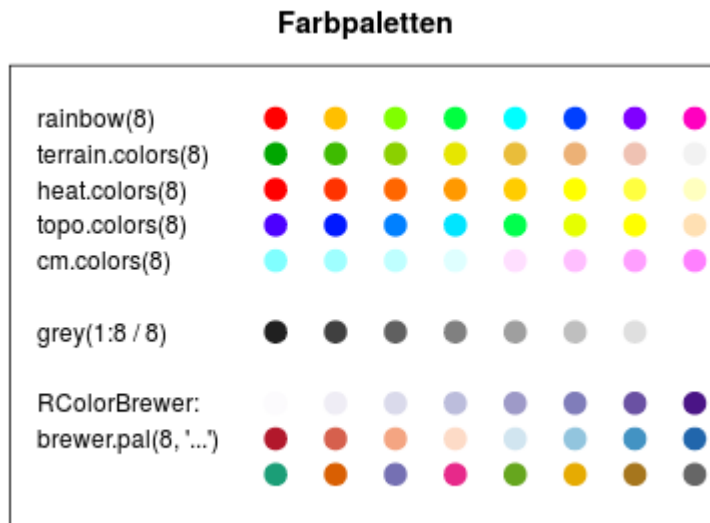


Abbildung 16 : Verschiedene Farbpaletten (hier mit jeweils 8 Ausprägungen)

Eine Farbangabe im RGB-Format beginnt mit einer Raute, gefolgt von einem zweistelligen hexadezimalen Wert (00 – FF) für die rote Farbsättigung. Dem folgen die jeweils zweistelligen Sättigungswerte für grün und blau. Schwarz ist "#000000", weiß ist "#FFFFFF".

RGB-Format

DIE TRANSPARENZ DER FARBEN können Sie in den Basis-Farbpaletten mit dem Argument `alpha` = bestimmen. Dabei steht 1 für keine Transparenz und 0 für vollständige Transparenz.

Transparenz

Im RGB-Format können Sie den sechs hexadezimalen Stellen für die Farbauswahl einfach zwei weitere für die Transparenz anhängen.

```
> plot.new()
> text(0.5, 0.5, "Transparenz", font = 2, cex = 5.5,
      col = "gray")
> text(seq(0.02, 0.98, length.out = 9), rep(0.5, 9),
      labels = paste(1:9 / 10),
      col = heat.colors(1, alpha = 1:9 / 10)),
      cex = 2, font = 2)
```

Transparenz

0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9

Abbildung 17 : Transparenz (alpha = 0.1 - 0.9)

LINIENTYPEN LASSEN SICH MIT `lty` = entsprechend der folgenden Abbildung definieren.

Linientyp bestimmen mit `lty` =

```
> plot(c(0.05, 1), c(1,1), xlim = c(0,1),
      ylim = c(-0.05,1.05), main = "Linientypen",
      type = "l", lwd = 3, xaxt = "n", xlab = "",
      yaxt = "n", ylab = "")
> segments(rep(0.05, 6), seq(1, 0, length.out = 6),
          rep(1, 6), lty = 1:6, cex = 3, lwd = 3)
> text(rep(0, 6), seq(1, 0, length.out = 6),
      labels = 1:6)
```

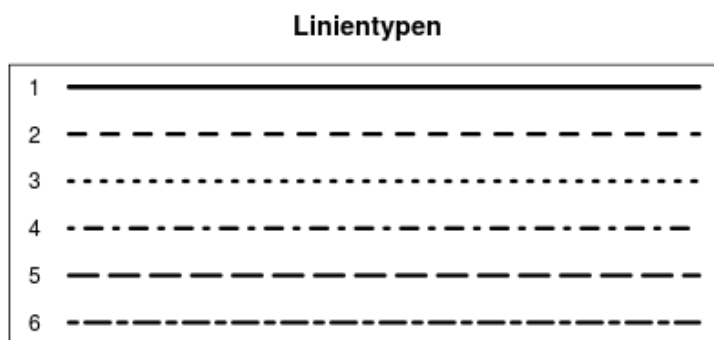


Abbildung 18 : Kodierung von Linientypen mit `lty` =

Weitere Linieneigenschaften können mit `lwd` = (Linienbreite), `lend` = (Linienende), `ljoin` = und `lmitr` = (Verbindung angrenzender Linien).

Weitere Linienparameter
`lwd`, `lend`, `ljoin`,
`lmitr`

EINIGE FEINEINSTELLUNGEN KÖNNEN SIE nur vornehmen, wenn sie der eigentlichen Grafik eine `par()`-Funktion voranstellen. Das gilt insbesondere für die folgenden Argumente.

Feineinstellung mit
`par()`

Den Rahmenbereich der Grafik, in dem Titel, und Achsenbeschriftungen dargestellt werden können Sie mit `mar` = oder `mai` = definieren. Bei `mar` = werden die Rahmenbreiten als Anzahl der Zeilen bestimmt, bei `mai` = als Inches. Die vier Breitenangaben werden nach dem Muster `c(unten, links, oben, rechts)` gemacht

`mar` =, `mai` =

Um mehrere Grafiken in einem Grafikfenster darzustellen, können Sie innerhalb von `par()` das Argument `mfrow = c(Zeilenzahl, Spaltenzahl)` verwenden.

`mfrow =`

KOMPLIZIERTERE KOMBINATIONEN VON MEHREREN GRAFIKEN können Sie mit dem Befehl `layout()` erzeugen.

`layout()`

AN DEN OBEN GEZEIGTEN BEISPIELGRAFIKEN ist erkennbar, dass sich nicht alle Elemente und Eigenschaften einer base-Grafik innerhalb der high-level-Funktion bestimmen lassen, sondern dass Einzelteile der Grafik mit low-level-Funktionen ergänzt werden. Hier ist ein Überblick über wichtige low-level-Funktionen.

Low-level-Funktionen

Funktion	Grafikelement
<code>plot.new()</code>	Erzeugt eine neue leere Grafik (Skalenbereich 0-1 auf X- und Y-Achse)
<code>axis(side = , ...)</code>	Achse, <code>side = 1</code> (unten), <code>2</code> (links), <code>3</code> (oben), <code>4</code> (rechts)
<code>text()</code>	Text
<code>mtext()</code>	Text im Rahmen der Grafik
<code>lines()</code>	Eine fortlaufende Linie
<code>segments()</code>	Ein oder mehr gerade Linien
<code>arrows()</code>	Pfeile
<code>curve()</code>	Geschwungene Linie
<code>rect()</code>	Rechteck(e)
<code>polygon()</code>	Polygon
<code>legend()</code>	Legende
<code>rug()</code>	Eindimensionales Verteilungsmuster am Rand der Grafik
<code>grid()</code>	Rasterlinien

Tabelle 7: Wichtige low-level-Funktionen des base-Grafiksystems

Innerhalb des base-Grafiksystems gibt es auch eine interaktive Funktion. `locator()` gibt die x- und y-Koordinaten an, die Sie in einer Grafik anklicken.

`locator()`

High-level-Grafikfunktionen berechnen automatisch Statistiken (ein Boxplot z.B. Quartilen, ein Balkendiagramm Häufigkeiten etc.). Auf diese Statistiken können Sie zugreifen, indem Sie den Plot einem Objekt zuweisen.

```
> agePlot <- boxplot(frami$AGE)
> agePlot
> text(rep(1.22, 5), agePlot$stats,
      labels = agePlot$stats, adj = c(0, 0.5))
```

```

$stats
      [,1]
[1,]    32
[2,]    48
[3,]    54
[4,]    62
[5,]    81
attr(,"class")
      1
"integer"

$n
[1] 11627

$conf
      [,1]
[1,] 53.79486
[2,] 54.20514

$out
numeric(0)

$group
numeric(0)

$names
[1] "1"

```

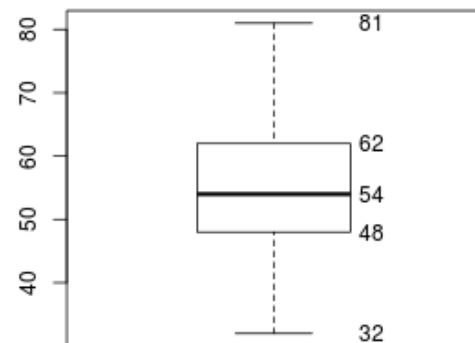


Abbildung 19 : Die in high-level-Funktionen enthaltenen Statistiken weiter verwenden

ggplot2

Das Grafikpaket *ggplot2* implementiert mit der *Grammar of Graphics*⁹ eine strukturierte Beschreibung statistischer Grafiken und ermöglicht es mit wenigen Befehlen sehr elegante Grafiken zu erstellen.

Die objektorientierte Schnellfunktion (entsprechend `summary()` und `plot()`) von *ggplot2* ist `qplot()`. Um die Struktur von *ggplot2*-Grafiken zu verstehen, ist es aber besser, einen etwas längeren, aber eben auch strukturierten Weg zu gehen.

Im ersten Schritt definieren Sie mit `ggplot()` die darzustellenden Daten. Dabei geben Sie zuerst den Datensatz und dann mit `aes()` die darzustellenden Variablen an und speichern das ganze als ein neues Objekt.

```

> library(ggplot2)
> agePlot <- ggplot(frami, aes(AGE))

```

Zunächst passiert erst einmal überhaupt nichts. Um eine Grafik zu `geom_...()`

9 Leland Wilkinson und Graham Wills, *The Grammar of Graphics* (New York: Springer, 2005), <http://public.eblib.com/choice/publicfullrecord.aspx?p=302755>.

erstellen, müssen Sie angeben, auf welche Weise die Daten dargestellt werden sollen. Das machen sie mit `geom_...()`. In diesem Beispiel `geom_histogram`.

```
> agePlot + geom_histogram()
```

`geom_histogram()`

Geoms in *ggplot2* entsprechen den high-level-Funktionen im base-Grafiksystem.

Beachten Sie, dass zuerst das vorher mit `ggplot()` erstellte Basisobjekt angegeben wird und weitere Angaben mit `+` hinzugefügt werden.

Wenn sie die Eigenschaften des Histogramms beeinflussen möchten, können Sie das innerhalb der Klammern tun. Sie können z.B. eine farbliche Unterscheidung zwischen Personen mit und ohne Angina Pectoris machen.

Zusätzliche
Datenrepräsentationen
mit `aes()` ergänzen

```
> agePlot + geom_histogram(aes(fill = ANGINA))
```

Jetzt haben Sie mit `aes()` die Art der Darstellung durch zusätzliche Daten beeinflusst. Sie können aber auch Werte festlegen, z.B. ob die Farben übereinander oder nebeneinander angezeigt werden sollen. Solche Angaben machen Sie außerhalb von `aes()`.

Eigenschaften außerhalb
von `aes()` festlegen

```
> agePlot + geom_histogram(aes(fill = ANGINA),  
                           position = "dodge")
```

Für dieses Beispiel ist

```
> agePlot + geom_histogram(aes(fill = ANGINA),  
                           position = "fill")
```

noch geeigneter

Mit `+` können Sie ganz einfach weitere Elemente hinzufügen, z.B. eine Facettierung nach Geschlecht.

Facetten hinzufügen

```
> agePlot +  
  geom_histogram(aes(fill = ANGINA),  
                position = "fill") +  
  facet_wrap(~ SEX)
```

Schließlich können Sie auch noch Beschriftungen festlegen.

Beschriftungen
hinzufügen

```
> agePlot +  
  geom_histogram(aes(fill = ANGINA),  
                position = "fill") +  
  facet_wrap(~ SEX) +  
  labs(title = "Altersabhängige Häufigkeit von Angina  
          Pectoris getrennt nach Geschlecht",  
        y = "Anteil")
```

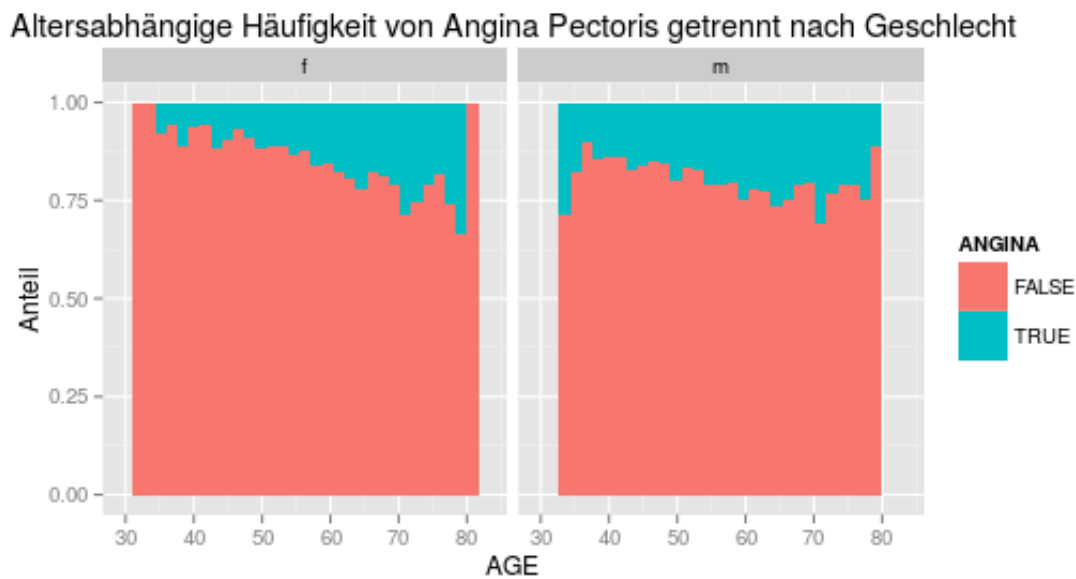


Abbildung 20: Beispielgrafik mit ggplot2

Hilfen zum Umgang mit ggplot2:

- ggplot2-Cheatsheet
<https://www.rstudio.com/resources/cheatsheets/>
- Chang, Winston. *R Graphics Cookbook*. Beijing; Sebastopol, CA: O'Reilly, 2013.

Übung:

- Laden Sie sich von <http://www.umass.edu/statdata/statdata/stat-logistic.html> den Datensatz "lowbwt" herunter, wandeln Sie nominale Daten in ein entsprechendes Format um und beschreiben Sie die Daten mit Zahlen und Grafiken.

Hypothesentests und Zusammenhangsmaße

R verfügt über zahlreiche Wahrscheinlichkeitsfunktionen, die in Hypothesentests verwendet werden können. Man kann aber auch direkt auf diese Verteilungsfunktionen zugreifen. Die folgende Tabelle zeigt die wichtigsten davon. Eine vollständige Liste finden Sie unter `?distributions`.

Funktion	Verteilung
<code>dnorm()</code>	Normalverteilung
<code>dt()</code>	T-Verteilung
<code>df()</code>	F-Verteilung
<code>dchisq()</code>	Chi ² -Verteilung
<code>dunif()</code>	Gleichverteilung
<code>dbinom()</code>	Binomialverteilung
<code>dpois()</code>	Poisson-Verteilung

Tabelle 8: Wichtige Verteilungsfunktionen in R

Das Vorzeichen `d` steht dabei jeweils für die Dichte der Funktion. Mit `q` können Sie analog die Quantilen der jeweiligen Funktion abrufen, mit `p` den p-Wert und mit `r` Zufallszahlen berechnen.

Die Dichtefunktion können Sie verwenden, um die Verteilungsfunktion über einen beliebigen Wertebereich darzustellen. Hier z.B. Die Chi²-Funktion mit einem Freiheitsgrad als Grafik.

Dichtefunktionen z.B.
`dchisq()`

```
> Bereich <- seq(0, 8, 0.01)
> plot(Bereich, dchisq(Bereich, df = 1), type = "l")
```

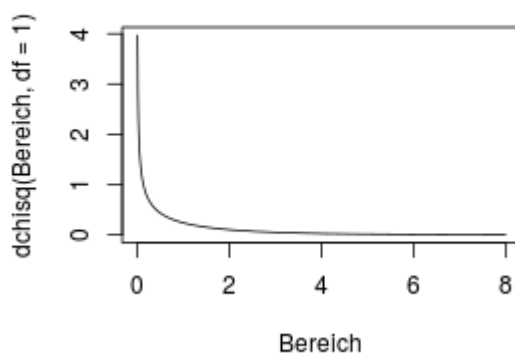


Abbildung 21 : Chi²-Dichtefunktion mit einem Freiheitsgrad

Mit der Quantilsfunktion können Sie kritische Werte für bestimmte Signifikanzniveaus berechnen. Hier z.B. den Kritischen Wert der Chi²-Funktion für $p = 0,05$ ($df = 1$).

Quantilsfunktion z.B.
`qchisq()`

```
> qchisq(0.05, df = 1, lower.tail = FALSE)
[1] 3.841459
```

Achten Sie darauf, dass hier `lower.tail = FALSE` angegeben werden muss, um den Kritischen Wert zu finden, bei dem eine 5%ige Wahrscheinlichkeit besteht, dass man ein größeres Ergebnis erhält.

Um p-Werte zu berechnen, verwenden Sie die Wahrscheinlichkeitsfunktion. Hier z.B. die Wahrscheinlichkeit bei einem Freiheitsgrad einen Chi²-Wert von 3,84 oder größer zu erhalten, wenn in Wirklichkeit die H_0 -Hypothese zutrifft.

Wahrscheinlichkeitsfunktion
z.B. `pchisq()`

```
> pchisq(3.84, df = 1, lower.tail = FALSE)
[1] 0.05004352
```

Auch hier müssen Sie wieder `lower.tail = FALSE` angeben.

Schließlich können Sie sich auf Basis der verschiedenen Verteilungsfunktionen Zufallszahlen berechnen lassen. Hier z.B. 100 Zufallszahlen der Chi²-Verteilung mit einem Freiheitsgrad.

Zufallszahlen z.B.
`rchisq()`

```
> ZufallChi <- rchisq(100, df = 1)
```

Lassen Sie sich die Verteilung dieser Zufallszahlen als Histogramm anzeigen ...

```
hist(ZufallChi, freq = FALSE)
```

... und vergleichen Sie es mit der theoretischen Dichtefunktion.

```
> lines(Bereich, dchisq(Bereich, df = 1))
```

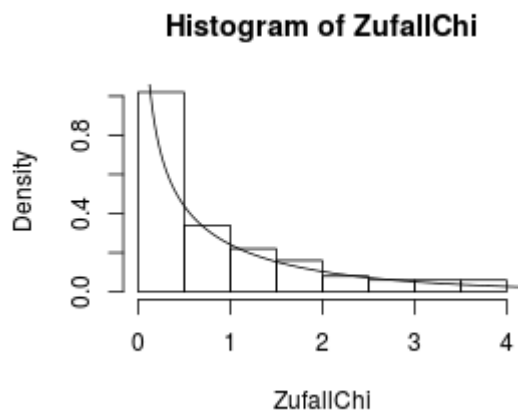


Abbildung 22 : Verteilung von 100 Zufallszahlen auf Basis der χ^2 -Verteilung und die theoretische Dichtefunktion

Zwei Besonderheiten fallen hier auf. Zum Einen wird die Funktion `hist()` mit dem Argument `freq = FALSE` ausgeführt, damit die Y-Achse die Dichte und nicht die absoluten Häufigkeiten abbildet. Das ist notwendig, damit später die theoretische Dichtefunktion im gleichen Skalenbereich dargestellt wird.

Zum Anderen wird Ihr Histogramm wahrscheinlich etwas anders aussehen, als das hier abgebildete. Das liegt daran, dass bei jedem Durchlauf etwas andere Zufallszahlen gezogen werden. Wenn Sie Zufallszahlen einmal exakt reproduzieren möchten, können Sie `set.seed()` verwenden. Mit

```
> set.seed(23)
> ZufallChi <- rchisq(100, 1)
```

sollte Ihr Histogramm meinem genau gleichen.

Zufallszahlen reproduzieren mit `set.seed()`

Annahmen prüfen

Die Annahme der Normalverteilung können Sie grafisch mit einem Histogramm mit überlagerter Normalverteilungskurve oder mit einem QQ-Plot überprüfen. Hier am Beispiel der Verteilung des Geburtsgewichtes in den Daten der Low-Birth-Weight-Studie.

```
> par(mfrow = c(1,2))
> hist(lowbwt$BWT, freq = FALSE)
> lines(500:5000,
        dnorm(500:5000, mean = mean(lowbwt$BWT),
              sd = sd(lowbwt$BWT)))
> qqnorm(lowbwt$BWT)
```

Normalverteilung grafisch prüfen mit

`hist()` und `qqnorm()`

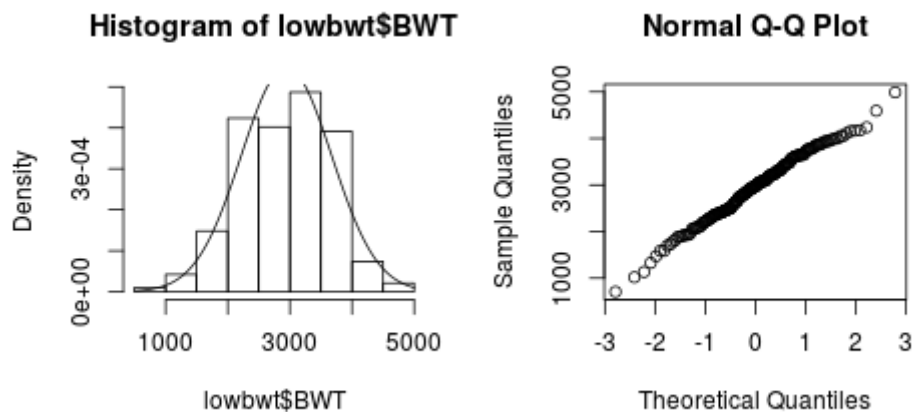


Abbildung 23 : Grafische Prüfung auf Normalverteilung

In R ist der Standard-Test zur Prüfung auf Normalverteilung der Shapiro-Wilk-Test.

Shapiro-Wilk-Test
`shapiro.test()`

```
> shapiro.test(lowbwt$BWT)

Shapiro-Wilk normality test

data:  lowbwt$BWT
W = 0.99247, p-value = 0.4383
```

Wie gewohnt wird auch bei diesem Test die H_0 -Hypothese getestet, dass die Grundgesamtheit normalverteilt ist.

Den Kolmogorov-Smirnov-Test verwenden wollen, sollten Sie eine Version mit Lilliefors-Korrektur des p-Wertes verwenden. Diesen Test finden Sie im Paket *nortest*¹⁰ als `lillie.test()`.

VARIANZGLEICHHEIT KÖNNEN SIE GRAFISCH mit einem Boxplot überprüfen. Die Abbildung zeigt die Verteilung des Geburtsgewichtes für rauchende und nicht-rauchende Mütter¹¹.

Varianzgleichheit grafisch
 prüfen mit

`boxplot()`

```
> boxplot(lowbwt$BWT ~ lowbwt$SMOKE)
```

¹⁰ Juergen Gross und Uwe Ligges, *nortest: Tests for Normality*, 2015, <https://CRAN.R-project.org/package=nortest>.

¹¹ Genau genommen müssten Sie die Normalverteilung auch in beiden Gruppen einzeln prüfen. Versuchen Sie das mit `tapply()`

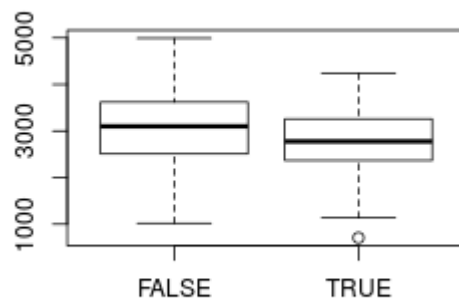


Abbildung 24 : Grafische Prüfung auf Varianzgleichheit

Den Levene-Test finden Sie im Paket *car* als `leveneTest()`.

`leveneTest()`

```
> library(car)
> leveneTest(lowbwt$BWT ~ lowbwt$SMOKE)
Levene's Test for Homogeneity of Variance (center = median)
      Df F value Pr(>F)
group  1  1.3901 0.2399
      187
```

BEACHTEN SIE SOWOHL IN DER FUNKTION für den Boxplot als auch in der für den Test die Schreibweise mit `~`. Das ist die sogenannte Formelschreibweise nach dem Muster abhängige Variable `~` unabhängige Variable(n). Sie taucht immer wieder auf, insbesondere bei den Regressionsmodellen.

Formelschreibweise:

`AV ~ UV`

Hypothesentests

Wichtige Hypothesentests:

Tabelle 9: Wichtige Hypothesentests

Funktion	Test
<code>t.test()</code>	T-Test
<code>wilcox.test()</code>	Wilcoxon signed rank bzw. rank sum Test (=Mann-Whitney-Test)
<code>chisq.test()</code>	Chi ² -Test
<code>fisher.test()</code>	Fisher exact Test
<code>aov()</code>	ANOVA
<code>kruskal.test()</code>	Kruskal-Wallis-Test

Die Funktion `t.test()` berechnet standardmäßig die ungepaarte

`t.test()`

Version des Tests. Mit dem Argument `paired = TRUE` können Sie auf einen gepaarten Test umschalten. Außerdem wird standardmäßig die Welch-Schätzung der Freiheitsgrade berechnet, die auch für ungleiche Varianzen gültig ist. Bei Varianzgleichheit kann man das mit `var.equal = TRUE` abschalten.

```
> t.test(BWT ~ SMOKE, data = lowbwt)
```

```
Welch Two Sample t-test
```

```
data: BWT by SMOKE
t = 2.7095, df = 170, p-value = 0.00743
alternative hypothesis: true difference in means is
not equal to 0
95 percent confidence interval:
 76.46677 486.95979
sample estimates:
mean in group FALSE mean in group TRUE
 3054.957          2773.243
```

Das non-parametrische Äquivalent zum T-Test ist der Wilcoxon-Test `wilcox.test()`. Standardmäßig wird er ungepaart als Rangsummentest ausgeführt, der dem Mann-Whiney-Test entspricht. Er kann auch mit `paired = TRUE` auf gepaarte Stichproben angewendet werden.

```
wilcox.test()
```

```
> wilcox.test(BWT ~ SMOKE, data = lowbwt)
```

```
Wilcoxon rank sum test with continuity correction
```

```
data: BWT by SMOKE
W = 5243.5, p-value = 0.007109
alternative hypothesis: true location shift is not
equal to 0
```

Den Chi²-Test `chisq.test()` wird standardmäßig mit Yates-Korrektur berechnet. Das können Sie mit dem Argument `correct = FALSE` abschalten.

```
chisq.test()
```

```
> chisq.test(lowbwt$SMOKE, lowbwt$LOW, correct =
FALSE)
```

```
Pearson's Chi-squared test
```

```
data: lowbwt$SMOKE and lowbwt$LOW
X-squared = 4.9237, df = 1, p-value = 0.02649
```

Den Fisher exact Test berechnet man mit `fisher.test()`.

```
fisher.test()
```

```
> fisher.test(lowbwt$LOW, lowbwt$SMOKE)
```

```
Fisher's Exact Test for Count Data
```



```
data: lowbwt$LOW and lowbwt$SMOKE
p-value = 0.03618
alternative hypothesis: true odds ratio is not equal
to 1
95 percent confidence interval:
1.028780 3.964904
sample estimates:
odds ratio
2.014137
```

Eine Varianzanalyse berechnen Sie mit `aov()`. Der zugehörige F-Test wird dann berechnet, indem man `summary()` auf das `aov()`-Objekt anwendet.

`aov()`

```
> summary(aov(BWT ~ RACE, data = lowbwt))
              Df    Sum Sq Mean Sq F value    Pr(>F)
RACE              1  3846362 3846362    7.487 0.00681 **
Residuals       187  96070691   513747
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.'
0.1 ' ' 1
```

Hier noch ein Beispiel für den Kruskal-Wallis-Test.

`kruskal.test()`

```
> kruskal.test(lowbwt$BWT ~ lowbwt$RACE)

Kruskal-Wallis rank sum test

data: lowbwt$BWT by lowbwt$RACE
Kruskal-Wallis chi-squared = 8.5909, df = 2, p-value =
0.01363
```

Zusammenhangsmaße

Korrelationen kann man mit `cor()` berechnen. Standardmäßig wird eine Pearson-Korrelation gerechnet. Mit dem Argument `method =` können Sie auch auf "kendall" oder "spearman" umschalten. Wenn Sie auch einen p-Wert berechnen möchten, benutzen Sie analog `cor.test()`.

`cor()`

`cor.test()`

```
> cor.test(lowbwt$BWT, lowbwt$LWT)

Pearson's product-moment correlation

data: lowbwt$BWT and lowbwt$LWT
t = 2.5856, df = 187, p-value = 0.01048
alternative hypothesis: true correlation is not equal
to 0
95 percent confidence interval:
0.04423134 0.32003247
sample estimates:
cor
0.1857887
```

Die Linearität des Zusammenhangs können sie einfach mit `scatter.smooth()` prüfen, wobei ein Streudiagramm mit LOESS-Anpassungslinie erstellt wird.

`scatter.smooth()`

```
> scatter.smooth(lowbwt$BWT ~ lowbwt$LWT)
```

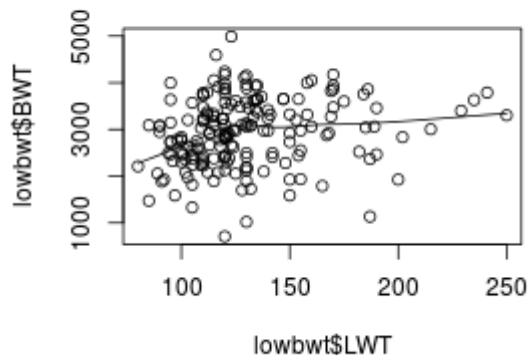


Abbildung 25 : Grafische Prüfung Linearität des Zusammenhangs

Wenn Sie Phi, Cramer's V und den Kontingenzkoeffizient nicht selber berechnen möchten, können Sie die Funktion `assocstats()` aus dem Paket *vcd*¹² verwenden.

`vcd::assocstats()`

```
> library(vcd)
> assocstats(table(lowbwt$LOW, lowbwt$SMOKE))
              X^2 df P(> X^2)
Likelihood Ratio 4.8674  1 0.027369
Pearson          4.9237  1 0.026491

Phi-Coefficient   : 0.161
Contingency Coeff.: 0.159
Cramer's V       : 0.161
```

Epidemiologische Maße

Das Paket *epiDisplay*¹³ berechnet die typischen epidemiologischen Maße aus 2x2-Tabellen. Dabei wird als erstes das Outcome und als zweites die Exposition angegeben. Die Daten müssen so kodiert sein, dass "nein" alphanumerisch vor "ja" sortiert wird, also z.B. 0/1 oder FALSE/TRUE.

Paket *epiDisplay*

Für relative Risiken, Risikodifferenzen und NNT können sie die

`cs()`

¹² David Meyer, Achim Zeileis, und Kurt Hornik, *vcd: Visualizing Categorical Data*, Version 1.4-1., 2015.

¹³ Virasakdi Chongsuvivatwong, *epiDisplay: Epidemiological Data Display Package*, 2015, <https://CRAN.R-project.org/package=epiDisplay>.

Funktion `cs()` anwenden.

```
> library(epiDisplay)
> cs(lowbwt$LOW, lowbwt$SMOKE)
```

Outcome	Exposure		
	Non-exposed	Exposed	Total
Negative	86	44	130
Positive	29	30	59
Total	115	74	189

	Rne	Re	Rt
Risk	0.25	0.41	0.31

	Lower95ci	Upper95ci	Estimate
Risk difference (attributable risk)	0.15	0.01	0.28
Risk ratio	1.61	1.02	2.53
Attr. frac. exp. -- (Re-Rne)/Re			0.38
Attr. frac. pop. -- (Rt-Rne)/Rt*100 %			19.22
Number needed to harm (NNH)	6.53	3.6	69.11
or 1/(risk difference)			

Um Odds Ratios zu berechnen, verwenden Sie `cc()`. Mit `design` = können Sie das Studiendesign festlegen. Das wirkt sich aber nur auf die zugehörige Grafik aus.

```
> cc(lowbwt$LOW, lowbwt$SMOKE)
```

	lowbwt\$SMOKE		
lowbwt\$LOW	FALSE	TRUE	Total
0	86	44	130
1	29	30	59
Total	115	74	189

```
OR = 2.02
95% CI = 1.08, 3.78
Chi-squared = 4.92, 1 d.f., P value = 0.026
Fisher's exact test (2-sided) P value = 0.036
```

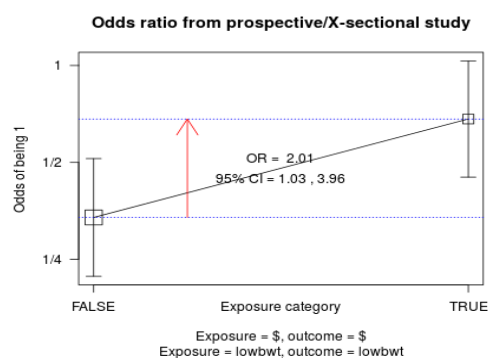


Abbildung 26 : OR-Grafik von `epiDisplay::cc()`

Stratifizierte Mantel-Haenszel-Berechnungen können mit `mhor()` durchgeführt werden.

`mhor()`

```
> mhor(lowbwt$LOW, lowbwt$SMOKE, lowbwt$HT)
```

```
Stratified analysis by Var3
      OR lower lim. upper lim. P value
Var3 0      2.12      1.0466      4.34 0.0281
Var3 1      1.11      0.0673     22.01 1.0000
M-H combined 2.03      1.0787      3.83 0.0276

M-H Chi2(1) = 4.86 , P value = 0.028
Homogeneity test, chi-squared 1 d.f. = 0.27 , P value
= 0.602
```

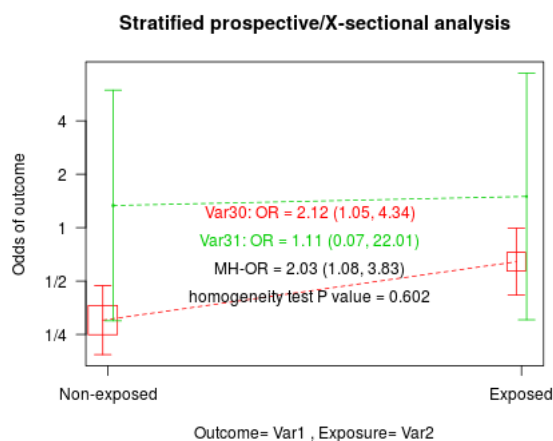


Abbildung 27 : Mantel-Haenszel-Berechnungen mit `mhor()`

5 Literatur

- Buuren, Stef, und Karin Groothuis-Oudshoorn. „mice: Multivariate imputation by chained equations in R“. *Journal of statistical software* 45, Nr. 3 (2011). <http://doc.utwente.nl/78938/>.
- Chang, Winston. *R Graphics Cookbook*. Beijing; Sebastopol, CA: O'Reilly, 2013.
- Chongsuvivatwong, Virasakdi. *epiDisplay: Epidemiological Data Display Package*, 2015. <https://CRAN.R-project.org/package=epiDisplay>.
- Crawley, Michael J. *The R book*. Chichester u.a.: Wiley, 2007.
- Field, Andy P, Jeremy Miles, und Zoë Field. *Discovering Statistics Using R*. London; Thousand Oaks, Calif.: Sage, 2012.
- Gross, Juergen, und Uwe Ligges. *nortest: Tests for Normality*, 2015. <https://CRAN.R-project.org/package=nortest>.
- Meyer, David, Achim Zeileis, und Kurt Hornik. *vcd: Visualizing Categorical Data* (Version 1.4-1.), 2015.

- R Core Team. *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing, 2015. <https://www.R-project.org/>.
- RStudio Team. *RStudio: Integrated Development Environment for R*. Boston, MA: RStudio, Inc., 2012. <http://www.rstudio.com/>.
- Sanchez, Gaston. *Handling and processing strings in R*. Berkley: Trowchez Editions, 2013. <http://lib.psylab.info/files/Sanchez2013.pdf>.
- Schendera, Christian F. G. *Datenqualität mit SPSS*. München u.a.: Oldenbourg, 2007.
- Wickham, Hadley. *Ggplot2 Elegant Graphics for Data Analysis*. Dordrecht; New York: Springer, 2009.
- . *stringr: Simple, Consistent Wrappers for Common String Operations*, 2015. <https://CRAN.R-project.org/package=stringr>.
- Wilkinson, Leland, und Graham Wills. *The Grammar of Graphics*. New York: Springer, 2005. <http://public.eblib.com/choice/publicfullrecord.aspx?p=302755>.