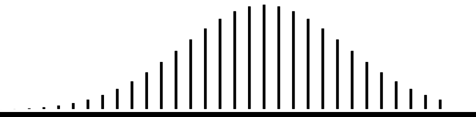
R Reader

Arbeiten mit dem Statistikprogramm R



von Jörg Groß & Benjamin Peters

1. Version: Juli 2009

erstellt von: Jörg Groß & Benjamin Peters

Kontakt: joerg@licht-malerei.de
peters@med.uni-frankfurt.de
http://www.random-stuff.de

Dieser Reader wurde erstellt mit LaTeX

Informationen zum Textsatzprogramm LaTeX: *http://www.latex-project.org*
Informationen zum Statistikprogramm R: *http://www.r-project.org*

Inhaltsverzeichnis

1 Variablen, Funktionen und Vektoren **4**

1.1 Rechnen mit R . . . 4
1.2 Variablen . . . 5
1.3 Funktionen definieren . . . 6
1.4 Vektoren und Vektoroperationen . . . 7
1.5 Bereits definierte Funktionen nutzen . . . 8
1.5.1 Hilfe und Erläuterungen finden . . . 8

2 Arbeiten mit Variablen- und Datentypen **10**

2.1 Grundlegende Datentypen . . . 10
2.2 Weitere Datentypen . . . 11
2.2.1 data.frame . . . 11
2.2.2 table . . . 14
2.2.3 matrix . . . 15
2.2.4 list . . . 17

3 Datenimport und -export **20**

3.1 Die read.table() Funktion . . . 20
3.2 Einlesen von anderen Dateitypen . . . 23
3.3 Export von Daten . . . 23

4 Deskriptivstatistik **25**

4.1 Zentrale Tendenz und Dispersionsmaße . . . 25
4.2 Bivariate Zusammenhangsmaße . . . 26
4.3 Anwendung von Funktionen auf Matrizen, Data-Frames und Listen . . . 28
4.4 Die summary() und describe() Funktionen . . . 30

5 Grafische Darstellung (High-Level Plots) **31**

5.1 Speichern von Grafiken . . . 34

6 Bivariate Inferenzstatistik und Verteilungen **36**

6.1 Verteilungen in R . . . 38
6.1.1 Funktionen der Verteilungen . . . 38
6.1.2 Weitere Verteilungen in R . . . 40

7 Kontrollstrukturen in R **41**

7.1 Wenn-Dann Beziehungen . . . 41
7.2 Schleifen . . . 42

8 Regressions- und Varianzanalyse **46**
8.1 Modelle in R formulieren 46
8.2 Darstellung der Ergebnisse 52
8.2.1 Stetige Prädiktoren 52
8.2.2 Kategoriale Prädiktoren 53
8.3 Grafische Beurteilung der Voraussetzungserfüllung 53
8.3.1 Homoskedastizität 54
8.3.2 Normalverteilung der Residuen 54

9 Low-Level und High-Level Plotting **57**
9.1 High-Level Plotting 57
9.1.1 Die par()-Funktion 57
9.2 Low-Level Plotting 59

Lösungsvorschläge **63**

Kapitel 1

Variablen, Funktionen und Vektoren

1.1 Rechnen mit R

In dem Hauptfenster von R lassen sich mithilfe der folgenden Symbole simple Berechnungen durchführen. R folgt dabei den üblichen arithmetischen Regeln (Punkt vor Strich Regel, Klammersetzung etc.).

+	Addition
–	Subtraktion
*	Multiplikation
/	Division
ˆ	Potenz
e	Zehnerpotenz

EINGABE

```
 3 + 4 - 2
 0.001 + 7  * 12
(0.001 + 7) * 12
 3.1 / 2.0
 2^2
 1e2
 1e2^0.5
```

AUSGABE

```
[1] 5
[1] 84.001
[1] 84.012
[1] 1.55
[1] 4
[1] 100
[1] 10
```

Neben grundlegenden Rechenoperationen lassen sich auch logische Operationen durchführen. Mithilfe verschiedener Symbole können Werte verglichen werden oder größer/kleiner Beziehungen überprüft werden.

==	ist gleich
!=	ist ungleich
>	ist größer
<	ist kleiner
>=	ist größer gleich
<=	ist kleiner gleich

EINGABE

```
 3 == 3
 3 == 4
 3 != 4
```

AUSGABE

```
[1] TRUE
[1] FALSE
[1] TRUE
```

Mehrere logische Vergleiche lassen sich mit den folgenden Zeichen verknüpfen:

<table>
<tr><td>&</td><td>und (∩)</td></tr>
<tr><td>|</td><td>oder (∪)</td></tr>
</table>

EINGABE
```
 3 <= 4 & 3 == 3
 3 <= 4 | 3 == 4
(3 <= 4 | 3 == 3) & 3 == 4
```

AUSGABE
```
[1] TRUE
[1] TRUE
[1] FALSE
```

1.2 Variablen

<-

Eine Variable ist eine Art Speicher, den man mit verschiedenen Datentypen füllen kann. Der Name einer Variable muss in R immer mit einem Buchstaben beginnen. R ist case-sensitiv, das heißt *MeineVariable* ist für R nicht das Gleiche wie *meinevariable*.
Man kann einer Variable auf unterschiedliche Art und Weise Inhalte zuweisen. Üblicherweise wird der <- Operator benutzt (so genannter Zuweisungsoperator). Wird keine Zuweisung durchgeführt, sondern einfach der Name einer bereits definierten Variable eingetippt, so wird der Inhalt dieser Variablen ausgegeben.

EINGABE
```
MeinVariablenName_numerisch <- 3 + 4
MeinVariablenName_numerisch

MeinVariablenName_string <- "Hello world!"
MeinVariablenName_string
```

AUSGABE
```
[1] 7
[1] "Hello world!"
```

Grundsätzlich kann mit Variablen genauso gerechnet werden wie mit Zahlen.

EINGABE
```
x <- 3 + 4
y <- 2
z <- x * y
z
z^2
```

AUSGABE
```
[1] 14
[1] 196
```

Wird einer Variablen ein neuer Wert zugewiesen, wird dabei der alte Wert überschrieben.

EINGABE

```
a <- 3 + 4
a <- 100000
a
```

AUSGABE

```
[1] 100000
```

1.3 Funktionen definieren

function(), return()

Zur Automatisierung von häufig durchgeführten Abläufen können Funktionen definiert werden. Eine Funktion kann bei ihrem Aufruf Funktionsparameter aufnehmen und einen (oder mehrere) Funktionswerte zurückgeben.
Hierfür ein Beispiel für die Definition einer Funktion zur Berechnung des Mittelwerts zweier Zahlen x und y; Im unteren Code-Beispiel wird in der ersten Zeile festgelegt, dass *mittelwert.aus.zwei* eine Funktion sein soll, die beim Aufruf zwei Parameter benötigt (in diesem Fall die beiden Zahlen x und y). Innerhalb der geschweiften Klammern befindet sich die Definition der Funktion. Alles was sich innerhalb der geschweiften Klammern befindet wird jedes mal ausgeführt sobald die Funktion aufgerufen wird. In unserer *mittelwert.aus.zwei* Funktion wird zunächst eine neue Variable *mittel* als Mittelwert von x und y definiert. In der nächsten Zeile wird mit *return(mittel)* die Funktion beendet und der Wert der Variablen *mittel* zurückgegeben. Anschließend lässt sich die Funktion aufrufen.

EINGABE

```
mittelwert.aus.zwei <- function(x, y)
        {
        mittel <- (x + y) / 2
        return(mittel)
        }

mittelwert.aus.zwei(3, 5)
```

AUSGABE

```
 [1] 4
```

Auf Variablen, die innerhalb der Funktion definiert werden, kann auch nur innerhalb dieser Funktion zugegriffen werden. Im oberen Beispiel betrifft dies die Variable *mittel*. Ein manueller Aufruf von *mittel* ausserhalb der Funktionsdefinition würde eine Fehlermeldung nach sich ziehen, die darauf hinweist, dass keine Variable mit der Bezeichnung *mittel* existiert.

Arbeiten mit dem Skript-Editor

Mehrzeiliger Code kann und sollte in einem, von dem Konsolenfenster getrennten, Skriptfenster eingeben werden. In Windows öffnet man ein leeres Skriptfenster über den Menüpunkt *Datei → Neues Skript*. Ausführen kann man den Code aus dem Skriptfenster am komfortabelsten über Tastaturkürzel. Mit dem Tastenkürzel *STRG + R* wird die aktive Zeile, in der sich der Cursor gerade befindet, vom Skriptfenster an das Konsolenfenster gesendet und ausgeführt. Mit der Kombination *STRG + A* und *STRG + R* wird zunächst der gesamte Code über alle Spalten hinweg im Skriptfenster ausgewählt und anschließend an das Konsolenfenster gesendet.

1.4 Vektoren und Vektoroperationen

Vektoren sind ein essentieller Bestandteil von R.
Im Prinzip haben wir bereits mit Vektoren gearbeitet;

EINGABE

```
a <- 3+4
a
```

Bei der Definition von a als die Summe von 3 und 4 wurde ein Vektor der Länge eins erstellt. Um einer Variablen einen Vektor mit Länge größer eins zuzuweisen, wird in R die Funktion *c()* benutzt. Das c steht dabei für concatenate (engl.: verbinden / verketten).
Das folgende Beispiel ist kommentiert. Alles, was nach dem Zeichen # steht, wird bei der Skriptausführung ignoriert.

c(), #

EINGABE

```
a <- c(1,2,3,4,5)              # ein Vektor mit fünf Elementen
a.reversed <- c(5,4,3,2,1)     # in umgekehrter Reihenfolge

b <- c(1:5)                    # das Gleiche wie a
b.reversed <- c(5:1)           # das Gleiche wie a.reversed

b == a
b.reversed == a.reversed

b
a.reversed

textvector <- c("hello","world!") # Länge = 2
textvector

logical.vector <- c(3 == 3,  3 != 4,  3 < 2)
logical.vector
```

AUSGABE

```
[1] TRUE TRUE TRUE TRUE TRUE
[1] TRUE TRUE TRUE TRUE TRUE
[1] 1 2 3 4 5
[1] 5 4 3 2 1
[1] "hello"  "world!"
[1] TRUE TRUE FALSE
```

Über die Indizierung mit *[]* lässt sich auf einzelne Elemente eines Vektors zugreifen.

[]

EINGABE

```
b.reversed[2]                  # das 2. Element von b
b.reversed[3] <- 100
b.reversed
b.reversed[2:3]                # das 2.-3. Element
b.reversed[c(1,3,5)]           # das 1., 3., 5. Element
```

AUSGABE

```
[1] 4
[1] 5 4 100 2 1
[1] 4 100
[1] 5 100 1
```

1.5 Bereits definierte Funktionen nutzen

length(), sum()

Neben der schon angesprochenen Funktion *c()*, mit der sich Objekte aneinanderreihen lassen, gibt es eine Vielzahl bereits definierter Funktionen, mit denen sich verschiedenste Operationen durchführen lassen.
Beispielsweise lässt sich mit der Funktion *length()* die Länge des Vektors bestimmen. Mit *sum()* wird die Summe eines Vektors errechnet.

EINGABE

```
1 length(b.reversed)
2 sum(b.reversed)
```

AUSGABE

```
1 [1] 5
2 [1] 112
```

1.5.1 Hilfe und Erläuterungen finden

?

Um herauszufinden, welche Funktionsparameter (so genannte Argumente) einer bestimmten Funktion übergeben werden müssen und was die Funktion ausgibt, kann die Hilfedatei zur Funktion mithilfe eines Fragezeichens vor dem Funktionsnamen aufgerufen werden:

EINGABE

```
1 ?sum      # öffnet die Hilfedatei der Funktion sum
```

In der Hilfedatei zu *sum()* lässt sich beispielsweise lesen, dass die Funktion neben numerischen Vektoren auch mit Vektoren vom Typ komplex oder logisch arbeiten kann. *sum(textvector)* würde also eine Fehlermeldung ausgeben. Bei der Summierung eines Vektors mit logischen Elementen wird *TRUE* als 1 und *FALSE* als 0 gewertet.

EINGABE

```
1 sum(logical.vector)
```

AUSGABE

```
1 [1] 2
```

help.search()

Die gesamten Hilfedateien zu Funktionen können auch nach Stichworten durchsucht werden. Dies geschieht über die Funktion *help.search()*. Insbesondere wenn man weiß welche Operation man durchführen will, allerdings den betreffenden Funktionsnamen, der diese Operation durchführt, nicht kennt ist die Funktion *help.search()* sehr hilfreich.

EINGABE

```
1 help.search("variance")
2
3 # durchsucht die Hilfedateien nach dem Stichwort "variance", in der Hoffnung
4 # eine Funktion zu finden, welche die Varianz eines Vektors, bzw. einer
5 # Zahlenreihe errechnet.
```

Übungen

1. Erstelle eine Funktion, die den Mittelwert aus vier Zahlen ausgibt.

2. Erstelle unter Einbeziehung der Funktionen *c(), length()* und *sum()* eine Funktion, die den Mittelwert aus beliebig vielen Zahlen berechnet.

3. Welcher Output wird durch den unten stehenden Code erzeugt?

EINGABE
```
1  z <- 2
2  y <- length(z)
3  x <- c(1:20)
4  y <- length(x[1:10])
5  y * z
```

4. In einer Untersuchung wurden die beiden folgenden Variablen mit jeweils 10 Messwerten gemessen. Berechne Mittelwert, Varianz und Kovarianz der beiden Variablen. Mach dir die Hilfefunktion zunutze, um die entsprechenden Funktionen zu finden.

EINGABE
```
1  x1 <- c(1, 2, 2, 2, 5, 6, 3, 2)
2  x2 <- c(9, 7, 7, 6, 3, 9, 9, 8)
```

5. In einem Genauigkeitstest werden von Probanden Darts auf ein Ziel geworfen. Die Teilnehmer können so oft werfen wie sie wollen. Am Ende errechnet sich für jeden Probanden ein Leistungsscore, der sich (aus empirisch optimaler Sicht) als die an den N Durchgängen relativierte Summe der Abweichung zur vierten Potenz berechnet. Hohe Scorewerte stehen für schlechte Leistung.

   Da durch ein hohes N an Durchgängen der Scorewert immer weiter verbessert werden kann, möchte man eine Gewichtung der Abweichungen in Abhängigkeit vom Durchgang einführen. Hierfür hat sich folgende Funktion bewährt:

$$\text{score} = \frac{\sum_{i=1}^{N} (ix_i)^4}{N}$$

   Erstelle eine Funktion nach dem Prinzip von Aufgabe 2, welche die Leistungswerte berechnet und ermittle diese für folgende Personen:

   Abweichungen von ...

   Person 1: 4.2, 3.1, 3, 3.4, 2.1, 1.1, 1.0, 1.0, 1.2, 1.1

   Person 2: 5.2, 3.0, 3.7, 2.1, 1.0, 0.5

Kapitel 2

Arbeiten mit Variablen- und Datentypen

2.1 Grundlegende Datentypen

Die Elemente, die man einer Variablen zuordnet, lassen sich in verschiedene Datentypen untergliedern. Auf der elementarsten Stufe unterscheidet man nummerische Daten, Daten vom Typ „Character“, logische Daten und faktorielle Daten. Nummerische Daten lassen sich des Weiteren unterteilen in Daten vom Typ *double* (Gleitkommazahlen) oder *integer* (ganze Zahlen).
In R muss man nicht vorher definieren, welchen Datentypus eine Variable aufnehmen kann oder soll. Stattdessen entscheidet R eigenständig bei der Verarbeitung zu welchem Typus eine Variable oder das Element einer Variable zählen soll.

is.numeric(), as.numeric(), is.character(), as.character(), is.logical(), as.logical(), is.factor(), as.factor(),

Mit den Funktionen *is.datentypus* lässt sich feststellen, zu welchem Typ ein Element oder der Inhalt einer Variable gehört. Über die Funktion *as.datentypus* lassen sich Variablen oder Objekte von einem Datentyp in einen anderen Datentyp umwandeln.

is.double(), as.double(), is.integer(), as.integer()

EINGABE

```
1  eine.character.variable <- "212"
2  is.character(eine.character.variable)
3
4  eine.numerische.variable <- as.numeric(eine character.variable)
5
6  is.character(eine.numerische.variable)
7  is.numeric(eine.numerische.variable)
```

AUSGABE

```
1  [1] TRUE
2  [1] FALSE
3  [1] TRUE
```

Übersicht über die fundamentalen Datentypen:

numeric	numerischer Datentyp (*is.numeric()*, *as.numeric()*)
character	Character-Datentyp (*is.character()*, *as.character()*)
logical	logischer Datentyp (*is.logical()*, *as.logical()*)
factor	faktorieller Datentyp (*is.factor()*, *as.factor()*)
double	nummerische Daten vom Typ double (*is.double()*, *as.double()*)
integer	Integer-Datentyp (*is.integer()*, *as.integer()*)

rep()

Gegenüber anderen Programmiersprachen, in denen ähnliche Datentypen und -bezeichnungen üblich sind, sind Daten vom Typ „factor“ eine Eigenheit von R. Statistische Daten, die auf Nominal- oder Ordinalskalenniveau vorliegen, werden in R meist als Daten diesen Typus definiert.
Die im unteren Beispiel verwendete Funktion *rep(x,n)* erzeugt einen Vektor, in dem das erste Argument x n-mal wiederholt wird (Beispiel: *rep(5,3)* erzeugt den selben Vektor wie *c(5,5,5)*)

EINGABE

```
x <- c(rep(1,10), rep(2,10))

geschl.factor.labeled <- factor(x, labels=c("w", "m"))
geschl.factor.labeled

is.factor(geschl.factor.labeled)
```

AUSGABE

```
[1] w w w w w w w w w w m m m m m m m m m m
Levels: w m
[1] TRUE
```

Bei einer Faktorvariable wird nicht zwischen Zahlen oder Zeichenketten unterschieden. Die erste Stufe kann die Bezeichnung „1“, „w“ oder „weiblich“ tragen - bei Berechnungen mit einer Faktorvariable macht dies keinen Unterschied. Weist man einer Faktorvariable einen Wert zu, der aus Buchstaben besteht, so werden diese im Output nicht durch Anführungszeichen ausgegeben, wie das bei reinen Charactervariablen der Fall ist. Der Name einer Faktorstufe kann über das optionale Argument „labels“ der Funktion *factor()* gewählt werden.

factor()

Über die Funktion *levels()* lassen sich die Faktorstufen einer bestehenden Faktorvariablen im Nachhinein noch verändern. Dabei wird die erste Level-Bezeichnung des Vektor (im oberen Fall: w) durch die erste Bezeichnung im neuen Vektor ersetzt (unten: weiblich) und so weiter.

levels()

EINGABE

```
levels(geschl.factor.labeled) <- c("weiblich", "männlich")
geschl.factor.labeled
```

AUSGABE

```
 [1] weiblich weiblich weiblich weiblich weiblich
 [6] weiblich weiblich weiblich weiblich weiblich
[11] männlich männlich männlich männlich männlich
[16] männlich männlich männlich männlich männlich
Levels: weiblich männlich
```

2.2 Weitere Datentypen

2.2.1 data.frame

Mithilfe von data-frames lassen sich mehrere Vektoren zu einer Datentabelle verknüpfen. Diese Datentabelle kann Vektoren verschiedenen Typus enthalten (faktoriell, numerisch etc.). Die Vektoren sollten allerdings gleich lang sein, also gleich viele Elemente beinhalten.

Mit der Funktion *data.frame()* lässt sich eine solche Datentabelle erstellen und zuweisen. Im unteren Beispiel wird des Weiteren die Funktion *rnorm(x,y,z)* benutzt, mit der sich x-Zufallsdaten einer Normalverteilung mit dem Mittelwert y und der Standardabweichung z generieren lassen. Mit der Funktion *round(x,y)* lassen sich die Elemente einer Variable x auf die y-te Stelle runden.

data.frame(), rnorm(), round()

EINGABE

```
messergebnisse.m <- rnorm(10, 5, 2)
                        # Erzeugung von 10 Zufallszahlen aus einer normal-
                        # verteilten Population mit Mittelwert 5 und SD = 2

messergebnisse.m

messergebnisse.m <- round(messergebnisse.m, 2)
                        # runden der Zufallszahlen auf zwei Nachkommastellen

messergebnisse.w <- round(rnorm(10, 8, 3),2)
                        # diesmal beide Funktion ineinander verschachtelt

messergebnisse <- c(messergebnisse.m, messergebnisse.w)
                        # Zusammenfügen der beiden Vektoren

datentabelle <- data.frame(geschlecht = geschl.factor.labeled,
                                ergebnis = messergebnisse)
                        # Erstellung einer Datentabelle mit der Faktorvariable
                        # „geschlecht“ und der numerischen Variable „ergebnis“

datentabelle            # Ausgabe
```

AUSGABE

```
   geschlecht ergebnis
1    weiblich     3.19
2    weiblich     3.07
3    weiblich     5.32
4    weiblich     4.89
5    weiblich     6.12
6    weiblich     3.10
7    weiblich    10.13
8    weiblich     7.54
9    weiblich     5.06
10   weiblich     2.73
11   männlich     6.08
12   männlich     4.17
13   männlich     9.81
14   männlich    12.71
15   männlich     5.95
16   männlich    14.62
17   männlich    10.27
18   männlich     8.21
19   männlich    10.45
20   männlich     7.10
```

Im oberen Codebeispiel wurden fiktive Werte für männliche und weibliche Probanden erzeugt (Vektoren „messergebnisse.m“ und „messergebnisse.w“) und zu einem data-frame („datentabelle“) zusammengefügt.

Die Indizierung bei data-frames mit *[]* benötigt im Gegensatz zu der Indizierung bei Vektoren nun zwei Informationen: Die Spalte, die ausgewählt werden soll sowie die Zeile. Diese beiden Informationen werden innerhalb der Klammern durch ein Komma getrennt. Zunächst wird die Zeilennummer angegeben, die ausgegeben werden soll, dann die Spaltennummer. Wird eine Nummer freigelassen, so werden alle Elemente der jeweiligen Dimension ausgegeben.

EINGABE

```
datentabelle[1,2]    # Element in erster Zeile, zweite Spalte
                     # (Ergebniswert der ersten Person)

datentabelle[,1]     # erste Spalte (Geschlecht aller Personen)

datentabelle[3:5,2]  # drittes bis fünftes Element der zweiten Spalte
                     # (Ergebniswert der Person 3 bis 5)
```

AUSGABE

```
[1] 3.19

 [1] weiblich weiblich weiblich weiblich weiblich
 [6] weiblich weiblich weiblich weiblich weiblich
[11] männlich männlich männlich männlich männlich
[16] männlich männlich männlich männlich männlich
Levels: weiblich männlich

[1] 5.32 4.89 6.12
```

Neben der Indizierung über *[]* lassen sich die einzelnen Spalten auch mit dem Operator *$* und dem Spaltennamen anwählen.

EINGABE

```
datentabelle$ergebnis
```

AUSGABE

```
 [1]  3.19  3.07  5.32  4.89  6.12  3.10 10.13  7.54
 [9]  5.06  2.73  6.08  4.17  9.81 12.71  5.95 14.62
[17] 10.27  8.21 10.45  7.10
```

Da *datentabelle$ergebnis* ein Vektor ist, lässt sich wieder mit der von den Vektoren bekannten Indizierung arbeiten:

EINGABE

```
datentabelle$geschlecht[9:11] # Element 9 bis 11 der Spalte „geschlecht“
```

AUSGABE

```
[1] weiblich weiblich männlich
Levels: weiblich männlich
```

attach() und detach()

attach(), detach(), search()

Das Wählen einer Spalte mit dem *$*-Operator wird bei komplexeren Analysen und langen Variablen- und Spaltenbezeichnungen sehr unhandlich. Mit der Funktion *attach(x)* lässt sich eine Datentabelle x (oder auch andere Datentypen) zu dem Suchpfad von R hinzufügen.
Der Suchpfad, den man über die Funktion *search()* einsehen kann, regelt, in welcher Reihenfolge in R Pakete und Arbeitsumgebung nach einem eingegebenem Namen durchsucht werden, um herauszufinden, was sich hinter dem eingetippten Namen verbirgt (eine Variable, eine Funktion etc.).
Wird über *attach()* eine Datentabelle zu dieser Suchliste hinzugefügt, so hat dies zur Folge, dass

man auf die einzelnen Variablen (bzw. Spalten) innerhalb einer Datentabelle direkt über den Namen zugreifen kann. Über *detach(x)* wird die Datentabelle x wieder aus dem Suchpfad gelöscht.

Ohne attach() und detach():

EINGABE
```
1  datentabelle$ergebnis[datentabelle$geschlecht == "weiblich"]
2                     # alle Ergebniswerte von weiblichen Probanden über eine
3                     # sehr lange und unhandliche Indizierung
```

AUSGABE
```
1  [1]  3.19  3.07  5.32  4.89  6.12  3.10 10.13  7.54
2  [9]  5.06  2.73
```

Mit attach() und detach():

EINGABE
```
1  attach(datentabelle)
2  ergebnis[geschlecht == "weiblich"] # Die Alternative über attach()
3                                     # ist weitaus kürzer
4  detach(datentabelle)
5
6  ergebnis
```

AUSGABE
```
1  [1]  3.19  3.07  5.32  4.89  6.12  3.10 10.13  7.54
2  [9]  5.06  2.73
3
4  Fehler: objekt "ergebnis" nicht gefunden
```

2.2.2 table

Tabellen sind in R nützlich, wenn man Daten aggregieren und damit flexibel weiterarbeiten will. Die Spalte *geschlecht* der Variable *datentabelle* vom Typ data.frame lässt sich zum Beispiel folgendermaßen in eine table-Variable umwandeln:

EINGABE
```
1  attach(datentabelle)
2  geschl.freq <- table(geschlecht)
3  detach(datentabelle)
4
5  geschl.freq
```

AUSGABE
```
1  geschlecht
2  weiblich männlich
3        10       10
```

Es lassen sich auch mehrdimensionale Tabellen erstellen. Die unten benutzte Funktion *sample(x,y)* generiert y ganze Zufallszahlen aus einem Wertebereich x, wobei das optionale Argument *replace* regelt, ob beim Ziehen einer Zahl diese wieder zurückgelegt wird, also doppelt auftreten darf, oder nicht.

sample()

EINGABE

```
1  alter <- sample(c(19:25), 20, replace=TRUE)
2       # Erstellung von 20 zufälligen „Alterswerten“ zwischen 19 und 25 Jahren
3
4  alter
5  datentabelle$geschlecht
6
7  table(datentabelle$geschlecht,alter)
8       # Verknüpfung des Alters mit dem Geschlecht in einer Tabelle
```

AUSGABE

```
1   [1] 24 21 25 19 22 24 24 25 24 21 19 24 25 23 21 25
2  [17] 20 21 19 22
3
4   [1] w w w w w w w w w w m m m m m m m m m m
5  Levels: w m
6
7     alter
8      19 20 21 22 23 24 25
9    w  1  2  0  2  3  1  1
10   m  1  4  2  1  1  0  1
```

2.2.3 matrix

matrix()

Mit der Funktion *matrix()* lassen sich Daten des Typs matrix erzeugen. Im Gegensatz zu einem data-frame können die Elemente einer Matrix nur aus einem Datentyp bestehen. Es lassen sich also nicht wie bei einem data-frame Vektoren von verschiedenen Datentypen spaltenweise koppeln. Mit den Argumenten *nrow* und *ncol* dieser Funktion lassen sich die Dimensionen der erzeugten Matrix definieren.

EINGABE

```
1  x <- c(1:9) # Vektor von 1 bis 9
2  x <- matrix(x, nrow=3, ncol=3)
3
4  x
```

AUSGABE

```
1       [,1] [,2] [,3]
2  [1,]    1    4    7
3  [2,]    2    5    8
4  [3,]    3    6    9
```

Bei der Umwandlung eines Vektors in eine Matrix werden die Elemente des Vektors spaltenweise auf die Matrix verteilt. Durch das optionale Argument *byrow* kann man dies auch umstellen. Das Argument erwartet eine logische Eingabe und ist standardmäßig auf *FALSE* gesetzt.

t(), solve()

Durch die Operatoren %*% bzw. %\% kann mit Matrizen nach den Regeln der Matrix-Algebra gerechnet werden. Des Weiteren lassen sich mit der Funktion *t()* Matrizen transponieren, mit *solve()* lässt sich die Inverse einer Matrix finden.

EINGABE

```
x <- sample(50, 9)
x <- matrix(x, nrow=3, ncol=3)

y <- matrix(sample(50,9), nrow=3, ncol=3)

x
y

x %*% y                       # Matrix Multiplikation


t(x)                          # Matrix Transposition
inverse.x <- solve(x)    # Berechnung der Inverse

identity.x <- x %*% inverse.x


round(identity.x)        # Ausgabe der Einheitsmatrix
```

AUSGABE

```
     [,1] [,2] [,3]
[1,]   46   12    1
[2,]   42   44    4
[3,]    6    9   19

     [,1] [,2] [,3]
[1,]   10    3   38
[2,]    1    9   25
[3,]    7   14   42

     [,1] [,2] [,3]
[1,]  479  260 2090
[2,]  492  578 2864
[3,]  202  365 1251

     [,1] [,2] [,3]
[1,]   46   42    6
[2,]   12   44    9
[3,]    1    4   19

     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

Bei einer Matrix, in der die Spalten voneinander linear abhängig sind, erzeugt der solve-Algorithmus einen Fehler und meldet, wie zu erwarten, dass die Matrix singular ist, also keine Inverse besitzt.

EINGABE

```
y <- c(1,2,3)
x <- c(y, y*2, y*3)
x <- matrix(x, nrow=3, ncol=3)
solve(x)
```

AUSGABE

```
Fehler in solve.default(x) :
  Lapackroutine dgesv: System ist genau singulär
```

2.2.4 list

Listen stellen einen weiteren grundlegenden Datentypus in R dar. Ein Objekt von diesem Typ lässt sich über die Funktion *list()* erstellen.

EINGABE
```
1 x <- c(1:10)
2 y <- rep(c("eins", "zwei"), 3)
3 eine.liste <- list(x,y)
4 eine.liste
```

AUSGABE
```
1 [[1]]
2  [1]  1  2  3  4  5  6  7  8  9 10
3
4 [[2]]
5 [1] "eins" "zwei" "eins" "zwei" "eins" "zwei"
```

Eine Liste kann mehrere Vektoren aufnehmen und besteht dann aus mehreren Unterlisten, die über die *[[]]*-Indizierung einzeln angewählt werden können. Mit doppelter Indizierung können auch einzelne Elemente einer Unterliste ausgewählt werden. So gibt *eine.liste[[1]][[2]]* das zweite Element der ersten Unterliste zurück. Die Unterlisten können sich jeweils aus verschiedenen Datentypen zusammensetzen und verschieden viele Elemente beinhalten. Die verschiedenen Datentypen können, wie schon erwähnt, durch die *as.datentyp*-Funktionen ineinander konvertiert werden.

EINGABE
```
1 as.list(datentabelle)
```

AUSGABE
```
1 $geschlecht
2  [1] w w w w w w w w w w m m m m m m m m m m
3 Levels: w m
4
5 $ergebnis
6  [1]  7.14  6.99  5.13  7.57  6.30  1.99  7.51  5.74
7       6.48  5.76  5.55  4.85
8 [13]  4.94  9.85  9.43 11.55  5.23  5.23  6.20  9.05
```

Bei der Umwandlung der data-frame Variable „datentabelle“ in eine Liste wurden die Spaltennamen direkt als Listenüberschriften übernommen. Dadurch kann man auf die einzelnen Unterlisten nun auch mit dem *$*-Operator zugreifen.

Die Umwandlung von einem Datentyp in einen anderen funktioniert aber nicht immer oder kann teilweise zu unerwünschten Ergebnissen führen. So werden bei der Umwandlung des Data-Frames „datentabelle“ in eine Matrix alle Elemente der beiden Spaltenvariablen in Characterelemente umgewandelt, da bei einem Objekt des Typs „matrix“ alle Elemente genau dem gleichen Datentypus angehören müssen;

EINGABE
```
1 as.matrix(datentabelle)
```

AUSGABE

```
1         geschlecht ergebnis
2    [1,] "w"        " 7.14"
3    [2,] "w"        " 6.99"
4    [3,] "w"        " 5.13"
5    [4,] "w"        " 7.57"
6    [5,] "w"        " 6.30"
7    [6,] "w"        " 1.99"
8    [7,] "w"        " 7.51"
9    [8,] "w"        " 5.74"
10   [9,] "w"        " 6.48"
11  [10,] "w"        " 5.76"
12  [11,] "m"        " 5.55"
13  [12,] "m"        " 4.85"
14  [13,] "m"        " 4.94"
15  [14,] "m"        " 9.85"
16  [15,] "m"        " 9.43"
17  [16,] "m"        "11.55"
18  [17,] "m"        " 5.23"
19  [18,] "m"        " 5.23"
20  [19,] "m"        " 6.20"
21  [20,] "m"        " 9.05"
```

Übungen

1. Ändere die erste Spalte von m durch Multiplikation mit 2.

   EINGABE
   ```
   m <- matrix(c(1,2,7,8),2,2)
   ```

2. Welche Datentypen werden bei folgendem Code erzeugt?

   EINGABE
   ```
   a <- c("1", 2, 3, 4, 5)
   b <- as.numeric(a)
   c <- as.character(as.factor(as.numeric(a)))
   ```

3. a) Versuche einen Zufallsvektor mit 10 Elementen im ganzzahligen Wertebereich von 1 bis 3 zu generieren.
   b) Wandle diese Elemente in Faktorstufen um.
   c) Benenne diese Faktorstufen „Gruppe1“ bis „Gruppe3“.
   d) Wandle die Faktorvariable in eine Variable vom Typ character um.

4. In der Variable „datentabelle“ befinden sich Werte einer Voruntersuchung von 23 Probanden. Für eine Nachuntersuchung sollen die Personen zufällig einer von zwei Gruppen zugeteilt werden, einer Experimental- und einer Kontrollgruppe.
   a) Generiere eine Zufallsvariable, welche die Gruppenzugehörigkeit codiert. Benutze dafür die Funktion *sample()*.
   b) Wandle die Gruppenvariable in eine Variable des Typs „factor“ um, mit den Stufen „Kontrollgruppe“ und „Experimentalgruppe“.
   c) Verbinde das bestehende data-frame „datentabelle“ mit der Gruppenvariable über die Funktion *cbind()*.
   d) Angenommen bei der Nachuntersuchung wurden durchschnittlich 12 Punkte bei einer Standardabweichung von 2 erreicht. Generiere mit der *rnorm()* Funktion diese Werte und hänge sie an die Datentabelle an.

5. Folgende Werte wurden in einem Persönlichkeitstest an 10 Personen gemessen:

   11, 14, 17, 10, 19, 20, 16, 11, 9, 15

   Erstelle eine Faktorvariable, bei der Personen mit Werten niedriger 15 der Stufe „niedrig“ zugeordnet werden und Personen mit Werten höher oder gleich 15 der Stufe „hoch“ zugeordnet werden.

Kapitel 3
Datenimport und -export

3.1 Die read.table() Funktion

read.table()

Ein Nachteil von R ist, dass sich in der Basisversion die Eingabe von vielen Rohdaten als relativ umständlich gestaltet. Meist werden Rohdaten daher von außen importiert. Am einfachsten ist es Textdateien (.txt oder .csv) einzulesen, bei denen die Daten durch bestimmte Zeichen voneinander getrennt sind. Mit der Funktion *read.table()* können einfache Textdateien eingelesen werden. Die optionalen Argumente *sep*, *header* und *dec* helfen dabei die Dateistruktur adäquat einzulesen.

Mit *sep* wird der Funktion übergeben, welches Separatorsymbol in der Datei verwendet wird, um die einzelnen Daten voneinander zu trennen. Das Argument *header* nimmt eine Anweisung des Typs „logical" entgegen (*TRUE* / *FALSE*) und regelt, ob die erste Zeile als Variablennamen eingelesen werden soll oder nicht. Durch das Argument *dec* kann man angeben, welches Dezimalzeichen in der Datei verwendet wird. Standardmäßig wird ein Punkt als Dezimalzeichen erwartet, aber gerade wenn man eine Datendatei einlesen will, die zum Beispiel mit der deutschen Version von Excel erstellt wurde, muss man hier angeben, dass ein Komma als Dezimalzeichen verwendet wurde.

Bei Pfadangaben in R kann entweder ein doppelter Backslash oder ein normaler Slash verwendet werden, der aus Windows bekannte Backslash erzeugt in R eine Fehlermeldung.

EINGABE
```
1  datentabelle <- read.table("Pfad/Dateiname.txt",
2                              header=TRUE, dec=",")
3
4       # Liest eine Datei Dateiname.txt ein und verwendet die erste Zeile als
5       # Bezeichnung der Spalten und wandelt Kommas in Punkte um bei ungeraden
6       # Zahlen. Die eingelesenen Daten werden der Variable „datentabelle"
7       # zugewiesen.
```

Bei den meisten Funktionen, mit denen man Dateien in R einlesen kann, werden die Daten als „data-frame"-Objekt ausgegeben (bzw. einer Variablen zugewiesen).

Angenommen folgende Datendatei soll eingelesen werden:

DATEI: Dateiname.txt

```
Geschlecht Voruntersuchung Alter Brillentraeger Code
1 7,24  32  TRUE  HW23
1 6,98  20  FALSE TR01
1 99    21  TRUE  FR31
1 3,55  35  FALSE WE13
1 7,06  27  FALSE QQ04
1       20  FALSE AR29
1 5,32  25  FALSE AZ11
1 7,04  24  TRUE  TR17
1 4,08  29  FALSE OP17
1 5,57  22  FALSE UU07
2 7,76  30  FALSE ZQ25
2 99    28  FALSE TA24
2 7,49  32  FALSE XW22
2 7,07  26  FALSE CQ29
2 7,27  24  TRUE  CO03
2 8,21  24  FALSE PH06
2 4,73  25  FALSE JT14
2 8,12  29  TRUE  KU18
2 7,37  28  FALSE LR22
```

Die Datei ist tabgetrennt. Das bedeutet, dass nach jedem Tab eine neue Variable bzw. ein neuer Datenpunkt kommt. In der ersten Spalte befinden sich die Variablennamen und das Dezimalzeichen ist nach der deutschen Konvention ein Komma. In Zeile 7 fehlt ein Datenwert.

Die Datei könnte man folgendermaßen einlesen:

EINGABE

```
datentabelle <- read.table("Pfad/Dateiname.txt", header=TRUE,
                           sep="\t", dec = ",")
```

Das Symbol *t* ist ein so genannter regulärer Ausdruck und steht für einen Tabstopp.
Die Variable *datentabelle* ist nun ein Objekt des Typs „data.frame“:

EINGABE

```
datentabelle
is.data.frame(datentabelle)
```

AUSGABE

```
   Geschlecht Voruntersuchung Alter Brillentraeger Code
1           1            7.24    32           TRUE HW23
2           1            6.98    20          FALSE TR01
3           1           99.00    21           TRUE FR31
4           1            3.55    35          FALSE WE13
5           1            7.06    27          FALSE QQ04
6           1              NA    20          FALSE AR29
7           1            5.32    25          FALSE AZ11
8           1            7.04    24           TRUE TR17
9           1            4.08    29          FALSE OP17
10          1            5.57    22          FALSE UU07
11          2            7.76    30          FALSE ZQ25
12          2           99.00    28          FALSE TA24
13          2            7.49    32          FALSE XW22
14          2            7.07    26          FALSE CQ29
15          2            7.27    24           TRUE CO03
16          2            8.21    24          FALSE PH06
17          2            4.73    25          FALSE JT14
18          2            8.12    29           TRUE KU18
19          2            7.37    28          FALSE LR22

[1] TRUE
```

In der siebten Zeile, zweite Spalte wurde der fehlende Wert automatisch durch *NA* („not available“) ersetzt, das in R übliche Zeichen für fehlende Werte.
An der importierten Datentabelle sollen noch zwei Änderungen vorgenommen werden. Zum einen steht auch der Wert 99 in der Variable „Voruntersuchung“ für einen fehlenden Wert. Dieser soll durch *NA* ersetzt werden. Zum anderen soll die Variable „Geschlecht“ in eine Faktorvariable mit den Stufen weiblich und männlich umgewandelt werden. Dies kann folgendermaßen realisiert werden:

EINGABE

```
attach(datentabelle)

Voruntersuchung[Voruntersuchung == 99.00] <- NA

Geschlecht <- as.factor(Geschlecht)
levels(Geschlecht) <- c("weiblich", "männlich")

datentabelle <- data.frame(Geschlecht, Voruntersuchung,
                           Alter, Brillentraeger, Code)

detach(datentabelle)

datentabelle
```

AUSGABE

```
1      Geschlecht Voruntersuchung Alter Brillentraeger Code
2   1    weiblich            7.24    32           TRUE HW23
3   2    weiblich            6.98    20          FALSE TR01
4   3    weiblich              NA    21           TRUE FR31
5   4    weiblich            3.55    35          FALSE WE13
6   5    weiblich            7.06    27          FALSE QQ04
7   6    weiblich              NA    20          FALSE AR29
8   7    weiblich            5.32    25          FALSE AZ11
9   8    weiblich            7.04    24           TRUE TR17
10  9    weiblich            4.08    29          FALSE OP17
11  10   weiblich            5.57    22          FALSE UU07
12  11   männlich            7.76    30          FALSE ZQ25
13  12   männlich              NA    28          FALSE TA24
14  13   männlich            7.49    32          FALSE XW22
15  14   männlich            7.07    26          FALSE CQ29
16  15   männlich            7.27    24           TRUE CO03
17  16   männlich            8.21    24          FALSE PH06
18  17   männlich            4.73    25          FALSE JT14
19  18   männlich            8.12    29           TRUE KU18
20  19   männlich            7.37    28          FALSE LR22
```

3.2 Einlesen von anderen Dateitypen

read.spss()

Mit dem foreign-Paket ist es möglich auch Dateitypen einzulesen, die von anderen Statistikprogrammen genutzt werden. So können über die Funktion *read.spss()* dieses Pakets die *.sav-Dateien, die von der SPSS-Software erzeugt werden, eingelesen werden. Diese Funktion gibt die Daten standardmäßig als Liste wieder. Durch das Argument *to.data.frame*, das standardmäßig auf *FALSE* eingestellt ist, kann dies umgestellt werden, sodass die Daten als data-frame eingelesen werden. Weitere Informationen dazu findet man in dem Manual und den Hilfedateien dieses Pakets.

3.3 Export von Daten

write.table()

Mit der Funktion *write.table()* können Berechnungsergebnisse oder Datentabellen exportiert werden. Beim Export über diese Funktion sind insbesondere die Argumente *file*, *append*, *row.names*, *col.names* und *sep* relevant:

EINGABE

```
1  write.table(datentabelle, "Pfad/neueDatei.txt", append=FALSE,
2              sep =";", col.names=TRUE, row.names=FALSE)
```

Der oben stehende Befehl generiert eine Textdatei mit dem Inhalt der Variable „datentabelle“ an dem definiertem Pfad. Die Daten werden, wie durch das Argument *sep* definiert, mit einem Semikolon voneinander getrennt. Das Argument *append*, das standardmäßig auf *FALSE* eingestellt ist, definiert, ob die Daten an eine schon bestehende Datei angehängt werden sollen, oder gegebenenfalls der schon bestehende Inhalt dieser Datei überschrieben werden soll. Mit den letzten beiden Argumenten wird eingestellt, ob die Variablennamen in die erste Zeile der Datei geschrieben werden sollen bzw. ob die Nummerierung der Zeilen mit in die Datei geschrieben werden soll.

Standardmäßig werden Faktorvariablen, Charactervariablen und Variablennamen in der gespeicherten Datei in Anführungszeichen gesetzt. Der Eintrag männlich wird also als „männlich“ abgespeichert. Dies kann verhindert werden, indem man das Argument *quote* der Funktion *write.table()* von *TRUE* auf *FALSE* stellt.

Kapitel 4
Deskriptivstatistik

4.1 Zentrale Tendenz und Dispersionsmaße

Je nach Skalenniveau stehen verschiedene Funktionen zur Berechnung von zentraler Tendenz und Dispersion zur Verfügung.

mean(), median(), quantile(), sd(), var(), range(), mad(), IQR()

mean()	Arithmetisches Mittel
median()	Median
quantile()	Quantile
sd()	Geschätzte Standardabweichung
var()	Geschätzte Varianz
mad()	Mittlere absolute Abweichung
range()	Range
IQR()	Interquartilrange

Bei den Funktionen *sd()* und *var()* ist zu beachten, dass hierbei die korrigierte (geschätzte) Standardabweichung und Varianz berechnet wird und nicht die unkorrigierte Standardabweichung oder Varianz der Stichprobe:

$$\hat{\sigma}^2 = \frac{\sum_{i=1}^{n}(x_i - \bar{x})}{n-1}$$

In dem folgenden Beispiel wird unter anderem die Funktion *sort(x)* verwendet, die den Vektor *x* in aufsteigender Reihenfolge sortiert. Die Funktion *diff()* berechnet die Differenz der Werte eines Vektors (hier der beiden Quantilwerte).

sort()
diff()

EINGABE

```
1  x <- round(rnorm(10,0,1),2) # 10 Zufallszahlen aus N(0,1)
2  x
3
4  mean(x)     # Arithmetisches Mittel
5  var(x)
6  mad(x, center = mean(x))   # mittlere absolute Abweichung vom AM
7
8  stichproben.var = var(x)*((length(x)-1)/length(x))
9  stichproben.var
10
11 x.ord <- round(100*x^3,0)
12 x.ord       # nicht-lineare Transformation ergibt ordinale Daten
13
14 sort(x.ord)
15 median(x.ord)
16 mad(x.ord) # mittlere absolute Abweichung vom Median
17
18 quantile(x.ord)
19 range(x.ord)
20
21 iqr <- as.numeric(diff(quantile(x.ord,c(0.25,0.75))))
22
23 IQR(x.ord) == iqr
```

AUSGABE

```
[1] 0.17 -0.47  0.66  0.46 -1.19
0.51  0.44 -0.16 -0.19 -1.37

[1] -0.114
[1] 0.5083822
[1] 0.8361864

[1] 0.457544

[1]    0  -10   29   10 -169   13    9    0   -1 -257

[1] -257 -169  -10   -1    0    0    9   10   13   29
[1] 0
[1] 14.826

     0%     25%     50%     75%    100%
-257.00   -7.75    0.00    9.75   29.00
[1] -257   29

[1] TRUE
```

4.2 Bivariate Zusammenhangsmaße

Je nach Skalenniveau stehen auch hier verschiedene Funktionen zur Verfügung.

cov(), cor(), spearman(), polychor()

cov()	Kovarianz
cor()	Produkt-Moment-Korrelationskoeffizient
spearman()	Spearman-Rho
polycor::polychor()	Polychorische Korrelation

polychor() berechnet die polychorische Korrelation anhand der Kontingenztafeln unter der Annahme, dass den beiden manifesten Variablen latente, bivariat normalverteilte Variablen zugrunde liegen. Diese Funktion entstammt dem Paket *polycor*.

cov() und *cor()* verarbeiten nicht nur einfache Vektoren, sondern nehmen auch Matrizen und Data-Frames auf. In diesem Fall geben sie eine Kovarianz- bzw. Korrelationsmatrix zurück. Die Anwendung von Funktionen auf Matrizen, Data-Frames und Listen wird im folgenden Abschnitt genauer behandelt.

rmvnorm(), mvtnorm()

Im unteren Beispiel werden mit der Funktion *rmvnorm(n,mean,sigma)* aus *k* multivariat normalverteilten Variablen *n* Fälle gezogen und in einer *n x k* Matrix zurückgegeben. Hierbei bestimmt sich *k* aus der Länge des Mittelwertevektors *mean*. Mit der symmetrischen Matrix *sigma* werden die Varianzen und Kovarianzen der Variablen in der Grundgesamtheit festgelegt.

library()

Die Funktion *rmvnorm()* ist im Paket *mvtnorm* definiert. Um die Funktion nutzen zu können, muss das Paket durch den *library()* Befehl zunächst geladen werden.

EINGABE

```
# Laden des Paketes mvtnorm
library(mvtnorm)

# Die Varianz der beiden Variablen ist jeweils 1 die Kovarianz
# (in diesem Fall = Korrelation) beträgt 0.3
S <- matrix(c(1,0.3,0.3,1),2,2)
S
zwei.korrelierte.v <- rmvnorm(10,mean=c(0,0),sigma=S)
zwei.korrelierte.v

cor(zwei.korrelierte.v)
```

AUSGABE

```
     [,1] [,2]
[1,]  1.0  0.3
[2,]  0.3  1.0

                [,1]       [,2]
 [1,]  2.20219180  0.6457782
 [2,] -0.17313329  2.1990490
 [3,]  0.73325348  1.8681796
 [4,]  0.02501693  0.5384568
 [5,]  0.11126044  2.1530846
 [6,]  0.86635714  1.7165953
 [7,] -0.20204287  0.6022951
 [8,]  0.89687825  0.8838052
 [9,]  0.64108152  0.8060353
[10,] -0.87239332 -1.2598818

         [,1]     [,2]
[1,] 1.000000 0.272199
[2,] 0.272199 1.000000
```

Bei mehreren Variablen wird die Eingabe der Kovarianzmatrix *sigma* schnell unübersichtlich. Hierfür lässt sich mit der Funktion *edit()* ein simpler Dateneditor öffnen, in dem die Daten im Tabellenformat bearbeitet werden können. *edit* gibt hierbei die (bearbeitete) Kopie zurück. Dabei werden ebenfalls automatisch Spaltennamen hinzugefügt.

edit()

Die Funktion *rmvnorm()* würde das Vorhandensein von Spaltennamen gegenüber dem Fehlen von Zeilennamen als Hinweis werten, dass dies keine symmetrische Matrix sei. Deswegen werden die Spaltennamen im folgenden Beispiel nach der Bearbeitung durch *edit()* auf *NULL* gesetzt.

EINGABE

```
S <- matrix(0,3,3) # symmetrische Matrix mit 9 Nullen

S <- edit(S)       # Öffnen des Editors
S
colnames(S) <- NULL
```

4.3 Anwendung von Funktionen auf Matrizen, Data-Frames und Listen

Alle oben genannten Funktionen sind generische Funktionen, d.h. sie können unterschiedliche Datentypen aufnehmen, liefern aber auch unterschiedlichen Output. Alle Funktionen nehmen numerische Vektoren und Matrizen als Argumente auf. Einige, wie z.B. *mean*, verarbeiten auch Data-Frames. Boolsche (logische) Vektoren werden als numerische Vektoren interpretiert.
Sollen Funktionen auf Matrizen, Data-Frames oder Listen angewandt werden, ergeben sich an manchen Stellen Probleme, da die generischen Funktionen die Intention des Users zu erahnen versuchen und dies nicht immer zu den gewünschten Ergebnissen führt.

EINGABE

```
1  # Illustration des Problems generischer Funktionen
2
3  # drei Variablen a,b,c mit jeweils 10 Zufallszahlen
4  dat <- data.frame(a=rnorm(10),b=rnorm(10),c=rnorm(10))
5  dat <- round(dat,2)
6  dat
7
8  mean(dat)              # Arithmetisches Mittel von a,b und c
9  mean(as.matrix(dat)) # Arithmetisches Mittel von c(a,b,c)
```

AUSGABE

```
1          a      b      c
2  1    0.75  -0.52  -1.16
3  2   -0.78   0.67   0.00
4  3    0.64   0.51  -1.22
5  4    0.69   1.96  -0.95
6  5    0.24  -0.44  -0.13
7  6   -1.85   1.18  -0.05
8  7    0.47   0.35  -0.71
9  8   -0.49  -0.47   0.00
10 9    0.59  -1.09  -0.66
11 10  -0.59   0.26   0.17
12
13      a       b       c
14 -0.033   0.241  -0.471
15
16 [1] -0.08766667
```

Die Funktionen *apply()* und *lapply()* ermöglichen mehr Kontrolle darüber, wie die entsprechenden Funktionen auf die Daten angewendet werden. Die Funktion *apply(X,MARGIN,FUN)* wendet die Funktion *FUN* auf die in *MARGIN* angegebene Dimension (1 = Zeilen, 2 = Spalten, *c(1,2)* = Spalten und Zeilen) von *X* an. Die Funktion *lapply(X,FUN)* wendet die Funktion *FUN* auf die Liste *X* an.

apply(), lapply()

EINGABE

```
1  apply(dat,2,var) # Varianz der Spalten
2  apply(dat,1,sum) # Summe der Zeilen
3
4  liste<-as.list(dat)
5  lapply(liste,range)
```

AUSGABE

```
1           a          b          c
2  0.7425567 0.8250322 0.2784544
3
4  [1] -0.93 -0.11 -0.07  1.70 -0.33 -0.72
5  [7]  0.11 -0.96 -1.16 -0.16
6
7  $a
8  [1] -1.85  0.75
9
10 $b
11 [1] -1.09  1.96
12
13 $c
14 [1] -1.22  0.17
```

tapply()

Sollen die Funktionen nicht auf den gesamten Datensatz sondern auf durch Faktoren spezifizierte Subgruppen angewendet werden, lässt sich dies über die Funktion *tapply(X,INDEX,FUN)* realisieren. Hierbei wird die Funktion *FUN* auf die verschiedenen Kombinationen der durch *INDEX* angegeben Faktoren auf den Vektor *X* angewendet. *INDEX* muss hierbei als eine Liste und *X* als Table übergeben werden.

Im folgenden Beispiel werden zunächst wieder Leistungswerte für Frauen und Männer (Faktor 1) und Brillenträger und Nicht-Brillenträger (Faktor 2) generiert und daraufhin deskriptive Maße der einzelnen Subgruppen berechnet.

EINGABE

```
1  geschlecht <- sample(c("m","w"),20,replace=TRUE)
2  geschlecht <- as.factor(geschlecht)
3
4  # es werden 40% Brillenträger in der Population angenommen:
5  brille <- sample(c("b","kb"),20,replace=TRUE,prob=c(0.4,0.6))
6  brille <- as.factor(brille)
7
8  # die Besetzung der Zellen
9  table(brille,geschlecht)
10
11 leistung <- rnorm(20,10,2) # AM=10, s=2
12
13 mean(leistung)
14
15 # "Haupteffekt": Brille
16 tapply(leistung,brille,mean)
17
18 # "Haupteffekt": Geschlecht
19 tapply(leistung,geschlecht,mean)
20
21 tapply(leistung,list(brille,geschlecht),mean)
```

AUSGABE

```
1          geschlecht
2   brille männlich weiblich
3       b         1        5
4       kb       10        4
5
6   [1] 9.6185
7
8          b       kb
9   9.979513 9.463781
10
11   männlich  weiblich
12   9.156775 10.182833
13
14      männlich   weiblich
15  b  12.346698  9.506077
16  kb  8.837782 11.028777
```

4.4 Die summary() und describe() Funktionen

Für einen schnellen ersten Überblick über größere Data-Frames oder Matrizen bietet sich die Funktion *summary()* an. *summary()* ist eine generische Funktion und findet sich in verschiedenen Paketen und statistischen Analysen. Je nach übergebenen Datentyp wird dann die entsprechende Routine ausgeführt. Sie liefert für Data-Frames und Matrizen Maße der zentralen Tendenz (Median, Mittelwert) und der Dispersion (Quantile, Range).

summary()

Für die psychologische Statistik besser geeignet ist die Funktion *describe()* des *psych* Paketes. Sie liefert unter anderem Maße der zentralen Tendenz, Dispersion und Maße der Verteilungsform (Kurtosis, Schiefe) in einem Data-Frame.

describe()

Im folgenden Beispiel wird zunächst die Funktion *summary()* angewandt. Für die Benutzung der *describe()* Funktion wird zunächst das *psych* Paket geladen. Die Ergebnisse werden in der Variablen *res* gespeichert. Durch den *$* Operator kann auf die Spalten des Data-Frames zugegriffen werden (S. 12, Abschnitt 2.2.1). Die Ergebnisse werden dann mit dem Befehl *write.csv2()* in die Datei „C:\ergebnisse.csv“ exportiert und können dann mit mit anderen Programmen wie zum Beispiel Excel geöffnet werden.

write.csv2()

EINGABE

```
1  summary(zwei.korrelierte.v)
2
3  library(psych) # Laden des Paktes psych
4  res <- describe(zwei.korrelierte.v)
5  res
6  res$mean        # nur die Mittelwerte
7
8  write.csv2(res, file = "c:/ergebnisse.csv", quote=FALSE)
```

Das *psych* Paket bietet neben dieser Funktion weitere nützliche Funktionen für psychologische Fragestellungen. Es ist deshalb zu empfehlen dieses Paket zu installieren und sich die Funktionsliste der Hilfedatei näher anzuschauen.

Kapitel 5
Grafische Darstellung (High-Level Plots)

Für die visuelle Darstellung von Daten und Ergebnissen steht eine sehr große Auswahl an Funktionen und Paketen zur Verfügung. In R unterscheidet man zwischen High-Level Plots, welche vollständige Abbildungen liefern und dem Low-Level Plotten, bei dem die einzelnen Elemente einer Grafik Stück für Stück zusammengesetzt werden. Hierdurch lassen sich flexiblere und anspruchsvollere Grafiken erstellen.

Im Folgenden wird zunächst nur auf das High-Level Plotten eingegangen.

plot()	generische Funktion
hist()	Histogramm
barplot()	Balkendiagramm
boxplot()	Box-Whisker Plot
curve()	Funktionskurven
pairs()	Matrix von Streudiagrammen
qqnorm()	Quantile-Quantile-Plot
psych::error.bars()	Mittelwert und Konfidenzintervall

Für die folgenden Beispiele werden die Daten zu Leistung, Geschlecht und Brille neu generiert. Diesmal mit einer größeren Stichprobe:

EINGABE
```
geschlecht <- sample(c("m","w"),200,replace=TRUE)
geschlecht <- as.factor(geschlecht)

brille <- sample(c("b","kb"),200, replace=TRUE,prob=c(0.4,0.6))
brille <- as.factor(brille)

leistung <- rnorm(200,10,2)
```

plot(), hist(), boxplot(), barplot()

Die Funktion *plot()* stellt die am häufigsten verwendete Plotfunktion in R dar, da diese wiederum eine generische Funktion ist und von den verschiedensten Paketen und statistischen Verfahren zur Darstellung der Ergebnisse verwendet wird. Wird *plot()* ein numerischer Vektor übergeben, wird dieser entlang seines Indexes geplottet. Bei zwei numerischen Vektoren wird ein bivariates Streudiagramm geplottet.

EINGABE
```
plot(leistung) # nicht sehr sinnvoll

# tausend Fälle in zwei Variablen, die zu 0.8 korreliert sind
zwei.hoch.kor.v <- rmvnorm(1000,mean=c(0,0),
           sigma=matrix(c(1,0.8,0.8,1),2,2))

plot(zwei.hoch.kor.v) # bivariates Streudiagramm
```

Zur Betrachtung von Verteilungen bietet sich bei stetigen Verteilungen die Funktionen *hist()* und *boxplot()*, bei diskreten Verteilungen (z.B. Faktoren) *barplot()* an.

Im folgenden Beispiel wird mit dem Befehl *par()* die Darstellungsform der Anzeige verändert. *mfrow=c(2,2)* bedeutet, dass Diagramme in einer *2x2* Matrix angezeigt werden. Mehr zur Bedeutung und Anwendung von *par()* wird der Abschnitt zum Low-Level Plotten erläutern.
Mit den Argumenten *main, xlab* und *ylab* lassen sich die Diagramm- und Achsenbeschriftung des Plots bestimmen. Die Argumente *xlim* und *ylim* erwarten einen Vektor der Länge zwei, der den Achsenbereich der x- bzw. y-Achse des Plots festlegt. Der ~ Operator ist ein Formeloperator. Genaueres zu Formeln folgt in Abschnitt 5 (Lineare Modelle).

par(), main, xlab, ylab, xlim, ylim, ~

EINGABE

```
1   # Darstellung stetiger Merkmale
2   hist(leistung)
3
4   # Darstellung getrennt nach Geschlecht
5   par(mfrow=c(2,2))
6   hist(leistung[geschlecht=="w"], main = "w",
7                 xlab="Leistung", xlim=c(5,15))
8   hist(leistung[geschlecht=="m"], main = "m",
9                 xlab="Leistung", xlim=c(5,15))
10
11  # oder schneller, aber „unschöner“
12  tapply(leistung, geschlecht, hist)
13
14  # Boxplots
15  par(mfrow=c(1,1))
16  boxplot(leistung~geschlecht)
17
18  # Darstellung diskreter Merkmale
19  table(brille,geschlecht)
20  barplot(table(brille,geschlecht))
```

Mit der Funktion *curve()* lassen sich Funktionen plotten. Dies ist zum Beispiel hilfreich, um eine theoretische Verteilungsform über eine empirische Verteilungsform zu plotten. Das logische Argument *freq* bestimmt im folgenden Beispiel, ob im Histogramm Häufigkeiten (Frequencies) oder Wahrscheinlichkeiten angezeigt werden sollen. Mit dem logischen Argument *add* wird festgelegt, ob die Funktionskurve in einem neuen Plot oder zu einem bestehenden Plot hinzugefügt wird.

curve(), freq, add

EINGABE

```
1   # irgendeine Parabel
2   curve(2*x^2)
3
4   # die Sinuskurve
5   curve(sin, xlim=c(-4,4))
6
7   # Normalverteilungskurve nur die erste Spalte der 1000 x 2 Matrix
8   v <- zwei.hoch.kor.v[,1]
9
10  hist(v, freq = FALSE)
11
12  curve(dnorm, add=TRUE)
```

Sehr hilfreich bei explorativen Datenanalysen von größeren Datensätzen ist neben der bereits beschriebenen *describe()* Funktion die Funktion *pairs()* bzw. die Funktion *pairs.panels()* des *psych* Paketes.

pairs(), pairs.panels()

Im folgenden Beispiel werden drei korrelierte Variablen zu je 1000 Fällen mit der bekannten Funktion *rmvnorm* erzeugt, zusätzlich wird eine schiefverteilte Variable zu 1000 Fällen mit dem Paket *sn* (Skew-Normal Distribution) über die Funktion *rsn* erzeugt. Die Funktion *rsn(n, location, scale, shape)* gibt *n* Zufallszahlen aus einer Verteilung mit den Verteilungsparametern *location*, *scale* und *shape* zurück. Wird *shape* auf Null gesetzt, entstammen die Zufallszahlen einer gewöhnlichen Normalverteilung. Genaueres zur Skew-Normal Distribution kann der Hilfefunktion und den dort angegeben Quellen entnommen werden. Die vier so generierten Variablen werden dann in einem Data-Frame verbunden und mit *describe()*, *pairs* und *pairs.panels* untersucht.

sn, rsn()

EINGABE

```
1  library(sn)
2
3  # Die Kovarianzmatrix der drei Variablen
4  S <- matrix(c(1,0.5,0.9,0.5,1,0.8,0.9,0.8,1),3,3)
5
6  dat.mvn <- rmvnorm(1000,rep(0,3),S)
7
8  # Die schiefe Variable
9  dat.skew <- rsn(1000,0,1,-5)
10
11 daten <- data.frame(dat.mvn,dat.skew)
12
13 pairs(daten)
14
15 pairs.panels(daten)
```

Die Funktion *error.bars()* des *psych* Paketes nimmt Matrizen oder Data-Frames auf und plottet die Mittelwerte der Spalten (Variablen) mit den entsprechenden Standardfehlerbalken. Die Funktion *error.bars.by(x, group)* des *psych* Paketes nimmt einen Vektor *x* auf und plottet Mittelwerte und Standardfehlerbalken für die verschiedenen Stufen der Variablen *group*.

error.bars()

error.bars.by()

Das im folgenden Beispiel verwendete Argument *by.var* ist ein funktionsspezifisches Argument der Funktion *error.bars.by()* und legt fest, ob die Mittelwerte auf unterschiedlichen Stellen in der x-Achse geplottet werden sollen.

by.var

EINGABE

```
1  # Die vier Zufallsvariablen
2  error.bars(daten)
3
4  # Männer gegenüber Frauen
5  error.bars.by(leistung, geschlecht, by.var=TRUE)
```

Abweichungen einer empirischen Verteilung von der Normalverteilung lassen sich grafisch über den Q-Q-Normal-Plot (Funktion *qqnorm()*) erkennen. Hierbei werden die Quantile der empirischen Verteilung gegenüber den Quantilen einer Normalverteilung in einem Streudiagramm abgetragen. Ist die empirische Verteilung normalverteilt, liegen die Punkte des Streudiagramms auf einer Geraden.

qqnorm()

EINGABE

```
par(mfrow=c(2,2)) # Setzen der Anzeige

hist(dat.mvn[,1]) # nur eine Variable (erste Spalte)
qqnorm(dat.mvn[,1])

hist(dat.skew)
qqnorm(dat.skew)
```

5.1 Speichern von Grafiken

Beim Erstellen von Grafiken stellt sich für R die Frage auf welchem Gerät (Device) diese Grafik ausgegeben werden soll. Standardmäßig werden Grafiken unter Windows in einem zusätzlichen Fenster angezeigt (*windows()*). Wird ein anderes Gerät gewählt, wird die erstellte Grafik dort ausgegeben. Andere Geräte können zum Beispiel sein:

pdf(), postscript(), bmp(), pictex(), jpeg(), win.print(), windows()

pdf()	PDF-Dokument
postscript()	Postscript-Datei
bmp()	Bitmap-Format
pictex()	für Latex-Dokumente
jpeg()	JPEG-Format
win.print()	Druckerausgabe
windows()	Standardausgabe in R

Soll eine Grafik etwa im PDF-Format gespeichert werden, so wird mit dem Befehl *pdf(file)* eine Datei mit Pfad und Name *file* im PDF-Format erstellt und diese als aktuelles Device geöffnet. Im Folgenden kann die Grafik anhand der üblichen Plot-Befehle erstellt werden. Durch den Befehl *dev.off()* wird das Device geschlossen und die Datei damit abgespeichert.

dev.off()

EINGABE

```
# Öffnen des Device
pdf("c:/Scatterplot.pdf")

# Erstellen der Grafik
plot(zwei.hoch.kor.v, main = "Streudiagramm", xlab = "x", ylab="y")

# Schließen des Device
dev.off()
```

Übungen

1. Verdeutliche die Erwartungstreue des Mittelwertes anhand eines *error.bars.by()* Diagramms. Generiere hierfür mehrere Stichproben unterschiedlicher Größe und beobachte den Standardfehler des Mittelwertes.

2. Korrelationen sind sehr leicht verfälschbar durch Ausreißer in den Daten. Veranschauliche dir die Problematik anhand selbst generierter Daten. Generiere Daten aus zwei korrelierten Variablen mit der Funktion *rmvnorm()*.

   a) Berechne die Korrelation zwischen den Variablen und plotte das bivariate Streudiagramm.

   b) Füge über die Funktion *rbind()* den Daten Ausreißer hinzu.

   c) Berechne die neue Korrelation und plotte erneut das Streudiagramm.

   d) Versuche eine möglichst große Verfälschung der wahren Korrelation zu erreichen.

3. In den Beispielen zu Leistungsunterschieden bei dem Faktor Geschlecht und dem Faktor Brille wurden die Daten bisher ohne Effekte generiert, d.h. die Gruppenmittelwerte unterscheiden sich nicht voneinander.

   a) Generiere Daten, welche Gruppeneffekte aufweisen.

   b) Wähle eine geeignete Darstellungsform (Standardfehler).

   c) Mache dir Gedanken zur Power. Wie ist der Zusammenhang zwischen Standardfehler, Stichprobengröße und Signifikanz?

   d) Stelle dir vor die Daten wurden in unterschiedlichen Schulen erhoben. Erstelle einen Faktor, der die Schulen (bei $n=200$ z.B. $m=8$ Schulen) kodiert.

   e) Stelle dir weiterhin vor, in den Schulen würde ein unterschiedliches Leistungsniveau herrschen. Generiere diese Unterschiede im Leistungsniveau und addiere sie zu den bisher generierten Daten.

   f) Beantworte für dich die Frage, was die Begriffe Fixed Effects und Random Effects an dieser Stelle bedeuten.

Kapitel 6

Bivariate Inferenzstatistik und Verteilungen

In dem Paket *stats*, das zur Basisinstallation von R gehört, sind viele Funktionen zur inferenzstatistischen Testungen von Unterschiedshypothesen zwischen zwei Stichproben implementiert. Hier die Funktionsnamen der wichtigsten Tests zur inferenzstatistischen Testung von Unterschiedshypothesen zwischen zwei Stichproben:

t.test()	Student t-Test
wilcox.test()	Wilcoxon Test
binom.test()	Binomial-Test
var.test()	Fisher's F-Test

Mithilfe des Arguments *paired* lässt sich bei der t-Test- und Wilcoxon-Test Funktion einstellen, ob es sich bei den Stichproben um abhängige oder unabhängige Stichproben handelt. Der nonparametrische Test bei unabhängigen Stichproben ist auch als Mann-Whitney U-Test bekannt. Wird nur ein Vektor der Funktion übergeben, so wird ein Einstichproben t-Test bzw. Wilcoxon-Test gerechnet.

t.test(), wilcox.test(), paired, alternative

Das Argument *alternative* dient dazu, die H1-Hypothese zu definieren und damit einhergehend einseitig oder zweiseitig zu testen. *alternative* kann drei Eingaben verarbeiten: *„less“*, *„greater“* oder *„two.sided“* (standardmäßig eingestellt).

Im unteren Code-Beispiel werden zwei Stichproben erzeugt, die voneinander abhängig sind. Anschließend wird ein einseitiger t-Test für abhängige Stichproben gerechnet.

EINGABE

```
1  library(mvtnorm) # Lädt das Paket mvtnorm für die Funktion rmvnorm
2
3  kovarianz <- matrix(c(5, 0.6, 0.6, 5), nrow = 2, ncol = 2)
4                      # Kovarianzmatrix der beiden Variablen
5
6  means <- c(5, 7) # Mittelwerte der zwei Variablen
7
8  data <- rmvnorm(50, means, sigma = kovarianz)
9                      # Datenerstellung
10
11 result <- t.test(data[,1], data[,2],
12                     paired=TRUE, alternative="less")
13
14 result              # Ausgabe der Ergebnisse
```

AUSGABE
```
1    Paired t-test
2
3  data:  data[, 1] and data[, 2]
4  t = -2.1365, df = 9, p-value = 0.03068
5  alternative hypothesis: true difference in
6  means is less than 0
7  95 percent confidence interval:
8          -Inf -0.3544248
9  sample estimates:
10 mean of the differences
11                -2.495834
```

Die Ausgabe zeigt die wichtigsten Ergebnisse wie Freiheitsgrade (df), t-Wert, p-Wert und Konfidenzintervall. Der Satz in der fünften Zeile fasst das Ergebnis zusammen, sofern der p-Wert unter der akzeptierten Irrtumswahrscheinlichkeit liegt.

Auf die Einzelergebnisse eines t-Tests kann durch den $-Operator zugegriffen werden. Dies kann dann sinnvoll sein, wenn man mit einzelnen Werten in weiteren Arbeitsschritten weiter rechnen will oder die Ergebnisse anders aufbereiten will.
Mit der Funktion *str()* (für structure) lassen sich die Bezeichnung, der Klassentypus und der gespeicherte Wert der einzelnen Elemente eines Objekts auflisten:

EINGABE
```
1  str(result)
2
3  result$p.value     # gibt den errechneten p-Wert zurück
4  result$statistic   # gibt den errechneten t-Wert zurück
```

AUSGABE
```
1  List of 9
2   $ statistic  : Named num -2.24
3    ..- attr(*, "names")= chr "t"
4   $ parameter  : Named num 9
5    ..- attr(*, "names")= chr "df"
6   $ p.value    : num 0.0259
7   $ conf.int   : atomic [1:2] -Inf -0.252
8    ..- attr(*, "conf.level")= num 0.95
9   $ estimate   : Named num -1.38
10   ..- attr(*, "names")= chr "mean of the differences"
11  $ null.value : Named num 0
12   ..- attr(*, "names")= chr "difference in means"
13  $ alternative: chr "less"
14  $ method     : chr "Paired t-test"
15  $ data.name  : chr "data[, 1] and data[, 2]"
16  - attr(*, "class")= chr "htest"
17
18
19  [1] 0.02585283
20
21         t
22 -2.241627
```

Mithilfe des Binomialtests lässt sich prüfen, ob ein Ereignis in einer Stichprobe überzufällig oft auftrat (diskrete Variable: ja/nein). Die Funktion *binom.test()* benötigt mindestens ein Argument:

binom.test()

Ein Vektor mit der Anzahl des Eintretens des Ereignisses und der Anzahl des Ausbleibens des Ereignisses. Alternativ kann man auch die Anzahl des Eintretens des Ereignisses und die Anzahl n der Messungen angeben. Darüber hinaus kann auch beim Binomialtest das optionale Argument *alternative* angegeben werden.
Dem entsprechend erzeugen folgende beiden Codezeilen das selbe Ergebnis:

EINGABE
```
1 binom.test(c(5, 15))
2 binom.test(5, 20)
```

AUSGABE
```
1  # Hier nur eine Kopie der erzeugten Ausgabe:
2
3   Exact binomial test
4
5 data:  c(5, 15)
6 number of successes = 5, number of trials = 20,
7 p-value = 0.04139
8
9 alternative hypothesis: true probability of success is
10                         not equal to 0.5
11
12 95 percent confidence interval:
13  0.08657147 0.49104587
14 sample estimates:
15 probability of success
16                   0.25
```

Die Funktion geht standardmäßig davon aus, dass Ereignis- und Gegenereignis jeweils eine Wahrscheinlichkeit von p = 0.5 hat. Mithilfe des Arguments *p* lässt sich dies ändern.

6.1 Verteilungen in R

Dem Anwender stehen in R verschiedene Verteilungen zur Verfügung. Zu jeder Verteilung existiert die Dichtefunktion, die Verteilungsfunktion, eine Funktion zur Berechnung von Quantilen und eine Funktion zur Erzeugung von Zufallszahlen. Diese Funktionen sind durch die vorangestellten Buchstaben *d* (density), *p* (probability = Verteilungsfunktion), *q* (quantile) und *r* (random) gekennzeichnet. Für die Normalverteilung gibt es also die Funktionen; *dnorm()*, *pnorm()*, *qnorm* und *rnorm()*.

6.1.1 Funktionen der Verteilungen

arrows(), lines()

Der Unterschied zwischen Dichtefunktion, Verteilungsfunktion, Quantilfunktion und der Funktion zur Generierung von Zufallszahlen soll anhand der Normalverteilung im Folgenden erläutert werden. Die dabei verwendeten Funktion *arrows()* und *lines()* fügen dem Plot Pfeile und Linien hinzu.

dnorm()

Dichtefunktion (d) Aus der Dichtefunktion lässt sich beispielsweise erkennen, welche Werte bei einer Ziehung aus dieser Verteilung am wahrscheinlichsten sind. Aus der in R geplotteten Dichtefunktion der Standardnormalverteilung lässt sich erkennen, dass Werte um 0 mit der größten Wahrscheinlichkeit und extreme Werte mit kleinerer Wahrscheinlichkeit bei einer Ziehung auftreten würden:

EINGABE

```
1 par(mfrow=c(2,1))
2 curve(dnorm(x), xlim = c(-4,4))
```

Verteilungsfunktion (p) Die Verteilungsfunktion $\Phi(x)$ wird auch häufig als kumulierte Verteilungsfunktion bezeichnet. Sie besagt, mit welcher Wahrscheinlichkeit die Dichtefunktion einen Wert von kleiner gleich x annimmt (mathematisch betrachtet ist sie im Fall der Normalverteilung das Integral der Dichtefunktion von $-\infty$ bis x).

pnorm()

EINGABE

```
1 curve(pnorm(x), xlim = c(-4,4))
```

Für den Wert $x = 0$ gibt die Funktion den Wert 0.5 aus. Dies bedeutet, dass die Wahrscheinlichkeit eines Wertes im Bereich von $-\infty$ bis 0 genau 50% beträgt.

EINGABE

```
1 curve(dnorm(x), xlim=c(-4,4))
2 curve(dnorm(x), from=-4, to=0, xlim=c(-4,4), n=50, type="h", add=T)
3 curve(pnorm(x), xlim = c(-4,4))
4
5 arrows(-0.5, 0.6, 0, pnorm(0), xpd=1, length=0.1, col="red")
6 lines(c(0, 0), c(0, pnorm(0)), lty=2)
```

Für den Wert $x = 1.69$ ergibt sich eine Wahrscheinlichkeit von ca. 95% (dies entspricht dem kritischen z-Wert bei einseitiger Testung).

EINGABE

```
1 curve(dnorm(x), xlim=c(-4,4))
2 curve(dnorm(x), from=-4, to=1.69, xlim=c(-4,4), n=70, type="h",
3       add=T)
4 curve(pnorm(x), xlim = c(-4,4))
5
6 arrows(1.9, 0.72, 1.69, pnorm(1.69), xpd=1,length=0.1, col="red")
7 lines(c(1.69, 1.69), c(0, pnorm(1.69)), lty=2)
```

Quantilfunktion (q) Die Quantilfunktion gibt die Quantile der Verteilungsfunktion zurück. Sie ist im Prinzip die invertierte Verteilungsfunktion. Während man bei der Verteilungsfunktion einen z-Wert eingibt und eine Wahrscheinlichkeit erhält, gibt man in die Quantilfunktion Wahrscheinlichkeiten ein und erhält z-Werte. Plottet man die Funktion *qnorm()*, erkennt man, dass sie die gleiche Form wie *pnorm()* besitzt. Das Koordinatensystem ist lediglich gespiegelt und um 90° gekippt.

qnorm()

EINGABE

```
1 par(mfrow=c(2,1))
2 curve(pnorm(x), xlim=c(-4,4), ylim=c(0,1))
3 curve(qnorm(x), xlim=c(0,1), ylim=c(-4,4))
```

Für die in der Verteilungsfunktion markierten Wahrscheinlichkeitswerte lassen sich dementsprechend die z-Werte erlangen.

EINGABE

```
1  qnorm(0.5)  # Für welchen Wert x ist die Wahrscheinlichkeit, dass ein Wert
2              # <= x auftritt genau 50%?
3
4  qnorm(0.95)  # Für welchen Wert x ist die Wahrscheinlichkeit, dass ein Wert
5               # <= x auftritt genau 95%?
6
7  qnorm(c(0.025,0.975))  # Die Grenzen bei zweiseitigem Testen
```

Zufallszahlengenerator (r) Die Funktion zur Generierung von Zufallszahlen *rnorm()* generiert Zufallszahlen aus der Dichtfunktion *pnorm()*.

rnorm()

EINGABE

```
1  hist(rnorm(1000))
```

6.1.2 Weitere Verteilungen in R

Neben der Normalverteilung existieren in R eine ganze Reihe weiterer Verteilungen. Im Folgenden seien einige von ihnen genannt:

norm	Normalverteilung
binom	Binomialverteilung
chisq	χ^2-Verteilung
f	F-Verteilung
t	t-Verteilung
pois	Poisson-Verteilung
unif	Gleichverteilung
wilcox	Wilcoxon-Verteilung
geom	geometrische Verteilung

Kapitel 7

Kontrollstrukturen in R

In allen Programmiersprachen findet man so genannte Kontrollstrukturen, mit deren Hilfe ein Programm flexibel auf verschiedene Zustände reagieren kann. So kann beispielsweise bei Bedingung x1 der Algorithmus y1 ausgeführt werden, bei Bedingung x2 dagegen der Algorithmus y2.

7.1 Wenn-Dann Beziehungen

Wenn-Dann Beziehungen kommen in R eher selten zum Einsatz, da auch implizite Wenn-Dann Beziehungen über die *[ ]*-Indizierung realisiert werden können.

EINGABE

```
1  x <- sample(1:6, 1)  # ein Würfelwurf
2
3  if(x == 6) { print("Sie haben eine sechs gewürfelt!") }
4
5
6  # Alternative mit der []-Indizierung:
7  x[x == 6] <- "Sie haben eine sechs gewürfelt"
8  x
```

In den Klammern nach der if-Anweisung befindet sich die Bedingung. Ist diese erfüllt, so wird der Code innerhalb der geschweiften Klammern ausgeführt. Ansonsten wird alles innerhalb der Klammern schlicht ignoriert.

Wenn-Dann Anweisungen können erweitert werden in Anweisungen die man als „Wenn-Dann-Ansonsten-Anweisungen“ bezeichnen könnte:

EINGABE

```
1  x <- sample(1:6, 1)  # ein Würfelwurf
2
3  if(x == 6) { print("Sie haben eine sechs gewürfelt!")
4  } else {  print("leider keine sechs.")  }
```

Der Code nach der else-Anweisung wird immer dann ausgeführt, wenn die oberhalb definierte if-Bedingung nicht erfüllt ist.
In den meisten Fällen ist es für R irrelevant, ob sich zwischen den verschiedenen Elementen Leerzeichen oder Zeilenumbrüche befinden. Insofern kann man seinen Code so strukturieren, wie man es für sich am übersichtlichsten empfindet. Bei der else-Anweisungen dagegen wird ein Fehler erzeugt, wenn vor dem else nicht die geschweifte Endklammer der oben stehenden if-Bedingungen steht.

Wenn-Dann-Ansonsten Anweisungen lassen sich um noch einen weiteren Schritt ausbauen, in dem man if und else zu else-if kombiniert:

EINGABE

```
x <- sample(1:6, 1)   # ein Würfelwurf

if(x == 6) { print("Sie haben eine sechs gewürfelt!")
} else if(x == 5) {  print("Immerhin eine fünf.")
} else if(x == 4) {  print("Noch eine vier.")
} else if(x == 3) {  print("Leider nur eine drei.")
} else {  print("Lieber noch mal würfeln.")  }
```

Dadurch können hierarchisch mehrere Bedingungszustände nacheinander abgefragt und entsprechend darauf reagiert werden. Im oberen Beispiel werden fünf der sechs möglichen Zustände auf unterschiedliche Art verarbeitet.

7.2 Schleifen

Während Wenn-Dann Anweisungen eher selten in R-Skripten zum Einsatz kommen, können Schleifen an manchen Stellen durchaus sehr nützlich sein.
Mithilfe von Schleifen kann man die Bearbeitung von einem festgelegtem Code-Abschnitt nach einer definierten Regel wiederholen lassen. Damit lassen sich beispielsweise Auswertungsroutinen realisieren, also die Anwendung von Berechnungen auf mehrere Datensätze oder Variablen. An dieser Stelle sollen die zwei Schleifenanweisungen *for()* und *while()* näher betrachtet werden. Zunächst ein Beispiel für eine *for()*-Schleife:

EINGABE

```
letters   # letters ist eine Konstante, die in R eingebaut ist. Die Variable
          # enthält alle Buchstaben des Alphabets als Vektor

for(i in 1:26)  {
                 print(letters[i])
                }


for(j in 26:1)  {
                 print(letters[j])
                }
```

AUSGABE

```
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m"
 [14] "n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"


[1] "a"
[1] "b"
[1] "c"
# usw. ...

[1] "z"
[1] "y"
[1] "x"
# usw. ...
```

In den Klammern nach der for-Anweisung steht die for-Definition. Diese Definition enthält zunächst eine Zählvariable, die hier mit i und j benannt wurde, aber auch jede andere für Variablen zulässige Bezeichnung tragen kann.

Die Zählvariable nimmt zunächst den ersten Wert an, der im zweiten Teil der Definition angegeben wird. Im oberen Beispiel wäre das für die erste Schleife der Wert 1 und für die zweite Schleife der Wert 26. Wurde der Code einmal ausgeführt, so nimmt i bzw. j den zweiten Wert an. Im oberen Fall wäre dies 2 bzw. 25 für die zweite Schleife. Dies wiederholt sich solange, bis i (bzw. j) den letzten Wert der angegeben Wertereihe angenommen hat.
Die Wertereihe, welche die Zählvariable im Laufe der Schleife annimmt, kann aufsteigend oder absteigend sein, aber auch aus einer vollkommen ungeordnete Zahlenreihe bestehen:

EINGABE

```
1  x <- c(8, 1, 12, 12, 15, 0)
2
3  for(i in x)  {
4                if(i == 0) { print(":)")
5               } else { print(letters[i]) }
6               }
```

AUSGABE

```
1  [1] "h"
2  [1] "a"
3  [1] "l"
4  [1] "l"
5  [1] "o"
6  [1] ":)"
```

Die while-Schleife unterscheidet sich von der for-Schleife dadurch, dass hierbei die Definition auf die Endbedingung beschränkt ist und keine Zählvariable enthält:

EINGABE

```
1  i <- 1
2  x <- rnorm(20, 0, 2)
3
4  while(i < 5)  {
5                 print(x[i])
6                 i <- i+1
7                }
```

AUSGABE

```
1  [1] 1.217908
2  [1] 1.305976
3  [1] -0.659858
4  [1] 1.636193
```

Der Code innerhalb der geschweiften Klammern wird also solange ausgeführt, bis i größer/gleich fünf ist. Erst dann bricht die Schleife ab. Die Zählvariable i muss hier also selbst innerhalb der geschweiften Klammern erhöht werden. Dadurch kann man flexibler mit Start- und Endbedingungen arbeiten, allerdings sollte man aufpassen, dass man keine Endlosschleifen erzeugt!

EINGABE

```
1  # Am Besten mal selbst ausprobieren :)
2  i <- 1
3  x <- rnorm(20, 0, 2)
4
5  while(i < 5)  {
6                 print(x[i])
7                }
```

AUSGABE

```
1 # Alt + F4 für Windows
2 # CMD + Q für Mac
3 # oder die Berechnung über das R-Menü abbrechen
```

Die *print()*-Funktion wurde in den oberen Beispielen öfter genutzt. Sie ist wichtig, da die einfache Eingabe von Variablennamen innerhalb einer Schleife oder einer bedingten Anweisung nicht zur Ausgabe des Inhalts der Variablen führt. Mit der *print()*-Funktion muss explizit eine Ausgabe angestoßen werden.

print()

Übungen

1. Welcher Test wird bei folgendem Code gerechnet?

   EINGABE
   ```
   wilcox.test(y1, y2, alternative = "greater")
   ```

2. Welche Möglichkeit gibt es, einen t-Test über ein Data-Frame mit einer Gruppenvariable zu rechnen?

   EINGABE
   ```
   grp <- c(rep("grp2", 5), rep("grp1", 5))
   abh.Var <- c(rnorm(5, 2, 3), rnorm(5, 5, 3))

   daten <- data.frame(as.factor(grp), abh.Var)
   ```

3. In einer Studie soll die Wirkung kurzfristiger Ereignisse auf Traitmessungen untersucht werden. Hierfür werden Versuchspersonen aufgefordert einen Fragebogen zum sozialen Optimismus auszufüllen. In der einen Bedingung begrüßt der Versuchsleiter die Versuchspersonen freundlich und ist während der Testung zuvorkommend. In der anderen Bedingung legt der Versuchsleiter ein ruppiges und in sich gekehrtes Verhalten an den Tag. Die Versuchspersonen werden gleichmäßig den beiden Bedingung per Randomisierung zugeordnet, jeder Versuchsperson wird eine Entschädigung bezahlt. Der Effekt der Instruktionsmanipulation auf das Antwortverhalten hat sich in der Literatur vielfach gezeigt und wird im Mittel mit einer Effektstärke von $d = 0.3$ angegeben.

   a) Einerseits hat die Untersuchung nur gute Aussichten auf Publikation, wenn ein signifikanter Effekt nachgewiesen werden kannst. Andererseits stehen dem Institut nur geringe Mittel zur Verfügung. Simuliere deswegen den Zusammenhang zwischen der Wahrscheinlichkeit einen signifikanten Effekt nachzuweisen und der gewählten Stichprobengröße.

   b) Ab welcher Stichprobengröße ist die Wahrscheinlichkeit eines signifikanten Effekts mindestens 80%?

4. Erstelle eine Schleife, mit der über die Variablen des folgenden Data-Frames deskriptive Statistiken wie Mittelwert, Standardabweichung etc. jeweils getrennt für die Faktorvariable „gruppe“ errechnet werden.

   EINGABE
   ```
   gruppe <- c(rep("grp2", 5), rep("grp1", 5))
   abh.Var1 <- c(rnorm(5, 2, 3), rnorm(5, 5, 3))
   abh.Var2 <- c(rnorm(5, 1, 3), rnorm(5, 5, 3))
   abh.Var3 <- c(rnorm(5, 0, 3), rnorm(5, 5, 3))

   gruppe <- as.factor(gruppe)

   daten <- data.frame(gruppe, abh.Var1, abh.Var2, abh.Var3)
   ```

Kapitel 8
Regressions- und Varianzanalyse

8.1 Modelle in R formulieren

In der Statistik wird oft versucht die Variation der Werte einer so genannten abhängigen Variable durch eine Linearkombination der Werte von unabhängigen Variablen zu erklären.
Im einfachsten Fall besteht die Modellgleichung ausschließlich aus einer abhängigen und einer unabhängigen Variable sowie den Gewichtungsparametern:

$$y = bx + a$$

Da die Messung von y und x meist durch einen Messfehler beeinträchtigt ist und die Varianzaufklärung von y durch x so gut wie nie perfekt ist, lässt sich die obige Formel für die einzelnen gemessenen y-Werte nur näherungsweise lösen. Es kann keine perfekte Linearkombination ausfindig gemacht werden. Stattdessen muss eine Linearkombination von x gefunden werden, bei der, wenn man einen gemessenen x-Wert in die Formel einsetzt, die Abweichung des errechneten y-Wertes vom tatsächlichen y-Wert möglichst gering ist. Für den Steigungsparameter b und die Konstante a müssen demnach Werte gesucht werden, die x so gewichten, dass dem Anspruch eines möglichst genauen Modells mit möglichst geringer Abweichung zur Realität genüge getan wird.

Ein Verfahren, um die Parameter nach diesem Kriterium zu schätzen ist die Methode der kleinsten Quadrate nach dem Mathematiker Carl Friedrich Gauß. Dabei werden Steigungsparameter und Konstante so geschätzt, dass die quadrierte Abweichung der empirisch gemessenen Werte, von der durch das Modell vorhergesagten Werte ein Minimum ergibt.

Mit moderner Statistiksoftware lässt sich diese Schätzung relativ schnell durchführen. Im Gegensatz zu anderen Statistikprogrammen mit grafischer Oberfläche muss in R allerdings zunächst die Modellgleichung ähnlich wie oben aufgestellt werden. So weiß R welche Variablen in das Modell eingehen und welche Parameter geschätzt werden müssen.

Eine Modellgleichung wird in R mit dem so genannten Tilde Symbol ($\sim$) aufgestellt, das in etwa dem Gleichheitszeichen in mathematischer Konnotation entspricht:

$\sim$

EINGABE

```
1  x <- rnorm(30, 0, 2)
2  y <- 2.5 * x + 8 + rnorm(30, 0, 2)
3
4  modell <- y ~ x
5  class(modell)
6
7  lm(modell)
```

AUSGABE

```
[1] "formula"

Call:
lm(formula = modell)

Coefficients:
(Intercept)            x
      8.095        2.097
```

Im oberen Code-Beispiel wurde zunächst eine unabhängige stetige Variable x erzeugt. Die abhängige Variable y setzt sich aus 2,5 mal dem x-Wert zusammen und einer Konstanten mit dem Wert 8. Da sich die y-Werte nicht perfekt durch x erklären lassen sollen (wie das in der Realität meist der Fall ist), wurde eine normalverteilte zufällige Fehlerstreuung über die generierten Werte von y gelegt.

lm()

Die Modellformel wird in Zeile 5 definiert und als Variable *modell* abgespeichert. R erkennt, dass es sich um eine Modellformel handelt und deklariert den Datentypus, der in der Variable abgespeichert wird, als formula-Objekt. Durch die Funktion *lm()* (für linear model) werden nun, indem man der Funktion die Modellgleichung übergibt, die Parameter des linearen Modells geschätzt. Wie man sieht, liegt die Schätzung der Konstanten a (im Output Intercept genannt) sehr nahe an der generierten Konstanten. Der Steigungsparameter b (im Output der Gewichtungsparameter für x) wurde dagegen leicht unterschätzt, was an der geringen Stichprobengröße liegen mag.

Der Output der Funktion *lm()* lässt sich noch erweitern, indem sie durch die Funktion *summary()* eingeklammert wird:

EINGABE

```
summary(lm(modell))
```

AUSGABE

```
Call:
lm(formula = modell)

Residuals:
    Min      1Q  Median      3Q     Max
-3.4764 -1.3724  0.1800  1.1778  2.9118

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   8.0946     0.3017   26.83  < 2e-16 ***
x             2.0970     0.1584   13.24 1.42e-13 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1

Residual standard error: 1.649 on 28 degrees of freedom
Multiple R-squared: 0.8622, Adjusted R-squared: 0.8573
F-statistic: 175.2 on 1 and 28 DF,  p-value: 1.421e-13
```

Der Output listet nun nicht nur die nach der Methode der kleinsten Quadrate geschätzten Parameter der Modellgleichung auf, sondern auch deren Standardfehler sowie die Prüfung, ob der geschätzte Parameter bedeutsam von null abweicht. Darüber hinaus werden Statistiken zu den Residuen sowie Zusammenhangsstatistiken ausgegeben.

Modelle mit mehreren Prädiktorvariablen

Durch die Formel-Schreibweise ist man in R relativ flexibel was die Erweiterung von linearen Modellen angeht. Während man bei Programmen mit grafischen Oberflächen meist andere Menüpunkte anwählen muss, um ein Modell mit mehreren UV's oder AV's auszurechnen, kann man in R einfach das Formelelement, das man der lm-Funktion übergibt, erweitern.

Unten stehend ein Beispiel mit drei unabhängigen Variablen zur Erklärung der Varianz einer abhängigen Variablen.

EINGABE

```
x1 <- rnorm(60, 0, 2)
x2 <- rnorm(60, 2, 2)
x3 <- rnorm(60, 4, 2)

y <- 1.2 * x1 + 0.3 * x2 + 0.6 * x3 + 8 + rnorm(60, 0, 2)

modell <- y ~ x1 + x2 + x3
summary(lm(modell))
```

AUSGABE

```
Call:
lm(formula = modell)

Residuals:
     Min       1Q   Median       3Q      Max
-4.98860 -1.13322  0.06602  1.63889  3.49744

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   8.6630     0.7124  12.160  < 2e-16 ***
x1            1.2173     0.1317   9.240  7.4e-13 ***
x2            0.1700     0.1501   1.132  0.26238
x3            0.4448     0.1406   3.163  0.00253 **
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 2.019 on 56 degrees of freedom
Multiple R-squared: 0.6267, Adjusted R-squared: 0.6067
F-statistic: 31.33 on 3 and 56 DF,  p-value: 5.048e-12
```

Im Output lässt sich ablesen, dass der Einfluss der x2 Variable leicht unterschätzt wurde und dadurch in diesem Beispiel nicht signifikant wurde. Würde man die Code-Zeilen ein paarmal wiederholen oder mit einer größeren Stichprobe durchführen, würden sich die Schätzungen natürlich aufgrund der zufälligen Fehlerstreuung verändern.

In linearen Modellen lassen sich auch nichtlineare Effekte einbauen. Diese so genannten Interaktionseffekte werden beim Aufrufen der Funktion *lm()* nicht mit modelliert bzw. geschätzt, sofern die Modellgleichung nach dem bisherigen Schema aufgebaut wird. Um Interaktionseffekte in das formula-Objekt aufzunehmen, muss dieses folgendermaßen erweitert werden:

EINGABE

```
modell1 <-  y ~ x1 + x2 + x3 + x1:x2
modell2 <-  y ~ x1 + x2 + x3 + x1:x2 + x1:x3
modell3 <-  y ~ x1 + x2 + x3 + x1:x2 + x1:x3 + x2:x3

modell4 <-  y ~ x1 + x2 + x3 + x1:x2 + x1:x3 + x2:x3 + x1:x2:x3
```

Die Definition von *modell1* würde neben den Haupteffekten auch den Interaktionseffekt von x1 und x2 schätzen. Würde man die Variable *modell3* der *lm*-Funktion übergeben, so würde man neben den drei Haupteffekten auch eine Schätzungen aller Interaktionseffekte erster Ordnung bekommen. Modellgleichung 4 definiert darüber hinaus noch eine Interaktion dritter Ordnung (*x1:x2:x3*).

Die vierte Modellgleichung kann man folgendermaßen verkürzen:

EINGABE

```
1  modell3 <-  y ~ x1 * x2 * x3
2  # wird als x1 + x2 + x3 + x1:x2 + x1:x3 + x2:x3 + x1:x2:x3 interpretiert
```

Durch die Ersetzung des + Zeichens mit einem Sternchen werden neben den Haupteffekten auch alle möglichen Interaktionseffekte erster Ordnung und höherer Ordnung errechnet.
Dazu die Analyse von Beispieldaten mit Interaktionseffekten:

EINGABE

```
1  n <- 60
2
3  x1 <- rnorm(n, 0, 2)
4  x2 <- rnorm(n, 2, 2)
5  x3 <- rnorm(n, 4, 2)
6
7  y <- 0 * x1 + 1 * x2 + 1.5 * x3   # Haupteffekte
8       + (0.5 * x1 * x2)            # Interaktionen 1. Ordnung
9       + (0.1 * x1 * x2 * x3)       # Interaktionen 2. Ordnung
10      + rnorm(n, 0, 2)             # Fehlerstreuung
11
12 modell <- y ~ x1 * x2 * x3
13 # bewirkt die Berechnung von Interaktionseffekten höherer Ordnung
14 summary(lm(modell))
```

AUSGABE

```
1  Call:
2  lm(formula = modell)
3
4  Residuals:
5      Min      1Q  Median      3Q     Max
6  -5.5682 -1.1712  0.3115  1.0900  4.2404
7
8  Coefficients:
9              Estimate Std. Error t value Pr(>|t|)
10 (Intercept) -0.03588    1.05653  -0.034  0.97304
11 x1          -0.73971    0.61074  -1.211  0.23131
12 x2           0.98107    0.31237   3.141  0.00278 **
13 x3           1.48470    0.23528   6.310  6.2e-08 ***
14 x1:x2        0.48955    0.19679   2.488  0.01610 *
15 x1:x3        0.16708    0.13896   1.202  0.23465
16 x2:x3       -0.01597    0.07273  -0.220  0.82702
17 x1:x2:x3     0.10483    0.04448   2.357  0.02223 *
18 ---
19 Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
20
21 Residual standard error: 2.127 on 52 degrees of freedom
22 Multiple R-squared: 0.8856, Adjusted R-squared: 0.8702
23 F-statistic: 57.53 on 7 and 52 DF,  p-value: < 2.2e-16
```

In diesen generierten Daten steckt ein so genannter Suppressoreffekt. Auch wenn x1 nicht direkt mit dem Kriterium zusammenhängt, so kann x1 in Verbindung mit x2 (Interaktion von x1 und x2) dennoch zur Erklärung der Variation der y-Werte beitragen.

Kategoriale Prädiktoren

Bisher wurden lediglich stetige (also mindestens intervallskalierte) Variablen generiert. Allerdings lassen sich auch kategoriale Daten (also Daten auf Nominalniveau) in ein Modell aufnehmen.
Angenommen es wurden drei Gruppen in dem Ausprägungsgrad einer abhängigen Variablen untersucht, so lässt sich dies durch den schon beschriebenen Weg tun:

EINGABE
```
1  n <- 60
2
3  y <- c(rnorm(n, 7, 2), rnorm(n, 1, 2), rnorm(n, 4, 2))
4  x <- c(rep("grp1", n), rep("grp2", n), rep("grp3", n))
5  x <- as.factor(x)
6
7  daten <- data.frame(x, y)
8  modell <- daten$y ~ daten$x - 1
9
10 summary(lm(modell))
```

AUSGABE
```
1  Call:
2  lm(formula = modell)
3
4  Residuals:
5       Min       1Q   Median       3Q      Max
6  -4.92699 -1.33722  0.02304  1.38071  4.09241
7
8  Coefficients:
9         Estimate Std. Error t value Pr(>|t|)
10 daten$xgrp1   7.2459     0.2510   28.87   <2e-16 ***
11 daten$xgrp2   0.5221     0.2510    2.08   0.0389 *
12 daten$xgrp3   4.1244     0.2510   16.43   <2e-16 ***
13 ---
14 Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
15
16 Residual standard error: 1.944 on 177 degrees of freedom
17 Multiple R-squared: 0.8622, Adjusted R-squared: 0.8599
18 F-statistic: 369.3 on 3 and 177 DF,
19 p-value: < 2.2e-16
```

Die Modellformel wurde ein wenig verändert: Das -1 im formula-Objekt bewirkt, dass keine Konstante (Intercept) geschätzt wird. Stattdessen werden die Gewichtungsparameter der drei Faktorstufen der x-Variable geschätzt. Da die x-Variable lediglich kategoriale Daten enthält, entspricht diese Gewichtung dem Mittelwert der abhängigen Variable y auf der jeweiligen Gruppenstufe. Der Mittelwert der zweiten Gruppe wurde dabei aufgrund der Zufallsziehung im obigen Beispiel etwas unterschätzt.

Der Output der *lm()*-Funktion ist vorwiegend regressionsanalytisch orientiert. Für eine varianzanalytische Konnotation und varianzanalytische Termini eignet sich die Funktion *aov()* (für analysis of variance).

aov(), TukeyHSD()

Zur Berechnung von Einzelkontrasten, also dem relativen Vergleich der Mittelwertsunterschiede auf den Stufen von x, eignet sich die Funktion *TukeyHSD()* zur Berechnung von Konfidenzintervallen und Inferenzstatistik bei der der Problematik des multiplen Testens und der damit verbundenen Alpha-Kumulierung Rechnung getragen wird.

Folgend ein Beispiel mit zwei kategorialen UV's und einer stetigen AV:

EINGABE

```
n <- 100

y <- c(rnorm(n/2, 8, 2),
       rnorm(n/2, 7, 2),
       rnorm(n/2, 7, 2),
       rnorm(n/2, 8, 2),
       rnorm(n/2, 8, 2),
       rnorm(n/2, 7, 2))

x1 <- c(rep("grp1", n), rep("grp2", n), rep("grp3", n))

x2 <- c(rep("m", n/2),
        rep("w", n/2),
        rep("m", n/2),
        rep("w", n/2),
        rep("m", n/2),
        rep("w", n/2))

x1 <- as.factor(x1)
x2 <- as.factor(x2)

daten2x3 <- data.frame(x1, x2, y)
modell <- daten2x3$y ~ daten2x3$x1 * daten2x3$x2

summary(aov(modell))
```

AUSGABE

```
                           Df  Sum Sq Mean Sq F value   Pr(>F)
daten2x3$x1                 2   19.21    9.60  2.0820 0.126513
daten2x3$x2                 1    4.45    4.45  0.9640 0.326987
daten2x3$x1:daten2x3$x2     2   50.11   25.06  5.4319 0.004825 **
Residuals                 294 1356.20    4.61
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
```

Im Output zeigt sich, dass die Variable x1 (Gruppenvariable) keinen bedeutsamen Anteil zur Varianzaufklärung von y beitragen kann, während dies bei Variable x2 (Geschlechtsvariable) der Fall ist. Darüber hinaus wurde die Interaktion der beiden Variablen signifikant. Dieses Ergebnis ist erwartungsgemäß, da bei der Generierung der Mittelwert der y-Variable innerhalb der Gruppen und innerhalb der Geschlechterstufen konstant gehalten wurde, während der Ausprägungsgrad der y-Variable auf den jeweiligen Stufen Geschlecht x Gruppe variiert wurde.

EINGABE

```
TukeyHSD(aov(modell))
```

AUSGABE

```
  Tukey multiple comparisons of means
    95% family-wise confidence level

Fit: aov(formula = modell)

$`daten2x3$x1`
                 diff       lwr       upr     p adj
grp2-grp1 -0.56645237 -1.281958 0.1490534 0.1507902
grp3-grp1 -0.50111261 -1.216618 0.2143932 0.2265466
grp3-grp2  0.06533977 -0.650166 0.7808456 0.9748144

$`daten2x3$x2`
         diff        lwr       upr     p adj
w-m -0.243499 -0.7315859 0.2445878 0.3269871

$`daten2x3$x1:daten2x3$x2`
                   diff        lwr          upr     p adj
grp2:m-grp1:m -1.02247765 -2.2547039  0.2097485 0.16655
grp3:m-grp1:m  0.04271398 -1.1895122  1.2749401 0.99999
grp1:w-grp1:m -0.18496480 -1.4171910  1.0472614 0.99810
grp2:w-grp1:m -0.29539190 -1.5276181  0.9368343 0.98322
grp3:w-grp1:m -1.22990399 -2.4621302  0.0023222 0.05074
grp3:m-grp2:m  1.06519162 -0.1670346  2.2974178 0.13348
grp1:w-grp2:m  0.83751284 -0.3947134  2.0697390 0.37411
grp2:w-grp2:m  0.72708574 -0.5051405  1.9593119 0.53774
grp3:w-grp2:m -0.20742635 -1.4396526  1.0247998 0.99673
grp1:w-grp3:m -0.22767878 -1.4599050  1.0045474 0.99492
grp2:w-grp3:m -0.33810588 -1.5703321  0.8941203 0.96956
grp3:w-grp3:m -1.27261797 -2.5048442 -0.0403917 0.03839
grp2:w-grp1:w -0.11042710 -1.3426533  1.1217991 0.99984
grp3:w-grp1:w -1.04493919 -2.2771654  0.1872870 0.14849
grp3:w-grp2:w -0.93451209 -2.1667383  0.2977141 0.25238
```

Die Einzelkontraste zeigen der Datengenerierung entsprechend kaum signifikante Ergebnisse. Lediglich innerhalb der Gruppen zeigten sich signifikante Unterschiede zwischen den Geschlechterstufen.

8.2 Darstellung der Ergebnisse

8.2.1 Stetige Prädiktoren

Wie jede statistische Auswertungsfunktion liefert *lm()* ein *list* Objekt zurück, in welchem die Rohdaten, Modellparameter, Residuen und vorhergesagte Werte gespeichert sind. Die Beschreibung der einzelnen Elemente des zurückgegebenen Objektes ist in der Hilfe zu *lm()* zu finden, die Verschachtelung der einzelnen Elemente ist mit dem Befehl *str()* nachzuvollziehen.

EINGABE

```
lm1 <- lm(modell)

class(lm1)
str(lm1)
```

abline()

Soll beispielsweise ein Streudiagramm mit eingezeichneter Regressionsgeraden erzeugt werden, können alle Informationen hierfür dem *lm()* Objekt entnommen werden. Die Funktion *abline(a, b)* fügt hierbei einem bestehenden Plot eine Linie mit dem y-Achsenabschnitt *a* und der Steigung *b* hinzu.

EINGABE

```
1  plot(x, y)
2
3  lm1$coefficients
4
5  abline(lm1$coefficients)
```

Das Weiterarbeiten mit den Residuen der Regression kann zwei Vorteile haben. Zum einen lassen sich Modellvoraussetzungen der linearen Regression, wie im folgenden Abschnitt beschrieben, anhand der Verteilung der Residuen überprüfen (Homoskedastizität, Normalverteilung der Residuen). Zum anderen stellen die Residuen einer Regression von *x* auf *y* den Anteil der Varianz von *y* dar, aus dem *x* heraus partialisiert wurde. Somit lassen sich relativ einfach Partialkorrelationen zu Drittvariablen berechnen.

lines()

Im folgenden Beispiel werden dem Streudiagramm die Residuen grafisch hinzugefügt. Die Funktion *lines(c(x0, x1), c(y0, y1))* fügt dem Plot eine Linie hinzu, welche von dem Punkt mit den Koordinaten *x0, y0* bis zum Punkt *x1, y1* läuft.

EINGABE

```
1  for (i in 1:length(lm1$residuals)) {
2                                      lines(c(x[i], x[i]),
3                                      # senkrechte Linien
4                                      c(y[i], y[i]-lm1$residual[i]))
5                                     }
```

8.2.2 Kategoriale Prädiktoren

interaction.plot()

Eine Möglichkeit zur Darstellung von Ergebnissen, denen kategoriale Prädiktoren zugrunde liegen, wurde bereits in Kapitel 3 mit der Funktion *error.bars.by()* des *psych* Paketes vorgestellt. Hierbei werden die Mittelwerte von Faktorstufen mit ihrem Konfidenzintervall abgetragen. Eine schnellere Variante besteht in der Funktion *interaction.plot(x.factor, trace.factor, response)*. Hierbei werden die Mittelwerte des Vektors *response* in den Kategorien des Faktors *x.factor* auf der x-Achse abgetragen. Die unterschiedlichen Kategorien des *trace.factor* werden durch verschiedene Linien kenntlich gemacht.

EINGABE

```
1  attach(data2x3)
2  interaction.plot(y, x1, x2)
3  detach(data2x3)
```

8.3 Grafische Beurteilung der Voraussetzungserfüllung

In R steht mit *plot(which)* eine generischen Funktion zu Verfügung, welche eine Reihe von Grafiken zur Beurteilung der Voraussetzungserfüllung liefert. Der Parameter *which* legt dabei fest,

welche Grafiken geplottet werden sollen. Es sei angemerkt, dass jede Grafik dieser generischen Funktion ohne größeren Aufwand auch „von Hand" erstellt werden kann. Die Funktion *plot()* bietet den Vorteil extreme Fälle im Streudiagramm mit der Fallnummer zu markieren. Um besser zu verstehen, was hinter den Grafiken steht, werden im Folgenden immer beide Varianten dargestellt, die Darstellung mit der Funktion *plot()* und das Plotten "von Hand".

8.3.1 Homoskedastizität

Homoskedastizität bedeutet im bivariaten Fall, dass die Varianz der einen Variablen über die Stufen der anderen Variablen hinweg konstant ist. Ist beispielsweise die Varianz der Variablen *y* bei kleinem *x* unterschiedlich von der Varianz von *y* bei großem *x*, spricht man von Heteroskedastizität. Liegt Heteroskedastizität vor, ist die Schätzung der Modellparameter nach der Gaußschen Methode der Kleinstquadrate beeinträchtigt. Manchmal bietet es sich hierbei an, eine Transformation der Daten (z.B. logarithmieren) vorzunehmen, um eine ausreichende Homoskedastizität zu erreichen.

Im Folgenden wird zunächst der homoskedastische Fall dargestellt. Der Parameter *which* der Funktion *plot()* wird hierbei auf 1 für den "Residuals vs. Fitted" Plot gesetzt. Die Heteroskedastizität wird generiert, indem die Varianz des Fehleranteils, welcher der Variablen *y* hinzugefügt wird, von der Ausprägung der Variablen *x* abhängig gemacht wird. Die Funktion *abs()* liefert den Absolutwert einer Variablen und verhindert hierbei, dass negative Werte für die Varianz in der Funktion *rnorm()* angegeben werden.

EINGABE

```
1  # Homoskedastizität
2  x <- rnorm(200, 0, 1)
3  y <- 5 + x + rnorm(200, 0, 1)
4  lm1 <- lm(y~x)
5
6  plot(lm1, which = 1, main = "Homoskedastizität")
7  # Residuals vs. Fitted
8
9  # Heteroskedastizität
10 y.het <- 5 + x + rnorm(200, 0, abs(x))
11 lm2 <- lm(y.het~x)
12
13 plot(lm2, which = 1, main = "Heteroskedastizität")
14 # Residuals vs. Fitted
```

EINGABE

```
1 # "von Hand"
2 plot(lm1$fitted.values, lm1$residuals, main = "Homoskedastizität")
3 plot(lm2$fitted.values, lm2$residuals, main = "Heteroskedastizität")
```

8.3.2 Normalverteilung der Residuen

Bei der Bestimmung von Konfidenzintervallen auf der Grundlage der Standardfehler des Modells wird von einer Normalverteilung der Fehler ausgegangen. Liegt somit eine Nicht-Normalität der Fehler vor, ist hiervon die Aussage über die Signifikanz der Regressionsschätzung betroffen. Nicht betroffen von der Nicht-Normalität ist die Parameterschätzung selbst. Heteroskedastizität

ist hierbei weder eine hinreichende noch eine notwendige Bedingung für eine Nicht-Normalität der Residualverteilung.

Um eine unverzerrte Aussage über die statistische Bedeutsamkeit der Parameterschätzung zu ermöglichen, muss deswegen eine Prüfung auf Normalverteilung der Fehler erfolgen. Prinzipiell muss hierbei auf jeder Stufe von x die Normalverteilung der Fehler gezeigt werden. Da dies aber sehr große Stichproben erfordert, wird in der Regel nur die Normalverteilung der Verteilung aller Residuen geprüft, auch wenn die Normalität der gesamten Residualverteilung nur eine notwendig, aber keine hinreichende Bedingung für die Normalität der Residuen auf allen Stufen von x ist.

Im folgenden Beispiel wird die Normalverteilung der Residuen anhand eines Q-Q-Plots im Falle von normalverteilten und nicht-normalverteilten Fehlern grafisch geprüft. Der Parameter *which* der Funktion *plot()* wird hierbei auf 2 für die Ausgabe eines Q-Q-Plots gesetzt. Zur Erzeugung von Nicht-Normalität der Fehler werden Zufallsvariablen aus einer F-Verteilung als Fehler zu der abhängigen Variablen addiert.

EINGABE
```
1  plot(lm1, which = 2)
2
3  y.nn <- 5 + x + rf(200, 10, 4)
4  lm3 <- lm(y.nn~x)
5
6  plot(lm3, which = 2)
```

EINGABE
```
1  # "von Hand"
2  qqnorm(lm1$residuals)
3  qqline(lm1$residuals) # Hinzügen der Linie
4
5  qqnorm(lm3$residuals)
6  qqline(lm3$residuals) # Hinzügen der Linie
```

Übungen

1. Generiere zwei Variablen x und y wie im Code-Beispiel auf Seite 46 und zeige, dass mit steigender Stichprobengröße die Abweichung der geschätzten Modellparametern von den festgelegten Parametern kleiner wird.

2. Lasse dir den Konfidenzbereich der Schätzung der Modellparamter anzeigen. Die Schätzung ist um den Mittelwert $\bar{x}$ herum am genausten, der Konfidenzbereich damit am engsten. Je weiter die Schätzung sich auf Werte, welche entfernter vom Mittelwert sind, bezieht, desto breiter wird der Konfidenzbereich.

   a) Generiere die hier aus einer Population mit konstantem n eine Anzahl N von Stichproben, bestimme für diese jeweils die Modellparameter.

   b) Plotte dir im nächsten Schritt alle Regressionsgeraden mit der Funktion *abline()*. Mache dir den Parameter *col* (Farbe) dieser Funktion zunutze und setze den Wert auf *col = rgb(0,0,0, 0.05)*. Dies erzeugt eine halbtransparente, schwarze Linie.

3. Wie lässt sich auf einzelne statistische Kennwerte bei der Funktion *lm()* zugreifen? Mache dir die Funktion *str()* dabei zunutze.

4. Welche Effekte werden bei folgender Formel untersucht?

   EINGABE
   ```
   1  y ~ x1 + x1:x2 + (x3+x1)^2
   ```

5. Generiere dir folgende Daten:

   EINGABE
   ```
   1  x1 <- rnorm(100, 0, 1)
   2  x2 <- 3 + 0.3*x1 + rnorm(100, 0, 1)
   3  y <- 10.4 - 0.3*x1 + 1.3*x2 + 0.4*x1*x2 + rnorm(100, 0, 1)
   ```

   a) Berechne die multiple Regression nur mit den beiden Haupteffekten.

   b) Berechne das volle Modell.

   c) Beurteile grafisch die Erfüllung der Voraussetzungen.

   d) Wie ließe sich die Interaktion x1*x2 grafisch darstellen?

6. In einer Simulationsstudie soll herausgefunden werden, wie stark sich Heteroskedastizität auf die Parameterschätzungen auswirkt. Angenommen dies soll in einem Design mit einem Faktor (zwei Stufen) als UV untersucht werden.

   a) Wie wäre das generelle Vorgehen?

   b) Wie ließe sich der Grad der Heteroskedastizität variieren?

   c) Wie ließe sich der Bias in der Parameterschätzung messen?

   d) Führe diese Simulationsstudie durch.

Kapitel 9
Low-Level und High-Level Plotting

9.1 High-Level Plotting

Funktionen, die einen vollständigen Plot erzeugen, werden in R auch High-Level Plotting Funktionen genannt. Im graphics-Paket, das zur Standardinstallation von R gehört, sind zum Beispiel die Funktionen *plot()*, *boxplot()* oder *hist()* enthalten. All diese Funktionen sind High-Level Plotting Funktionen.

Die Darstellungsweise eines Plots lässt sich meist nur bedingt durch optionale Argumente innerhalb der High-Level Plotting Funktionen verändern oder bestimmen.

9.1.1 Die par()-Funktion

par()

In R müssen bestimmte Informationen vorangeschickt werden, bevor der tatsächliche Plot erstellt wird. Dies erreicht man mit der Funktion *par()* (für parameters). Die Funktion *par()* besitzt eine Vielzahl an Argumenten, mit denen die Darstellungsweise eines Plots verändert werden kann.

Hier eine Beschreibung der wichtigsten Argumente der *par()*-Funktion mit den jeweiligen Standardwerten. Die angegeben Unterargumente *main*, *lab* und *axis* stehen für den Plottitel, den Achsentitel sowie die Achsenbeschriftung.

Abstände und Boxen

bty = „o“ Definiert die Umrandungsbox
mögliche Werte: l, 7, c, u,], o (symbolisch für die Umrandungslinien)
oder n für keinerlei Boxlinien

mar = c(5, 4, 4, 2) + 0.1 Abstand des Plotbereichs (definiert durch die Umrandungsbox) zu den vier Außenseiten des Plotfensters (Definitionsreihenfolge: unten, links, oben, rechts), gemessen in Textlinienhöhen. Ein Abstand von 1 nach unten würde genügend Platz für die Achsenbeschriftung der x-Achse frei machen. Ein Abstand von 3 nach unten würde genügend Platz für die Achsenbeschriftung, ein freies Feld und den Achsentitel nach unten schaffen.
Alternativ können die Außenabstände auch mit dem Argument *mai* in Inch angegeben werden.

mgp = c(3, 1, 0) Der Abstand des Achsentitels (erstes Element), der Achsenbeschriftung (zweites Element) und der Achsenlinie (drittes Element) zur Umrandungsbox bzw. dem Plotbereich.

oma = c(0, 0, 0, 0) Definition der Außenabstände des Beschriftungsbereichs in Textlinienhöhen (Elementenreihenfolge: unten, links, oben, rechts)
Alternativ können die Außenabstände des Beschriftungsbereichs auch mit dem Argument *omi* in Inch angegeben werden.

Schrift und Größe

cex = 1 Relative Veränderung der Größe von Plotsymbolen und Beschriftungen
(Unterargumente: *cex.main*, *cex.lab*, *cex.axis*)

family Definition der Schriftart
Standardwert: Standardschrift des Systems
Mögliche Werte (unter anderem): *„serif“*, *„sans“*, *„mono“*

font = 1 Definition des Schrifttyps
1 = normal (Standardwert); 2 = bold; 3 = italic; 4 = bold/italic
(Unterargumente: *font.main*, *font.lab*, *font.axis*)

adj = 0.5 Definition der Ausrichtung der Plotbeschriftungen (Achsen, Titel etc.).
Mögliche Werte: 0 = linksbündig; 0.5 = zentriert; 1 = rechtsbündig

las = 0 Ausrichtung der Achsenbeschriftung relativ zur Achse
0 = Parallel zur Achse; 1 = horizontal; 2 = senkrecht zur Achse; 3 = vertikal

Farben

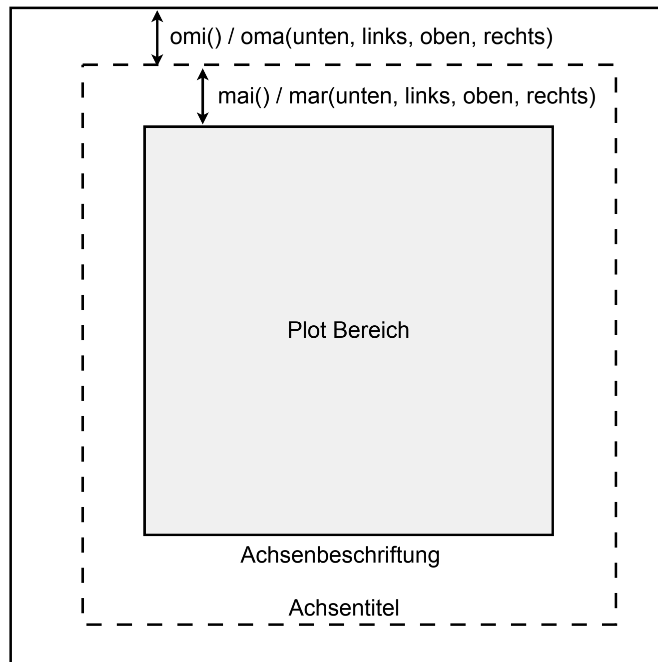
bg = „white“ Die Hintergrundfarbe des gesamten Plots.
Für Alternativwerte: siehe *?colors()*

col = „black“ Farbe für Achsen, Boxen, Beschriftungen etc.
Für Alternativwerte: siehe *?colors()*
(Unterargumente: *col.main*, *col.lab*, *col.axis*)

Eine vollständige Auflistung der *par()*-Argumente kann der Hilfedatei zur Funktion entnommen werden.
In der unten stehenden Abbildung sind die einzelnen Plotbereiche, die durch die Argumente *mar*, *mai*, *oma* und *omi* verändert werden können, grafisch veranschaulicht.

Die Trennung von Darstellungsdefinition und tatsächlicher Plot-Erstellung hat einen großen Vorteil: Es muss lediglich einmal durch die *par()*-Funktion definiert werden, wie die einzelnen Elemente eines Plots angeordnet und angezeigt werden sollen. Alle High-Level Plot Funktionen, die auf eine *par()*-Funktion folgen, werden sich an diesen Definitionen orientieren. Dadurch ist eine konsistente und sogleich an die eigenen Wünsche angepasste Darstellung von Abbildungen möglich.

Im unteren Beispiel werden durch die *par()*-Funktion die Standardwerte einiger Plotparameter überschrieben. Die zwei nachfolgenden Plots, die durch die Funktionen *boxplot()* und *hist()* aufgerufen werden, übernehmen diese Definitionen.
Obwohl die Darstellungsweise der Daten sich unterscheidet, sind die Plots in ihrem Aussehen vergleichbar und vermitteln einen konsistenten - wenn auch nicht sehr vorteilhaften - Eindruck.



EINGABE

```
1  par(bg = "lightblue", col.axis = "white", bty = "n",
2      mar = c(3, 3, 3, 1), cex = 1.2, font.axis = 2, las = 1,
3      ask = TRUE)
4
5  x <- rnorm(1000, 0, 2)
6
7  boxplot(x)
8
9  hist(x)
```

Die Definition *ask=TRUE* bewirkt, dass R dazu auffordert, eine Taste zu drücken bevor ein bereits bestehender Plot durch einen neuen Plot-Befehl überschrieben wird.

9.2 Low-Level Plotting

Mit der *par()*-Funktion lassen sich Abstände und Elemente des Plots beeinflussen. Was sich meist nicht beeinflussen lässt, ist die Art der Darstellung der Daten, die von der High-Level Plot Funktion erzeugt wird.

axis(), legend(), text(), title(), points(), abline(), lines(), polygon()

Um noch mehr Einfluss auf die Darstellung zu haben, gibt es in R die Methode des Low-Level Plottings. Dabei wird zunächst mit einem High-Level Plot Befehl ein Plotfenster erzeugt. Mithilfe von Low-Level Plot Funktionen lassen sich nun einzelne Elemente zum schon bestehendem Plot-Fenster hinzufügen. Low-Level Plot Funktionen rufen also im Gegensatz zu High-Level Plot Funktionen kein neues Plot-Fenster auf.

Eine Auflistung der wichtigsten Low-Level Plot Funktionen:

axis()	Fügt eine Achse zum Plot hinzu
legend()	Fügt eine Legende zum Plot hinzu
text()	Fügt Text an einer definierten Stelle ein
title()	Erstellung eines Plot-Titels
points()	Fügt Datenpunkte ein
abline()	Erstellung einer Regressionslinie
lines()	Erstellung einer Linie
polygon()	Erstellung eines Polygons

Mithilfe von optionalen Argumenten kann man wie bei der *par()*-Funktion das Aussehen verändern. Hier einige der wichtigsten Argumente, die von mehreren Low-Level Plotting Funktionen genutzt werden:

lty = 1 Die Linienart für Plotlinien
1 = durchgezogen; 2 = gestrichelt, 3 = gepunktet, 4 = gestrichelt und gepunktet, 5 = lange gestrichelte Linien, 6 = lange und kurze gestrichelte Linien

lwd = 1 Die Linienbreite für Plotlinien

pch = 21 Definition des Symbols für Datenpunkte
Mögliche Werte: 19 = einfacher Kreis; 20 = kleiner Kreis; 21 = gefüllter Kreis; 22 = gefülltes Viereck (siehe *?points()* für weitere mögliche Werte)

xlog / ylog = FALSE Definiert ob eine lineare oder logarithmische Achse gezeichnet werden soll (mögliche Werte: *TRUE* / *FALSE*)

Nachfolgend ein Codebeispiel für einen Plot mithilfe von Low-Level Funktionen:

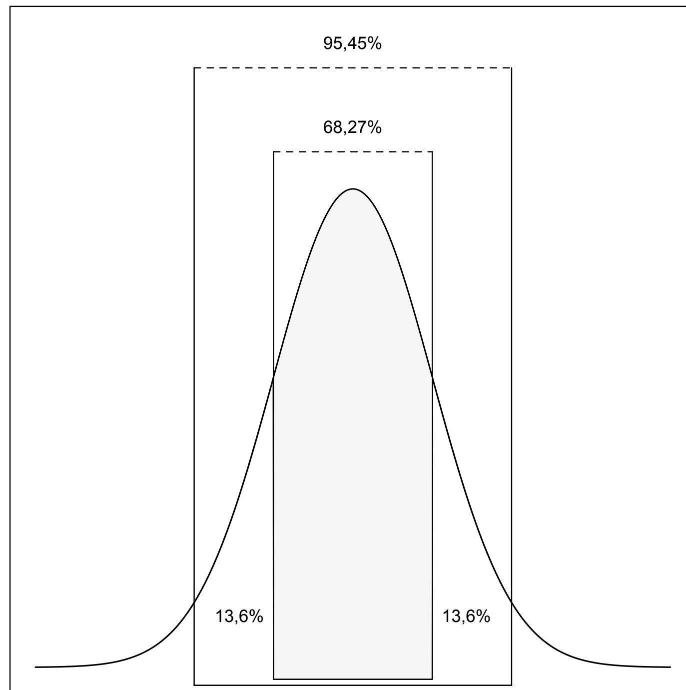
EINGABE

```
par(bty="n", cex.lab=1.2, cex.axis=1.0, mar=c(0,0,0,0),
    oma=c(0.2,0.2,0.2,0.2), mgp=c(4,1.5,0))

x <- seq(-4,4, length=300)
y <- dnorm(x, m=0, sd=1)

plot(x,y, type="l", ylim=c(0,0.53), xaxp=c(-4,4,8), xaxt="n",
     yaxt="n")


lines(x=c(-1,+1), y=c(0.43,0.43), lty="dashed")
lines(x=c(-1,-1), y=c(-0.010,0.43))
lines(x=c(+1,+1), y=c(-0.010,0.43))

lines(x=c(-2,+2), y=c(0.50,0.50), lty="dashed", lwd=1)
lines(x=c(-2,-2), y=c(-0.015,0.50))
lines(x=c(+2,+2), y=c(-0.015,0.50))

lines(x=c(-2,+2), y=c(-0.015,-0.015), lwd=1.1)

polygon(x=c(-1,x[113:188],1), y=c(-0.010, y[113:188], -0.010),
        col="#F2F2F2")

text(x=0, y=0.45, labels="68,27%", cex=0.9)
text(x=0, y=0.52, labels="95,45%", cex=0.9)


# Die Angabe x und y können auch weggelassen werden, sofern die Werte
# in dieser Reihenfolge am Anfang der Argumentenauflistung auftauchen:

text(-1.43, 0.042, labels="13,6%", cex=0.9)
text(+1.43, 0.042, labels="13,6%", cex=0.9)

box(which = "inner", lty = "solid")
```

Zunächst werden Abstände und Schriftgrößen mit der *par()*-Funktion definiert. Dann wird ein Plotfenster mit der High-Level Plotting Funktion *plot()* erzeugt, in dem lediglich eine Normalverteilungslinie dargestellt wird. Dann werden mit der Funktion *lines()* vertikale und horizontale Linien in das bestehende Fenster gezeichnet, welche die Standardabweichungsgrenzen der Normalverteilung kennzeichnen. Mithilfe der Funktion *polygon()* wird die Fläche zwischen dem Abstand einer Standardabweichung vom Mittelwert mit einem hellen Grau ausgefüllt. Mit der Funktion *text()* werden die einzelnen Flächen dann beschriftet. Mit der Funktion *box()* in der letzten Zeile wird noch eine Box um den Plot gezeichnet.

box()

Die erzeugte Abbildung sieht so aus:



Anhang

Lösungsvorschläge

Kapitel 1

1. Aufgabe

EINGABE

```
1  mittel.aus.vier.zahlen <- function(a,b,c,d) {
2    mittel <- (a+b+c+d) / 4
3    return(mittel)
4    }
5
6  mittel.aus.vier.zahlen(1,2,3,4)
```

2. Aufgabe

EINGABE

```
1  mittelwert <- function(a) {
2    mittel <- sum(a) / length(a)
3    return(mittel)
4    }
5
6  mittelwert(c(1:4))
```

3. Aufgabe

AUSGABE

```
1  [1] 20
```

4. Aufgabe

EINGABE

```
1  help.search("mean")
2  help.search("covariance")
3
4  mean(x1)
5  mean(x2)
6
7  var(x1)
8  var(x2)
9
10 var(x1,x2)
```

5. Aufgabe

EINGABE

```
score <- function(x) {
        gewichtung <- c(1:length(x))
        x.gewichtet <- x * gewichtung
        return(sum(x.gewichtet)^4 / length(x))
        }

Person1 <- c(4.2, 3.1, 3, 3.4, 2.1, 1.1, 1.0, 1.0, 1.2, 1.1)
Person2 <- c(5.2, 3.0, 3.7, 2.1, 1.0, 0.5)

score(Person1)
score(Person2)
```

AUSGABE

```
[1] 5702681
[1] 373845.9
```

Kapitel 2

1. Aufgabe

EINGABE

```
m[,1] <- m[,1]*2
m
```

2. Aufgabe

EINGABE

```
is.character(a)
is.numeric(b)
is.character(c)
```

AUSGABE

```
[1] TRUE
[1] TRUE
[1] TRUE
```

3. Aufgabe

EINGABE

```
zufall <- sample(c(1:3), 10, replace=TRUE)                  # (a)

gruppe <- as.factor(zufall)                                 # (b)

levels(gruppe) <- c("Gruppe1", "Gruppe2", "Gruppe3")        # (c)
gruppe

as.character(gruppe)                                        # (d)
```

AUSGABE

```
 [1] Gruppe2 Gruppe2 Gruppe2 Gruppe1 Gruppe3 Gruppe1 Gruppe1
 [8] Gruppe3 Gruppe3 Gruppe1
Levels: Gruppe1 Gruppe2 Gruppe3

 [1] "Gruppe2" "Gruppe2" "Gruppe2" "Gruppe1" "Gruppe3"
 [6] "Gruppe1" "Gruppe1" "Gruppe3" "Gruppe3" "Gruppe1"
```

4. Aufgabe

EINGABE

```
gruppe <- sample(2, 23, replace=TRUE)                              # (a)
gruppe <- as.factor(gruppe)                                        # (b)
levels(gruppe) <- c("Kontrollgruppe", "Experimentalgruppe")

datentabelle <- cbind(datentabelle, gruppe)                        # (c)
nachuntersuchung <- round(rnorm(23, 12, 2), 2)                     # (d)
datentabelle <- cbind(datentabelle, nachuntersuchung)

head(datentabelle)
# die Funktion head() gibt nur den Anfangsteil eines Vektors, einer Matrix,
# einer Tabelle, oder eines data-frames zurück

 write.table(datentabelle, "Pfad/Testdatei.txt", sep="\t")  # (e)
```

AUSGABE

```

  Geschlecht Voruntersuchung Alter Brillentraeger Code
1   weiblich            7.24    32           TRUE HW23
2   weiblich            6.98    20          FALSE TR01
3   weiblich              NA    21           TRUE FR31
4   weiblich            3.55    35          FALSE WE13
5   weiblich            7.06    27          FALSE QQ04
6   weiblich              NA    20          FALSE AR29
              gruppe nachuntersuchung
1 Experimentalgruppe            14.53
2 Experimentalgruppe            11.22
3     Kontrollgruppe            11.81
4 Experimentalgruppe            11.84
5     Kontrollgruppe            15.58
6     Kontrollgruppe            13.84
```

5. Aufgabe

EINGABE

```
ergebnis <- c(11, 14, 17, 10, 19, 20, 16, 11, 9, 15)
ergebnis.dichotom <- ergebnis

ergebnis.dichotom[ergebnis.dichotom < 15] <- 0
ergebnis.dichotom[ergebnis.dichotom >= 15] <- 1

ergebnis.dichotom <- as.factor(ergebnis.dichotom)

levels(ergebnis.dichotom) <- c("niedrig", "hoch")
```

Kapitel 5

1. Aufgabe

EINGABE

```
par(mfrow=c(1,1))
stpr.groesse <- c(rep(3,3), rep(5,5), rep(10,10), rep(20,20),
                  rep(50,50), rep(100,100),rep(300,300))
stpr.groesse <- as.factor(stpr.groesse)

x003 <- rnorm(3,0,1)
x005 <- rnorm(5,0,1)
x010 <- rnorm(10,0,1)
x020 <- rnorm(20,0,1)
x050 <- rnorm(50,0,1)
x100 <- rnorm(100,0,1)
x300 <- rnorm(300,0,1)

x <- c(x003, x005, x010, x020, x050, x100, x300)

error.bars.by(x, stpr.groesse, by.var=T, xlab="Stichprobengröße",
              ylab="Konfidenzintervall des Mittelwerts")
```

2. Aufgabe

a)

EINGABE

```
S <- matrix(c(1,0.2,0.2,1), 2, 2)
dat <- as.data.frame(rmvnorm(30, mean = c(0,0), sigma = S))

colnames(dat) <- c("x", "y")

# Da cor() an dieser Stelle eine Matrix zurückgibt, wird mit [1,2] auf die
# Korrelation zwischen den beiden Variablen innerhalb der Korrelationsmatrix
# zugegriffen
korrelation <- cor(dat)[1,2]

# Die Überschrift wird festgelegt.
plot(dat, main=paste("Korrelation:", round(korrelation,2)))
```

b)

EINGABE

```
outlier1 <- c(5,5)
outlier2 <- c(7,0)

new.dat <- rbind(dat, outlier1, outlier2)
```

c)

EINGABE

```
par(mfrow=c(1,2))
korrelation.neu <- cor(new.dat)[1,2]

plot(new.dat, main=paste("Korrelation.neu:",
                         round(korrelation.neu,2)))
plot(dat, main=paste("Korrelation.alt:", round(korrelation,2)))
```

d)

EINGABE

```
outlier1 <- c(-4,-4)
outlier2 <- c(4,4)

new.dat <- rbind(dat, outlier1, outlier2)

par(mfrow=c(1,2))
korrelation.neu <- cor(new.dat)[1,2]

plot(new.dat, main=paste("Korrelation.neu:",
                          round(korrelation.neu,2)))
plot(dat, main=paste("Korrelation.alt:", round(korrelation,2)))
```

3. Aufgabe

a)

EINGABE

```
# Die beiden Haupteffekte
effect.geschlecht <- 2.5
effect.brille <- -3.1

# Die Faktoren
geschlecht <- sample(c("m","w"),200,replace=TRUE)
geschlecht <- as.factor(geschlecht)

brille <- sample(c("b","kb"),200,replace=TRUE,prob=c(0.4,0.6))
brille <- as.factor(brille)

# Die Leistung ohne Effekte
leistung <- rnorm(200,10,2) # Mittelwert = 10, Standardabweichung = 2

# Effekte werden aufaddiert
leistung[geschlecht=="m"] <- leistung[geschlecht=="m"]
                              + effect.geschlecht
leistung[brille=="b"] <- leistung[brille=="b"] + effect.brille

daten <- data.frame(brille, geschlecht, leistung)
```

b)

EINGABE

```
par(mfrow=c(1,1))

# blaue Brillenträger
error.bars.by(leistung[brille=="b"], geschlecht[brille=="b"],
              by.var=TRUE, col="blue")

# rote nicht-Brillenträger
error.bars.by(leistung[brille=="kb"], geschlecht[brille=="kb"],
              by.var=TRUE, add=TRUE, col="red")
```

d)

EINGABE

```
# Der Schulfaktor
schule <- sample(c(1:8), 200, replace = TRUE)
schule <- as.factor(schule)
```

e)

EINGABE

```
# Die 8 Klasseneffekte
schule.effects <- rnorm(8, 0, 0.5)

# Aufaddieren der Schuleffekte
leistung[schule==1] <- leistung[schule==1] + schule.effects[1]
leistung[schule==2] <- leistung[schule==2] + schule.effects[2]
leistung[schule==3] <- leistung[schule==3] + schule.effects[3]
leistung[schule==4] <- leistung[schule==4] + schule.effects[4]
leistung[schule==5] <- leistung[schule==5] + schule.effects[5]
leistung[schule==6] <- leistung[schule==6] + schule.effects[6]
leistung[schule==7] <- leistung[schule==7] + schule.effects[7]
leistung[schule==8] <- leistung[schule==8] + schule.effects[8]

daten <- data.frame(schule, brille, geschlecht, leistung)
```

Kapitel 7

1. Aufgabe

Ein gerichteter Mann-Whitney U-Test.

2. Aufgabe

a)

EINGABE

```
rep <- 1000  # repetitions / wiederholungen
samplesizes <- seq(20, 300, 20) # hier pro Gruppe
effect <- 0.3

pvalues <- matrix(NA, nrow=repetitions, ncol=length(samplesizes))

for (i in 1:length(samplesizes)) {
  for (j in 1:repetitions) {
    sample1 <- rnorm(samplesizes[i], 0, 1)
    sample2 <- rnorm(samplesizes[i], effect, 1)
    pvalues[j, i] <- t.test(sample1, sample2)$p.value
  }
}
significantrepetitions <- pvalues <= 0.05
percentsignificant <- apply(significantrepetitions, 2, sum) / rep

names(percentsignificant) <- samplesizes
percentsignificant

plot(samplesizes, percentsignificant, type="l", ylab="power")
```

b) Aus der Grafik aus Teilaufgabe a) würde man eine Power von mindestens 80% bei Stichprobengrößen von etwas weniger als 200 Personen pro Gruppe, also insgesamt unter 400 Personen, erwarten. Um den Wert genauer zu bestimmen, könnte man die Simulation noch einmal für andere Stichprobengrößen laufen lassen:

EINGABE

```
1  samplesizes <- seq(150, 250, 10)
```

3. Aufgabe

EINGABE

```
1  t.test(daten$abh.Var[daten[,1] == "grp1"],
2         daten$abh.Var[daten[,1] == "grp2"])
3
4  # oder einfacher:
5  t.test(daten[,2] ~ daten[,1])
```

4. Aufgabe

EINGABE

```
1   for(i in 2:dim(daten)[2])   {
2           print("")
3           print("")
4           print(names(daten[i]))
5           print("Mittelwerte")
6           print(tapply(daten[,i], daten$gruppe, mean))
7           print("Standardabweichung")
8           print(tapply(daten[,i], daten$gruppe, sd))
9           }
```

Kapitel 8

1. Aufgabe

EINGABE

```
stichproben <- seq(5,100, by=5)
wiederholungen <- 100

steigung.population <- 2.5
intercept.population <- 8

abweichungen.intercept.gesamt <- matrix(data=NA,
                                  ncol=length(stichproben),
                                  nrow=wiederholungen,
                                  dimnames=list(1:100, stichproben))

abweichungen.steigung.gesamt <- matrix(data=NA,
                                  ncol=length(stichproben),
                                  nrow=wiederholungen,
                                  dimnames=list(1:100, stichproben))


for(i in 1:length(stichproben)) {

  for(j in 1:wiederholungen) {
    x <- rnorm(stichproben[i], 0, 2)
    y <- x * rnorm(stichproben[i], steigung.population, 2)
                    + rnorm(stichproben[i], intercept.population, 2)

    modell <- y ~ x
    lm.ergebnis <- lm(modell)

    abweichung.intercept <- intercept.population -
                            lm.ergebnis$coefficients[1]
    abweichung.steigung <- steigung.population -
                            lm.ergebnis$coefficients[2]

    abweichungen.intercept.gesamt[j,i] <- abweichung.intercept
    abweichungen.steigung.gesamt[j,i] <- abweichung.steigung
    }
  }


plot(matrix(stichproben, ncol=length(stichproben),
            nrow=wiederholungen, byrow=TRUE),
            abweichungen.intercept.gesamt,
            xlab="Stichprobengroesse",
            ylab="Abweichung vom Populationswert")

plot(matrix(stichproben, ncol=length(stichproben),
            nrow=wiederholungen, byrow=TRUE),
            abweichungen.steigung.gesamt,
            xlab="Stichprobengroesse",
            ylab="Abweichung vom Populationswert")
```

2. Aufgabe

EINGABE

```
n <- 1200
wiederholungen <- 1000

x <- rnorm(n, 0, 3)

plot(c(), c(), xlim=c(-10,10), ylim=c(-10,10))              # (b)

steigung <- 1 + rnorm(wiederholungen, 0, 0.4)
intercept <- 1 + rnorm(wiederholungen, 0, 0.4)

for(i in 1: wiederholungen) {
  y <- steigung[i] * x + intercept[i]
  abline(lm(x ~ y), col=rgb(0,0,0, 0.05))                   # (b)
  }
```

3. Aufgabe

EINGABE

```
x <- rnorm(10, 0, 3)
y <- x * 0.5 + rnorm(10, 0, 2)

modell <- y ~ x

str(lm(modell))

lm(modell)$coefficients
lm(modell)$residuals
```

4. Aufgabe

Der Einfluss von x1 auf y (Haupteffekt), der Einfluss von x3 auf y (Haupteffekt), der Einfluss der Interaktion zwischen x1 und x2, sowie der Einfluss der Interaktion zwischen x1 und x3 auf y.

5. Aufgabe

EINGABE

```
x1 <- rnorm(100, 0, 1)
x2 <- 3 + 0.3*x1 + rnorm(100, 0, 1)
y <- 10.4 - 0.3*x1 + 1.3*x2 + 0.4*x1*x2 + rnorm(100, 0, 1)

# (a)
modell <- y ~ x1 + x2
summary(lm(modell))

# (b)
modell <- y ~ x1 * x2
summary(lm(modell))

# (c)
plot(lm(modell), which = 1,
     main = "Prüfung auf Heteroskedastizität")
plot(lm(modell), which = 2,
     main = "Prüfung auf Normalverteilung der Residuen")

# d)
?coplot
coplot(y~x1|x2, panel = panel.smooth)
```

6. Aufgabe

a) Zunächst müssen zwei kontinuierliche Variablen mit unterschiedlichem Mittelwert generiert werden. Darüber hinaus muss eine kategoriale Variable mit zwei Stufen generiert werden (siehe dazu Kapitel 2.2.1).

b) Heteroskedastizität mit einer kategorialen unabhängigen Variable zeichnet sich durch Varianzheterogenität zwischen den Gruppen aus. Heteroskedastizität lässt sich also variieren bzw. erzeugen, indem die zwei kontinuierliche Variablen unterschiedliche Varianzen aufweisen.

c) Der Bias zeigt sich in der ungenaueren Schätzung der Regressionsparameter. Der Standardfehler der Regressionsparameter sollte demnach unterschiedlich groß ausfallen, je nach Ausmaß der Heteroskedastizität.

d)

EINGABE

```
1  n <- 50
2
3  # Simulation varianzhomogener Daten
4  intercept.ohne <- c()
5  steigung.ohne <- c()
6
7  for(i in 1:500) {
8    y <- c(rnorm(n/2, 7, 2),  rnorm(n/2, 8, 2))
9    x1 <- c(rep("m", n/2), rep("w", n/2))
10   x1 <- as.factor(x1)
11
12   daten <- data.frame(x1, y)
13   modell <- daten$y ~ daten$x1
14
15   intercept.ohne <- c(intercept.ohne, lm(modell)$coefficients[1])
16   steigung.ohne <- c(steigung.ohne, lm(modell)$coefficients[2])
17   }
18
19 # Simulation varianzheterogener Daten
20 intercept.mit <- c()
21 steigung.mit <- c()
22
23 for(i in 1:500) {
24   y <- c(rnorm(n/2, 7, 2),  rnorm(n/2, 8, 4))
25   x1 <- c(rep("m", n/2), rep("w", n/2))
26   x1 <- as.factor(x1)
27
28   daten <- data.frame(x1, y)
29   modell <- daten$y ~ daten$x1
30
31   intercept.mit <- c(intercept.mit, lm(modell)$coefficients[1])
32   steigung.mit <- c(steigung.mit, lm(modell)$coefficients[2])
33   }
34
35 # Darstellung
36 boxplot(intercept.ohne, intercept.mit, xlab="Datenset",
37         names=c("var(m) = var(w)", "var(m) < var(w)"),
38         ylab="Verteilung des geschätzten intercepts (n = 500)")
39
40 boxplot(steigung.ohne, steigung.mit, xlab="Datenset",
41         names=c("var(m) = var(w)", "var(m) < var(w)"),
42         ylab="Verteilung des geschätzten intercepts (n = 500)")
```