

Daniel Wollschläger

Grundlagen der Datenanalyse mit R

Eine
anwendungsorientierte
Einführung

Reihenherausgeber:

Prof. Dr. Holger Dette · Prof. Dr. Wolfgang Härdle

Statistik und ihre Anwendungen

Weitere Bände dieser Reihe finden Sie unter <http://www.springer.com/series/5100>

Daniel Wollschläger

Grundlagen der Datenanalyse mit R

Eine anwendungsorientierte Einführung



Daniel Wollschläger
Christian-Albrechts-Universität zu Kiel
Institut für Psychologie
Olshausenstr. 62
24098 Kiel
Deutschland
dwoll@psychologie.uni-kiel.de

ISBN 978-3-642-12227-9 e-ISBN 978-3-642-12228-6
DOI 10.1007/978-3-642-12228-6
Springer Heidelberg Dordrecht London New York

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

© Springer-Verlag Berlin Heidelberg 2010

Dieses Werk ist urheberrechtlich geschützt. Die dadurch begründeten Rechte, insbesondere die der Übersetzung, des Nachdrucks, des Vortrags, der Entnahme von Abbildungen und Tabellen, der Funksendung, der Mikroverfilmung oder der Vervielfältigung auf anderen Wegen und der Speicherung in Datenverarbeitungsanlagen, bleiben, auch bei nur auszugsweiser Verwertung, vorbehalten. Eine Vervielfältigung dieses Werkes oder von Teilen dieses Werkes ist auch im Einzelfall nur in den Grenzen der gesetzlichen Bestimmungen des Urheberrechtsgesetzes der Bundesrepublik Deutschland vom 9. September 1965 in der jeweils geltenden Fassung zulässig. Sie ist grundsätzlich vergütungspflichtig. Zu widerhandlungen unterliegen den Strafbestimmungen des Urheberrechtsgesetzes.

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

Einbandentwurf: WMXDesign GmbH, Heidelberg

Gedruckt auf säurefreiem Papier

Springer ist Teil der Fachverlagsgruppe Springer Science+Business Media (www.springer.com)

Vorwort

Das vorliegende Buch liefert eine an human- und sozialwissenschaftlichen Anwendungen orientierte Einführung in die Datenauswertung mit dem Statistikprogramm R. R ist eine freie Umgebung zur Analyse und graphischen Darstellung von Datensätzen, die befehlsoorientiert arbeitet. Die Motivation für dieses Buch entstand aus dem Eindruck, dass sich R zwar unter statistischen Experten großer Beliebtheit erfreut, Anwender statistischer Verfahren aus Gebieten der empirischen Datenanalyse dagegen das Potential von R noch nicht gleichermaßen nutzen. Dieser Umstand scheint zumindest teilweise dem Mangel an deutschsprachiger Literatur geschuldet, die sich explizit an ein Publikum wendet, das in erster Linie einen Grundkanon bestehender Auswertungsverfahren anwenden möchte und nicht über technische Vorkenntnisse mit befehlsgesteuerten Programmen verfügt.

Dieses Buch ist deshalb nicht auf eine Leserschaft zugeschnitten, die an fortgeschrittenen Themen komputationaler Statistik interessiert ist und aufbauend auf Erfahrungen in Programmiersprachen auch R einsetzen möchte. Stattdessen soll es jenen den Einstieg in R ermöglichen, die zwar mit den statistischen Grundlagen vertraut sind, nicht aber mit deren Umsetzung mit Hilfe befehlsgesteuerter Software.

Die hier getroffene Auswahl an statistischen Verfahren orientiert sich an den Anforderungen der Psychologie, sollte damit aber auch die wichtigsten Auswertungsmethoden anderer Human- und Sozialwissenschaften abdecken. Vorgestellt wird die Umsetzung graphischer und deskriptiver Datenauswertung, nonparametrischer Verfahren, univariater linearer Modelle (Regression, Varianzanalysen) und ausgewählter multivariater Methoden. Dabei liegt der Fokus des Buches auf der Anwendung der Verfahren mit R, nicht aber auf der Vermittlung statistischer Grundkenntnisse – auf hierfür geeignete Literatur wird zu Beginn der Abschnitte jeweils hingewiesen. Eine nähere Erläuterung der Tests etwa hinsichtlich ihrer mathematischen Grundlagen, Anwendungsbereiche und Interpretation der Ergebnisse erfolgt dort, wo es für die Beschreibung der Testanwendung unabdingbar ist. In den meisten Beispielen wird davon ausgegangen, dass die vorliegenden Daten bereits geprüft sind und eine hohe Datenqualität vorliegt: Fragen der Einheitlichkeit etwa hinsichtlich der Codierung von Datum und Uhrzeit, potentiell unvollständige Datensätze, fehlerhaft eingegebene oder unplausible Daten sowie doppelte Werte oder Ausreißer sollen ausgeschlossen sein. Besondere Aufmerksamkeit wird jedoch in einem eigenen Abschnitt dem Thema fehlender Werte geschenkt.

Kapitel 1 bis 3 dienen der Einführung in den generellen Umgang mit R, in die zur Datenanalyse notwendigen Grundkonzepte sowie in die Syntax der Befehlsteuerung. Inhaltlich werden in Kap. 2 Methoden zur deskriptiven Datenauswertung behandelt, Kap. 3 befasst sich mit der Organisation vollständiger Datensätze, die Daten aus mehreren Variablen zusammenfassen. Das sich an Kap. 4 zur Verwaltung von Befehlen und Daten anschließende Kap. 5 stellt Hilfsmittel für die inferenzstatistischen Methoden bereit. Diese werden in Kap. 6 (nonparametrische Verfahren), 7 (Regression) und 8 (*t*-Tests und Varianzanalysen) behandelt. Einen Überblick über ausgewählte multivariate Verfahren gibt Kap. 9. Das Buch schließt mit Kap. 10 zum Erstellen von Diagrammen und einem kurzen Ausblick auf den Einsatz von R als Programmiersprache in Kap. 11. Die gewählte Reihenfolge der Themen ist bei der Lektüre keinesfalls zwingend einzuhalten. Da in der praktischen Auswertung statistische Analysen meist gemeinsam mit der Datenorganisation und graphischen Illustrationen durchzuführen sind, empfiehlt es sich vielmehr, bereits zu Beginn auch Kap. 4 und 10 selektiv parallel zu lesen.

Um die Ergebnisse von R-eigenen Auswertungsfunktionen besser nachvollziehbar zu machen, wird ihre Anwendung an vielen Stellen durch manuelle Kontrollrechnungen begleitet. Der gewählte Rechenweg soll dabei die aus der Statistik bekannten Formeln umsetzen, vernachlässigt aber zusätzliche Fragen, wie sie bei der Behandlung empirischer Datensätze auftreten, etwa wie mit fehlenden Werten umzugehen ist. Die eigene Umsetzung soll zudem zeigen, wie auch Testverfahren, für die zunächst keine vorbereitete Funktion vorhanden ist, mit elementaren Mitteln prinzipiell selbst umgesetzt werden können.

Im Buch wird an verschiedenen Punkten Bezug zu anderer Software genommen. Die folgenden dabei verwendeten Namen sind durch eingetragenes Warenzeichen der jeweiligen Eigentümer geschützt: Eclipse, Excel, Java, Linux, MacOS X, Mathematica, MySQL, MATLAB, Octave, ODBC, OpenGL, OpenOffice, Oracle, S, S+, SAS, SPSS, SQLite, Stata, TIBCO, Trellis, Unix, Windows.

Das Buch bezieht sich auf Version 2.10.1 von R. Die verwendeten Daten sowie alle Befehle des Buches können Sie unter dieser Adresse beziehen:

<http://www.uni-kiel.de/psychologie/dwoll/r/>

Korrekturen, Ergänzungen und Anregungen sind herzlich willkommen – bitte schicken Sie diese an dwoll@psychologie.uni-kiel.de.

Danksagung

Mein besonderer Dank gilt den Personen, die an der Entstehung des Buches in frühen und späteren Phasen mitgewirkt haben: Abschn. 1.1 bis 1.2.3 entstanden auf der Grundlage eines Manuskripts zur Begleitung der Methoden-Veranstaltungen von Dieter Heyer und Gisela Müller-Plath am Institut für Psychologie der Martin-Luther-Universität Halle-Wittenberg, denen ich für die Überlassung des Textes danken möchte. Zahlreiche Korrekturen und viele Verbesserungsvorschläge wurden dankenswerterweise von Erwin Grüner, Johannes Andres, Sabrina Flindt und Su-

sanne Wollschläger beigesteuert. Johannes Andres danke ich für seine ausführlichen Erläuterungen der statistischen Grundlagen. Die Entstehung des Buches wurde beständig durch die selbstlose Unterstützung von Heike Jores und Vincent van Houten begleitet. Schließlich ist den Entwicklern von R zu danken, die in freiwillig geleisteter Arbeit eine offene Umgebung zur statistischen Datenauswertung geschaffen haben, deren mächtige Funktionalität hier nur zu einem Bruchteil vermittelt werden kann.

Kiel, Germany
März 2010

Daniel Wollschläger

Inhaltsverzeichnis

1 Erste Schritte	1
1.1 Vorstellung	1
1.1.1 Was ist R?	1
1.1.2 Typographische Konventionen	3
1.1.3 Wo erhalte ich R und Dokumentation zu R?	4
1.1.4 Installation von R unter Windows	5
1.2 Grundlegende Elemente	6
1.2.1 Starten und beenden, die Konsole	6
1.2.2 Einstellungen	9
1.2.3 Umgang mit dem Workspace	10
1.2.4 Einfache Arithmetik	11
1.2.5 Funktionen mit Argumenten aufrufen	13
1.2.6 Hilfe-Funktionen	14
1.2.7 Zusatzpakete verwenden	15
1.3 Datenstrukturen: Klassen, Objekte, Datentypen	17
1.3.1 Objekte benennen	18
1.3.2 Zuweisungen an Objekte	19
1.3.3 Objekte ausgeben	20
1.3.4 Objekte anzeigen lassen und entfernen	20
1.3.5 Datentypen	20
1.3.6 Logische Werte, Operatoren und Verknüpfungen	22
2 Elementare Dateneingabe und -verarbeitung	25
2.1 Vektoren	25
2.1.1 Vektoren erzeugen	25
2.1.2 Elemente auswählen und verändern	26
2.1.3 Datentypen in Vektoren	28
2.1.4 Reihenfolge von Elementen kontrollieren	29
2.1.5 Elemente benennen	30
2.1.6 Elemente löschen	31
2.1.7 Rechenoperationen mit Vektoren	31
2.2 Logische Operatoren	35

2.2.1	Logische Operatoren zum Vergleich von Vektoren	35
2.2.2	Logische Indexvektoren	37
2.2.3	Werte ersetzen oder recodieren	38
2.3	Mengen	40
2.3.1	Duplizierte Werte behandeln	40
2.3.2	Mengenoperationen	41
2.3.3	Kombinatorik	42
2.4	Numerische Sequenzen und feste Wertefolgen erzeugen	44
2.4.1	Numerische Sequenzen erstellen	44
2.4.2	Wertefolgen wiederholen	46
2.5	Zufallszahlen und zufällige Reihenfolgen generieren	46
2.5.1	Zufällig aus einer Urne ziehen	47
2.5.2	Zufallszahlen aus bestimmten Verteilungen erzeugen . .	47
2.5.3	Unterauswahl einer Datenmenge bilden	48
2.5.4	Zufällige Reihenfolgen erstellen	49
2.6	Deskriptive Kennwerte numerischer Vektoren	49
2.6.1	Summen, Differenzen und Produkte	50
2.6.2	Extremwerte	51
2.6.3	Mittelwert, Median und Modalwert	52
2.6.4	Quartile, Quantile, Interquartilabstand	53
2.6.5	Varianz, Streuung, Schiefe und Wölbung	54
2.6.6	Kovarianz, Korrelation und Partialkorrelation	55
2.6.7	Funktionen auf geordnete Paare von Werten anwenden.	57
2.7	Gruppierungsfaktoren	57
2.7.1	Ungeordnete Faktoren	58
2.7.2	Faktorstufen hinzufügen, entfernen und zusammenfassen	60
2.7.3	Geordnete Faktoren	61
2.7.4	Reihenfolge von Faktorstufen	61
2.7.5	Faktoren nach Muster erstellen	63
2.7.6	Quantitative Variablen in Faktoren umwandeln	64
2.7.7	Funktionen getrennt nach Gruppen anwenden	64
2.8	Matrizen	66
2.8.1	Datentypen in Matrizen	67
2.8.2	Dimensionierung, Zeilen und Spalten	68
2.8.3	Elemente auswählen und verändern	70
2.8.4	Weitere Wege, um Elemente auszuwählen und zu verändern	71
2.8.5	Matrizen verbinden	72
2.8.6	Randkennwerte	73
2.8.7	Beliebige Funktionen auf Matrizen anwenden	73
2.8.8	Matrix zeilen- oder spaltenweise mit Kennwerten verrechnen	74
2.8.9	Kovarianz- und Korrelationsmatrizen	75
2.8.10	Matrizen sortieren	76

2.9	Lineare Algebra	78
2.9.1	Matrix-Algebra	78
2.9.2	Lineare Gleichungssysteme lösen	81
2.9.3	Norm und Abstand von Vektoren und Matrizen	81
2.9.4	Orthogonale Projektion	85
2.9.5	Kennwerte und Zerlegungen von Matrizen	86
2.10	Arrays	90
2.11	Häufigkeitsauszählungen	91
2.11.1	Einfache Tabellen absoluter und relativer Häufigkeiten .	91
2.11.2	Häufigkeiten natürlicher Zahlen	93
2.11.3	Iterationen zählen	94
2.11.4	Absolute, relative und bedingte relative Häufigkeiten in Kreuztabellen	94
2.11.5	Randkennwerte von Kreuztabellen	97
2.11.6	Kumulierte relative Häufigkeiten und Prozentrang	98
2.12	Codierung, Identifikation und Behandlung fehlender Werte	100
2.12.1	Fehlende Werte codieren und ihr Vorhandensein prüfen	101
2.12.2	Fehlende Werte ersetzen oder umcodieren	102
2.12.3	Behandlung fehlender Werte bei der Berechnung einfacher Kennwerte	103
2.12.4	Behandlung fehlender Werte in Matrizen	104
2.12.5	Behandlung fehlender Werte beim Sortieren von Daten	106
2.13	Zeichenketten verarbeiten	106
2.13.1	Objekte in Zeichenketten umwandeln	107
2.13.2	Zeichenketten erstellen und ausgeben	107
2.13.3	Zeichenketten manipulieren	109
2.13.4	Zeichenfolgen finden	111
2.13.5	Zeichenfolgen ersetzen	112
2.13.6	Zeichenketten als Befehl ausführen	113
2.14	Datum und Uhrzeit	113
2.14.1	Datum	114
2.14.2	Uhrzeit	115
3	Datensätze	117
3.1	Listen	117
3.1.1	Komponenten auswählen und verändern	117
3.1.2	Listen mit mehreren Ebenen	120
3.2	Datensätze	121
3.2.1	Datentypen in Datensätzen	123
3.2.2	Elemente auswählen und verändern	124
3.2.3	Datensätze in den Suchpfad einfügen	126
3.2.4	Namen von Variablen und Beobachtungen	127
3.2.5	Variablen einem Datensatz hinzufügen oder aus diesem entfernen	128

3.2.6	Teilmengen von Daten auswählen	130
3.2.7	Organisationsform von Daten ändern	133
3.2.8	Organisationsform eines Datensatzes ändern	134
3.2.9	Datensätze teilen	138
3.2.10	Datensätze zusammenfügen	139
3.2.11	Funktionen auf Variablen anwenden	143
3.2.12	Funktionen für mehrere Variablen anwenden	145
3.2.13	Funktionen getrennt nach Gruppen anwenden	146
3.2.14	Doppelte und fehlende Werte	147
3.2.15	Datensätze sortieren	149
4	Befehle und Daten verwalten	151
4.1	Befehlssequenzen im Editor bearbeiten	151
4.2	Daten importieren und exportieren	153
4.2.1	Daten in der Konsole einlesen	153
4.2.2	Daten im Editor eingeben	154
4.2.3	Im Textformat gespeicherte Daten	155
4.2.4	R-Objekte	157
4.2.5	Daten mit anderen Programmen austauschen	157
5	Hilfsmittel für die Inferenzstatistik	163
5.1	Lineare Modelle formulieren	163
5.2	Funktionen von Zufallsvariablen	165
5.2.1	Dichtefunktionen	166
5.2.2	Verteilungsfunktionen	167
5.2.3	Quantilfunktionen	168
5.3	Behandlung fehlender Werte in inferenzstatistischen Tests	169
6	Nonparametrische Methoden	171
6.1	Anpassungstests	171
6.1.1	Binomialtest	172
6.1.2	Test auf Zufälligkeit (Runs Test)	174
6.1.3	Kolmogorov-Smirnov-Anpassungstest	176
6.1.4	χ^2 -Test auf eine feste Verteilung	179
6.1.5	χ^2 -Test auf eine Verteilungsklasse	180
6.2	Analyse von gemeinsamen Häufigkeiten kategorialer Variablen	181
6.2.1	χ^2 -Test auf Unabhängigkeit	182
6.2.2	χ^2 -Test auf Gleichheit von Verteilungen	183
6.2.3	χ^2 -Test für mehrere Auftretenswahrscheinlichkeiten	184
6.2.4	Fishers exakter Test auf Unabhängigkeit	185
6.2.5	Fishers exakter Test auf Gleichheit von Verteilungen	186
6.2.6	Kennwerte für 2×2 Kontingenztafeln	187
6.3	Maße für Zusammenhang und Übereinstimmung kategorialer Daten	189

6.3.1	Zusammenhang ordinaler Variablen: Spearmans ρ und Kendalls τ	189
6.3.2	Weitere Zusammenhangsmaße: φ , Cramérs V , Kontingenzkoeffizient, Goodman und Kruskals γ , Somers' d , ICC, r_{WG}	191
6.3.3	Inter-Rater-Übereinstimmung: Cohens κ , Fleiss' κ , Kendalls W	192
6.4	Tests auf Übereinstimmung von Verteilungen	199
6.4.1	Vorzeichen-Test	199
6.4.2	Wilcoxon-Vorzeichen-Rang-Test für eine Stichprobe	200
6.4.3	Wald-Wolfowitz-Test für zwei Stichproben	203
6.4.4	Kolmogorov-Smirnov-Test für zwei Stichproben	203
6.4.5	Wilcoxon-Rangsummen-Test/Mann-Whitney-U-Test für zwei unabhängige Stichproben	205
6.4.6	Wilcoxon-Test für zwei abhängige Stichproben	207
6.4.7	Kruskal-Wallis-H-Test für unabhängige Stichproben	207
6.4.8	Friedman-Rangsummen-Test für abhängige Stichproben	208
6.4.9	Cochran-Q-Test für abhängige Stichproben	210
6.4.10	Bowker-Test für zwei abhängige Stichproben	211
6.4.11	McNemar-Test für zwei abhängige Stichproben	213
6.4.12	Stuart-Maxwell-Test für zwei abhängige Stichproben	214
7	Korrelation und Regressionsanalyse	217
7.1	Test auf Korrelation	217
7.2	Einfache lineare Regression	218
7.2.1	Regressionsanalyse	221
7.2.2	Regressionsdiagnostik	224
7.2.3	Vorhersage bei Anwendung auf andere Daten	225
7.2.4	Partialkorrelation	227
7.2.5	Kreuzvalidierung	227
7.3	Multiple lineare Regression	228
7.3.1	Modell verändern	230
7.3.2	Modell auswählen	231
7.3.3	Auf Multikollinearität prüfen	232
7.4	Logistische Regression	233
8	Parametrische Tests für Dispersions- und Lageparameter von Verteilungen	239
8.1	Tests auf Varianzhomogenität	239
8.1.1	F -Test auf Varianzhomogenität bei zwei Stichproben	239
8.1.2	Fligner-Killeen-Test und Bartlett-Test	241
8.1.3	Levene-Test	241
8.2	t -Tests	242
8.2.1	t -Test für eine Stichprobe	242

8.2.2	<i>t</i> -Test für zwei unabhängige Stichproben	243
8.2.3	<i>t</i> -Test für zwei abhängige Stichproben	245
8.3	Einfaktorielle Varianzanalyse (CR- <i>p</i>)	246
8.3.1	Regression und Varianzanalyse als lineare Modelle	246
8.3.2	Auswertung mit <code>oneway.test()</code>	248
8.3.3	Auswertung mit <code>aov()</code>	249
8.3.4	Graphische Prüfung der Voraussetzungen	251
8.3.5	Auswertung mit <code>anova()</code>	252
8.3.6	Einzelvergleiche (Kontraste)	254
8.4	Einfaktorielle Varianzanalyse mit abhängigen Gruppen (RB- <i>p</i>)	261
8.4.1	Univariat formulierte Auswertung mit <code>aov()</code>	262
8.4.2	Zirkularität der Kovarianzmatrix prüfen	264
8.4.3	Multivariat formulierte Auswertung mit <code>Anova()</code>	266
8.4.4	Einzelvergleiche (Kontraste)	268
8.5	Zweifaktorielle Varianzanalyse (CRF- <i>pq</i>)	268
8.5.1	Auswertung mit <code>aov()</code>	269
8.5.2	Quadratsummen vom Typ I und III	271
8.5.3	Beliebige a-priori Kontraste	275
8.5.4	Beliebige post-hoc Kontraste nach Scheffé	278
8.6	Zweifaktorielle Varianzanalyse mit zwei Intra-Gruppen Faktoren (RBF- <i>pq</i>)	279
8.6.1	Univariat formulierte Auswertung mit <code>aov()</code>	280
8.6.2	Zirkularität der Kovarianzmatrizen prüfen	283
8.6.3	Multivariat formulierte Auswertung mit <code>Anova()</code>	284
8.6.4	Einzelvergleiche (Kontraste)	285
8.7	Zweifaktorielle Varianzanalyse mit Split-Plot-Design (SPF- <i>p · q</i>)	286
8.7.1	Univariat formulierte Auswertung mit <code>aov()</code>	286
8.7.2	Voraussetzungen und Prüfen der Zirkularität	287
8.7.3	Multivariat formulierte Auswertung mit <code>Anova()</code>	288
8.7.4	Einzelvergleiche (Kontraste)	289
8.7.5	Erweiterung auf dreifaktorielles SPF- <i>p · qr</i> Design	290
8.7.6	Erweiterung auf dreifaktorielles SPF- <i>pq · r</i> Design	291
8.8	Kovarianzanalyse	292
8.8.1	Test der Effekte von Gruppenzugehörigkeit und Kovariaten	292
8.8.2	Beliebige a-priori Kontraste	297
8.8.3	Beliebige post-hoc Kontraste nach Scheffé	299
8.9	Power und notwendige Stichprobengrößen berechnen	299
8.9.1	Binomialtest	300
8.9.2	<i>t</i> -Test	301
8.9.3	Einfaktorielle Varianzanalyse	304
9	Multivariate Verfahren	309
9.1	Multivariate Multiple Regression	309

9.2	Hauptkomponentenanalyse	311
9.3	Faktorenanalyse	317
9.4	Multidimensionale Skalierung	323
9.5	Hotellings T^2	325
9.5.1	Test für eine Stichprobe	325
9.5.2	Test für zwei Stichproben	327
9.6	Multivariate Varianzanalyse (MANOVA)	329
9.6.1	Einfaktorielle MANOVA	329
9.6.2	Mehr faktorielle MANOVA	332
10	Diagramme erstellen	335
10.1	Graphik-Devices	335
10.1.1	Aufbau und Verwaltung von Graphik-Devices	335
10.1.2	Graphiken speichern	337
10.2	Streu- und Liniendiagramme	339
10.2.1	Streudiagramme mit <code>plot()</code>	339
10.2.2	Datenpunkte eines Streudiagramms identifizieren	342
10.2.3	Streudiagramme mit <code>matplotlib()</code>	342
10.3	Diagramme formatieren	343
10.3.1	Graphikelemente formatieren	343
10.3.2	Farben spezifizieren	346
10.3.3	Achsen formatieren	348
10.4	Säulen- und Punktdiagramme	348
10.4.1	Einfache Säulendiagramme	349
10.4.2	Gruppierte und gestapelte Säulendiagramme	350
10.4.3	Cleveland Dotchart	352
10.5	Elemente einem bestehenden Diagramm hinzufügen	354
10.5.1	Koordinaten in einem Diagramm identifizieren	354
10.5.2	Punkte	355
10.5.3	Linien	355
10.5.4	Polygone	358
10.5.5	Funktionsgraphen	362
10.5.6	Text und mathematische Formeln	362
10.5.7	Achsen	365
10.5.8	Fehlerbalken	366
10.6	Verteilungsdiagramme	369
10.6.1	Histogramm und Schätzung der Dichtefunktion	369
10.6.2	Stamm-Blatt-Diagramm	371
10.6.3	Box-Whisker-Plot	372
10.6.4	<code>stripchart()</code>	374
10.6.5	Quantil-Quantil-Diagramme	375
10.6.6	Empirische kumulative Häufigkeitsverteilungen	376
10.6.7	Kreisdiagramm	377
10.6.8	Gemeinsame Verteilung zweier Variablen	378

10.7	Datenpunkte interpolieren	381
10.7.1	Lineare Interpolation und polynomiale Glätter	381
10.7.2	Splines	382
10.8	Multivariate Daten visualisieren	383
10.8.1	Höhenlinien und variable Datenpunktssymbole	384
10.8.2	Dreidimensionale Gitter und Streudiagramme	387
10.8.3	Simultane Darstellung mehrerer Diagramme gleichen Typs	388
10.8.4	Matrix aus Streudiagrammen	390
10.9	Mehrere Diagramme in einem Graphik-Device darstellen	393
10.9.1	<code>layout()</code>	393
10.9.2	<code>par(mfrow, mfcoll, fig)</code>	395
10.9.3	<code>split.screen()</code>	398
11	R als Programmiersprache	401
11.1	Eigene Funktionen erstellen	401
11.1.1	Funktionskopf mit Argumentliste	402
11.1.2	Funktionsrumpf mit Befehlen	403
11.1.3	Rückgabewert	403
11.1.4	Eigene Funktionen verwenden	404
11.1.5	Generische Funktionen	405
11.2	Kontrollstrukturen	407
11.2.1	Bedingungen zur Fallunterscheidung prüfen	407
11.2.2	Schleifen	409
Literaturverzeichnis	413
Sachverzeichnis	419

Kapitel 1

Erste Schritte

1.1 Vorstellung

1.1.1 Was ist R?

R ist ein freies und kostenloses Programm paket zur statistischen Datenverarbeitung (Ihaka und Gentleman, 1996; R Development Core Team, 2009b): es integriert eine Vielzahl von Möglichkeiten, um Daten speichern, organisieren, transformieren, auswerten und visualisieren zu können. Dabei bezeichnet R sowohl das Programm selbst als auch die Sprache, in der die Auswertungsbefehle geschrieben werden.¹ In R bestehen Auswertungen nämlich aus einer Abfolge von Befehlen in Textform, die der Benutzer unter Einhaltung einer bestimmten Syntax selbst einzugeben hat. Jeder Befehl stellt dabei einen eigenen Auswertungsschritt dar, wobei eine vollständige Datenanalyse meist durch die sequentielle Abfolge vieler solcher Schritte gekennzeichnet ist. So könnten Daten zunächst aus einer Datei gelesen und zwei Variablen zu einer neuen verrechnet werden, ehe eine Teilmenge von Beobachtungen agewählt und mit ihr ein statistischer Test durchgeführt wird, dessen Ergebnisse im Anschluss graphisch aufzubereiten sind.

Während in Programmen, die über eine graphische Benutzeroberfläche gesteuert werden, die Auswahl von vorgegebenen Menüpunkten einen wesentlichen Teil der Arbeit ausmacht, ist es in R die aktive Produktion von Befehlsausdrücken. Diese Eigenschaft ist gleichzeitig ein wesentlicher Vorteil wie auch eine Herausforderung beim Einstieg in die Arbeit mit R. Folgende Aspekte lassen R für die Datenauswertung besonders geeignet erscheinen:

- Die befehlsgesteuerte Arbeitsweise erhöht durch die Wiederverwendbarkeit von Befehlssequenzen für häufig wiederkehrende Arbeitsschritte langfristig die Effizienz. Zudem steigt die Zuverlässigkeit von Analysen, wenn über die Zeit

¹ Genauer gesagt ist R eine eigenständige Implementierung der Sprache S, deren kommerzielle Umsetzung das Programm S+ ist (TIBCO Software Inc., 2008). R teilt damit weitgehend die Syntax von S, besitzt aber einen erweiterten Funktionsumfang. Sich auf S beziehende Texte und Auswertungsbeispiele lassen sich weitestgehend direkt für R nutzen.

bewährte Bausteine immer wieder verwendet und damit Fehlerquellen ausgeschlossen werden.

- Die Möglichkeit zur Weitergabe von Befehlssequenzen an Dritte (gemeinsam mit empirischen Datensätzen) kann die Auswertung für andere transparent sowie prüf- und replizierbar im Sinne einer höheren Auswertungsobjektivität machen.
- Als *Open Source*-Programm² wird R beständig von vielen Personen evaluiert, weiterentwickelt und den praktischen Erfordernissen angepasst. Da der Quelltext frei verfügbar ist und zudem viele Auswertungsfunktionen ihrerseits in R geschrieben sind, ist die Art der Berechnung statistischer Kennwerte vollständig transparent. Sie kann damit vom Benutzer bei Interesse analysiert, aber auch auf Richtigkeit kontrolliert werden.
- Datensätze können unter Einhaltung vorgegebener Wertebereiche und anderer Randbedingungen leicht mit Zufallswerten simuliert werden. Dies lässt es zu, Auswertungsschritte bereits vor der eigentlichen Datenerhebung zu entwickeln und daraufhin zu prüfen, ob sie sich zur Beantwortung einer Fragestellung eignen.
- Dank seines modularen Aufbaus bietet R die Möglichkeit, die Basisfunktionalität für spezielle Anwendungszwecke durch eigenständig entwickelte Zusatzkomponenten zu erweitern, von denen bereits mehrere hundert frei verfügbar sind.
- Auch ohne tiefergehende Programmierkenntnisse lassen sich in R eigene Funktionen erstellen und so Auswertungen flexibel an individuelle Anforderungen anpassen. Da Funktionen in derselben Syntax wie normale Auswertungen erstellt werden, sind dafür nur wenige Zusatzschritte, nicht aber eine eigene Makrosprache zu erlernen.

R hält für Anwender allerdings auch Hürden bereit, insbesondere für Einsteiger, die nur über Erfahrungen mit Programmen verfügen, die über eine graphische Benutzeroberfläche bedient werden³:

- Die einzelnen Befehle und ihre Syntax zu erlernen, erfordert die Bereitschaft zur Einübung. Es muss zunächst ein Grundverständnis für die Arbeitsabläufe sowie ein gewisser „Wortschatz“ häufiger Funktionen und Konzepte geschaffen werden, ehe Daten umfassend ausgewertet werden können. Sind jedoch einmal einzelne Befehlsbausteine aus einfachen Auswertungen vertraut, lassen sie sich leicht Schritt für Schritt zu komplexen Analysen zusammenstellen.
- Im Gegensatz zu Programmen mit graphischer Benutzeroberfläche müssen Befehle aktiv erinnert werden. Es stehen keine Gedächtnissstützen im Sinne von Elementen einer graphischen Umgebung zur Verfügung, die interaktiv mögliche

² Der Open Source-Programm zugrundeliegende Quelltext ist frei erhältlich, zudem darf die Software frei genutzt, verbreitet und verändert werden. Genaueres erläutert der Befehl `licence()`.

³ Als technische Beschränkung bei der Analyse extrem großer Datenmengen besteht gegenwärtig im Gegensatz zu S+ noch das Problem, dass Datensätze zur Bearbeitung im Arbeitsspeicher vorgehalten werden müssen. Dies schränkt die Größe von praktisch auswertbaren Datensätzen ein, vgl. `help(Memory)`. Ansätze, diese Einschränkung aufzuheben, befinden sich in der Entwicklung. Generell profitiert die Geschwindigkeit der Datenauswertung von einem großzügig dimensionierten Hauptspeicher.

Vorgehensweisen anbieten und zu einem Wiedererkennen führen könnten. Dies betrifft sowohl die Auswahl von geeigneten Analysen wie auch die konkrete Anwendung bestimmter Verfahren.

- Im Gegensatz zur natürlichen Sprache ist die Befehlssteuerung nicht fehlertolerant, Befehle müssen also exakt richtig eingegeben werden. Dies kann zu Beginn der Beschäftigung mit R frustrierend und auch zeitraubend sein, weil schon vermeintlich unwesentliche Fehler verhindern, dass Befehle ausgeführt werden. Im Fall falsch eingegebener Befehle liefert R aber Fehlermeldungen, die Rückschlüsse auf die Ursache erlauben können. Zudem nimmt die Zahl der Syntaxfehler im Zuge der Beschäftigung mit R von allein deutlich ab.
- Die Ergebnisse von Auswertungsschritten, etwa der Anwendung inferenzstatistischer Testverfahren, werden von R in Textform ohne besondere Formatierungen ausgegeben. Um Ergebnisse in eigene Publikationen übernehmen zu können, sind daher alle Formatierungsarbeiten selbst durchzuführen.⁴

Es existieren verschiedene Ansätze, um die Funktionalität von R auch über eine graphische Benutzeroberfläche zugänglich zu machen (Grosjean, 2006). Zu unterscheiden sind dabei zum einen solche Programme, die Analysefunktionen über graphische Menüs und damit ohne Befehlseingabe nutzbar machen, etwa das Statistiklabor (Schlittgen, 2005), Rcmdr (Fox, 2005), Rattle (Williams, 2010) oder RKWard (Friedrichsmeier et al., 2009). Zum anderen sind Programme verfügbar, die zwar nur eine graphische Umgebung zur Befehlseingabe bieten, darin aber weit komfortabler als der in R mitgelieferte Texteditor sind (vgl. Abschn. 4.1), z. B. Tinn-R (Faria et al., 2010) oder die Entwicklungsumgebung Eclipse mit StatET Plugin (Wahlbrink, 2009).

1.1.2 Typographische Konventionen

Zur besseren Lesbarkeit sollen zunächst einige typographische Konventionen für die Notation vereinbart werden. Um zwischen den Befehlen und Ausgaben von R sowie der zugehörigen Beschreibung innerhalb dieses Textes unterscheiden zu können, werden Befehle und Ausgaben im Schrifttyp Schreibmaschine dargestellt. Eine Befehlszeile mit zugehöriger Ausgabe könnte also z. B. so aussehen:

```
> 1 + 1
[1] 2
```

Die erste Zeile bezeichnet dabei die Eingabe des Anwenders, die zweite Zeile die (in diesem Text bisweilen leicht umformatierte) Ausgabe von R. Fehlen Teile der Ausgabe im Text, ist dies mit . . . als Auslassungszeichen angedeutet. Zeilenumbrüche innerhalb von R-Befehlen sind durch Pfeile in der Form Befehl Anfang ↴ Fortsetzung gekennzeichnet, um deutlich zu machen, dass die getrennten Teile

⁴ Anwender von OpenOffice Writer oder L^AT_EX erhalten jedoch Unterstützung durch die `Sweave()` Funktion (Leisch, 2002) und das Paket `odfWeave` (Kuhn und Weaston, 2009, vgl. Abschn. 1.2.7).

unmittelbar zusammen gehören. Platzhalter, wie z. B. `(Dateiname)`, die für einen Typ von Objekten stehen (hier Dateien) und mit einem beliebigen konkreten Objekt dieses Typs gefüllt werden können, werden in stumpfwinklige Klammern `()` gesetzt.

Schaltflächen des Programmfensters werden in serifloser Schrift dargestellt. Bei Vorgängen, die eine aufeinanderfolgende Aktivierung mehrerer Schaltflächen erfordern, werden die Schaltflächen in der Reihenfolge genannt, in der sie angeklickt werden sollen, wobei die Art des Klicks nachgestellt wird. Die linke Maustaste wird dabei durch einen Doppelpunkt : dargestellt, die rechte durch ein Semikolon;. Internet-URLs, Tastenkombinationen sowie Dateien und Verzeichnisse werden im Schrifttyp Schreibmaschine dargestellt, wobei die Unterscheidung zu R-Befehlen aus dem Kontext hervorgehen sollte.

1.1.3 Wo erhalte ich R und Dokumentation zu R?

Zentrale Anlaufstelle für Nachrichten über die Entwicklung von R, für den Download des Programms selbst, für Zusatzpakete sowie für frei verfügbare Literatur ist die R-Projektseite im WWW:

<http://www.r-project.org/>

Um Zugang zur dort erhältlichen Literatur, zur Installationsdatei des Programms oder zu den Zusatzpaketen zu bekommen, folgt man dem Verweis Download, Packages/CRAN, der auf eine Übersicht von CRAN-Servern verweist, von denen die Dateien erhältlich sind.⁵

Häufige Fragen zu R sowie speziell zur Verwendung von R unter Windows werden in den FAQs (Frequently Asked Questions) beantwortet (Hornik, 2009; Ripley und Murdoch, 2009). Für individuelle Fragen existiert auch die Mailing-Liste R-help, deren Adresse auf der Projektseite unter R Project/Mailing Lists genannt wird. Bevor sie für eigene Hilfegesuche genutzt wird, sollte aber eine umfangreiche selbständige Recherche vorausgehen, zudem sind die Hinweise des Posting-Guides unter

<http://www.r-project.org/posting-guide.html>

zu beherzigen. Die Suche innerhalb von Beiträgen auf dieser Liste, sowie innerhalb von Funktionen und der Hilfe erleichtern die folgenden auf R-Inhalte spezialisierten Suchseiten:

<http://www.rseek.org/>
<http://search.r-project.org/>

Unter dem Link Documentation/Manuals auf der Projektseite von R findet sich die vom R Development Core Team herausgegebene offizielle Dokumentation in

⁵ CRAN steht für „Comprehensive R Archive Network“ und bezeichnet ein Netzwerk von mehreren sog. Mirror-Servern mit gleichem Angebot, die die aktuellen Dateien und Informationen zu R zur Verfügung stellen. Aus der Liste der verfügbaren Server sollte einer nach dem Kriterium der geographischen Nähe ausgewählt werden.

Form von Handbüchern im PDF- und im HTML-Format. Sie liefern einerseits einen umfassenden, aber sehr konzisen Überblick über R selbst (Venables, Smith und the R Development Core Team, 2009) und befassen sich andererseits mit Spezialthemen wie dem Datenaustausch mit anderen Programmen (vgl. Abschn. 4.2). Diese Handbücher sind in einer vollständigen Installation von R enthalten und lassen sich unter Windows über das Menü des R Programmfensters mittels Hilfe: Handbücher (PDF) erreichen. Weitere, von Anwendern beigesteuerte Literatur zu R findet sich auf der Projektseite unter dem Verweis Documentation/Other.

Darüber hinaus existiert eine Reihe von Büchern mit unterschiedlicher inhaltlicher Ausrichtung, etwa zur anwendungsorientierten Einführung in R anhand grundlegender statistischer Themen (Dalgaard, 2008; Mairdonald und Braun, 2007; Verzani, 2005) oder anhand fortgeschritten Anwendungen (Everitt und Hothorn, 2006; Venables und Ripley, 2002). Wie jeweils bestimmte statistische Auswertungen in R umgesetzt werden können, behandeln eigene Bücher, etwa zu multivariaten Verfahren (Everitt, 2005), gemischten Modellen (Faraway, 2006; Pinheiro und Bates, 2000), robusten Methoden (Jurečková und Picek, 2006) und verschiedenen Typen von Regression (Faraway, 2004; Fox, 2002). Das in diesem Text nur angedeutete Gebiet der Programmierung mit R thematisieren Chambers (2008) und Ligges (2009) ausführlich. Eine umfassende, laufend aktualisierte Literaturübersicht bietet die R-Projektseite unter Documentation/Books.

1.1.4 Installation von R unter Windows

Die folgenden Ausführungen beziehen sich auf die Installation des R-Basispaket unter Windows XP, vgl. dazu auch die offizielle Installationsanleitung (R Development Core Team, 2009d) und die Windows-FAQ (Ripley und Murdoch, 2009).⁶ Um die Installationsdatei herunterzuladen, folgt man von der R-Projektseite dem Verweis Download/CRAN und wählt einen CRAN-Server. Über Download and Install R/Windows: base gelangt man zum Verzeichnis mit der Installationsdatei, die mit R-2.10.1-win32.exe; Ziel speichern unter unter Angabe des Zielordners auf dem eigenen Rechner gespeichert werden kann.⁷

Um R zu installieren, ist die gespeicherte Installationsdatei R-<Version>-win32.exe auszuführen und den Aufforderungen des Setup-Assistenten zu folgen.

⁶ Auch bei der Beschreibung von Elementen der graphischen Oberfläche von R wird im folgenden von einer deutschsprachigen Installation unter Windows ausgegangen. Für eine ähnliche, aber plattformunabhängige Oberfläche vgl. JGR (Helbig et al., 2005). Abgesehen von der Oberfläche bestehen nur unwesentliche Unterschiede zwischen der Arbeit mit R unter verschiedenen Betriebssystemen.

⁷ R-2.10.1-win32.exe ist die im März 2010 aktuelle Version von R für Windows. 2.10.1 ist die Versionsnummer. Wenn Sie R zu einem späteren Zeitpunkt herunterladen, ist u. U. eine neuere Version verfügbar. Es sind dann leichte, für den Benutzer jedoch in den allermeisten Fällen nicht merkliche Abweichungen zur in diesem Manuskript beschriebenen Arbeitsweise von Funktionen möglich.

Wenn keine Änderungen am Installationsordner von R vorgenommen wurden, sollte R daraufhin im Verzeichnis C:\Programme\R\R-<Version>\ installiert und eine zugehörige Verknüpfung auf dem Desktop vorhanden sein.

1.2 Grundlegende Elemente

1.2.1 Starten und beenden, die Konsole

Nach der Installation lässt sich R unter Windows über die Datei rgui.exe im Unterordner bin/des Programmordners starten, aber auch über die bei der Installation erstellte Verknüpfung mit dieser Datei auf dem Desktop. Durch den Start öffnen sich zwei Fenster, ein großes, das Programmfenster, und darin ein kleineres, die sog. Konsole. Auf der Konsole werden im interaktiven Wechsel von eingegebenen Befehlen und der Ausgabe von R die Verarbeitungsschritte vorgenommen.⁸ Hier erscheint nach dem Start unter einigen Hinweisen hinter dem sog. Prompt-Zeichen > ein roter Cursor, um zu signalisieren, dass Befehle vom Benutzer entgegengenommen und verarbeitet werden können. Das Ergebnis einer Berechnung wird in der auf das Prompt-Zeichen folgenden Zeile ausgegeben, nachdem ein Befehl mit der Return Taste beendet wurde. Dabei wird in eckigen Klammern stets zunächst die laufende Nummer des ersten in der jeweiligen Zeile angezeigten Objekts aufgeführt.

```
> 1 + 1
[1] 2
>
```

Pro Zeile wird im Normalfall ein Befehl eingegeben. Sollen mehrere Befehle in eine Zeile geschrieben werden, so sind sie durch ein Semikolon; zu trennen. Das Symbol # markiert den Beginn eines sog. Kommentars und verhindert, dass der dahinter auftauchende Text in derselben Zeile als Befehl interpretiert wird.

War die Eingabe fehlerhaft, erscheint statt einer Ausgabe eine Fehlermeldung mit einer Beschreibung der Ursache. Anlass für Fehler sind häufig fehlende Kommata, nicht paarweise verwendete Klammern, Klammern des falschen Typs oder falsch geschriebene Objekt- bzw. Funktionsnamen. Befehle können auch Warnungen verursachen, die die Auswertung zwar nicht wie Fehler verhindern, aber immer daraufhin untersucht werden sollten, ob sie Symptom für falsche Berechnungen sind.

Das folgende Beispiel soll eine kleine Auswertung in Form mehrerer aufeinanderfolgender Arbeitsschritte demonstrieren. An dieser Stelle ist es dabei nicht wichtig, schon zu verstehen, wie die einzelnen Befehle funktionieren. Vielmehr soll das Beispiel zeigen, wie eine realistische Sequenz von Auswertungsschritten

⁸ Für automatisierte Auswertungen vgl. Abschn. 4.1. Die Ausgabe lässt sich mit der sink() Funktion entweder gänzlich oder im Sinne eines Protokolls aller Vorgänge als Kopie in eine Datei umleiten (Argument split=TRUE). Ebenso lassen sich alle Konsoleninhalte (eingegebene Befehle und Output) über das Menü mit Datei: Speichern in Datei in einer Textdatei speichern. Befehle des Betriebssystems sind mit shell("Befehl") ausführbar, so können etwa die Netzwerkverbindungen mit shell("netstat") angezeigt werden.

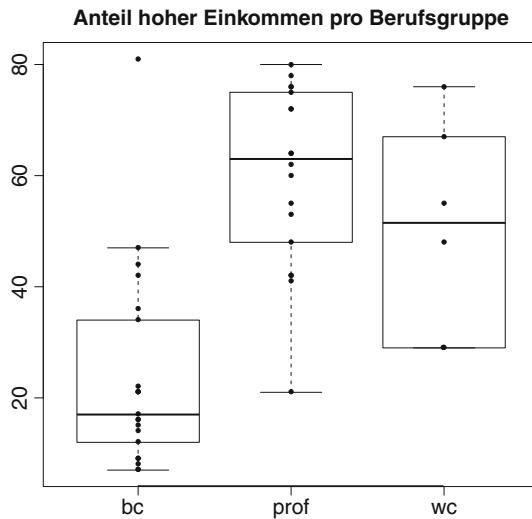


Abb. 1.1 Anteil der Angehörigen eines Berufs mit hohem Einkommen in Abhängigkeit vom Typ des Berufs. Daten des Datensatzes Duncan

inkl. der von R erzeugten Ausgabe aussehen kann. Die Analyse bezieht sich auf den Datensatz Duncan, der Daten von Personen jeweils eines Berufs speichert. Ein Beruf kann dabei zur Gruppe bc (Blue Collar), wc (White Collar) oder prof (Professional) gehören – die Stufen der Variable type. Erhoben wurde der prozentuale Anteil von Personen eines Berufs mit einem hohen Einkommen (Variable income), einem hohen Ausbildungsgrad (education) und einem hohen Prestige (prestige). Die Daten je eines Berufs befinden sich in einer Zeile, die Daten einer Variable in einer Spalte des Datensatzes.

Zunächst sollen wichtige deskriptive Kennwerte von income in nach Gruppen getrennten Boxplots dargestellt und dabei auch die Rohdaten selbst abgebildet werden (Abb. 1.1). Es folgt die Berechnung der Gruppenmittelwerte für education und die Korrelationsmatrix der drei Abhängigen Variablen. Als inferenzstatistische Auswertung schließt sich eine Varianzanalyse mit der Abhängigen Variable prestige und dem Gruppierungsfaktor type an. Im folgenden t-Test der Variable education sollen nur die Gruppen bc und wc berücksichtigt werden.

```
> data(Duncan, package="car")      # lade Datensatz Duncan
> attach(Duncan)                 # mache Variablen aus Duncan bekannt
> head(Duncan)                  # gebe die ersten Zeilen von Duncan aus
   type income education prestige
accountant prof    62      86     82
pilot       prof    72      76     83
architect   prof    75      92     90
author      prof    55      90     76
chemist     prof    64      86     90
minister    prof    21      84     87
```

```
#####
# nach Gruppen getrennte Boxplots und Rohdaten der Variable income
> boxplot(income ~ type, main="Anteil hoher Einkommen pro Berufsgruppe")
> stripchart(income ~ type, pch=20, vert=TRUE, add=TRUE)

#####
# Gruppenmittelwerte von education
> tapply(education, type, mean)
      bc      prof       wc
25.33333 81.33333 61.50000

#####
# Korrelationsmatrix der Variablen income, education, prestige
> cor(cbind(income, education, prestige))
      income education prestige
income   1.0000000 0.7245124 0.8378014
education 0.7245124 1.0000000 0.8519156
prestige  0.8378014 0.8519156 1.0000000

#####
# einfaktorielle Varianzanalyse - AV prestige, UV type
> anova(lm(prestige ~ type))
Analysis of Variance Table
Response: prestige
  Df Sum Sq Mean Sq F value    Pr(>F)
type     2   33090   16545   65.571 1.207e-13 ***
Residuals 42  10598     252

```

```
#####
# nur Berufe der Gruppen bc oder wc auswählen
> BCandWC <- (type == "bc") | (type == "wc")

#####
# linksseitiger t-Test der Variable education
# für die zwei ausgewählten Berufsgruppen bc und wc
> t.test(education ~ type, alternative="less", subset=BCandWC)
Welch Two Sample t-test
data: education by type
t = -4.564, df = 5.586, p-value = 0.002293
alternative hypothesis: true difference in means is less than 0
95 percent confidence interval:
 -Inf -20.56169
sample estimates:
mean in group bc  mean in group wc
25.33333          61.50000

> detach(Duncan) # entferne Variablen des Datensatzes
```

Einzelne Befehlsbestandteile müssen meist nicht unbedingt durch Leerzeichen getrennt werden, allerdings erhöht dies die Übersichtlichkeit und ist mitunter auch erforderlich. Insbesondere das Zuweisungssymbol `<-` sollte stets von Leerzeichen

umschlossen sein. Wenn ein Befehl die sichtbare Zeilenlänge überschreitet, verlängert R die Zeile automatisch, schränkt aber auf diese Weise die Sichtbarkeit des Befehlsbeginns ein. Ein langer Befehl kann darum auch durch Drücken der Return Taste in der nächsten Zeile fortgesetzt werden, solange er syntaktisch nicht vollständig ist – z. B. wenn eine geöffnete Klammer noch nicht geschlossen wurde. Es erscheint dann ein + zu Beginn der folgenden Zeile, um zu signalisieren, dass sie zur vorangehenden gehört. Falls dieses + ungewollt in der Konsole auftaucht, muss der Befehl entweder vervollständigt oder mit der ESC Taste (Windows) bzw. der CTRL+C Tastenkombination (Unix/Linux) abgebrochen werden.

```
> 2 * (4  
+ -5)  
[1] -2
```

In der Konsole können Befehle mit der Maus markiert und über die Tastenkombination Strg+C in die Zwischenablage hinein bzw. mit Strg+V aus der Zwischenablage heraus in die Befehlszeile hinein kopiert werden.

Eine weitere Funktionalität der Konsole ist die sog. Tab Vervollständigung. Wird der Anfang eines Befehlsnamens in die Konsole eingegeben und dann die Tabulator-Taste gedrückt, zeigt R diejenigen Funktionen an, die den begonnenen Befehl vervollständigen könnten. Es existieren noch weitere vergleichbare Kurzbefehle, über die das Menü unter Hilfe: Konsole informiert.

R wird entweder über den Befehl q() in der Konsole, über den Menüpunkt Datei: Beenden oder durch Schließen des Programmfensters beendet. Zuvor erscheint die Frage, ob man den sog. Workspace, also alle erstellten Daten, speichern möchte (vgl. Abschn. 1.2.3). Mit Ja wird der Workspace im aktuellen Arbeitsverzeichnis gespeichert und R beendet, mit Abbrechen kehrt man zu R zurück.

1.2.2 Einstellungen

R wird immer in einem sog. Arbeitsverzeichnis ausgeführt, das durch getwd() ausgegeben werden kann. In der Voreinstellung handelt es sich um das Heimverzeichnis des Benutzers. Alle während einer Sitzung gespeicherten Dateien werden, sofern nicht explizit anders angegeben, in eben diesem Verzeichnis abgelegt. Wenn Informationen aus Dateien geladen werden sollen, greift R ebenfalls auf dieses Verzeichnis zu, sofern kein anderes ausdrücklich genannt wird. Um das voreingestellte Arbeitsverzeichnis abzuändern, existieren folgende Möglichkeiten:

- Unter Windows durch eine Veränderung der Eigenschaften der auf dem Desktop erstellten Verknüpfung mit R ((Name der Verknüpfung); Eigenschaften und anschließend bei Ausführen in ein neues Arbeitsverzeichnis angeben).
- Im Menü des Programmfensters von R kann Datei: Verzeichnis wechseln gewählt und anschließend ein neues Arbeitsverzeichnis angegeben werden. Dieser Schritt ist dann nach jedem Start von R zu wiederholen.

- In der Konsole lässt sich das aktuelle Arbeitsverzeichnis mit `setwd("Pfad")` ändern, unter Windows also z. B. mit `setwd("c:/work/r/")`.

R wird mit einer Reihe von Voreinstellungen gestartet, die sich über selbst editierbare Textdateien steuern lassen, über die ?Startup Auskunft gibt (R Development Core Team, 2009d). Sollen etwa bestimmte Pakete in jeder Sitzung geladen werden (vgl. Abschn. 1.2.7), können die entsprechenden `library()` Befehle in die Datei `Rprofile.site` im `etc/` Ordner des Programmordners geschrieben werden.⁹ Gleiches gilt für befehlübergreifende Voreinstellungen, die mit dem `options()` Befehl angezeigt und auch verändert werden können.

```
> options("<Option>")      # gibt aktuellen Wert für <Option> aus
> options(<Option>=<Wert>)  # setzt <Option> auf <Wert>
```

Mit `options(width=<Anzahl>)` kann z. B. festgelegt werden, mit wie vielen Zeichen pro Zeile die Ausgabe von R erfolgt. `options()` liefert dabei den Wert zurück, den die Einstellung vor ihrer Änderung hatte. Wird dieser Wert in einem Objekt gespeichert, kann die Einstellung später wieder auf ihren ursprünglichen Wert zurückgesetzt werden.

```
> op <- options(width=70)    # Option ändern und alten Wert in op speichern
> options(op)              # Option auf alten Wert zurücksetzen
```

Einstellungen, die die graphische Oberfläche von R unter Windows betreffen (etwa Schriftart und -größe der Ausgabe), werden im Menü unter **Bearbeiten: GUI Einstellungen** vorgenommen.

1.2.3 Umgang mit dem Workspace

Da R alle während einer Sitzung ausgeführten Befehle und erstellten Daten automatisch in einem sog. Workspace zwischenspeichert, kann eine Kopie von Befehlen und Daten nachträglich in externen Dateien abgespeichert und bei einer neuen Sitzung wieder genutzt werden. Auf bereits eingegebene Befehle kann auch bereits während derselben Sitzung erneut zugegriffen werden: über die Pfeiltasten nach oben und unten lassen sich verwendete Befehle wieder aufrufen. Diese Funktion wird im folgenden als Befehlshistorie bezeichnet. Eine Liste der letzten Befehle liefert auch die Funktion `history()`.

Es gibt mehrere, in ihren Konsequenzen unterschiedliche Wege, Befehle und Daten der aktuellen Sitzung zu sichern. Eine Kopie des aktuellen Workspace inklusive der Befehlshistorie wird etwa gespeichert, indem man die beim Beenden von R

⁹ Unter Unix-artigen Systemen auch in die Datei `.Rprofile` im Heimverzeichnis des Benutzers. Hier können auch eigene Funktionen namens `.First` bzw. `.Last` mit beliebigen Befehlen definiert werden, die dann beim Start als erstes bzw. bei Beenden als letztes ausgeführt werden, vgl. Abschn. 11.1.

erscheinende diesbezügliche Frage mit **Ja** beantwortet. Als Folge wird der sich – falls vorhanden – in diesem Verzeichnis befindende, bereits während einer früheren Sitzung gespeicherte Workspace automatisch überschrieben. R legt bei diesem Vorgehen zwei Dateien an: eine, die lediglich die erzeugten Daten der Sitzung enthält (Datei `.RData`) und eine, die die Befehlshistorie enthält (Datei `.Rhistory`). Der so gespeicherte Workspace wird beim nächsten Start von R automatisch geladen.

Der Workspace kann auch im Menü des Programmfensters über **Datei: Speichere Workspace** gespeichert und über **Datei: Lade Workspace** wieder aufgerufen werden. Bei diesem Vorgehen ist es möglich, Dateinamen für die zu speichernden Dateien anzugeben und damit ein Überschreiben zuvor angelegter Dateien zu vermeiden. Eine so angelegte Datei `<Dateiname>.RData` enthält nur die erstellten Objekte der Sitzung, die Befehlshistorie wird dabei also nicht automatisch gespeichert. Um die Befehlshistorie unter einem bestimmten Dateinamen abzuspeichern, wählt man **Datei: Speichere History**, und um eine Befehlshistorie zu laden **Datei: Lade History**.¹⁰ Die manuell gespeicherten Workspaces werden bei einem Neustart von R nicht automatisch geladen.

Beim Laden eines Workspace während einer laufenden Sitzung ist zu beachten, dass die in dieser Sitzung definierten Objekte von jenen Objekten aus dem geladenen Workspace überschrieben werden, die dieselbe Bezeichnung tragen. Die Befehlshistorie der aktuellen Sitzung wird hingegen nicht durch eine geladene History überschrieben, die Befehlshistorie ist also kumulativ.

1.2.4 Einfache Arithmetik

In R sind die grundlegenden arithmetischen Operatoren, Funktionen und Konstanten implementiert, über die auch ein Taschenrechner verfügt. Für eine Übersicht vgl. die mit dem Befehl `?Syntax` aufzurufende Hilfeseite. Punktrechnung geht dabei vor Strichrechnung, das Dezimaltrennzeichen ist unabhängig von den Länderereinstellungen des Betriebssystems immer der Punkt. Nicht ganzzahlige Werte werden in der Voreinstellung mit sieben relevanten Stellen ausgegeben, was mit dem `options(digits=<Anzahl>)` Befehl veränderbar ist.

Alternativ können Werte auch in wissenschaftlicher, also verkürzter Exponentialschreibweise ausgegeben werden.¹¹ Dabei ist z. B. der Wert `2e-03` als $2 \cdot (10^{-3})$, also $2/(10^3)$, entsprechend $2/1000$, mithin 0.002 zu lesen. Auch die Eingabe von Zahlen ist in diesem Format möglich.

¹⁰ Tatsächlich rufen auch die meisten Einträge des Menüs im Programmfenster lediglich die zugehörigen R-Funktionen auf. In der Konsole stehen zum Speichern und Laden der Befehlshistorie die Funktionen `savehistory(file="<Dateiname>")` und `loadhistory(file="<Dateiname>")` zur Verfügung.

¹¹ Sofern diese Formatierung nicht mit `options(scipen=999)` ganz unterbunden wird. Allgemein kann dabei mit ganzzahligen positiven Werten für `scipen` (Scientific Penalty) die Schwelle erhöht werden, ab der R die wissenschaftliche Notation für Zahlen verwendet, vgl. `?options`.

Tabelle 1.1 Arithmetische Funktionen, Operatoren und Konstanten

Operator/Funktion/Konstante	Bedeutung
<code>+</code> , <code>-</code>	Addition, Subtraktion
<code>*</code> , <code>/</code>	Multiplikation, Division
<code>%/%</code>	ganzzahlige Division (ganzzahliges Ergebnis einer Division ohne Rest)
<code>%%</code>	Modulo Division (Rest einer ganzzahligen Division, verallgemeinert auf Dezimalzahlen) ¹²
<code>~</code>	potenzieren
<code>sign()</code>	Vorzeichen (-1 bei negativen, 1 bei positiven Zahlen, 0 bei der Zahl 0)
<code>abs()</code>	Betrag
<code>sqrt()</code>	Quadratwurzel
<code>round(</code> <i>Zahl</i> <code>, digits=(</code> Anzahl <code>)</code>	runden (mit Argument zur Anzahl der Dezimalstellen) ¹³
<code>floor()</code> , <code>ceiling()</code> , <code>trunc()</code>	auf nächsten ganzzahligen Wert abrunden, aufrunden, tranchieren (Nachkommastellen abschneiden)
<code>log()</code> , <code>log10()</code> , <code>log2()</code> , <code>log(</code> <i>Zahl</i> <code>, base=</code> <i>Basis</i> <code>)</code>	natürlicher Logarithmus, Logarithmus zur Basis 10, zur Basis 2, zu beliebiger Basis
<code>exp()</code>	Exponentialfunktion
<code>exp(1)</code>	Eulersche Zahl e
<code>sin()</code> , <code>cos()</code> , <code>tan()</code> , <code>asin()</code> , <code>acos()</code> , <code>atan()</code> , <code>atan2()</code>	trigonometrische Funktionen sowie ihre Umkehrfunktionen
<code>factorial()</code>	Fakultät
<code>pi</code>	Kreiszahl π
<code>Inf</code> , <code>-Inf</code>	unendlich großer, kleiner Wert (Infinity)
<code>NA</code>	fehlender Wert (Not Available)
<code>NaN</code>	nicht definiert (Not a Number), verhält sich weitestgehend wie NA
<code>NULL</code>	leere Menge

Die in Tabelle 1.1 aufgeführten Funktionen, Operatoren und Konstanten sind in mathematisch-logischen Zusammenhängen frei kombinierbar und können auch ineinander verschachtelt werden. Um die Reihenfolge der Auswertung eindeutig zu halten, sollten in Zweifelsfällen Klammern gesetzt werden.¹⁴

```
> 12^2 + 1.5*10
[1] 159
```

¹² Der Dezimalteil einer Dezimalzahl ergibt sich also als `(Zahl) %% 1`.

¹³ R rundet in der Voreinstellung nicht nach dem vielleicht vertrauten Prinzip des kaufmännischen Rundens, sondern *unverzerrt* (Bronstein et al., 2008). Durch negative Werte für `digits` kann auch auf Zehnerpotenzen gerundet werden.

¹⁴ Für die zur Bestimmung der Ausführungsreihenfolge wichtige Assoziativität von Operatoren vgl. ?Syntax.

```
> sin(pi/2) + sqrt(abs(-4))
[1] 3

> exp(1)^((0+1i)*pi)
[1] -1+0i
```

$\langle \text{Realteil} \rangle + \langle \text{Imaginärteil} \rangle i$ ist die Notation zur Eingabe komplexer Zahlen. $0+1i$ ist also die imaginäre Zahl i , $-1+0i$ hingegen einfach die reelle Zahl -1 . Die angegebene Formel ist die sog. Eulersche Identität. Den Realteil einer komplexen Zahl liefert $\text{Re}(\langle \text{Zahl} \rangle)$, den Imaginärteil $\text{Im}(\langle \text{Zahl} \rangle)$. Die Funktionen $\text{Mod}(\langle \text{Zahl} \rangle)$ und $\text{Arg}(\langle \text{Zahl} \rangle)$ geben die Polarkoordinaten in der komplexen Ebene aus, die komplexe Konjugierte ermittelt $\text{Conj}(\langle \text{Zahl} \rangle)$.

Um zu überprüfen, ob ein Objekt einen „besonderen“ numerischen Wert speichert, stellt R Funktionen bereit, die nach dem Muster `is.(Prüfwert)(⟨Zahl⟩)` aufgebaut sind und einen Wahrheitswert zurückgeben (vgl. Abschn. 1.3.6).

```
> is.infinite(<Zahl>)      # ist <Zahl> unendlich?
> is.finite(<Zahl>)        # ist <Zahl> endlich?
> is.nan(<Zahl>)          # ist <Zahl> nicht definiert?
> is.na(<Zahl>)           # ist <Zahl> ein fehlender Wert?
> is.null(<Zahl>)         # ist <Zahl> die leere Menge?

> is.infinite(1/0)
[1] TRUE

> is.nan(0/0)
[1] TRUE
```

1.2.5 Funktionen mit Argumenten aufrufen

Beim Aufruf von Funktionen in R sind die Werte, die der Funktion als Berechnungsgrundlage dienen, in runde Klammern einzuschließen: `<Funktionsname>(<Werte>)`. Auch wenn eine Funktion keine Werte benötigt, müssen die runden Klammern vorhanden sein, z. B. `q()`.¹⁵

Weitere Argumente sind beim Funktionsaufruf häufig in der Form `<Funktionsname>(<Werte>, <Argumentliste>)` anzugeben. Die Argumentliste besteht aus Zuweisungen an Argumente in der Form `<Argumentname>=<Wert>`, die der Funktion die notwendigen Eingangsinformationen liefern. Es können je nach Funktion ein oder mehrere durch Komma getrennte Argumente angegeben werden, die ihrerseits obligatorisch oder nur optional sein können.¹⁶

¹⁵ In R sind auch Operatoren wie `+`, `-`, `*` oder `/` Funktionen, für die lediglich eine bequemere und vertrautere Kurzschreibweise zur Verfügung steht. Operatoren lassen sich aber auch in der üblichen Präfix-Form benutzen, wenn sie in Anführungszeichen gesetzt werden. `"/"(1, 10)` ist also äquivalent zu `1/10`.

¹⁶ In diesem Text werden nur die wichtigsten Argumente der behandelten Funktionen vorgestellt, eine vollständige Übersicht liefert jeweils `args(<Funktionsname>)` sowie die zugehörige Hilfeseite `?<Funktionsname>`.

Argumente sind benannt und machen so häufig ihre Bedeutung für die Arbeitsweise der Funktion deutlich. Um z. B. eine Zahl zu runden, muss der Funktion `round(<Zahlen>, digits=0)` mindestens ein zu rundender Wert übergeben werden, z. B. `round(1.271)`. Weiterhin besteht die Möglichkeit, über das zusätzliche Argument `digits` die gewünschte Zahl an Nachkommastellen zu bestimmen: mit `round(pi, digits=2)` wird die Zahl π auf zwei Dezimalstellen gerundet. Das Argument `digits` ist optional, wird es nicht angegeben, kommt der auf 0 voreingestellte Wert (sog. Default) zur Rundung auf ganze Zahlen zum Einsatz.

Der Name von Argumenten muss nicht unbedingt vollständig angegeben werden, wenn eine Funktion aufgerufen wird – eine Abkürzung auf den zur eindeutigen Identifizierung notwendigen Namensanfang reicht aus.¹⁷ Von dieser Möglichkeit sollte jedoch mit Blick auf die Verständlichkeit des Funktionsaufrufs kein Gebrauch gemacht werden. Fehlt der Name eines Arguments ganz, so erfolgt die Zuordnung eines im Funktionsaufruf angegebenen Wertes über seine Position in der Argumentliste: beim Befehl `round(pi, 3)` wird die 3 als Wert für das `digits` Argument interpretiert, weil sie an zweiter Stelle steht. Allgemein empfiehlt es sich, nur den Namen des ersten Hauptarguments wegzulassen und die übrigen Argumentnamen zu nennen, insbesondere, wenn viele Argumente an die Funktion übergeben werden können.

1.2.6 Hilfe-Funktionen

R hat ein integriertes Hilfesystem, das vom Benutzer auf verschiedene Arten zurate gezogen werden kann: zum einen ruft `help.start()` eine HTML-Oberfläche im Browser auf, von der aus dann spezifische Hilfeseiten erreichbar sind. Zum anderen kann auf diese Seiten mit der Funktion `help(topic=<Befehlsname>)` zugegriffen werden, gleichbedeutend auch mit `?<Befehlsname>`. Operatoren müssen bei beiden Versionen in Anführungszeichen gesetzt werden, etwa `?"/"`. Die Hilfe-Funktion lässt sich auch über das Menü des Programmfensters mit **Hilfe: HTML Hilfe** aufrufen.

Die Inhalte der Hilfe sind meist knapp und eher technisch geschrieben, zudem setzen sie häufig Vorkenntnisse voraus. Dennoch stellen sie eine wertvolle und reichhaltige Ressource dar, deren Wert sich mit steigender Vertrautheit mit R stärker erschließt. Im Abschnitt Usage der Hilfeseiten werden verschiedene Anwendungsmöglichkeiten der Funktion beschrieben. Dazu zählen auch unterschiedliche Varianten im Fall von generischen Funktionen, deren Arbeitsweise von der Klasse der übergebenen Argumente abhängt (vgl. Abschn. 11.1.5). Unter Arguments wird erläutert, welche notwendigen sowie optionalen Argumente die Funktion besitzt und welcher Wert für ein optionales Argument voreingestellt ist. Der Abschnitt Value

¹⁷ Gleichermaßen gilt für die Werte von Argumenten, sofern sie aus einer festen Liste von Zeichenketten stammen. Statt `cov(Matrix, use="pairwise.complete.obs")` ist also auch `cov(Matrix, use="pairwise")` als Funktionsaufruf möglich.

erklärt, welche Werte die Funktion als Ergebnis zurückliefert. Weiterhin wird die Benutzung der Funktion im Abschnitt Examples mit Beispielen erläutert. Um diese Beispiele auch samt ihrer Ergebnisse vorgeführt zu bekommen, kann die Funktion `example(topic=(Befehlsnahme))` verwendet werden.

Wenn der genaue Name einer gesuchten Funktion unbekannt ist, können die Hilfeseiten mit der Funktion `help.search(pattern="Stichwort")` nach Stichworten gesucht werden. Das Ergebnis führt jene Funktionsnamen auf, in deren Hilfeseiten *<Stichwort>* vorhanden ist. Auch diese Funktion lässt sich alternativ über das Menü mit dem Eintrag Hilfe: Durchsuche Hilfe aufrufen. Mit `apropos(what="Stichwort")` werden Funktionen ausgegeben, die *<Stichwort>* in ihrem Funktionsnamen tragen.

1.2.7 Zusatzpakete verwenden

Die Grundfunktionalität von R lässt sich über eigenständig entwickelte Zusatzkomponenten modular erweitern, die in Form von sog. Paketen inhaltlich spezialisierte Funktionen zur Datenanalyse sowie vorgefertigte Datensätze bereitstellen (Ligges, 2003). Die thematisch geordnete Übersicht *Task Views* auf CRAN führt dazu etwa unter **Psychometrics** und **SocialSciences** viele für Psychologen und Sozialwissenschaftler relevante Pakete an (R Development Core Team, 2009a; Zeileis, 2005). Dort finden sich auch Pakete zu Inhalten, die in diesem Text weitgehend ausgeklammert bleiben, etwa zur Test- und Fragebogenanalyse oder zur Analyse von Zeitreihen.

Während einige Pakete bereits in einer Standardinstallation enthalten sind, aber aus Effizienzgründen erst im Bedarfsfall geladen werden müssen, sind die meisten Zusatzpakete zunächst manuell zu installieren. Auf Rechnern mit Online-Zugriff lassen sich Zusatzpakete direkt über das Menü des Programmfensters mit **Pakete: Installiere Paket(e)** und anschließender Wahl eines CRAN-Mirrors installieren. Gleiches kann in der Konsole mit dem `install.packages()` Befehl die Installation eines Pakets angefordert werden.¹⁸

```
> install.packages(pkgs="Paketname", dependencies=NA)
```

Der Paketname ist in Anführungszeichen eingeschlossen für das Argument `pkgs` zu nennen. Zusätzlich sollte abweichend von der Voreinstellung `dependencies=TRUE` gesetzt werden, damit vom zu installierenden Paket ggf. benötigte andere Pakete automatisch ebenfalls mit installiert werden.

Für Rechner ohne Internetanbindung lassen sich die Installationsdateien der Zusatzpakete von einem anderen Rechner mit Online-Zugriff herunterladen, um sie dann manuell auf den Zielrechner übertragen und dort installieren zu können. Um ein Paket auf diese Weise zu installieren, muss von der R-Projektseite kommend einer der CRAN-Mirrors und anschließend **Contributed extension packages**

¹⁸ Mit dem Argument `repos` von `install.packages()` können temporär, mit der Funktion `setRepositories()` auch dauerhaft andere Server als Paketquelle verwendet werden.

gewählt werden. Die sich daraufhin öffnende Seite führt alle verfügbaren Zusatzpakete inklusive einer kurzen Beschreibung auf. Durch Anklicken eines Paketnamens öffnet sich die zugehörige Downloadseite. Sie enthält eine längere Beschreibung sowie u. a. den Quelltext des Pakets (Dateiendung `.tar.gz`), eine zur Installation geeignete Archivdatei (Dateiendung `.zip`) sowie Dokumentation im PDF-Format, die u. a. die Funktionen des Zusatzpaketes erläutert. Nachdem die Archivdatei auf den Zielrechner übertragen ist, lässt sie sich im Menü des R-Programmfensters über **Pakete: Installiere Paket(e) aus lokalen Zip-Dateien** installieren.

R installiert die Zusatzpakete in der Voreinstellung automatisch im Unterverzeichnis `library/` des R-Ordners.¹⁹ Alle installierten Pakete lassen sich simultan über das Menü des R-Programmfensters mit **Pakete: Aktualisiere Pakete** aktualisieren, sofern eine neue Version auf den CRAN-Servern vorhanden ist. Diesem Vorgehen entspricht auf der Konsole der Befehl `update.packages()`.

Damit die Funktionen und Datensätze eines installierten Zusatzpaketes auch zur Verfügung stehen, muss es bei jeder neuen R-Sitzung manuell geladen werden. Durch **Pakete: Lade Paket** im Menü des Programmfensters öffnet sich ein Fenster, in dem alle installierten Zusatzpakete aufgelistet sind und durch Anklicken des entsprechenden Pakets geladen werden können. Alternativ lassen sich die installierten Zusatzpakete in der Konsole mit folgenden Befehlen auflisten und laden²⁰:

```
> installed.packages()
> library(package=<Paketname>)
```

Die `installed.packages()` sowie die `library()` Funktion ohne Angabe von Argumenten zeigt alle installierten und damit ladbaren Pakete an.

Kurzinformationen zu einem ladbaren Paket, etwa die darin enthaltenen Funktionen, liefert `help(package=<Paketname>)`. Viele Pakete bringen darüber hinaus noch ausführlichere Dokumentation im PDF-Format mit, die mit `vignette(topic=<Thema>)` aufgerufen werden kann. Ein für `topic` bestimmtes Thema kann etwa der Paketname sein, aber auch für Spezialthemen existieren eigene Dokumente. Verfügbare Themen können durch `vignette()` ohne Angabe von Argumenten angezeigt werden.

Aus dem Output der Funktion `search()` ist u. a. ersichtlich, welche Pakete geladen sind (vgl. Abschn. 1.3.1). Über `detach(package:<Paketname>)` kann ein geladenes Paket auch wieder entfernt werden.

Eine Übersicht darüber, welche Datensätze in einem bestimmten Zusatzpaket vorhanden sind, wird mit der Funktion `data(package=<Paketname>)`

¹⁹ Bei der Installation einer neuen R-Version müssen zuvor manuell hinzugefügte Pakete erneut installiert werden. Alternativ können Pakete auch in einem separaten Verzeichnis außerhalb des R-Programmverzeichnisses installiert werden. Dafür muss eine Textdatei `Renvironment.site` im Unterordner `etc/` des R-Programmordners existieren und eine Zeile der Form `R_LIBS="<Pfad>"` (z. B. `R_LIBS="c:/rlibs/"`) mit dem Pfad zu den Paketen enthalten.

²⁰ Wird versucht, ein nicht installiertes Paket zu laden, erzeugt `library()` einen Fehler und gibt ein später zur Fallunterscheidung verwendbares `FALSE` zurück, sofern das Argument `logical.return=TRUE` gesetzt wird (vgl. Abschn. 11.2.1). Soll in einem solchen Fall nur eine Warnung ausgegeben werden, ist `require()` zu verwenden.

geöffnet (vgl. Abschn. 3.2). Diese Datensätze können mit `data(<Datensatz>, package="<Paketname>")` auch unabhängig von den Funktionen des Pakets geladen werden. Ohne Angabe von Argumenten öffnet `data()` eine Liste mit bereits geladenen Datensätzen. Viele Datensätze sind mit einer kurzen Beschreibung ausgestattet, die `help(<Datensatz>)` ausgibt.

1.3 Datenstrukturen: Klassen, Objekte, Datentypen

Die Datenstrukturen, die in R Informationen repräsentieren, sind im wesentlichen eindimensionale Vektoren (`vector`), zweidimensionale Matrizen (`matrix`), verallgemeinerte Matrizen mit auch mehr als zwei Dimensionen (`array`), Listen (`list`), Datensätze (`data.frame`) und Funktionen (`function`). Die gesammelten Eigenschaften jeweils einer dieser Datenstrukturen werden als Klasse bezeichnet (für Details vgl. Chambers, 2008; Ligges, 2009).

Daten werden in R in benannten Objekten gespeichert. Jedes Objekt ist die konkrete Verkörperung einer der o. g. Klassen, die Art und Struktur der im Objekt gespeicherten Daten festlegt. Die Klasse eines Objekts kann mit dem Befehl `class(<Objekt>)` erfragt werden, wobei als Klasse von Vektoren der Datentyp der in ihm gespeicherten Werte gilt, z. B. `numeric` (s. u.). Die Funktionen, deren Namen nach dem Muster `is.<Klasse>(<Objekt>)` aufgebaut sind, prüfen, ob ein vorliegendes Objekt von einer gewissen Klasse ist. So gibt etwa die `is.matrix(<Objekt>)` Funktion an, ob `<Objekt>` die Klasse `matrix` hat (Ausgabe `TRUE`) oder nicht (Ausgabe `FALSE`).

Bestehende Objekte einer bestimmten Klasse können unter gewissen Voraussetzungen in Objekte einer anderen Klasse konvertiert werden. Zu diesem Zweck stellt R eine Reihe von Funktionen bereit, deren Namen nach dem Schema `as.<Klasse>(<Objekt>)` aufgebaut sind. Um ein Objekt in einen Vektor umzuwandeln, wäre demnach die Funktion `as.vector(<Objekt>)` zu benutzen. Mehr Informationen zu diesem Thema finden sich bei der Behandlung der einzelnen Klassen.

Intern werden die Daten vieler Objekte durch einen Vektor, d. h. durch eine linear geordnete Menge einzelner Werte repräsentiert. Jedes Objekt besitzt auch eine Länge, die meist der Anzahl der im internen Vektor gespeicherten Elemente entspricht und durch den Befehl `length(<Objekt>)` abgefragt werden kann.

Objekte besitzen darüber hinaus einen Datentyp oder Modus, der sich auf die Art der im Objekt gespeicherten Informationen bezieht und mit `mode(<Objekt>)` ausgegeben werden kann – unterschieden werden vornehmlich numerische, alphanumerische und logische Modi (vgl. Abschn. 1.3.5). Ein Objekt der Klasse `matrix` könnte also z. B. mehrere Wahrheitswerte speichern und somit den Datentyp `logical` besitzen.

Ein Objekt kann zudem sog. Attribute aufweisen, die zusätzliche Informationen über die in einem Objekt enthaltenen Daten speichern. Sie können mit dem Befehl `attributes(<Objekt>)` und `attr(<Objekt>, which=<Attribut>)`

abgefragt und über `attr()` auch geändert werden. Meist versieht R von sich aus bestimmte Objekte mit Attributen, so ist etwa die Klasse eines Objekts als Attribut gespeichert. Man kann Attribute aber auch selbst im Sinne einer freien Beschreibung nutzen, die man mit einem Objekt assoziieren möchte – hierfür eignet sich alternativ auch die `comment()` Funktion.²¹

Über die interne Struktur eines Objekts, also seine Zusammensetzung aus Werten samt ihrer Datentypen, gibt die Funktion `str(<Objekt>)` Auskunft.

1.3.1 Objekte benennen

Objekte tragen i. d. R. einen Namen beliebiger Länge, über den sie in Befehlen identifiziert werden. Objektnamen sollten mit einem Buchstaben beginnen, können aber ab der zweiten Stelle neben Buchstaben auch Zahlen, Punkte und Unterstriche enthalten. Von der Verwendung anderer Sonderzeichen wie auch von deutschen Umlauten ist abzuraten, selbst wenn dies bisweilen möglich ist.²² Groß- und Kleinschreibung werden bei Objektnamen und Befehlen unterschieden, so ist das Objekt `asdf` ein anderes als `Asdf`. Objekte dürfen nicht den Namen spezieller Schlüsselwörter wie `if` tragen, die in der Hilfeseite `?Reserved` aufgeführt sind.

Ebenso sollten keine Objekte mit Namen versehen werden, die gleichzeitig Funktionen in R bezeichnen, selbst wenn dies möglich ist. Kommt es dennoch zu einer sog. Maskierung von bereits durch R vergebenen Namen durch selbst angelegte Objekte, ist dies i. d. R. unproblematisch. Dies liegt daran, dass es nicht nur einen, sondern mehrere Workspaces als voneinander abgegrenzte Einheiten gibt. Sie werden auch als Umgebungen (Environments) bezeichnet und sind intern linear geordnet. Objekte, die in unterschiedlichen Umgebungen beheimatet sind, können denselben Namen tragen, ohne dass ihre Inhalte wechselseitig überschrieben würden. Existieren mehrere Objekte desselben Namens in unterschiedlichen Umgebungen, bezeichnet der einfache Name das Objekt in der früheren Umgebung.²³ Die Reihenfolge, in der R die Umgebungen nach Objekten durchsucht, ist der sog. Suchpfad, der von `search()` ausgegeben wird. Die erste Umgebung mit dem Namen `.GlobalEnv` ist dabei jene, in der R automatisch alle während einer Sitzung ausgeführten Befehle und erstellten Daten sichert (vgl. Abschn. 1.2.3). Welche Objekte eine Umgebung speichert, lässt sich mittels `ls("Umgebung")` feststellen.

```
> ls("package:stats")      # nenne Objekte des Pakets stats ...
```

²¹ Die Funktion `label()` aus dem `Hmisc` Paket (Harrell, 2009a) erweitert dieses Konzept und macht es den etwa in SPSS gebräuchlichen Variablen-Labels ähnlicher.

²² Wenn ein Objektname dennoch nicht zulässige Zeichen enthält, kann man nichtsdestotrotz auf das Objekt zugreifen, indem man den Namen in rückwärts gerichtete Hochkommata setzt (``<Objektname>``).

²³ Um gezielt Objekte aus einer bestimmten Umgebung zu erhalten vgl. `?environment`.

Ob Namenskonflikte, also mehrfach vergebene Objektnamen, vorliegen, kann mit der Funktion `conflicts(detail=TRUE)` geprüft werden. Die Funktion `exists("<Name>")` gibt an, ob `<Name>` schon als Bezeichner verwendet wird.

1.3.2 Zuweisungen an Objekte

Um Ergebnisse von Berechnungen zu speichern und wiederverwenden zu können, müssen diese einem benannten Objekt zugewiesen werden. Objekte können dabei einzelne Zahlen aufnehmen, aber auch Text oder andere komplexe Inhalte haben. Zuweisungen, z. B. an ein Objekt `x1`, können auf drei verschiedene Arten geschehen:

```
> x1 <- 4.5
> x1 = 4.5
> 4.5 -> x1
```

Zur Vermeidung von Mehrdeutigkeiten bei komplizierteren Eingaben sollte die erste Methode bevorzugt werden. Das vielleicht vertrautere Gleichheitszeichen sollte der Zuweisung von Funktionsargumenten vorbehalten bleiben (vgl. Abschn. 1.2.5), um Verwechslungen mit der Prüfung auf Gleichheit zweier Objekte durch `==` vorzubeugen und die Richtung der Zuweisung eindeutig zu halten. Im Fall des `<Objekt> <- <Wert>` Operators erfolgt die Zuweisung von rechts nach links in Richtung des Pfeils und kann sich auch über mehrere Objekte erstrecken:

```
> x2 <- x3 <- 10
> x2
[1] 10

> x3
[1] 10
```

Objekte können in Befehlen genauso verwendet werden, wie die Daten, die in ihnen gespeichert sind, d. h. Objektnamen stehen in Berechnungen für die im Objekt gespeicherten Werte.

```
> x1 * 2
[1] 9

> x1^x1 - x2
[1] 859.874
```

Die Übersichtlichkeit ist ein entscheidender Faktor beim Erstellen und nachträglichen Verändern einer gespeicherten Abfolge von Befehlen (vgl. Abschn. 4.1). Es ist deshalb dringend zu raten, die Befehlssequenzen so zu erstellen, dass ein einfaches Verständnis der Vorgänge gewährleistet ist. Dies kann u. a. durch folgende Maßnahmen erreicht werden:

- Leerzeichen zwischen Befehlen und Objektnamen verwenden
- zusammengehörende Elemente eines Befehls in runde Klammern () einschließen
- Objekte sinnvoll (inhaltlich aussagekräftig) benennen

- komplexe Berechnungen in einzelne Teilschritte aufteilen, deren Zwischenergebnisse separat geprüft werden können
- Kommentare einfügen – dies sind von R nicht als Befehl interpretierte Texte, die mit dem # Zeichen beginnen und sich für Erläuterungen der Befehle eignen (vgl. Abschn. 4.1)

1.3.3 Objekte ausgeben

Bei Zuweisungen zu Objekten erfolgt durch R keine Ausgabe des Wertes, der letztlich im Zielobjekt gespeichert wurde. Um sich den Inhalt eines Objekts anzeigen zu lassen, gibt es drei Möglichkeiten:

```
> print(x1)          # print(<Objektname>) Funktion
> x1                # Objektnamen nennen - ruft implizit print() auf
> (x1 <- 4.5)       # Befehl in runde Klammern setzen - zeigt nur
                     # die durch den Befehl veränderten Werte an
```

Es ist allgemein dazu zu raten, häufig runde Klammern um einen Befehl zu setzen. Dadurch können die in Zwischenrechnungen veränderten Werte mit ausgegeben werden, was die Kontrolle der Richtigkeit einzelner Arbeitsschritte erleichtert.

Wurde vergessen, das Ergebnis eines Rechenschritts als Objekt zu speichern, so lässt sich das letzte ausgegebene Ergebnis mit .Last.value erneut anzeigen und einem Objekt zuweisen.

1.3.4 Objekte anzeigen lassen und entfernen

Um sich einen Überblick über alle im Workspace vorhandenen Objekte zu verschaffen, dient die Funktion ls(). Objekte, deren Name mit einem Punkt beginnt, sind dabei versteckt – sie werden mit ls(all=TRUE) angezeigt.

```
> ls()
[1] "x1" "x2" "x3"
```

Vorhandene Objekte können mit der rm(<Objekt>) Funktion (Remove) gelöscht werden. Sollen alle bestehenden Objekte entfernt werden, kann dies mit dem Befehl rm(list=ls(all=TRUE)) oder über das Menü Verschiedenes: Entferne alle Objekte geschehen.

```
> age <- 22
> rm(age); age
Fehler: Objekt "age" nicht gefunden
```

1.3.5 Datentypen

Der Datentyp eines Objekts bezieht sich auf die Art der in ihm gespeicherten Informationen. Er lässt sich mit mode(<Objekt>) ausgeben. Neben den in Tabelle 1.2 aufgeführten Datentypen existieren noch weitere, über die ?mode Auskunft gibt.

Tabelle 1.2 Datentypen

Beschreibung	Beispiel	Datentyp
leere Menge	NULL	NULL
logische Werte	TRUE, FALSE (T, F)	logical
ganze und reelle Zahlen	3.14	numeric ²⁴
komplexe Zahlen	3.14 + 1i	complex
Buchstaben und Zeichenfolgen (immer in Anführungszeichen einzugeben) ²⁵	"Hello"	character

```
> charVar <- "asdf"
> mode(charVar)
[1] "character"
```

Die Funktionen, deren Namen nach dem Muster `is.<Datentyp>(<Objekt>)` aufgebaut sind, prüfen, ob ein Objekt Werte von einem bestimmten Datentyp speichert. `is.logical(<Objekt>)` gibt etwa an, ob die Werte in `<Objekt>` vom Datentyp logical sind.

So wie Objekte einer bestimmten Klasse u. U. in Objekte einer anderen Klasse umgewandelt werden können, so lässt sich auch der Datentyp der in einem Objekt gespeicherten Werte in einen anderen konvertieren. Die Funktionen zur Umwandlung des Datentyps sind nach dem Muster `as.<Datentyp>(<Objekt>)` benannt. Um etwa eine Zahl in den zugehörigen Text umzuwandeln, ist also der Befehl `as.character(<Zahl>)` zu benutzen.

```
> is.character(1.23)
[1] FALSE
> as.character(1.23)
[1] "1.23"

> as.logical(2)
[1] TRUE
```

Bei Umwandlung von Datentypen besteht eine Hierarchie entsprechend der in Tabelle 1.2 aufgeführten Reihenfolge. Weiter unten stehende Datentypen können Werte aller darüber stehenden Datentypen ohne Informationsverlust repräsentieren, nicht jedoch umgekehrt: jede reelle Zahl lässt sich z. B. genauso gut als komplexe Zahl mit imaginärem Anteil 0 (1.23 ist gleich $1.23 + 0i$) speichern, jeder logische

²⁴ Für reelle Zahlen (`numeric`) existieren zwei Möglichkeiten, sie in einem Computer intern zu repräsentieren: ganze Zahlen können mit einem L hinter der Zahl gekennzeichnet werden (z. B. 5L), wodurch R sie dann auch als solche speichert (`integer`). Andernfalls werden alle Zahlen als Gleitkommazahlen mit doppelter Genauigkeit gespeichert (`double`). Dies lässt sich mit dem Befehl `typeof(<Objekt>)` abfragen. Ob ein Objekt einen bestimmten Speichertyp aufweist, wird mit Funktionen der `is.<Speicherart>(<Objekt>)` Familie geprüft (z. B. `is.double(<Objekt>)`).

²⁵ Dies können einfache ('Zeichen') oder doppelte ("Zeichen") Anführungszeichen sein. Innerhalb einfacher Anführungszeichen können auch Zeichenketten stehen, die ihrerseits doppelte Anführungszeichen beinhalten ('a"b'), während diese innerhalb doppelter Anführungszeichen als sog. Escape-Sequenz durch einen vorangestellten Backslash zu schreiben sind ("a\"b").

Wert entsprechend einer bestimmten Konvention als ganze Zahl (TRUE entspricht der 1, FALSE der 0). Umgekehrt jedoch würden viele unterschiedliche komplexe Zahlen nur als gleiche reelle Zahl gespeichert werden können, und viele unterschiedliche ganze Zahlen würden als gleicher logischer Wert repräsentiert (alle Zahlen ungleich 0 als TRUE, die 0 als FALSE). Während sich also alle Zahlen mit `as.character(<Zahl>)` in die zugehörige Zeichenkette umwandeln lassen, ist dies umgekehrt nicht allgemein möglich. `as.numeric("<Text>")` ergibt nur für Zeichenketten der Form "`<Zahl>`" den entsprechenden numerischen Wert, andernfalls NA als Konstante, die für einen fehlenden Wert steht (vgl. Abschn. 2.12).

1.3.6 Logische Werte, Operatoren und Verknüpfungen

Das Ergebnis eines logischen Vergleichs mit den in Tabelle 1.3 genannten Operatoren ist ein Wahrheitswert, der entweder WAHR (TRUE) oder FALSCH (FALSE) sein kann. Wahrheitswerte können auch in numerischen Rechnungen genutzt werden, dem Wert TRUE entspricht dann die 1, dem Wert FALSE die 0.

```
> TRUE == TRUE    > TRUE == FALSE    > ! TRUE        > ! FALSE
[1] TRUE          [1] FALSE         [1] FALSE        [1] TRUE
> TRUE != TRUE   > TRUE != FALSE   > isTRUE(TRUE)  > isTRUE(FALSE)
[1] FALSE         [1] TRUE          [1] TRUE         [1] FALSE

> TRUE & TRUE    > TRUE & FALSE    > FALSE & FALSE  > FALSE & TRUE
[1] TRUE          [1] FALSE         [1] FALSE        [1] FALSE

> TRUE | TRUE    > TRUE | FALSE    > FALSE | FALSE  > FALSE | TRUE
[1] TRUE          [1] TRUE          [1] FALSE        [1] TRUE

> 4 < 8          > 7 < 3          > 4 > 4          > 4 >= 4
[1] TRUE          [1] FALSE         [1] FALSE        [1] TRUE
```

Statt des logischen Vergleichsoperators `==` kann zum Prüfen zweier Objekte auf Gleichheit auch der Befehl

Tabelle 1.3 Logische Operatoren, Funktionen und Konstanten

Operator/Funktion/Konstante	Beschreibung
<code>!=, ==</code>	Vergleich: ungleich, gleich
<code>>, >=, <, <=</code>	Vergleich: größer, größer-gleich, kleiner, kleiner-gleich
<code>!</code>	logisches NICHT (Negation)
<code>&, &&</code>	Verknüpfung: logisches UND
<code> , </code>	Verknüpfung: logisches ODER (einschließend)
<code>xor()</code>	logisches ENTWEDER-ODER (ausschließend)
<code>TRUE, FALSE (T, F)</code>	logische Wahrheitswerte: WAHR, FALSCH (abgekürzt)
<code>isTRUE(<Objekt>)</code>	gibt an, ob zusammengesetztes <code><Objekt></code> dem Wert TRUE entspricht

```
> all.equal(target=<Objekt1>, current=<Objekt2>,
+           tolerance=(relative Abweichung))
```

verwendet werden. Dieser bestätigt die Gleichheit der Objekte `target` und `current` auch, wenn sie nur ungefähr (d. h. mit einer durch `tolerance` festgelegten Genauigkeit) gleich sind, während der `==` Operator nur bei exakter Gleichheit `TRUE` ergibt.²⁶ Allerdings liefert `all.equal()` im Fall der Ungleichheit nicht den Wahrheitswert `FALSE` zurück, sondern ein Maß der Abweichung. Wird ein einzelner Wahrheitswert als Ergebnis benötigt, muss `all.equal()` deshalb mit der Funktion `isTRUE(<Objekt>)` verschachtelt werden, die komplexere Objekte darauf prüft, ob sie dem Wert `TRUE` entsprechen:

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

> all.equal(0.12345001, 0.12345000)
[1] "Mean relative difference: 8.100445e-08"

> isTRUE(all.equal(0.12345001, 0.12345000))
[1] FALSE

> isTRUE(all.equal(0.123450001, 0.123450000))
[1] TRUE

> 0.123400001 == 0.123400000
[1] FALSE
```

²⁶ Aufgrund der Art, in der Computer Gleitkommazahlen intern speichern und verrechnen, sind kleine Abweichungen in Rechenergebnissen schon bei harmlos wirkenden Ausdrücken möglich. So ergibt der Vergleich $0.1 + 0.2 == 0.3$ fälschlicherweise `FALSE` und $1 \% \% 0.1$ ist 9 statt 10. `sin(pi)` wird als $1.224606e-16$ und nicht exakt 0 berechnet, ebenso ist $1 - ((1/49)*49)$ nicht exakt 0, sondern $1.110223e-16$. Dagegen ist $1 - ((1/48)*48)$ exakt 0. Dies sind keine R-spezifischen Probleme, sie können nicht allgemein verhindert werden (Cowlishaw, 2008; Goldberg, 1991).

Kapitel 2

Elementare Dateneingabe und -verarbeitung

Im folgenden Abschnitt sollen gleichzeitig die grundlegenden Datenstrukturen in R sowie Möglichkeiten zur deskriptiven Datenauswertung erläutert werden.

2.1 Vektoren

R ist eine vektorbasierte Sprache, ist also auf die Verarbeitung von in Vektoren angeordneten Daten ausgerichtet.¹ Ein Vektor ist dabei lediglich eine Datenstruktur für eine sequentiell geordnete Menge einzelner Werte und nicht mit dem mathematischen Konzept eines Vektors zu verwechseln. Da sich empirische Daten einer Variable meist als eine linear anzuordnende Wertemenge betrachten lassen, sind Vektoren als Organisationsform gut für die Datenanalyse geeignet. Vektoren sind die einfachste Datenstruktur für Werte, d. h. auch jeder Skalar oder andere Einzelwert ist ein Vektor der Länge 1.

2.1.1 Vektoren erzeugen

Vektoren werden durch Funktionen erzeugt, die den Namen eines Datentyps tragen und als Argument die Anzahl der zu speichernden Elemente erwarten, also etwa `numeric(<Anzahl>)`. Die Elemente des Vektors werden hierbei auf eine Voreinstellung gesetzt, die vom Datentyp abhängt.²

```
> numeric(4)
[1] 0 0 0 0
```

Als häufiger genutzte Alternative lassen sich Vektoren auch mit der Funktion `c(<Wert1>, <Wert2>, ...)` erstellen (Concatenate – aneinanderhängen), die die

¹ R ähnelt in der Vektorbasiertheit anderen befehlsgesteuerten Programmen zur allgemeinen wissenschaftlichen Datenverarbeitung wie z. B. Mathematica oder MATLAB (bzw. GNU Octave).

² Ein leerer Vektor entsteht analog, z. B. durch `logical(0)`.

Angabe der zu speichernden Werte benötigt. Ein das Alter von sechs Personen speichernder Vektor könnte also so erstellt werden:

```
> (age <- c(18, 20, 30, 24, 23, 21))
[1] 18 20 30 24 23 21
```

Dabei werden die Werte in der angegebenen Reihenfolge gespeichert und intern mit fortlaufenden Indizes für ihre Position im Vektor versehen. Sollen bereits bestehende Vektoren zusammengefügt werden, ist ebenfalls die `c()` Funktion zu nutzen, wobei statt eines einzelnen Wertes auch der Name eines bereits bestehenden Vektors angegeben werden kann.

```
> addAge <- c(27, 21, 19)
> (ageNew <- c(age, addAge))
[1] 18 30 30 25 23 21 27 21 19
```

Mit der Funktion `length(<Objektname>)` wird die Länge eines Vektors, d. h. die Anzahl der in ihm gespeicherten Elemente, erfragt.

```
> length(age)
[1] 6
```

Elemente eines Vektors können auch Zeichenketten sein. Während dann der `length()` Befehl jede Zeichenkette als ein Element betrachtet, gibt die `nchar("<Zeichenkette>")` Funktion an, aus wie vielen einzelnen Zeichen die Zeichenkette besteht. Wird ein Vektor von Zeichenketten übergeben, besteht die Ausgabe aus einem numerischen Vektor, der für jedes Element die Wortlänge anzeigt.

```
> length("ABCDEF")
[1] 1

> nchar("ABCDEF")
[1] 6

> nchar(c("A", "BC", "DEF"))
[1] 1 2 3
```

2.1.2 Elemente auswählen und verändern

Um ein einzelnes Element eines Vektors abzurufen, wird seine Position im Vektor (sein Index) in eckigen Klammern, dem `[<Index>]` Operator, hinter dem Objektnamen angegeben.³ Indizes beginnen bei 1 für die erste Position⁴ und enden bei der

³ Für Hilfe zu diesem Thema vgl. `?Extract`. Auch der Index-Operator ist eine Funktion, kann also gleichermaßen in der Form `"[<Vektor>, <Index>]"` verwendet werden (vgl. Abschn. 1.2.5, Fußnote 15).

⁴ Dies mag zunächst selbstverständlich erscheinen, in anderen Sprachen wird jedoch oft der Index 0 für die erste Position und allgemein der Index $n - 1$ für die n -te Position verwendet.

Länge des Vektors. Werden größere Indizes angegeben, erfolgt als Ausgabe die für einen fehlenden Wert stehende Konstante NA (vgl. Abschn. 2.12).

```
> age[4]
[1] 24

> (ageLast <- age[length(age)])
[1] 21

> age[length(age) + 1]
[1] NA
```

Ein Vektor muss nicht unbedingt einem Objekt zugewiesen werden, um indiziert werden zu können, dies ist auch für unbenannte Vektoren möglich.

```
> c(11, 12, 13, 14)[2]
[1] 12
```

Mehrere Elemente eines Vektors lassen sich gleichzeitig abrufen, indem ihre Indizes in Form eines Indexvektors in die eckigen Klammern eingeschlossen werden. Dazu kann zunächst ein eigener Vektor erstellt werden, dessen Name dann in die eckigen Klammern geschrieben wird. Ebenfalls kann der Befehl zum Erzeugen eines Vektors direkt in die eckigen Klammern verschachtelt werden. Der Indexvektor kann auch länger als der indizierte Vektor sein, wenn einzelne Elemente mehrfach ausgegeben werden sollen. Das Weglassen eines Index mit `(Vektor)[]` führt dazu, dass alle Elemente des Vektors ausgegeben werden.

```
> idx <- c(1, 2, 4)
> age[idx]
[1] 18 20 24

> age[c(3, 5, 6)]
[1] 30 23 21

> age[c(1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6)]
[1] 18 18 20 20 30 30 24 24 23 23 21 21
```

Beinhaltet der Indexvektor fehlende Werte (NA), erzeugt dies in der Ausgabe ebenfalls einen fehlenden Wert an der entsprechenden Stelle.

```
> age[c(4, NA, 1)]
[1] 25 NA 17
```

Wenn alle Elemente bis auf ein einzelnes abgerufen werden sollen, ist dies am einfachsten zu bewerkstelligen, indem der Index des nicht erwünschten Elements mit negativem Vorzeichen in die eckigen Klammern geschrieben wird.⁵ Sollen mehrere Elemente nicht ausgegeben werden, verläuft der Aufruf analog zum Aufruf gewünschter Elemente, wobei mehrere Variationen mit dem negativen Vorzeichen möglich sind.

⁵ Als Indizes dürfen in diesem Fall keine fehlenden Werte (NA) vorkommen, ebenso darf der Indexvektor nicht leer sein, muss also eine Länge größer als 0 besitzen.

```
> age[-3]                      # alle Elemente bis auf das 3.
[1] 18 20 24 23 21

> age[c(-1, -2, -4)]          # alle Elemente bis auf das 1., 2. und 4.
[1] 30 23 21

> age[-c(1, 2, 4)]            # alle Elemente bis auf das 1., 2. und 4.
[1] 30 23 21

> age[-idx]                   # alle Elemente bis auf die Indizes im Vektor idx
[1] 30 23 21
```

Die in einem Vektor gespeicherten Werte können nachträglich verändert werden. Dazu muss der Position des zu ändernden Wertes der neue Wert zugewiesen werden.

```
> age[4] <- 25; age
[1] 18 20 30 25 23 21
```

Das Verändern von mehreren Elementen gleichzeitig geschieht analog. Dazu lassen sich die Möglichkeiten zur Auswahl mehrerer Elementen nutzen und diesen in einem Arbeitsschritt passend viele neue Werte zuweisen. Dabei müssen die zugewiesenen Werte ebenfalls durch einen Vektor repräsentiert sein.

```
> age[idx] <- c(17, 30, 25); age
[1] 17 30 30 25 23 21
```

Um Vektoren zu verlängern, also mit neuen Elementen zu ergänzen, kann zum einen der [$\langle\text{Index}\rangle$] Operator benutzt werden, wobei als Index nicht belegte Positionen angegeben werden.⁶ Zum anderen kann auch hier die `c($\langle\text{Wert1}\rangle$, $\langle\text{Wert2}\rangle$, ...)` Funktion Verwendung finden. Als Alternative steht auch die `append(x= $\langle\text{Vektor}\rangle$, values= $\langle\text{Vektor}\rangle$)` Funktion zur Verfügung, die an einen Vektor x die Werte eines unter `values` genannten Vektors anhängt.

```
> charVec1 <- c("Z", "Y", "X")
> charVec1[c(4, 5, 6)] <- c("W", "V", "U"); charVec1
[1] "Z" "Y" "X" "W" "V" "U"

> (charVec2 <- c(charVec1, "T", "S", "R"))
[1] "Z" "Y" "X" "W" "V" "U" "T" "S" "R"

> (charVec3 <- append(charVec2, c("Q", "P", "O")))
[1] "Z" "Y" "X" "W" "V" "U" "T" "S" "R" "Q" "P" "O"
```

2.1.3 Datentypen in Vektoren

Vektoren können Werte unterschiedlicher Datentypen speichern, etwa `numeric`, wenn sie Zahlen beinhalten oder `character` im Fall von Zeichenketten (z.B.

⁶ Bei der Verarbeitung sehr großer Datenmengen ist zu bedenken, dass die schrittweise Vergrößerung von Objekten aufgrund der dafür notwendigen internen Kopiervorgänge ineffizient ist. Objekte sollten deshalb bevorzugt bereits mit der Größe angelegt werden, die sie später benötigen.

"asdf"). Letztere müssen dabei immer in Anführungszeichen eingegeben werden. Jeder einzelne Vektor kann aber nur einen Datentyp besitzen, alle Elemente haben also denselben Datentyp. Fügt man einem numerischen Vektor eine Zeichenkette hinzu, so werden seine numerischen Elemente automatisch in Zeichenketten umgewandelt,⁷ was man an den hinzugekommenen Anführungszeichen erkennt und mit `mode(<Vektor>)` überprüfen kann.

```
> charVec4 <- "word"
> numVec     <- c(10, 20, 30)
> (combVec  <- c(charVec4, numVec))
[1] "word" "10" "20" "30"

> mode(combVec)
[1] "character"
```

Zwei aus Zeichen bestehende Vektoren sind in R bereits vordefiniert, `LETTERS` und `letters`, die jeweils alle Buchstaben A–Z bzw. a–z in alphabetischer Reihenfolge als Elemente besitzen.

```
> LETTERS[c(1, 2, 3)]          # Alphabet in Großbuchstaben
[1] "A" "B" "C"

> letters[c(4, 5, 6)]          # Alphabet in Kleinbuchstaben
[1] "d"  "e"  "f"
```

2.1.4 Reihenfolge von Elementen kontrollieren

Um die Reihenfolge eines Vektors umzukehren, kann die `rev(<Vektor>)` Funktion (Reverse) benutzt werden.

```
> vec <- c(10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20)
> rev(vec)
[1] 20 19 18 17 16 15 14 13 12 11 10
```

Die Elemente von Vektoren können auch entsprechend ihrer Reihenfolge sortiert werden, die wiederum vom Datentyp des Vektors abhängt: bei numerischen Vektoren bestimmt die Größe der gespeicherten Zahlen, bei Vektoren aus Zeichenketten die alphabetische Reihenfolge der Elemente die Ausgabe. Zum Sortieren stehen die Funktionen `sort()` und `order()` zur Verfügung.⁸

```
> sort(x=<Vektor>, decreasing=FALSE)
> order(<Vektor>, decreasing=FALSE)
```

⁷ Allgemein gesprochen werden alle Elemente in den umfassendsten Datentyp umgewandelt, der notwendig ist, um alle Werte ohne Informationsverlust zu speichern (vgl. Abschn. 1.3.5).

⁸ Beide Funktionen sortieren stabil: Elemente gleicher Größe behalten ihre Reihenfolge relativ zueinander bei, werden also beim Sortiervorgang nicht zufällig vertauscht.

`sort()` gibt den sortierten Vektor direkt aus. Dagegen ist das Ergebnis von `order()` ein Indexvektor, der die Indizes des zu ordnenden Vektors in der Reihenfolge seiner Elemente enthält. Im Gegensatz zu `sort()` gibt `order()` also nicht schon die sortierten Datenwerte, sondern nur die zugehörigen Indizes aus, die anschließend zum Indizieren des Vektors verwendet werden können. Daher ist bei Vektoren `sort(<Vektor>)` äquivalent zu `<Vektor>[order(<Vektor>)]`. Der Vorteil von `order()` tritt beim Umgang mit Matrizen und Datensätzen zutage (vgl. Abschn. 2.8.10). Die Sortierreihenfolge wird über das Argument `decreasing` kontrolliert. Per Voreinstellung auf `FALSE` gesetzt, wird aufsteigend sortiert. Mit `decreasing=TRUE` ist die Reihenfolge absteigend.

```
> vec <- c(10, 12, 1, 12, 7, 16, 6, 19, 10, 19)
> sort(vec)
[1] 1 6 7 10 10 12 12 16 19 19

> (idxDec <- order(vec, decreasing=TRUE))
[1] 8 10 6 2 4 1 9 5 7 3

> vec[idxDec]
[1] 19 19 16 12 12 10 10 7 6 1
```

Die `rank(<Vektor>)` Funktion gibt für jedes Element eines Vektors seinen Rang an, der sich an der Position des Wertes im sortierten Vektor orientiert und damit der Ausgabe von `order()` ähnelt. Anders als bei `order()` erhalten identische Werte jedoch denselben Rang, wobei dieses Verhalten mit dem Argument `ties.method` kontrolliert werden kann.

```
> rank(c(3, 1, 2, 3))
[1] 3.5 1.0 2.0 3.5

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

Wenn Vektoren vom Datentyp `character` sortiert werden, so geschieht dieses in alphabetischer Reihenfolge. Auch als Zeichenkette gespeicherte Zahlen werden hierbei alphabetisch sortiert, d. h. die Zeichenkette "10" käme vor "4".

```
> sort(c("D", "E", "E", "A", "F", "E", "D", "A", "E", "A"))
[1] "A" "A" "A" "D" "D" "E" "E" "E" "E" "F"
```

2.1.5 Elemente benennen

Es ist möglich, die Elemente eines Vektors bei seiner Erstellung zu benennen. Die Elemente können dann nicht nur über ihren Index, sondern auch über ihren in Anführungszeichen gesetzten Namen angesprochen werden.⁹ In der Ausgabe wird der Name eines Elements in der über ihm stehenden Zeile mit aufgeführt.

⁹ Namen werden als Attribut gespeichert und sind mit `attributes(<Vektor>)` sichtbar (vgl. Abschn. 1.3).

```
> (namedVec1 <- c(elem1="first", elem2="second"))
elem1   elem2
"first" "second"

> namedVec1["elem1"]
elem1
"first"
```

Auch im nachhinein lassen sich Elemente benennen, bzw. vorhandene Benennungen ändern – beides geschieht mit der `names(<Vektor>)` Funktion.

```
> (namedVec2 <- c(val1=10, val2=-12, val3=33))
val1  val2  val3
 10   -12    33

> names(namedVec2)
[1] "val1" "val2" "val3"

> names(namedVec2) <- c("A", "B", "C"); namedVec2
 A     B     C
10   -12   33
```

2.1.6 Elemente löschen

Elemente eines Vektors lassen sich nicht im eigentlichen Sinne löschen. Derselbe Effekt kann stattdessen über zwei mögliche Umwege erreicht werden. Zum einen kann ein bestehender Vektor mit einer Auswahl seiner eigenen Elemente überschrieben werden.

```
> vec <- c(10, 20, 30, 40, 50)
> vec <- vec[c(-4, -5)]; vec
[1] 10 20 30
```

Zum anderen kann ein bestehender Vektor über die `length()` Funktion verkürzt werden, indem ihm eine Länge zugewiesen wird, die kleiner als seine bestehende ist. Gelöscht werden dabei die überzähligen Elemente am Ende des Vektors.

```
> vec           <- c(1, 2, 3, 4, 5)
> length(vec) <- 3; vec
[1] 1 2 3
```

2.1.7 Rechenoperationen mit Vektoren

Auf Vektoren lassen sich alle elementaren Rechenoperationen anwenden, die in Abschn. 1.2.4 für Skalare aufgeführt wurden. Vektoren können also in den meisten Rechnungen wie Einzelwerte verwendet werden. Die Berechnungen einer Funktion werden dann elementweise durchgeführt: die Funktion wird zunächst auf das erste

Element des Vektors angewendet, dann auf das zweite, usw., bis zum letzten Element. Das Ergebnis ist ein Vektor, der als Elemente die Einzelergebnisse besitzt. In der Konsequenz ähnelt die Schreibweise zur Transformation von in Vektoren gespeicherten Werten in R sehr der aus mathematischen Formeln gewohnten.

```
> age <- c(18, 20, 30, 24, 23, 21)
> age/10
[1] 1.8 2.0 3.0 2.4 2.3 2.1

> (age/2) + 5
[1] 14.0 15.0 20.0 17.0 16.5 15.5
```

Die Verwendbarkeit von Vektoren analog zu Einzelwerten erstreckt sich auch auf die Situation, dass mehrere Vektoren in einer Rechnung auftauchen. Die Vektoren werden dann elementweise entsprechend der gewählten Rechenoperation miteinander verrechnet. Dabei wird das erste Element des ersten Vektors mit dem ersten Element des zweiten Vektors z.B. multipliziert, ebenso das zweite Element des ersten Vektors mit dem zweiten Element des zweiten Vektors, usw.

```
> vec1 <- c(3, 4, 5, 6)
> vec2 <- c(-2, 2, -2, 2)
> vec1*vec2
[1] -6 8 -10 12

> vec3 <- c(10, 100, 1000, 10000)
> (vec1+vec2) / vec3
[1] 1e-01 6e-02 3e-03 8e-04
```

Die Zahlen der letzten Ausgabe sind in verkürzter Exponentialschreibweise dargestellt (vgl. Abschn. 1.2.4).

2.1.7.1 Zyklische Verlängerung von Vektoren (Recycling)

Die Verrechnung mehrerer Vektoren scheint aufgrund der elementweisen Zuordnung zunächst vorauszusetzen, dass die Vektoren dieselbe Länge haben. Tatsächlich ist dies nicht unbedingt notwendig, weil R in den meisten Fällen diesen Zustand ggf. selbsttätig herstellt. Dabei wird der kürzere Vektor intern von R zyklisch wiederholt (also sich selbst angefügt, sog. Recycling), bis er mindestens die Länge des längeren Vektors besitzt.¹⁰ Eine Warnmeldung wird in einem solchen Fall nur dann ausgegeben, wenn die Länge des längeren Vektors kein ganzzahliges Vielfaches der Länge des kürzeren Vektors ist. Dies ist gefährlich, weil meist Vektoren gleicher Länge miteinander verrechnet werden sollen und die Verwendung von Vektoren ungleicher Längen ein Hinweis auf einen Fehler in den Berechnungen sein kann.

¹⁰ Dieser Vorgang geschieht auch, wenn ein Vektor mit einer Zahl verrechnet wird – in diesem Fall wird die Zahl in einen Vektor passender Länge umgewandelt, dessen Elemente alle aus dieser Zahl bestehen.

```
> vec1 <- c(2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24)
> vec2 <- c(2, 4, 6, 8, 10)

> c(length(age), length(vec1), length(vec2))
[1] 6 12 5

> vec1*age
[1] 36 80 180 192 230 252 252 320 540 480 506 504

> vec2*age
[1] 36 80 180 192 230 42

Warning message:
Länge des längeren Objektes ist kein Vielfaches der Länge des kürzeren
Objektes in: vec2 * age
```

2.1.7.2 *z*-Transformation

Durch eine *z*-Transformation wird eine quantitative Variable x so normiert, dass sie den Mittelwert $M_x = 0$ und die Standardabweichung $s_x = 1$ besitzt. Dies geschieht für jeden Einzelwert x_i durch $(x_i - M_x)/s_x$. Die Funktion `mean(<Vektor>)` berechnet den Mittelwert (vgl. Abschn. 2.6.3), `sd(<Vektor>)` ermittelt die korrigierte Streuung (vgl. Abschn. 2.6.5).

```
> (zAge <- (age - mean(age)) / sd(age))
[1] -1.1166106 -0.6380632 1.7546739 0.3190316 0.0797579 -0.3987895
```

Eine andere Möglichkeit bietet die `scale(x=<Vektor>)` Funktion. Diese berechnet die *z*-Werte mit Hilfe der korrigierten Streuung, gibt sie jedoch nicht in Form eines Vektors, sondern als Matrix mit einer Spalte aus.¹¹ Weiterhin werden Mittelwert und korrigierte Streuung von x in Form von Attributen mit angegeben.

```
> (zAge <- scale(age))
[,1]
[1,] -1.1166106
[2,] -0.6380632
[3,] 1.7546739
[4,] 0.3190316
[5,] 0.0797579
[6,] -0.3987895

attr("scaled:center")
[1] 22.66667

attr("scaled:scale")
[1] 4.179314
```

¹¹ Für x kann auch eine Matrix übergeben werden, deren *z*-transformierte Spalten dann die Spalten der ausgegebenen Matrix ausmachen (vgl. Abschn. 2.8).

Um die ausgegebene Matrix wieder in einen Vektor zu verwandeln, muss sie wie in Abschn. 2.8.2 dargestellt mit `as.vector(Matrix)` konvertiert werden.

```
> as.vector(zAge)
[1] -1.1166106 -0.6380632 1.7546739 0.3190316 0.0797579 -0.3987895
```

Durch Umkehrung des Prinzips der z -Transformation lassen sich empirische Datenreihen so skalieren, dass sie einen beliebigen Mittelwert M_{neu} und eine beliebige Streuung s_{neu} besitzen. Dies geschieht für eine z -transformierte Variable z mit $z \cdot s_{\text{neu}} + M_{\text{neu}}$.

```
> newSd    <- 15
> newMean  <- 100
> (newAge  <- (as.vector(zAge)*newSd) + newMean)
[1] 83.25084 90.42905 126.32011 104.78547 101.19637 94.01816

> mean(newAge)
[1] 100

> sd(newAge)
[1] 15
```

2.1.7.3 Neue aus bestehenden Variablen bilden

Das elementweise Verrechnen mehrerer Vektoren kann, analog zur z -Transformation, allgemein zur flexiblen Neubildung von Variablen aus bereits bestehenden Daten genutzt werden.

Ein Beispiel sei die Berechnung des Body-Mass-Index (BMI) einer Person, für den ihr Körpergewicht in kg durch das Quadrat ihrer Körpergröße in m geteilt wird.

```
> height <- c(1.78, 1.91, 1.89, 1.83, 1.64)
> weight <- c(65, 89, 91, 75, 73)
> (bmi   <- weight / (height^2))
[1] 20.51509 24.39626 25.47521 22.39541 27.14158
```

In einem zweiten Beispiel soll die Summenvariable aus drei dichotomen Items („trifft zu“: TRUE, „trifft nicht zu“: FALSE) eines an 8 Personen erhobenen Fragebogens gebildet werden. Dies ist die Variable, die jeder Person den Summenscore aus ihren vorherigen Antworten zuordnet, also angibt, wie viele Items von der Person als zutreffend angekreuzt wurden. Logische Werte verhalten sich bei numerischen Rechnungen wie 1 (TRUE) bzw. 0 (FALSE).

```
> quest1 <- c(FALSE, FALSE, FALSE, TRUE, FALSE, TRUE, FALSE, TRUE)
> quest2 <- c(TRUE, FALSE, FALSE, FALSE, TRUE, TRUE, TRUE, FALSE)
> quest3 <- c(TRUE, TRUE, TRUE, TRUE, FALSE, TRUE, FALSE, FALSE)
> (sumVar <- quest1 + quest2 + quest3)
[1] 2 1 1 2 1 3 1 1
```

2.2 Logische Operatoren

Verarbeitungsschritte mit logischen Vergleichen und Werten treten häufig bei der Auswahl von Teilmengen von Daten sowie bei der Recodierung von Datenwerten auf. Dies liegt vor allem an der Möglichkeit, in Vektoren und anderen Datenstrukturen gespeicherte Werte auch mit logischen Indexvektoren auszuwählen.

2.2.1 Logische Operatoren zum Vergleich von Vektoren

Vektoren werden oft mit Hilfe logischer Operatoren mit einem bestimmten Wert oder auch mit anderen Vektoren verglichen um zu prüfen, ob die Elemente gewisse Bedingungen erfüllen. Als Ergebnis der Prüfung wird ein logischer Vektor mit Wahrheitswerten ausgegeben, der die Resultate der elementweisen Anwendung des Operators beinhaltet.

Als Beispiel seien im Vektor age wieder die Daten von sechs Versuchspersonen (VPn) gespeichert. Zunächst werden jene VPn identifiziert, die jünger als 25 Jahre sind. Dazu wird der < Operator verwendet, der als Ergebnis einen Vektor mit Wahrheitswerten liefert, der für jedes Element separat angibt, ob die Bedingung < 25 zutrifft. Andere Vergleichsoperatoren, wie gleich (==), ungleich (!=), etc. funktionieren analog.

```
> age <- c(17, 30, 30, 24, 23, 21)
> age < 24
[1] TRUE FALSE FALSE FALSE TRUE TRUE
```

Wenn analog zwei Vektoren miteinander logisch verglichen werden, wird der Operator immer auf ein zueinander gehörendes Wertepaar angewendet, also auf Werte, die sich an derselben Position in ihrem jeweiligen Vektor befinden.

```
> x <- c(2, 4, 8)
> y <- c(3, 4, 5)
> x == y
[1] FALSE TRUE FALSE

> x < y
[1] TRUE FALSE FALSE
```

Auch die Prüfung jedes Elements auf mehrere Kriterien ist möglich. Wenn zwei Kriterien gleichzeitig erfüllt sein sollen, wird & als Symbol für das logische UND verwendet, wenn nur eines von zwei Kriterien erfüllt sein muss, das Symbol | für das logische, d. h. einschließende, ODER. Um sicherzustellen, dass R die zusammengehörenden Ausdrücke auch als Einheit erkennt, ist die Verwendung runder Klammern zu empfehlen.

```
> (age <= 20) | (age >= 30)          # Werte im Bereich bis 20 ODER ab 30?
[1] TRUE TRUE TRUE FALSE FALSE FALSE

> (age > 20) & (age < 30)           # Werte im Bereich zwischen 20 und 30?
[1] FALSE FALSE FALSE TRUE TRUE TRUE
```

Während die elementweise Prüfung von Vektoren den häufigsten Fall der Anwendung logischer Kriterien ausmacht, sind vor allem zur Fallunterscheidung (vgl. Abschn. 11.2.1) auch Prüfungen notwendig, die in Form eines einzelnen Wahrheitswertes eine summarische Auskunft darüber liefern, ob Kriterien erfüllt sind. Diese auch bei Anwendung auf Vektoren nur einen Wahrheitswert ergebenden Prüfungen lassen sich mit `&&` für das logische UND bzw. mit `||` für das logische ODER formulieren. Beide Vergleiche werten nur das jeweils erste Element aus, wenn Vektoren beteiligt sind.

```
> c(TRUE, FALSE, FALSE) && c(TRUE, TRUE, FALSE)
[1] TRUE

> c(FALSE, FALSE, TRUE) || c(FALSE, TRUE, FALSE)
[1] FALSE
```

Sollen Werte nur auf ungefähre Übereinstimmung geprüft werden, kann dies mit `all.equal()` geschehen (vgl. Abschn. 1.3.6). Dabei ist im Fall von zu vergleichenden Vektoren zu beachten, dass die Funktion nicht die Ergebnisse der elementweisen Einzelvergleiche als Vektor ausgibt. Stattdessen liefert sie nur einen einzelnen Wert zurück, entweder TRUE im Fall der paarweisen Übereinstimmung aller Elemente oder das mittlere Abweichungsmaß im Fall der Ungleichheit. Um auch in letzterem Fall einen Wahrheitswert als Ausgabe zu erhalten, sollte die `isTRUE()` Funktion verwendet werden.

```
> x <- c(4, 5, 6)
> y <- c(4, 5, 6)
> z <- c(1, 2, 3)
> all.equal(x, y)
[1] TRUE
> all.equal(y, z)
[1] "Mean relative difference: 0.6"

> isTRUE(all.equal(y, z))
[1] FALSE
```

Bei der Prüfung von Elementen auf Kriterien kann mit Hilfe spezialisierter Funktionen summarisch analysiert werden, ob diese Kriterien zutreffen. Ob mindestens ein Element eines logischen Vektors den Wert TRUE besitzt, zeigt `any(<Vektor>)`, ob alle Elemente den Wert TRUE haben, gibt `all(<Vektor>)` an.

```
> res <- age > 30
> any(res)
[1] FALSE

> any(age < 18)
[1] TRUE

> all(x == y)
[1] TRUE
```

Um zu zählen, auf wie viele Elemente eines Vektors ein Kriterium zutrifft, wird die Funktion `sum(<Vektor>)` verwendet, die alle Werte eines Vektors aufaddiert (vgl. Abschn. 2.6.1).

```
> res <- age < 24
> sum(res)
[1] 3
```

Alternativ kann verschachtelt in `length()` die `which(<Vektor>)` Funktion genutzt werden, die die Indizes der Elemente mit dem Wert TRUE ausgibt (vgl. Abschn. 2.2.2).

```
> which(age < 24)
[1] 1 5 6

> length(which(age < 24))
[1] 3
```

2.2.2 Logische Indexvektoren

Vektoren von Wahrheitswerten können zur Indizierung anderer Vektoren benutzt werden. Diese Art zu indizieren kann z. B. zur Auswahl von durch bestimmte Merkmale definierte Teilstichproben genutzt werden. Hat ein Element des logischen Indexvektors den Wert TRUE, so wird dabei das sich an dieser Position befindliche Element des indizierten Vektors ausgegeben. Hat der logische Indexvektor an einer Stelle den Wert FALSE, so wird das zugehörige Element des indizierten Vektors ausgelassen.

```
> age[c(TRUE, FALSE, TRUE, TRUE, FALSE, TRUE)]
[1] 17 30 24 21
```

Wie numerische können auch logische Indizes zunächst in einem Vektor gespeichert werden, mit dem die Indizierung dann später geschieht. Statt der Erstellung eines separaten logischen Indexvektors, z. B. als Ergebnis einer Überprüfung von Bedingungen, kann der Schritt aber auch übersprungen und der logische Ausdruck direkt innerhalb des `[<Index>]` Operators benutzt werden. Dabei ist jedoch abzuwägen, ob der Übersichtlichkeit und Nachvollziehbarkeit der Befehle mit einer separaten Erstellung von Indexvektoren besser gedient ist.

```
> (idx <- (age <= 20) | (age >= 30))    # Werte im Bereich bis 20 ODER ab 30?
[1] TRUE TRUE TRUE FALSE FALSE FALSE

> age[idx]
[1] 17 30 30

> age[(age >= 30) | (age <= 20)]
[1] 17 30 30
```

Logische Indexvektoren bringen zwei Nachteile mit sich: zum einen ist die Ausgabe von Wahrheitswerten nach einer Überprüfung von Kriterien zwar bei nur wenigen Elementen noch übersichtlich. Je länger der Vektor jedoch ist, desto schwieriger wird es, die Wahrheitswerte den geprüften Elementen zuzuordnen. Zum anderen können logische Indexvektoren dann zu Problemen führen, wenn der zu prüfende Vektor fehlende Werte enthält. Überall dort, wo dieser NA ist, wird i. d. R. auch das Ergebnis eines logischen Vergleichs NA sein, d. h. der resultierende logische Indexvektor enthält seinerseits fehlende Werte (vgl. Abschn. 2.1.3, Fußnote 48).

```
> vecNA <- c(-3, 2, 0, NA, -7, 5)          # Vektor mit fehlendem Wert
> (idx <- vecNA > 0)                      # prüfe auf Werte größer 0
[1] FALSE TRUE FALSE NA FALSE TRUE
```

Enthält ein Indexvektor einen fehlenden Wert, erzeugt er beim Indizieren eines anderen Vektors an dieser Stelle ebenfalls ein NA in der Ausgabe (vgl. Abschn. 2.1.2). Dies führt dazu, dass sich der Indexvektor nicht mehr dazu eignet, ausschließlich die Werte auszugeben, die eine bestimmte Bedingung erfüllen.

```
> vecNA[idx]                                # Verwendung von idx erzeugt NA
[1] 2 NA 5
```

In Situationen, in denen fehlende Werte möglich sind, ist deshalb ein anderes Vorgehen sinnvoller: statt eines logischen Indexvektors sollten die numerischen Indizes der Elemente zum Indizieren verwendet werden, die die geprüfte Bedingung erfüllen, an deren Position der logische Vektor also den Wert TRUE besitzt. Logische in numerische Indizes wandelt in diesem Sinne die which(<Vektor>) Funktion um, die die Indizes der TRUE Werte zurückgibt.¹²

```
> (logIdx <- (age < 24))                  # prüfe auf Werte kleiner 24
[1] TRUE FALSE FALSE FALSE TRUE TRUE

> (numIdx <- which(logIdx))                # Indizes der TRUE Werte
[1] 1 5 6
```

2.2.3 Werte ersetzen oder recodieren

Mitunter werden Variablen zunächst auf eine bestimmte Art codiert, die sich für manche Auswertungen dann als nicht zweckmäßig erweist und deswegen geändert werden soll. Ebenso können vorhandene Variablen als Basis für neu zu erstellende benutzt werden, deren Werte durch Recodierung von Wertebereichen der ursprünglichen Variable entstehen sollen – etwa zur Umwandlung einer quantitativen in eine kategoriale Variable. In beiden Fällen müssen bestimmte Werte gesucht und durch

¹² Umgekehrt lassen sich auch die in <Indexvektor> gespeicherten numerischen Indizes für <Vektor> in logische verwandeln: seq(along=<Vektor>)%in% <Indexvektor> (vgl. Abschn. 2.4.1 und 2.3.2).

andere ersetzt werden, was sich mit logischen Indexvektoren erreichen lässt.¹³ Dabei sollte immer zunächst ein neues Objekt passender Länge für die recodierten Werte erstellt werden, statt die Werte des alten Objekts zu überschreiben.

In einem Vektor seien die Lieblingsfarben von sechs englischsprachigen VPn erhoben worden. Später soll die Variable auf deutsche Farbnamen recodiert werden.

```
> myColors <- c("red", "blue", "blue", "chartreuse", "red", "green")
> farben <- character(length(myColors)) # neuen Vektor erstellen
> farben[myColors == "red"] <- "rot"
> farben[myColors == "blue"] <- "blau"
> farben[myColors == "chartreuse"] <- "gelbgrün"
> farben[myColors == "green"] <- "grün"
> farben
[1] "rot" "blau" "blau" "gelbgrün" "rot" "grün"
```

An anderen VPn sei der IQ-Wert erhoben worden, der zu einer Klasseneinteilung genutzt werden soll.¹⁴

```
> IQ <- c(112, 103, 87, 86, 90, 101, 90, 89, 122, 103)
> IQclass <- numeric(length(IQ)) # neuen Vektor erstellen
> IQclass[IQ <= 100] <- 1
> IQclass[(IQ > 100) & (iq <= 115)] <- 2
> IQclass[IQ > 115] <- 3
> IQclass
[1] 2 2 1 1 1 2 1 1 3 2
```

Mit der `replace()` Funktion können auf sehr ähnliche Weise Werte eines Vektors ausgetauscht werden.

```
> replace(x=<Vektor>, list=<Indexvektor>, values=<neue Werte>)
```

Der Vektor mit den auszutauschenden Elementen ist unter `x` zu nennen. Welche Werte geändert werden sollen, gibt der Indexvektor `list` an, bei dem es sich auch um einen logischen Vektor, etwa als Resultat eines Vergleichs, handeln kann. Der Vektor `values` definiert, welche Werte an den durch `list` bezeichneten Indizes neu einzufügen sind. Da `replace()` den unter `x` angegebenen Vektor nicht verändert, muss das Ergebnis ggf. einem neuen Objekt zugewiesen werden.

```
> replace(c(1, 2, 3, 4, 5), list=c(2, 4), values=c(200, 400))
[1] 1 200 3 400 5
```

Gilt es, Werte entsprechend einer dichotomen Entscheidung durch andere zu ersetzen, kann dies auch mit der `ifelse()` Funktion geschehen.

```
> ifelse(test=<logischer Ausdruck>, yes=<Wert>, no=<Wert>)
```

Für das Argument `test` muss ein Ausdruck angegeben werden, der sich zu einem logischen Wert auswerten lässt, der also WAHR (TRUE) oder FALSCH (FALSE) ist.

¹³ Die Funktion `recode()` aus dem `car` Paket (Fox, 2009) bietet eine weitere Möglichkeit, Werte nach einem bestimmten Muster durch andere zu ersetzen.

¹⁴ Bei numerischen Variablen ist dies einfacher mit der Funktion `cut()` zu erreichen, vgl. Abschn. 2.7.6.

Ist `test` WAHR, wird der unter `yes` eingetragene Wert zurückgegeben, andernfalls der unter `no` genannte. Ist `test` ein Vektor, wird jedes seiner Elemente daraufhin geprüft, ob es TRUE oder FALSE ist und ein Vektor mit den passenden, unter `yes` und `no` genannten Werten als Elementen zurückgegeben. Die Ausgabe hat also immer dieselbe Länge wie die von `test`.

```
> ifelse(c(6, 2, 4, 5, 4, 2, 5, 4, 6, 6) >= 4, "hi", "lo")
[1] "hi" "lo" "hi" "hi" "hi" "lo" "hi" "hi" "hi" "hi"
```

Die Argumente `yes` und `no` können selbst Vektoren derselben Länge wie `test` sein – ist dann etwa das dritte Element von `test` gleich TRUE, wird als drittes Element des Ergebnisses das dritte Element von `yes` zurückgegeben, andernfalls das dritte Element von `no`. Indem für `yes` ebenfalls der in `test` zu prüfende Vektor eingesetzt wird, können so bestimmte Werte eines Vektors ausgetauscht, andere dagegen unverändert gelassen werden. Dies erlaubt es etwa, alle Werte größer einem Cutoff-Wert auf denselben Maximalwert zu setzen und die übrigen Werte beizubehalten.

```
> orgVec <- c(5, 9, 11, 8, 9, 3, 1, 13, 9, 12, 5, 12, 6, 3, 17, 5, 8, 7)
> cutoff <- 10
> (reVec <- ifelse(orgVec <= cutoff, orgVec, cutoff))
[1] 5 9 10 8 9 3 1 10 9 10 5 10 6 3 10 5 8 7
```

2.3 Mengen

Werden Vektoren als Wertemengen im mathematischen Sinn betrachtet, ist zu beachten, dass die Elemente einer Menge nicht geordnet sind und mehrfach vorkommende Elemente wie ein einzelnes behandelt werden. Anders gesagt werden duplizierte Elemente einer Menge gelöscht – so ist z. B. die Menge {1, 1, 2, 2} gleich der Menge {2, 1}.

2.3.1 Duplizierte Werte behandeln

Die `duplicated(x=(Vektor))` Funktion gibt für jedes Element eines Vektors an, ob der Wert bereits an einer früheren Stelle des Vektors aufgetaucht ist. Die Funktion `unique(x=(Vektor))` nennt alle voneinander verschiedenen Werte eines Vektors, mehrfach vorkommende Werte werden also nur einmal aufgeführt. Die Funktion eignet sich in Kombination mit `length()` zum Zählen der tatsächlich vorkommenden unterschiedlichen Werte in einer Variable.

```
> duplicated(c(1, 1, 1, 3, 3, 4, 4))
[1] FALSE TRUE TRUE FALSE TRUE FALSE TRUE

> unique(c(1, 1, 1, 3, 3, 4, 4))
[1] 1 3 4

> length(unique(c("A", "B", "C", "C", "B", "B", "A", "C", "C", "A")))
[1] 3
```

2.3.2 Mengenoperationen

Die `union(x=<Vektor1>, y=<Vektor2>)` Funktion bildet die Vereinigungsmenge von x und y. Das Ergebnis sind die Werte, die Element mindestens einer der beiden Mengen sind, wobei duplizierte Werte gelöscht werden. Wird das Ergebnis als Menge betrachtet, spielt die Reihenfolge, in der x und y genannt werden, keine Rolle.

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

> union(x, y)
[1] 2 1 3 5 4

> union(y, x)
[1] 5 3 1 4 2
```

Die Schnittmenge zweier Mengen wird mit `intersect(x=<Vektor1>, y=<Vektor2>)` erzeugt. Das Ergebnis sind die Werte, die sowohl Element von x als auch Element von y sind, wobei duplizierte Werte gelöscht werden. Auch hier ist die Reihenfolge von x und y unerheblich, wenn das Ergebnis als Menge betrachtet wird.

```
> intersect(x, y)
[1] 1 3

> intersect(y, x)
[1] 3 1
```

Mit `setequal(x=<Vektor1>, y=<Vektor2>)` lässt sich prüfen, ob als Mengen betrachtete Vektoren identisch sind.

```
> setequal(c(1, 1, 2, 2), c(2, 1))
[1] TRUE
```

Die Funktion `setdiff(x=<Vektor1>, y=<Vektor2>)` liefert als Ergebnis all jene Elemente der von x gebildeten Menge, die nicht Element von y sind. Im Unterschied zu den oben behandelten Mengenoperationen ist die Reihenfolge, in der x und y angegeben sind, bedeutsam, auch wenn das Ergebnis als Menge betrachtet wird.¹⁵

```
> setdiff(x, y)
[1] 2

> setdiff(y, x)
[1] 5 4
```

Soll jedes Element eines Vektors daraufhin geprüft werden, ob es Element einer Menge ist, kann die `is.element(el=<Menge1>, set=<Menge2>)` Funktion genutzt werden. Unter el ist der Vektor mit den zu prüfenden Elementen einzutragen

¹⁵ Die symmetrische Differenz zweier Mengen ergibt sich also aus `union(setdiff(<Menge1>, <Menge2>), setdiff(<Menge2>, <Menge1>))`.

und unter `set` die durch einen Vektor definierte Menge. Als Ergebnis wird ein logischer Vektor ausgegeben, der für jedes Element von `e1` angibt, ob es in `set` enthalten ist. Die Kurzform in Operator-Schreibweise lautet `(Menge1) %in% (Menge2)`.

```
> is.element(c(29, 23, 30, 17, 30, 10), c(30, 23))
[1] FALSE TRUE TRUE FALSE TRUE FALSE
```

```
> c("A", "Z", "B") %in% c("A", "B", "C", "D", "E")
[1] TRUE FALSE TRUE
```

Durch `all((A) %in% (B))` lässt sich so prüfen, ob `(A)` eine Teilmenge von `(B)`, ob also jedes Element von `(A)` auch Element von `(B)` ist. Dabei ist `(A)` eine echte Teilmenge von `(B)`, wenn sowohl `all((A) %in% (B))` gleich TRUE als auch `all((B) %in% (A))` gleich FALSE ist.

```
> A <- c(4, 5, 6)
> B <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
> (AinB <- all(A %in% B))           # A Teilmenge von B?
[1] TRUE
```

```
> (BinA <- all(B %in% A))          # B Teilmenge von A?
[1] FALSE
```

```
> AinB & !BinA                      # A echte Teilmenge von B?
[1] TRUE
```

In Kombination mit dem `ifelse()` Befehl kann `%in%` beispielsweise genutzt werden, um eine kategoriale Variable so umzucodieren, dass alle nicht in einer bestimmten Menge auftauchenden Werte in einer Kategorie „Sonstiges“ zusammengefassst werden. Im konkreten Beispiel sollen nur die ersten 15 Buchstaben des Alphabets als solche beibehalten, alle anderen Werte zu "other" recodiert werden.

```
> targetSet <- c("A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K")
> response <- c("Z", "E", "O", "W", "H", "G", "I", "G", "A", "O", "B")
> (respRec <- ifelse(response %in% targetSet, response, "other"))
[1] "other" "E" "other" "other" "H" "C" "I" "G" "A" "other" "B"
```

2.3.3 Kombinatorik

Aus dem Bereich der Kombinatorik sind im Rahmen der Datenauswertung bisweilen zwei Themen von Bedeutung, nämlich zum einen die Zusammenstellung von Teilmengen aus Elementen einer Grundmenge. Dabei ist die Reihenfolge der Elemente innerhalb einer Teilmenge meist nicht bedeutsam, d. h. es handelt sich um eine sog. Kombination.¹⁶ Zum anderen kann die Zusammenstellung von Elementen aus verschiedenen Grundmengen notwendig sein, wobei jeweils ein Element aus jeder Grundmenge beteiligt sein soll.

¹⁶ Werden alle Elemente einer Grundmenge ohne Zurücklegen unter Beachtung der Reihenfolge gezogen, handelt es sich um eine sog. Permutation. Die Funktion `permn()` aus dem `combinat` Paket (Chasalow, 2009) stellt alle $n!$ Permutationen einer Grundmenge mit n Elementen als Komponenten einer Liste zusammen (vgl. Abschn. 3.1).

Die Kombination entspricht dem Ziehen aus einer Grundmenge ohne Zurücklegen sowie ohne Berücksichtigung der Reihenfolge. Oft wird die Anzahl der Elemente der Grundmenge mit n , die Anzahl der gezogenen Elemente mit k und die Kombination deshalb mit k -Kombination bezeichnet. Insgesamt gibt es $n!/(k!(n - k)!)$ viele k -Kombinationen, dies entspricht den Binomialkoeffizienten $\binom{n}{k}$. Die Fakultät einer Zahl wird mit `factorial(<Zahl>)` berechnet (vgl. Abschn. 2.6.1). Da eine k -Kombinationen die Anzahl der Möglichkeiten darstellt, aus einer Menge mit n Elementen k auszuwählen, spricht man im Englischen beim Binomialkoeffizienten von „ n choose k “, woraus sich der Name der `choose(n=<Zahl>, k=<Zahl>)` Funktion ableitet, die ihn ermittelt.

```
> myN <- 5
> myK <- 4
> factorial(myN) / (factorial(myK)*factorial(myN-myK))
[1] 5

> choose(myN, myK)
[1] 5
```

Möchte man alle k -Kombinationen einer gegebenen Grundmenge x auch explizit anzeigen lassen, kann dies mit der Funktion `combn()` geschehen.

```
> combn(x=<Vektor>, m=<Zahl>, simplify=TRUE, FUN=<Funktion>, ...)
```

Die Zahl m entspricht dabei dem k der bisherigen Terminologie. Mit `simplify=TRUE` erfolgt die Ausgabe auf möglichst einfache Weise, d. h. nicht als Liste (vgl. Abschn. 3.1), sondern als Vektor oder wie hier als Matrix, in der in jeder Spalte eine der k -Kombinationen aufgeführt ist (vgl. Abschn. 2.8).

```
> combn(c("a", "b", "c", "d", "e"), myK)
[,1] [,2] [,3] [,4] [,5]
[1,] "a"  "a"  "a"  "a"  "b"
[2,] "b"  "b"  "b"  "c"  "c"
[3,] "c"  "c"  "d"  "d"  "d"
[4,] "d"  "e"  "e"  "e"  "e"
```

Die `combn()` Funktion lässt sich darüber hinaus anwenden, um in einem Arbeitsschritt eine frei wählbare Funktion auf alle gebildeten k -Kombinationen von x anzuwenden. Das Argument `FUN` erwartet hierfür eine Funktion, die einen Kennwert jedes sich als Kombination ergebenden Vektors bestimmt. Benötigt `FUN` ihrerseits weitere Argumente, so können diese unter ... durch Komma getrennt an `combn()` übergeben werden.¹⁷

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

¹⁷ Dasselbe Ergebnis ließe sich auch durch Verwendung der `apply()` Funktion erzielen (vgl. Abschn. 2.8.7): `apply(combn(c(1, 2, 3, 4), 3), 2, sum)`.

```
> combn(c(1, 2, 3, 4), 3, sum)
[1] 6 7 8 9

# gewichtetes Mittel jeder Kombination mit Argument w für die Gewichte
> combn(c(1, 2, 3, 4), 3, weighted.mean, simplify=TRUE, w=c(0.1, 0.2, 0.3))
[1] 2.333333 2.833333 3.166667 3.333333
```

Die Funktion `expand.grid()` bildet alle Kombinationen von Elementen mehrerer Grundmengen, wobei jeweils ein Element aus jeder Grundmenge stammt und die Reihenfolge nicht berücksichtigt wird. Dies entspricht der Situation, dass aus den Stufen mehrerer Unabhängiger Variablen (UVn) alle Kombinationen von Faktorstufen gebildet werden. Das Ergebnis von `expand.grid()` ist ein Datensatz (vgl. Abschn. 3.2), bei dem jede Kombination in einer Zeile steht. Die zuerst genannte Variable variiert dabei am schnellsten über die Zeilen, die anderen entsprechend ihrer Position im Funktionsaufruf langsamer.

```
> IV1 <- c("control", "treatment")
> IV2 <- c("f", "m")
> IV3 <- c(1, 2)
> expand.grid(IV1, IV2, IV3)
   Var1 Var2 Var3
1   control   f    1
2 treatment   f    1
3   control   m    1
4 treatment   m    1
5   control   f    2
6 treatment   f    2
7   control   m    2
8 treatment   m    2
```

Soll es sich beim Ergebnis von `expand.grid()` tatsächlich um vollständig gekreuzte Faktorstufen im versuchsplanerischen Sinn handeln, so sollten die einzelnen Variablen Objekte der Klasse `factor` sein (vgl. Abschn. 2.7, insbesondere 2.7.5).

2.4 Numerische Sequenzen und feste Wertefolgen erzeugen

Ein häufig auftretender Arbeitsschritt in R ist die Erstellung von Zahlenfolgen nach vorgegebenen Gesetzmäßigkeiten, die sich etwa auf die sequentielle Abfolge von Zahlen oder die Wiederholung von Wertemustern beziehen.

2.4.1 Numerische Sequenzen erstellen

Zahlenfolgen mit Einerschritten, etwa für eine fortlaufende Numerierung, können mit Hilfe des Operators `(Startwert):(Endwert)` erzeugt werden – in aufsteigender wie auch in absteigender Reihenfolge.

```
> 20:26
[1] 20 21 22 23 24 25 26
```

```
> 26:20
[1] 26 25 24 23 22 21 20
```

Bei Zahlenfolgen im negativen Bereich sollten Klammern Verwendung finden, um nicht versehentlich eine nicht gemeinte Sequenz zu produzieren.

```
> -4:2                  # negatives Vorzeichen bezieht sich nur auf die 4
[1] -4 -3 -2 -1 0 1 2
```

```
> -(4:2)                 # negatives Vorzeichen bezieht sich auf Sequenz 4:2
[1] -4 -3 -2
```

Eine Zahlenfolge ist nicht darauf beschränkt, in Einerschritten erstellt zu werden. Man kann durch Verwendung der

```
> seq(from=<Zahl>, to=<Zahl>, by=<Schrittweite>, length.out=<Länge>)
```

Funktion auch Start- (*from*) und Endpunkt (*to*) des durch die Sequenz abzudeckenden Intervalls ebenso wählen wie die gewünschte Schrittweite (*by*) bzw. stattdessen die gewünschte Anzahl der Elemente der Zahlenfolge (*length.out*).

```
> seq(from=2, to=12, by=2)
[1] 2 4 6 8 10 12
```

```
> seq(from=0, to=-1, length.out=5)
[1] 0.00 -0.25 -0.50 -0.75 -1.00
```

Eine weitere Möglichkeit zum Erstellen einer bei 1 beginnenden Sequenz in Einerschritten, die genauso lang ist wie ein bereits vorhandener Vektor, besteht mit `seq(along=<Vektor>)`. Dabei muss `along=<Vektor>` das einzige Argument von `seq()` sein. Dies ist die bevorzugte Art, für einen vorhandenen Vektor den passenden Vektor seiner Indizes zu erstellen. Vermieden werden sollte dagegen die `1:length(<Vektor>)` Sequenz, deren Behandlung von Vektoren der Länge 0 meist nicht sinnvoll ist.

```
> age <- c(18, 20, 30, 24, 23, 21)
> seq(along=age)
[1] 1 2 3 4 5 6
```

```
> vec <- numeric(0)          # leeren Vektor (Länge 0) erzeugen
> 1:length(vec)            # hier unerwünschtes Ergebnis: Sequenz 1:0
[1] 1 0

> seq(along=vec)           # sinnvolleres Ergebnis: leerer Vektor
[1] integer(0)
```

Eine seltener einsetzbare Funktion ist `sequence(<Vektor>)`, die für jedes Element (*Zahl*) von (*Vektor*) die Sequenz `1:<Zahl>` erzeugt und alle so erstellten Einzelsequenzen in der Ausgabe aneinanderhängt.

```
> sequence(c(3, 2))        # entspricht c(1:3, 1:2)
[1] 1 2 3 1 2
```

```
> sequence(1:3)            # entspricht c(1:1, 1:2, 1:3)
[1] 1 1 2 1 2 3
```

2.4.2 Wertefolgen wiederholen

Eine andere Art von Wertefolgen kann mit der `rep()` Funktion (Repeat) erzeugt werden, die Elemente wiederholt ausgibt.

```
> rep(x=<Vektor>, times=<Anzahl>, each=<Anzahl>)
```

Für `x` ist der zu wiederholende Vektor einzutragen, etwa eine Zahlenfolge. Die Replikation von `x` kann auf zwei verschiedene Arten geschehen. Mit dem Argument `times` wird der Vektor als Ganzes so oft aneinander gehängt wie gefordert.

```
> rep(1:3, times=5)
[1] 1 2 3 1 2 3 1 2 3 1 2 3
```

Wird das Argument `each` verwendet, wird jedes Element von `<Vektor>` einzeln mit der gewünschten Häufigkeit wiederholt, bevor das nächste Element von `<Vektor>` einzeln wiederholt und angehängt wird.

```
> rep(age, each=2)
[1] 18 18 20 20 30 30 24 24 23 23 21 21
```

Solche wiederholten Sequenzen können ihrerseits wieder benutzt werden, um z.B. die Elemente anderer Vektoren wiederholt auszugeben.

```
> vec <- c(1, 4, 8)
> (idx <- rep(c(1, 3, 1, 2, 3), times=2))
[1] 1 3 1 2 3 1 3 1 2 3

> vec[idx]
[1] 1 8 1 4 8 1 8 1 4 8
```

Wird als zweites Argument von `rep()` ebenfalls ein Vektor angegeben, so muss dieser dieselbe Länge wie der zu wiederholende Vektor besitzen – hier wird ein kürzerer Vektor durch R nicht selbsttätig zyklisch wiederholt. Jedes Element des zweiten Vektors gibt an, wie häufig das an gleicher Position stehende Element des ersten Vektors wiederholt werden soll, ehe das nächste Element des zu replizierenden Vektors wiederholt und angehängt wird.

```
> rep(c("A", "B", "C"), c(2, 3, 4))
[1] "A" "A" "B" "B" "C" "C" "C" "C"
```

2.5 Zufallszahlen und zufällige Reihenfolgen generieren

Zufallszahlen sind ein unverzichtbares Hilfsmittel der Datenauswertung, wobei ihnen insbesondere in der Planung von Analysen anhand simulierter Daten vor einer tatsächlichen Datenerhebung eine große Bedeutung zukommt. Zufällige Datensätze können unter Einhaltung vorgegebener Wertebereiche und anderer Randbedingungen erstellt werden. So können sie empirische Gegebenheiten realistisch widerspiegeln und statistische Voraussetzungen der eingesetzten Verfahren berücksichtigen.¹⁸

¹⁸ Wenn hier und im folgenden von Zufallszahlen die Rede ist, sind immer sog. Pseudozufallszahlen gemeint. Diese kommen nicht im eigentlichen Sinn zufällig zustande, sind aber von tatsächlich

Aber auch bei der zufälligen Auswahl von Teilstichproben eines Datensatzes oder beim Erstellen zufälliger Reihenfolgen zur Zuordnung von VPn auf experimentelle Bedingungen kommen Zufallszahlen zupass.

2.5.1 Zufällig aus einer Urne ziehen

Die einfachste Art, einen aus zufälligen Werten bestehenden Vektor zu erstellen, stellt die Funktion `sample()` zur Verfügung, die das Ziehen aus einer Urne simuliert.

```
> sample(x=<Vektor>, size=<Anzahl>, replace=FALSE, prob=NULL)
```

Für `x` wird ein Vektor eingetragen, der die Elemente der Urne festlegt, aus der gezogen wird. Dies sind die Werte, aus denen sich die Zufallsfolge zusammensetzen soll. Es können Vektoren vom Datentyp `numeric` (etwa `1:50`), `character` (`c("Kopf", "Zahl")`) oder auch `logical` (`c(TRUE, FALSE)`) verwendet werden. Unter `size` ist die gewünschte Anzahl der zu ziehenden Elemente einzutragen. Mit dem Argument `replace` wird die Art des Ziehens festgelegt: auf `FALSE` gesetzt (Voreinstellung) wird ohne, andernfalls (`TRUE`) mit Zurücklegen gezogen. Natürlich kann ohne Zurücklegen aus einem Vektor der Länge n nicht häufiger als n mal gezogen werden. Wenn `replace=FALSE` und dennoch `size` größer als `length(x)` ist, liefert R deswegen eine Fehlermeldung. Für den Fall, dass nicht alle Elemente der Urne dieselbe Auftretenswahrscheinlichkeit besitzen sollen, existiert das Argument `prob`. Es benötigt einen Vektor derselben Länge wie `x`, dessen Elemente die Auftretenswahrscheinlichkeit für jedes Element von `x` bestimmen.

```
> sample(c(1:6), 20, replace=TRUE)
[1] 4 1 2 5 6 5 3 6 6 5 1 6 1 5 1 4 5 4 4 2

> sample(c("rot", "grün", "blau"), 8, replace=TRUE)
[1] "grün" "blau" "grün" "rot" "rot" "blau" "grün" "blau"
```

2.5.2 Zufallszahlen aus bestimmten Verteilungen erzeugen

Abgesehen vom zufälligen Ziehen aus einer vorgegebenen Menge endlich vieler Werte lassen sich auch Zufallszahlen mit bestimmten Eigenschaften generieren. Dazu können mit Funktionen, deren Name nach dem Muster `r<Funktionsfamilie>` aufgebaut ist, Realisierungen von Zufallsvariablen mit verschiedenen Verteilungen

zufälligen Zahlenfolgen im Ergebnis fast nicht zu unterscheiden. Pseudozufallszahlen hängen deterministisch vom sog. Zustand des die Zahlen produzierenden Generators ab. Wird sein Zustand über die Funktion `set.seed(<Zahl>)` festgelegt, kommt bei gleicher `<Zahl>` bei späteren Aufrufen von Zufallsfunktionen immer dieselbe Folge von Werten zustande. Dies gewährleistet die Reproduzierbarkeit von Auswertungsschritten bei Simulationen mit Zufallsdaten. Nach welcher Methode Zufallszahlen generiert werden, ist konfigurierbar, vgl. `?RNGkind`. Für tatsächliche Zufallszahlen vgl. das Paket `random` (Eddelbuettel, 2009).

erstellt werden (vgl. Abschn. 5.2). Diese Möglichkeit ist insbesondere für die Simulation empirischer Daten nützlich.

```
> runif(n=<Anzahl>, min=0, max=1)                      # Gleichverteilung
> rbinom(n=<Anzahl>, size, prob)                      # Binomialverteilung
> rnorm(n=<Anzahl>, mean=0, sd=1)                      # Normalverteilung
> rchisq(n=<Anzahl>, df, ncp=0)                        # chi^2-Verteilung
> rt(n=<Anzahl>, df, ncp=0)                            # t-Verteilung
> rf(n=<Anzahl>, df1, df2, ncp=0)                      # F-Verteilung
```

Unter `n` ist immer die gewünschte Anzahl an Zufallszahlen anzugeben. Bei `runif()` definiert `min` die untere und `max` die obere Grenze des Zahlenbereichs, aus dem gezogen wird. Beide Argumente akzeptieren auch Vektoren der Länge `n`, die für jede einzelne Zufallszahl den zulässigen Wertebereich angeben.

Bei `rbinom()` entsteht jede der `n` Zufallszahlen als Anzahl der Treffer in einer simulierten Serie von gleichen Bernoulli-Experimenten, die ihrerseits durch die Argumente `size` und `prob` charakterisiert sind. `size` gibt an, wie häufig ein einzelnes Bernoulli-Experiment wiederholt werden soll, `prob` legt die Trefferwahrscheinlichkeit in jedem dieser Experimente fest. Sowohl `size` als auch `prob` können Vektoren der Länge `n` sein, die dann die Bernoulli-Experimente charakterisieren, deren Simulation zu jeweils einer Zufallszahl führt.

Bei `rnorm()` sind der Erwartungswert `mean` und die theoretische Streuung `sd` der normalverteilten Variable anzugeben, die simuliert werden soll.¹⁹ Auch diese Argumente können Vektoren sein und für jede Zufallszahl andere Parameter vorgeben.

Sind Verteilungen über Freiheitsgrade und Nonzentralitätsparameter charakterisiert, werden diese mit den Argumenten `df` (Degrees of Freedom) respektive `ncp` (Non-Centrality Parameter) ggf. in Form von Vektoren übergeben.

```
> runif(5, min=1, max=6)
[1] 4.411716 3.893652 2.412720 5.676668 2.446302

> rbinom(20, size=5, prob=0.3)
[1] 2 0 3 0 2 2 1 0 1 0 2 1 1 4 2 2 1 1 3 3

> round(rnorm(12, mean=100, sd=15))
[1] 93 107 87 117 91 84 90 86 93 99 109 87
```

2.5.3 Unterauswahl einer Datenmenge bilden

Soll aus einer vorhandenen Datenmenge eine Substichprobe gezogen werden, hängt das Vorgehen von der genau intendierten Art des Ziehens ab. Eine rein zufällige Unterauswahl bestimmten Umfangs ohne weitere Nebenbedingungen kann mit `sample()` erstellt werden. Dazu werden die Indizes des Datenvektors als Urne betrachtet, aus der ohne Zurücklegen die gewünschte Anzahl von Fällen gezogen wird.

¹⁹ Der die Breite (Dispersion) einer Normalverteilung charakterisierende Parameter ist in R-Funktionen immer die Streuung σ , in der Literatur dagegen häufig die Varianz σ^2 .

```
> vec      <- rep(c("rot", "grün", "blau"), 1000)
> selIdx1 <- sample(seq(along=vec), 5, replace=FALSE)
> vec[selIdx1]
[1] "blau" "grün" "blau" "grün" "rot"
```

Ein anderes Ziel könnte darin bestehen, z. B. jedes zehnte Element einer Datenreihe auszugeben. Hier bietet sich die `seq()` Funktion an.

```
> selIdx2 <- seq(1, length(vec), by=10)
```

Soll nicht genau, sondern nur im Mittel jedes zehnte Element ausgegeben werden, eignet sich die Funktion `rbinom()` zur Erstellung eines geeigneten Indexvektors. Dazu kann der Vektor der Trefferanzahlen aus einer Serie von jeweils nur einmal durchgeföhrten Bernoulli-Experimenten mit Trefferwahrscheinlichkeit 1/10 in einen logischen Indexvektor umgewandelt werden:

```
> selIdx3 <- rbinom(length(vec), 1, 0.1) == 1
```

2.5.4 Zufällige Reihenfolgen erstellen

Zufällige Reihenfolgen können mit Kombinationen von `rep()` und `sample()` erstellt werden. Sie sind z. B. bei der randomisierten Zuteilung von VPn zu Gruppen, beim Randomisieren der Reihenfolge von Bedingungen oder beim Ziehen einer Zufallsstichprobe aus einer Datenmenge nützlich.

```
# teile 12 VPn auf 3 gleich große Gruppen auf
> nGroups      <- 3                                # Anzahl Gruppen
> groupSize    <- 4                                # Gruppengröße
> (groupsRep   <- rep(1:nGroups, groupSize))       # Gruppenzugehörigkeiten
[1] 1 2 3 1 2 3 1 2 3 1 2 3

> (distrib <- sample(groupsRep, length(groupsRep), replace=FALSE))
[1] 2 3 3 3 2 2 1 1 3 2 1 1

# randomisiere Reihenfolge von 5 Farben
> myColors     <- c("red", "green", "blue", "yellow", "black")
> (randCols    <- sample(myColors, length(myColors), replace=FALSE))
[1] "yellow" "green" "red" "blue" "black"
```

2.6 Deskriptive Kennwerte numerischer Vektoren

Die deskriptive Beschreibung von Variablen ist ein wichtiger Teil der Analyse empirischer Daten, die gemeinsam mit der graphischen Darstellung (vgl. Kap. 10) gerade zu Beginn der Auseinandersetzung mit einem Datensatz hilft, seine Struktur besser zu verstehen. Die hier umgesetzten statistischen Konzepte und Techniken seien als bekannt vorausgesetzt und finden sich in vielen Lehrbüchern der Statistik (Bortz, 2005; Hartung et al., 2005; Hays, 1994).

R stellt für die Berechnung aller gängigen Kennwerte separate Funktionen bereit, die meist erwarten, dass die Daten in Vektoren gespeichert sind. Es sei an dieser Stelle daran erinnert, dass sich logische Wahrheitswerte ebenfalls in einem numerischen Kontext verwenden lassen, wobei der Wert TRUE wie eine 1, der Wert FALSE wie eine 0 behandelt wird.

Mit der Funktion `summary(<Vektor>)` können die wichtigsten deskriptiven Kennwerte einer Datenreihe abgerufen werden, dies sind Minimum, erstes Quartil, Median, Mittelwert, drittes Quartil und Maximum. Der Output ist ein Vektor mit benannten Elementen.

```
> summary(c(17, 30, 30, 25, 23, 21))
Min. 1st Qu. Median Mean 3rd Qu. Max.
17.00   21.50  24.00 24.33   28.75 30.00
```

2.6.1 Summen, Differenzen und Produkte

Mit der `sum(<Vektor>)` Funktion wird die Summe aller Elemente eines Vektors berechnet. Die kumulierte Summe erhält man mit `cumsum(<Vektor>)`.

```
> sum(c(17, 30, 30, 25, 23, 21))
[1] 146

> sum(c(17, 30, 30, 25, 23, 21) >= 25)           # wie viele Elemente >= 25?
[1] 3

> cumsum(c(17, 30, 30, 25, 23, 21))
[1] 17 47 77 102 125 146
```

Um die Differenzen aufeinanderfolgender Elemente eines Vektors (also eines Wertes zu einem vorhergehenden Wert) zu berechnen, dient die Funktion `diff(x=<Vektor>, lag=1)`. Über das Argument `lag` wird kontrolliert, über welchen Abstand die Differenz gebildet wird. Die Voreinstellung 1 bewirkt, dass die Differenz eines Wertes zum unmittelbar vorhergehenden berechnet wird. Die Ausgabe umfasst `lag` Werte weniger, als `x` Elemente besitzt.

```
> diff(c(17, 30, 30, 25, 23, 21))
[1] 13 0 -5 -2 -2

> diff(c(17, 30, 30, 25, 23, 21), lag=2)
[1] 13 -5 -7 -4
```

Das Produkt aller Elemente eines Vektors wird mit `prod(<Vektor>)` berechnet, das kumulierte Produkt mit `cumprod(<Vektor>)`. `factorial(<Zahl>)` ermittelt die Fakultät $n!$ einer Zahl n , ist für natürliche Zahlen also identisch zu `prod(1:<Zahl>)`.²⁰

²⁰ Genauso gilt für natürliche Zahlen $n! = \Gamma(n+1)$, in R als `gamma(<Zahl> + 1)` berechenbar. Bei Dezimalzahlen begünstigt die wiederholte Multiplikation die Kumulation von Rundungsfehlern,

```
> prod(c(17, 30, 30, 25, 23, 21))
[1] 184747500

> cumprod(c(17, 30, 30, 25, 23, 21))
[1] 17 510 15300 382500 8797500 184747500

> factorial(5)
[1] 120
```

2.6.2 Extremwerte

Mit `min(<Vektor>)` und `max(<Vektor>)` können die Extremwerte eines Vektors erfragt werden. `range(<Vektor>)` gibt den größten und kleinsten Wert zusammengefasst als Vektor aus. Den Index des größten bzw. kleinsten Wertes liefern die Funktionen `which.max(<Vektor>)` bzw. `which.min(<Vektor>)`. Die letzteren Funktionen lässt sich etwa nutzen, um herauszufinden, welches Element eines Vektors am nächsten an einem vorgegebenen Wert liegt.

```
> max(c(17, 30, 30, 25, 23, 21))
[1] 30

> which.min(c(17, 30, 30, 25, 23, 21))
[1] 1

> vec <- c(-5, -8, -2, 10, 9)
> val <- 0
> which.min(abs(vec-val))           # welches Element liegt am nächsten an 0?
[1] 3

> range(c(17, 30, 30, 25, 23, 21))
[1] 17 30
```

Um die Spannweite (Range) von Werten eines Vektors, also die Differenz von kleinstem und größtem Wert zu ermitteln, ist die `diff()` Funktion nützlich.

```
> diff(range(c(17, 30, 30, 25, 23, 21)))
[1] 13
```

Die Funktionen `pmin(<Vektor1>, <Vektor2>, ...)` und `pmax(<Vektor1>, <Vektor2>, ...)` (Parallel Min/Max) vergleichen zwei oder mehr Vektoren elementweise hinsichtlich der Größe der in ihnen gespeicherten Werte und liefern einen Vektor aus den pro Position größten bzw. kleinsten Werten zurück.

die durch die interne Darstellung solcher Zahlen unvermeidlich sind (vgl. Abschn. 1.3.6, Fußnote 26). Numerisch stabiler als `prod(<Vektor>)` ist deswegen u. U. die Rücktransformation der Summe der logarithmierten Werte mit `exp(sum(log(<Vektor>)))`.

```
> vec1 <- c(5, 2, 0, 7)
> vec2 <- c(3, 3, 9, 2)
> pmax(vec1, vec2)
[1] 5 3 9 7

> pmin(vec1, vec2)
[1] 3 2 0 2
```

2.6.3 Mittelwert, Median und Modalwert

Mit der Funktion `mean(x=(Vektor))` wird das arithmetische Mittel $M_x = (1/n) \cdot \sum x_i$ eines Vektors `x` der Länge n berechnet, manuell also auch mit `sum((Vektor))/length((Vektor))`.²¹ Hier ist zu beachten, dass `x` tatsächlich ein etwa mit `c(...)` gebildeter Vektor ist: der Aufruf `mean(1, 7, 3)` gibt nämlich anders als `mean(c(1, 7, 3))` nicht den Mittelwert der Daten 1, 7, 3 aus. Stattdessen ist die Ausgabe gleich dem ersten übergebenen Argument.

Der Mittelwert kann gestutzt, d. h. ohne einen bestimmten Anteil an Extremwerten berechnet werden. Dies kann dann sinnvoll sein, wenn die Daten Ausreißer aufweisen, die den Mittelwert stark verzerrn, so dass er die zentrale Tendenz der Daten nicht mehr gut repräsentiert. Mit dem Argument `trim=(Zahl)` kann der gewünschte Anteil an Extremwerten aus der Berechnung ausgeschlossen werden. `(Zahl)` gibt dabei den Anteil der Werte an, der auf jeder der beiden Seiten der empirischen Verteilung verworfen werden soll. Wenn insgesamt die extremen 5% der Daten ausgeschlossen werden sollen, ist mithin `trim=0.025` zu setzen.

```
> mean(c(17, 30, 30, 25, 23, 21))
[1] 24.33333
```

Um ein gewichtetes Mittel zu berechnen, bei dem die Gewichte nicht wie bei `mean()` für alle Werte identisch sind, dient die Funktion `weighted.mean(x=(Vektor), w=(Gewichte))`. Ihr zweites Argument `w` muss ein Vektor derselben Länge wie `x` sein und die Gewichte benennen. Der Einfluss jedes Werts auf den Mittelwert entspricht dann dem Verhältnis seines Gewichts zur Summe aller Gewichte.

```
> weights <- c(0.6, 0.6, 0.3, 0.2, 0.4, 0.6)
> weighted.mean(c(17, 30, 30, 25, 23, 21), weights)
[1] 23.70370
```

Die Funktion `median(x=(Vektor))` gibt den Median, also das 50%-Quantil einer empirischen Verteilung aus. Dies ist der Wert, für den die empirische kumulative Häufigkeitsverteilung von `x` den Wert 0.5 besitzt (vgl. Abschn. 2.11.6), der also

²¹ Oder auch mit `sum(prop.table(table((Vektor)))*sort(unique((Vektor))))` als Umsetzung der Formel $\sum x_i \cdot h(x_i)$, wenn $h(x_i)$ die relative Häufigkeit eines Wertes x_i ist (vgl. Abschn. 2.11). Für das geometrische und harmonische Mittel vgl. die Funktionen `geometric.mean()` bzw. `harmonic.mean()` aus dem `psych` Paket (Revelle, 2009).

größer oder gleich 50% (und kleiner oder gleich 50%) der Werte ist. Im Fall einer geraden Anzahl von Elementen in x wird zwischen den mittleren beiden Werten von `sort(<Vektor>)` gemittelt, andernfalls das mittlere Element von `sort(<Vektor>)` ausgegeben.

```
> age <- c(17, 30, 30, 25, 23, 21)
```

```
> sort(age)
```

```
[1] 17 21 23 25 30 30
```

```
> median(age)
```

```
[1] 24
```

```
> ageNew <- c(age, 22)
```

```
> sort(ageNew)
```

```
[1] 17 21 22 23 25 30 30
```

```
> median(ageNew)
```

```
[1] 23
```

Der am häufigsten vorkommende Wert eines Vektors wird als Modalwert bezeichnet, für dessen Berechnung R keine separate Funktion bereitstellt. Eine Möglichkeit, um den Modalwert auf andere Weise zu erhalten, bietet die `table()` Funktion zur Erstellung von Häufigkeitstabellen (vgl. Abschn. 2.11). Der Modalwert ist jener Wert, bei dem das Maximum der Häufigkeiten auftritt. Die folgende Methode gibt zunächst den Index der Häufigkeitstabelle aus, unter dem der häufigste Wert verzeichnet ist. Den Modalwert erhält man zusammen mit seiner Auftretenshäufigkeit durch Indizieren der Tabelle mit diesem Index.

```
> vec <- c(11, 22, 22, 33, 33, 33, 33)      # häufigster Wert: 33
> (tab <- table(vec))                      # Häufigkeitstabelle
```

```
vec
```

```
11 22 33
```

```
1 2 4
```

```
> (modIdx <- which.max(tab))          # Modalwert und sein Index
```

```
33
```

```
3
```

```
> tab[modIdx]                         # Modalwert mit Häufigkeit
```

```
33
```

```
4
```

2.6.4 Quartile, Quantile, Interquartilabstand

Mit der Funktion `quantile(x=<Vektor>, probs=seq(0, 1, 0.25))` werden in der Voreinstellung die Quartile eines Vektors bestimmt. Dies sind jene Werte, die größer oder gleich einem ganzzahligen Vielfachen von 25% der Datenwerte und kleiner oder gleich den Werten des verbleibenden Anteils der Daten sind. Das erste Quartil ist $\geq 25\%$ (und $\leq 75\%$) der Daten, das zweite Quartil (der Median) $\geq 50\%$

(und $\leq 50\%$) und das dritte Quartil $\geq 75\%$ (und $\leq 25\%$) der Werte. Der Output von `quantile()` ist ein Vektor mit benannten Elementen.

```
> (quantOld <- quantile(c(17, 30, 30, 25, 23, 21)))
  0%   25%   50%   75%  100%
17.00 21.50 24.00 28.75 30.00

> quantOld[c("25%", "50%")]
  25%   50%
21.5 24.0
```

Über das `probs=(Vektor)` Argument können statt der Quartile auch andere Anteile eingegeben werden, deren Wertegrenzen gewünscht sind. Bei Berechnung der Werte, die einen bestimmten Anteil der Daten abschneiden, wird ggf. zwischen den in `x` tatsächlich vorkommenden Werten interpoliert.²²

```
> vec <- sample(seq(0, 1, by=0.01), 1000, replace=TRUE)
> quantile(vec, probs=c(0, 0.2, 0.4, 0.6, 0.8, 1))
  0%   20%   40%   60%   80%  100%
0.000 0.190 0.400 0.604 0.832 1.000
```

Mit der Funktion `IQR(x=(Vektor))` wird der Interquartilabstand erfragt, also die Differenz von drittem und erstem Quartil.

```
> IQR(c(17, 30, 30, 25, 23, 21))
[1] 7.25
```

2.6.5 Varianz, Streuung, Schiefe und Wölbung

Mit der Funktion `var(x=(Vektor))` wird die korrigierte Varianz $s_x^2 = (1/(n-1)) \cdot \sum_i (x_i - M_x)^2$ zur erwartungstreuen Schätzung der Populationsvarianz auf Basis einer empirischen Datenreihe x der Länge n ermittelt. Die Umrechnungsformel zur Berechnung der unkorrigierten Varianz $S_x^2 = (1/n) \cdot \sum_i (x_i - M_x)^2$ aus der korrigierten lautet $S_x^2 = ((n-1)/n) \cdot s_x^2$, in R also `((length(Vektor))-1)/length(Vektor))*var(Vektor)`.²³

```
> age <- c(17, 30, 30, 25, 23, 21)
> var(age)                                     # korrigierte Varianz
[1] 26.26667

> sum((age-mean(age))^2) / ((length(age)-1))    # manuelle Berechnung
[1] 26.26667

> ((length(age)-1) / length(age)) * var(age)      # unkorrigierte Varianz
[1] 21.88889
```

²² Zur Berechnung von Quantilen stehen verschiedene Verfahren zur Verfügung, vgl. `?quantile`.

²³ Als Alternative ließe sich die `cov.wt()` Funktion verwenden, vgl. Abschn. 2.8.9.

```
> sum((age-mean(age))^2) / (length(age))           # manuelle Berechnung
[1] 21.88889
```

Die korrigierte Streuung s_x kann durch Ziehen der Wurzel aus der korrigierten Varianz oder mit der `sd(x=<Vektor>)` Funktion berechnet werden. Auch hier basiert das Ergebnis auf der bei der Varianz erläuterten Korrektur zur Schätzung der Populationsstreuung auf Basis einer empirischen Stichprobe. Die Umrechnungsformel zur Berechnung der unkorrigierten Streuung $S_x = \sqrt{(1/n) \cdot \sum_i (x_i - M_x)^2}$ aus der korrigierten lautet $S_x = \sqrt{(n-1)/n} \cdot s_x$, in R also `sqrt((length(<Vektor>)-1)/length(<Vektor>))*sd(<Vektor>)`.

```
> sqrt(var(age))                                # Wurzel aus Varianz
[1] 5.125102

> sd(age)                                       # korrigierte Streuung
[1] 5.125102

> sqrt((length(age)-1) / length(age)) * sd(age)  # unkorrigierte Streuung
[1] 4.678556

> sqrt(sum((age-mean(age))^2) / (length(age)))    # manuelle Berechnung
[1] 4.678556
```

Die mittlere absolute Abweichung vom Median ist manuell zu berechnen, während für den Median der absoluten Abweichungen vom Median die Funktion `mad(x=<Vektor>, constant=1.4826)` bereit steht. Deren Ergebnis beinhaltet die Multiplikation mit dem Faktor 1.4826, der über das Argument `constant` auf einen anderen Wert gesetzt werden kann.

```
> mean(abs(age-median(age)))      # mittlere absolute Abweichung vom Median
[1] 4

> mad(age)                         # Median der abs. Abweichungen vom Median
[1] 6.6717
```

Schiefe und Wölbung empirischer Verteilungen lassen sich über die Funktionen `skewness()` und `kurtosis()` aus dem `moments` Paket ermitteln (Komsta und Novomestky, 2007).

2.6.6 Kovarianz, Korrelation und Partialkorrelation

Mit der Funktion `cov(x=<Vektor1>, y=<Vektor2>)` wird die korrigierte Kovarianz $(1/(n-1)) \cdot \sum_i ((x_i - M_x) \cdot (y_i - M_y))$ zweier Datenreihen x und y derselben Länge n berechnet. Die unkorrigierte Kovarianz muss nach der bereits für die Varianz genannten Umrechnungsformel ermittelt werden.²⁴

²⁴ Als Alternative ließe sich die `cov.wt()` Funktion verwenden, vgl. Abschn. 2.8.9.

```

> age      <- c(17, 30, 30, 25, 23, 21)
> semester <- c(1, 12, 8, 10, 5, 3)
> cov(age, semester)                      # korrigierte Kovarianz
[1] 19.2

# korrigierte Kovarianz manuell berechnen
> sum((age-mean(age)) * (semester-mean(semester))) / (length(age)-1)
[1] 19.2
# unkorrigierte Kovarianz aus korrigierter berechnen
> ((length(age)-1) / length(age)) * cov(age, semester)
[1] 16

# unkorrigierte Kovarianz manuell berechnen
> sum((age-mean(age)) * (semester-mean(semester))) / length(age)
[1] 16

```

Neben der voreingestellten Berechnungsmethode für die Kovarianz nach Pearson kann auch die Rang-Kovarianz nach Spearman oder Kendall berechnet werden (vgl. Abschn. 6.3.1).

Analog zur Kovarianz kann mit `cor(x=⟨Vektor1⟩, y=⟨Vektor2⟩)` die herkömmliche Produkt-Moment-Korrelation oder die Rangkorrelation berechnet werden.²⁵ Für die Korrelation gibt es keinen Unterschied beim Gebrauch von korrigierten und unkorrigierten Streuungen, so dass sich nur ein Kennwert ergibt.

```

> cor(age, semester)
[1] 0.8854667

> cov(age, semester) / (sd(age)*sd(semester))      # manuelle Berechnung
[1] 0.8854667

```

Für die Berechnung der Partialkorrelation zweier Variablen x und y ohne eine dritte Variable z kann die Formel $(r_{xy} - (r_{xz} \cdot r_{yz})) / \sqrt{(1 - r_{xz}^2) \cdot (1 - r_{yz}^2)}$ umgesetzt werden, da die Basisinstallation von R hierfür keine eigene Funktion bereitstellt.²⁶ Für eine alternative Berechnungsmethode, die sich die Eigenschaft der Partialkorrelation als Korrelation der Residuen der Regressionen von x auf z und y auf z zunutze macht, vgl. Abschn. 7.2.4.

```

> nObs <- 100
> z     <- runif(nObs)
> x     <- z + rnorm(nObs, 0, 0.5)
> y     <- z + rnorm(nObs, 0, 0.5)
> (cor(x,y)-(cor(x,z)*cor(y,z))) / sqrt((1-cor(x,z)^2) * (1-cor(y,z)^2))
[1] 0.0753442

```

²⁵ Die kanonische Korrelation zweier Gruppen von an denselben Beobachtungsobjekten erhobenen Variablen ermittelt `cancor()`. Für die multiple Korrelation im Sinne der Wurzel aus dem Determinationskoeffizienten R^2 in der multiplen linearen Regression vgl. Abschn. 7.2.1.

²⁶ Wohl aber das Paket `ggm` mit der `pcor()` Funktion (Marchetti und Drton, 2010).

2.6.7 Funktionen auf geordnete Paare von Werten anwenden

Eine Verallgemeinerung der Anwendung einer Funktion auf jeden Wert eines Vektors stellt die Anwendung einer Funktion für zwei Argumente auf alle geordneten Paare aus den Werten zweier Vektoren dar.

```
> outer(x=<Vektor1>, Y=<Vektor2>, FUN="*", ...)
```

Für die Argumente X und Y ist dabei jeweils ein Vektor einzutragen, unter FUN eine Funktion, die zwei Argumente in Form von Vektoren verarbeiten kann.²⁷ Da Operatoren nur Funktionen mit besonderer Schreibweise sind, können sie hier ebenfalls eingesetzt werden, müssen dabei aber in Anführungszeichen stehen (vgl. Abschn. 1.2.5, Fußnote 15). Voreinstellung ist die Multiplikation, für diesen Fall existiert auch die Kurzform in Operatorschreibweise X %o% Y. Sollen an FUN weitere Argumente übergeben werden, kann dies an Stelle der ... geschehen, wobei mehrere Argumente durch Komma zu trennen sind. Die Ausgabe erfolgt in Form einer zweidimensionalen Matrix (vgl. Abschn. 2.8).

Als Beispiel soll das kleine 1×1 ausgegeben werden, also alle Produkte der Zahlen 1–10.

```
> outer(1:10, 1:10, "*")
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1    2    3    4    5    6    7    8    9   10
[2,]    2    4    6    8   10   12   14   16   18   20
[3,]    3    6    9   12   15   18   21   24   27   30
[4,]    4    8   12   16   20   24   28   32   36   40
[5,]    5   10   15   20   25   30   35   40   45   50
[6,]    6   12   18   24   30   36   42   48   54   60
[7,]    7   14   21   28   35   42   49   56   63   70
[8,]    8   16   24   32   40   48   56   64   72   80
[9,]    9   18   27   36   45   54   63   72   81   90
[10,]   10   20   30   40   50   60   70   80   90  100
```

2.7 Gruppierungsfaktoren

Um die Eigenschaften kategorialer Variablen abzubilden, existiert die Klasse `factor`. Sie eignet sich insbesondere für Gruppierungsfaktoren im versuchsplanerischen Sinn. Ein Objekt dieser Klasse nimmt die jeweilige Gruppenzugehörigkeit von Beobachtungsobjekten auf und enthält Informationen darüber, welche Stufen die Variable prinzipiell umfasst. Für den Gebrauch in inferenzstatistischen Analysefunktionen ist es wichtig, dass Gruppierungsvariablen auch tatsächlich die Klasse `factor` besitzen. Insbesondere bei numerisch codierter Gruppenzugehörigkeit besteht sonst die Gefahr der Verwechslung mit echten quantitativen Variablen, was

²⁷ Vor dem Aufruf von FUN verlängert `outer()` jeweils X und Y mit Hilfe von `rep()` so, dass beide die Länge `length(X)*length(Y)` besitzen und sich aus der Kombination der Elemente mit gleichem Index alle geordneten Paare ergeben.

etwa bei linearen Modellen (z. B. lineare Regression oder Varianzanalyse) unentdeckt bleiben und für falsche Ergebnisse sorgen könnte.

2.7.1 Ungeordnete Faktoren

Als Beispiel für eine Gruppenzugehörigkeit soll das qualitative Merkmal Geschlecht dienen, dessen Ausprägungen in einer Stichprobe zunächst als character Werte eingegeben und in einem Vektor gespeichert werden.

```
> sex <- c("m", "f", "f", "m", "m", "m", "f", "m", "f", "f")
```

Um den aus Zeichen bestehenden Vektor zu einem Gruppierungsfaktor mit zwei Ausprägungen zu machen, dient der Befehl `factor()`.

```
> factor(x=<Vektor>, levels=<Stufen>, labels=levels, exclude=NA)
```

Unter `x` ist der umzuwandelnde Vektor einzutragen. Welche Stufen der Faktor prinzipiell annehmen kann, bestimmt das Argument `levels`. In der Voreinstellung werden die Faktorstufen automatisch anhand der in `x` tatsächlich vorkommenden Werte mit `sort(unique(x))` bestimmt. Fehlende Werte werden nicht als eigene Faktorstufe gewertet, es sei denn das Argument `exclude=NULL` wird gesetzt – an `exclude` übergebene Werte werden nämlich nicht als Stufe berücksichtigt, wenn der Faktor erstellt wird.

```
> (sexFac <- factor(sex))
[1] m f f m m m f m f f
Levels: f m
```

Da die in `x` gespeicherten empirischen Ausprägungen nicht notwendigerweise auch alle theoretisch möglichen Kategorien umfassen müssen, kann an `levels` auch ein Vektor mit allen möglichen Stufen übergeben werden.

```
# 2 und 5 kommen nicht vor, sollen aber mögliche Ausprägungen sein
> factor(c(1, 1, 3, 3, 4, 4), levels=1:5)
[1] 1 1 3 3 4 4
Levels: 1 2 3 4 5
```

Die Stufenbezeichnungen stimmen in der Voreinstellung mit den `levels` überein, sie können aber auch durch einen für das Argument `labels` bestimmten Vektor umbenannt werden. Dies könnte etwa sinnvoll sein, wenn die Faktorstufen in einem Vektor numerisch codiert sind, wenn der Faktor erstellt werden soll aber inhaltlich aussagekräftigere Namen erhalten sollen.

```
> (sexNum <- rbinom(10, 1, 0.5))
[1] 1 0 1 1 1 0 1 0 1 0
> factor(sexNum, labels=c("man", "woman"))
[1] woman man woman woman woman man woman man
Levels: man woman
```

Wenn die Stufenbezeichnungen eines Faktors im nachhinein geändert werden sollen, so kann dem von der Funktion `levels(<Faktor>)` ausgegebenen Vektor ein Vektor mit passend vielen neuen Namen zugewiesen werden.

```
> levels(sexFac) <- c("fem", "male"); sexFac      # vorherige Stufen: f, m
[1] male fem fem male male male female male fem fem
Levels: fem male
```

Die Anzahl der Stufen eines Faktors wird mit der Funktion `nlevels(<Faktor>)` ausgegeben; wie häufig jede Stufe vorkommt, erfährt man durch `summary <-(<Faktor>)`.

```
> nlevels(sexFac)
[1] 2

> summary(sexFac)
f   m
5   5
```

Die im Faktor gespeicherten Werte werden intern auf zwei Arten repräsentiert – zum einen mit den Namen der Faktorstufen, zum anderen mit einer internen Codierung der Stufen über fortlaufende natürliche Zahlen, die der (ggf. alphabetischen) Reihenfolge der Ausprägungen entspricht. Dies wird in der Ausgabe der internen Struktur eines Faktors mit `str(<Faktor>)` deutlich. Die Namen der Faktorstufen werden mit `levels(<Faktor>)` ausgegeben, die interne numerische Repräsentation mit `unclass(<Faktor>)`.²⁸

```
> levels(sexFac)
[1] "f" "m"

> str(sexFac)
Factor w/ 2 levels "f", "m": 2 1 1 2 2 2 1 2 1 1

> unclass(sexFac)
[1] 2 1 1 2 2 2 1 2 1 1

attr(levels)
[1] "f" "m"
```

Bei der Umwandlung eines Faktors mit der `as.character(<Faktor>)` Funktion erhält der Ergebnisvektor des Datentyps `character` als Elemente die Namen der entsprechenden Faktorstufen. Sind die Namen aus Zahlen gebildet und sollen letztlich in numerische Werte umgewandelt werden, so ist dies durch

²⁸ Trotz dieser Codierung können Faktoren keinen mathematischen Transformationen unterzogen werden. Wenn die Namen der Faktorstufen aus Zahlen gebildet werden, kann es zu Diskrepanzen zwischen Levels und interner Codierung kommen: `unclass(factor(10:15))` ergibt `1 2 3 4 5 6`. Dies ist bei der üblichen Verwendung von Faktoren aber irrelevant.

einen zusätzlichen Schritt mit `as.numeric(as.character(<Faktor>))` oder mit `as.numeric(levels(<Faktor>)[<Faktor>])` möglich.²⁹

```
> as.character(sexFac)
[1] "m" "f" "f" "m" "m" "m" "f" "m" "f" "f"
```

2.7.2 Faktorstufen hinzufügen, entfernen und zusammenfassen

Einem bestehenden Faktor können nicht beliebige Werte als Element hinzugefügt werden, sondern lediglich solche, die einer bereits existierenden Faktorstufe entsprechen. Bei Versuchen, einem Faktorelement einen anderen Wert zuzuweisen, wird das Element auf NA gesetzt und eine Warnung ausgegeben. Die Menge möglicher Faktorstufen kann jedoch über `levels()` erweitert werden, ohne dass es bereits zugehörige Beobachtungen gäbe.

```
> (status <- factor(c("hi", "lo", "hi")))
[1] hi lo hi
Levels: hi lo

> status[4] <- "mid"; status
Warning message:
In `<-factor`(`*tmp*`, 4, value = "mid") :
  invalid factor level, NAs generated

[1] hi lo hi <NA>
Levels: hi lo

> levels(status) <- c(levels(status), "mid")    # Stufe "mid" hinzufügen
> status[4] <- "mid"; status                      # Beobachtung "mid" hinzufügen
[1] hi lo hi mid
Levels: hi lo mid
```

Stufen eines bestehenden Faktors lassen sich nicht ohne weiteres löschen. Die erste Möglichkeit, um einen gegebenen Faktor in einen Faktor mit weniger möglichen Stufen umzuwandeln, besteht im Zusammenfassen mehrerer ursprünglicher Stufen zu einer gemeinsamen neuen Stufe. Hierzu muss dem von `levels()` ausgegebenen Objekt eine Liste zugewiesen werden, die nach dem Muster `list(<neueStufe>=c("<alteStufe1>", "<alteStufe2>", ...))` aufgebaut ist (vgl. Abschn. 3.1).

```
# kombiniere Stufen "mid" und "lo" zu "notHi", "hi" bleibt unverändert
> hiNotHi <- status
> levels(hiNotHi) <- list(hi="hi", notHi=c("mid", "lo")); hiNotHi
[1] hi notHi hi notHi
Levels: hi notHi
```

²⁹ Sind die Namen der Faktorstufen dagegen nicht aus Zahlen, sondern aus anderen Zeichen gebildet, ist das Ergebnis NA, vgl. Abschn. 1.3.5.

Sollen dagegen Beobachtungen samt ihrer Stufen gelöscht werden, muss eine Teilmenge der Elemente des Faktors ausgegeben werden, die nicht alle Faktorstufen enthält. Zunächst umfasst diese Teilmenge jedoch noch wie vor alle ursprünglichen Stufen, wie in der Ausgabe unter `Levels` deutlich wird.

```
> status[1:2]
[1] hi lo
Levels: hi lo mid
```

Sollen nur die in der gewählten Teilmenge tatsächlich auftretenden Ausprägungen auch mögliche `Levels` sein, kann dies mit dem Argument `drop=TRUE` für den `[<Index>]` Operator erreicht werden. Eine andere Möglichkeit besteht in der Neubildung eines Faktors durch Einschließen der ausgegebenen Teilmenge in den `factor()` Befehl.

```
> status[1:2, drop=TRUE]
[1] hi lo
Levels: hi lo

> (newStatus <- factor(status[1:2]))
[1] hi lo
Levels: hi lo
```

2.7.3 Geordnete Faktoren

Besteht eine Reihenfolge in den Stufen eines Gruppierungsfaktors im Sinne einer ordinalen Variable, so lässt sich dieser Umstand mit der Funktion `ordered(x=<Vektor>, levels)` abbilden, die einen geordneten Gruppierungsfaktor erstellt. Dabei muss die inhaltliche Reihenfolge im Argument `levels` explizit angegeben werden, weil R sonst die Reihenfolge selbst bestimmt und ggf. die alphabetische heranzieht.

```
> ordered(status, levels=c("lo", "mid", "hi"))
[1] hi lo mid mid hi
Levels: lo < mid < hi
```

Manche Funktionen zur inferenzstatistischen Analyse nehmen bei geordneten Faktoren Gleichabständigkeit in dem Sinne an, dass die inhaltliche Unterschiedlichkeit zwischen zwei benachbarten Stufen immer dieselbe ist. Trifft dies nicht zu, sollte im Zweifel auf ungeordnete Faktoren zurückgegriffen werden.

2.7.4 Reihenfolge von Faktorstufen

Beim Sortieren von Faktoren wie auch in manchen statistischen Analysefunktionen ist die Reihenfolge der Faktorstufen bedeutsam. Werden die Faktorstufen beim Erstellen eines Faktors explizit mit `labels` angegeben, bestimmt die Reihenfolge der

Elemente in `labels` die Reihenfolge der Stufen. Ohne Verwendung von `labels` ergibt sich die Reihenfolge aus den sortierten Bezeichnungen des Vektors, der die Gruppenzugehörigkeiten enthält.³⁰

Um die Reihenfolge der Stufen nachträglich zu ändern, kann ein Faktor in einen geordneten Faktor unter Verwendung des `levels` Arguments konvertiert werden (s.o.). Als weitere Möglichkeit wird mit `relevel(x=<Faktor>, ref=<Referenzstufe>)` die für `ref` übergebene Faktorstufe zur ersten Stufe des Faktors `x`. Die `reorder()` Funktion ändert die Reihenfolge der Faktorstufen ebenfalls nachträglich und ordnet sie so, dass ihre Reihenfolge den empirischen Kennwerten einer Variable entspricht, wie sie jeweils in den durch den Faktor definierten Gruppen berechnet wurden.

```
> reorder(x=<Faktor>, X=<Vektor>, FUN=<Funktion>)
```

Als erstes Argument `x` wird der Faktor mit den zu ordnenden Stufen erwartet. Für das Argument `X` ist ein numerischer Vektor derselben Länge wie `x` zu übergeben, der auf Basis von `x` in Gruppen eingeteilt wird. Pro Gruppe wird die mit `FUN` bezeichnete Funktion angewendet, die einen Vektor zu einem skalaren Kennwert verarbeiten muss. Als Ergebnis werden die Stufen von `x` entsprechend den sich aus Anwendung von `FUN` pro Gruppe ergebenden Kennwerten geordnet.

```
> fac <- factor(rep(LETTERS[1:3], each=10))
> vec <- rnorm(30, rep(c(10, 5, 15), each=10), 3)
> tapply(vec, fac, mean)                                # Mittelwerte pro Gruppe
   A          B          C
10.18135  6.47932 13.50108

> reorder(fac, vec, mean)
[1] A A A A A A A A A B B B B B B B B C C
[23] C C C C C C C C
Levels: B < A < C
```

Beim Sortieren von Faktoren wird die Reihenfolge der Elemente durch die Reihenfolge der Faktorstufen bestimmt, die nicht mit der numerischen oder alphabetischen Reihenfolge der Stufenbezeichnungen übereinstimmen muss. Damit kann die Reihenfolge in Faktoren von der Reihenfolge in Vektoren abweichen, auch wenn sie oberflächlich dieselben Elemente enthalten.

```
> (fac <- factor(sample(1:2, 10, replace=TRUE), labels=c("B", "A")))
[1] A A A B B A B B B B
Levels: B A

> sort(fac)
[1] B B B B B A A A A
```

³⁰ Sind die Bezeichnungen Zeichenketten mit numerischer Bedeutung, so ist zu beachten, dass die Reihenfolge dennoch alphabetisch bestimmt wird – die Stufe "10" käme demnach vor der Stufe "4".

```
> as.vector(fac)
[1] "A" "A" "A" "B" "B" "A" "B" "B" "B" "B"

> sort(as.vector(fac))
[1] "A" "A" "A" "A" "B" "B" "B" "B" "B" "B"
```

2.7.5 Faktoren nach Muster erstellen

Da Faktoren im Zuge der Einteilung von Beobachtungsobjekten in Gruppen meist nach festem Muster erstellt werden müssen, lassen sich als Alternative zur manuellen Anwendung von `rep()` und `factor()` mit der `gl()` Funktion Faktoren anhand weniger Argumente automatisiert erstellen.

```
> gl(n=<Stufen>, k=<Zellbesetzung>, labels=1:n, ordered=FALSE)
```

Das Argument `n` gibt die Anzahl der Stufen an, die eine UV besitzen soll. Mit `k` wird festgelegt, wie häufig jede Faktorstufe realisiert werden soll, wie viele Beobachtungen also jede Bedingung umfasst. Für `labels` kann ein Vektor mit so vielen Gruppenbezeichnungen angegeben werden, wie Stufen vorhanden sind. In der Voreinstellung werden die Gruppen numeriert. Um einen geordneten Faktor zu erstellen, ist `ordered=TRUE` als Argument anzugeben.

```
> (myFac1 <- factor(rep(c("A", "B"), c(5, 5)))) # manuell
[1] A A A A A B B B B B
Levels: A B

> (myFac2 <- gl(2, 5, labels=c("less", "more"), ordered=TRUE))
[1] less less less less less more more more more more
Levels: less < more
```

Sollen die Elemente des Faktors in eine zufällige Reihenfolge gebracht werden, um die Zuordnung von VPn zu Gruppen zu randomisieren, kann dies wie in Abschn. 2.5.4 beschrieben geschehen.

```
> sample(myFac2, length(myFac2), replace=FALSE)
[1] more more less more less less less more more less
Levels: less < more
```

Bei mehreren UVn mit vollständig gekreuzten Faktorstufen kann die Funktion `expand.grid()` verwendet werden, um alle Stufenkombinationen zu erstellen (vgl. Abschn. 2.3.3). Dabei ist die angestrebte Gruppenbesetzung pro Zelle nur bei einem der hier im Aufruf durch `gl()` erstellten Faktoren anzugeben, beim anderen ist sie auf 1 zu setzen. Das Ergebnis ist ein Datensatz (vgl. Abschn. 3.2).

```
> expand.grid(IV1=gl(2, 2, labels=c("a", "b")), IV2=gl(3, 1))
  IV1 IV2
1     a   1
2     a   1
3     b   1
4     b   1
```

```

5   a   2
6   a   2
7   b   2
8   b   2
9   a   3
10  a   3
11  b   3
12  b   3

```

2.7.6 Quantitative Variablen in Faktoren umwandeln

Aus den in einem Vektor gespeicherten Werten einer quantitativen Variable lässt sich mit `cut(x=<Vektor>, breaks)` ein nicht geordneter Gruppierungsfaktor erstellen.³¹ Dazu muss zunächst der Wertebereich des Vektors in disjunkte Intervalle eingeteilt werden. Die einzelnen Werte werden dann entsprechend ihrer Zugehörigkeit zu diesen Intervallen in Faktorstufen umgewandelt. Die Intervalle werden über das Argument `breaks` festgelegt, wobei entweder ihre Anzahl oder die Intervallgrenzen in Form eines Vektors anzugeben sind. Intervalle werden in der Form $(a, b]$ gebildet, das Intervall ist also nach unten offen und nach oben geschlossen. Anders gesagt ist die untere Grenze nicht Teil des Intervalls, die obere schon. Die Untergrenze des insgesamt möglichen Wertebereichs müssen als Grenzen berücksichtigt werden, ggf. sind dies `-Inf` und `Inf` für negativ und positiv unendliche Werte.

Wenn die Faktorstufen andere Namen als die zugehörigen Intervallgrenzen tragen sollen, müssen sie über das Argument `labels` explizit angegeben oder mit `levels()` nachträglich geändert werden. Hier ist auf die Entsprechung der Reihenfolge der Werte im ursprünglichen Vektor und der Reihenfolge der Benennungen bei Verwendung von `levels()` zu achten.

```

> IQ      <- rnorm(100, mean=100, sd=15)
> IQfac  <- cut(IQ, breaks=c(0, 85, 115, Inf)); IQfac[1:5]
[1] (115,Inf] (0,85] (85,115] (85,115] (85,115] (0,85]
Levels: (0,85] (85,115] (115,Inf]

> levels(IQfac) <- c("lo", "mid", "hi"); IQfac[1:5]
[1] hi lo mid mid mid lo
Levels: lo mid hi

```

2.7.7 Funktionen getrennt nach Gruppen anwenden

Oft sind die Werte einer in verschiedenen Bedingungen erhobenen Variable in einem Vektor `x` gespeichert, wobei sich die zu jedem Wert gehörende Beobachtungsbedingung aus einem Faktor oder der Kombination mehrerer Faktoren ergibt. Jeder Faktor

³¹ Eine erweiterte Version dieser Funktion ist `cut2()` aus dem `Hmisc` Paket.

besitzt dabei dieselbe Länge wie x und codiert die Zugehörigkeit der Beobachtungen in x zu den Stufen einer Gruppierungsvariable. Dabei müssen nicht für jede Bedingung auch gleich viele Beobachtungen vorliegen.

Sollen Kennwerte von x jeweils getrennt für jede Bedingung bzw. Kombination von Bedingungen berechnet werden, können die Funktionen `ave()` und `tapply()` herangezogen werden.

```
> ave(x=<Vektor>, <Faktor1>, <Faktor2>, ..., FUN=<Funktion>)
> tapply(X=<Vektor>, INDEX=<Liste mit Faktoren>, FUN=<Funktion>, ...)
```

Als Argumente werden neben dem zuerst zu nennenden Datenvektor die Faktoren übergeben. Bei `ave()` geschieht dies einfach in Form mehrerer durch Komma getrennter Gruppierungsfaktoren. Bei `tapply()` müssen die Faktoren in einer Liste zusammengefasst werden, die als Komponenten je einen Faktor enthält (vgl. Abschn. 3.1). Mit dem Argument `FUN` wird schließlich die pro Gruppe auf die Daten anzuwendende Funktion angegeben. Der Argumentname `FUN` ist bei `ave()` immer zu nennen, andernfalls wäre nicht ersichtlich, dass kein weiterer Faktor gemeint ist. In der Voreinstellung von `ave()` wird die `mean()` Funktion angewendet, sie kann jedoch durch eine beliebige andere Funktion ersetzt werden.

In der Ausgabe von `ave()` wird jeder Einzelwert von x durch den für die entsprechende Gruppe berechneten Kennwert ersetzt, was etwa in der Berechnung von Quadratsummen linearer Modelle Anwendung finden kann.

Im Beispiel sei ein IQ-Test mit Personen durchgeführt worden, die aus einer Treatment- (T), Wartelisten- (WL) oder Kontrollgruppe (CG) stammen. Weiterhin sei das Geschlecht als Faktor berücksichtigt worden.

```
> nSubj  <- 2                                # Zellbesetzung
> P      <- 2                                # Anzahl Stufen Geschlecht
> Q      <- 3                                # Anzahl Stufen Treatment
> sex    <- factor(rep(c("f", "m"), Q*nSubj))
> group  <- factor(rep(c("T", "WL", "CG"), each=P*nSubj))
> table(sex, group)

group
sex CG T WL
  f   2 2 2
  m   2 2 2

> IQ <- round(rnorm(P*Q*nSubj, mean=100, sd=15))
> ave(IQ, sex, FUN=mean)
[1] 100.3333 101.5000 100.3333 101.5000 100.3333 101.5000 100.3333
[8] 101.5000 100.3333 101.5000 100.3333 101.5000
```

Die Ausgabe von `tapply()` dient der Übersicht über die gruppenweise berechneten Kennwerte, wobei das Ergebnis die Klasse `array` besitzt (vgl. Abschn. 2.10). Bei einem einzelnen Gruppierungsfaktor ist dies einem benannten Vektor ähnlich und bei zweien einer zweidimensionalen Kreuztabelle (vgl. Abschn. 2.11).

```
> (tapRes <- tapply(IQ, group, FUN=mean))
      CG      T      WL
103.75 103.00 96.00
```

```
> tapply(IQ, list(sex, group), FUN=mean)
      CG      T      WL
f 100.0 108.5 92.5
m 107.5 97.5 99.5
```

Auch die Ausgabe von `tapply()` lässt sich verwenden, um jeden Wert durch einen für seine Gruppe berechneten Kennwert zu ersetzen, da die Elemente der Ausgabe im Fall eines eindimensionalen Arrays als Namen die zugehörigen Gruppenbezeichnungen tragen und sich über diese Namen indizieren lassen. Die als Indizes verwendbaren Gruppenbezeichnungen finden sich im Faktor, wobei jeder Index entsprechend den Gruppengrößen mehrfach auftaucht.

```
> tapRes[group]
      T      T      T      T      WL      WL      WL      WL
103.00 103.00 103.00 103.00 96.00 96.00 96.00 96.00
      CG      CG      CG      CG
103.75 103.75 103.75 103.75
```

Da für `FUN` beliebige Funktionen an `tapply()` übergeben werden können, muss man sich nicht darauf beschränken, für Gruppen getrennt einzelne Kennwerte zu berechnen. Genauso sind Funktionen zugelassen, die pro Gruppe mehr als einen einzelnen Wert ausgeben, wobei das Ergebnis von `tapply()` dann eine Liste ist (vgl. Abschn. 3.1): jede Komponente der Liste beinhaltet die Ausgabe von `FUN` für eine Gruppe.

So ist es etwa auch möglich, sich die Werte jeder Gruppe selbst ausgeben zu lassen, indem man sich den Umstand zunutze macht, dass auch der `[<Index>]` Operator als Funktion "`[]`" geschrieben werden kann (vgl. Abschn. 1.2.5, Fußnote 15).

```
> tapply(IQ, sex, "[")                      # IQ-Werte pro Geschlecht
$f
[1] 108 109 81 104 85 115

$ m
[1] 106 89 108 91 105 110

> IQ[sex == "f"]                           # Kontrolle ...
> IQ[sex == "m"]                           # Kontrolle ...
```

2.8 Matrizen

Wenn für jedes Beobachtungsobjekt Daten von mehreren Variablen vorliegen, können die Werte jeder Variable in einem separaten Vektor gespeichert werden. Eine bessere Möglichkeit zur Zusammenstellung des gesamten Datensatzes bieten Objekte der Klasse `matrix`.³²

³² Eine Matrix ist in R zunächst nur eine rechteckige Anordnung von Werten und nicht mit dem gleichnamigen mathematischen Konzept zu verwechseln. Wie `attributes(<Matrix>)` zeigt, sind

```
> matrix(data=<Vektor>, nrow=<Anzahl>, ncol=<Anzahl>, byrow=FALSE)
```

Unter `data` ist der Vektor einzutragen, der alle Werte der zu bildenden Matrix enthält. Mit `nrow` wird die Anzahl der Zeilen dieser Matrix festgelegt, mit `ncol` die der Spalten. Die Länge des Vektors muss gleich dem Produkt von `nrow` und `ncol` sein, das gleich der Zahl der Zellen ist. Mit dem auf `FALSE` voreingestellten Argument `byrow` wird die Art des Einlesens der Daten aus dem Vektor in die Matrix bestimmt – es werden zunächst die Spalten nacheinander gefüllt. Mit `byrow=TRUE` werden die Werte über die Zeilen eingelesen.

```
> age <- c(17, 30, 30, 25, 23, 21)
> matrix(age, nrow=3, ncol=2, byrow=FALSE)
 [,1] [,2]
[1,] 17   25
[2,] 30   23
[3,] 30   21

> (ageMat <- matrix(age, nrow=2, ncol=3, byrow=TRUE))
 [,1] [,2] [,3]
[1,] 17   30   30
[2,] 25   23   21
```

2.8.1 Datentypen in Matrizen

Wie Vektoren können Matrizen verschiedene Datentypen besitzen, etwa `numeric`, wenn sie Zahlen beinhalten, oder `character` im Fall von Zeichenketten. Jede einzelne Matrix kann dabei aber ebenso wie ein Vektor nur einen einzigen Datentyp haben, alle Matrixelemente müssen also vom selben Datentyp sein. Fügt man einer numerischen Matrix eine Zeichenkette als Element hinzu, so werden die numerischen Matrixelemente automatisch in Zeichenketten umgewandelt,³³ was an den hinzugekommenen Anführungszeichen zu erkennen ist. Auf die ehemals numerischen Werte können dann keine statistischen Funktionen mehr angewendet werden. Dieser Umstand macht Matrizen letztlich weniger geeignet für empirische Datensätze, für die stattdessen Objekte der Klasse `data.frame` bevorzugt werden sollten (vgl. Abschn. 3.2).³⁴

Matrizen intern lediglich Vektoren mit einem Attribut, das Auskunft über die Dimensionierung der Matrix, also die Anzahl ihrer Zeilen und Spalten liefert, vgl. Abschn. 1.3.

³³ Allgemein gesprochen werden alle Elemente in den umfassendsten Datentyp umgewandelt, der notwendig ist, um alle Werte ohne Informationsverlust zu speichern (vgl. Abschn. 1.3.5).

³⁴ Da Matrizen numerisch effizienter als Objekte der Klasse `data.frame` verarbeitet werden können, sind sie dagegen bei der Analyse sehr großer Datenmengen vorzuziehen.

2.8.2 Dimensionierung, Zeilen und Spalten

Die Dimensionierung einer Matrix (also die Anzahl ihrer Zeilen und Spalten) liefert die Funktion `dim(<Matrix>)`, die auch auf Arrays (vgl. Abschn. 2.10) oder Datensätze (vgl. Abschn. 3.2) anwendbar ist. Sie gibt einen Vektor aus, der die Anzahl der Zeilen und Spalten in dieser Reihenfolge als Elemente besitzt. Über die Funktionen `nrow(<Matrix>)` und `ncol(<Matrix>)` kann die Anzahl der Zeilen bzw. Spalten auch einzeln ausgegeben werden.

```
> age      <- c(17, 30, 30, 25, 23, 21)
> ageMat  <- matrix(age, nrow=2, ncol=3, byrow=FALSE)
> dim(ageMat)                      # Dimensionierung
[1] 2 3

> nrow(ageMat)                    # Anzahl der Zeilen
[1] 2

> ncol(ageMat)                    # Anzahl der Spalten
[1] 3

> prod(dim(ageMat))              # Anzahl der Elemente
[1] 6
```

Die `dim()` Funktion kann auch zum Umwandeln eines bestehenden Vektors in eine Matrix genutzt werden, indem ihr die gewünschte Dimensionierung explizit über `dim(<Vektor>)<- c(<Zeilen>, <Spalten>)` zugewiesen wird. Dabei gibt die erste Zahl des zugewiesenen Vektors die Anzahl der Zeilen, die zweite die der Spalten vor. Die Elemente des Wertevektors werden bei dieser Methode über die Spalten in die zu bildende Matrix eingelesen.

```
> dim(age) <- c(2, 3)
> age
 [,1] [,2] [,3]
[1,] 17   30   23
[2,] 30   25   21
```

Wird ein Vektor über die Konvertierungsfunktion `as.matrix(<Vektor>)` in eine Matrix umgewandelt, entsteht als Ergebnis eine Matrix mit einer Spalte und so vielen Zeilen, wie der Vektor Elemente enthält.

```
> as.matrix(1:4)
 [,1]
[1,]    1
[2,]    2
[3,]    3
[4,]    4
```

Um eine Matrix in einen Vektor umzuwandeln, sollte entweder die Konvertierungsfunktion `as.vector(<Matrix>)` oder einfach `c(<Matrix>)` verwendet

werden.³⁵ Die Anordnung der Elemente entspricht dabei der Umkehrung des Einlesens über die Spalten.

```
> c(ageMat)
[1] 17 30 30 25 23 21
```

Mitunter ist es nützlich, zu einer gegebenen Matrix zwei zugehörige Matrizen zu erstellen, in denen jedes ursprüngliche Element durch seinen Zeilen- bzw. Spaltenindex ersetzt wurde. Während sich dieses Ziel auch manuell erreichen lässt, ist die einfachste Möglichkeit, die dafür bereitgestellten Funktionen `row(<Matrix>)` und `col(<Matrix>)` zu verwenden.

```
> P      <- 2                      # Anzahl Zeilen
> Q      <- 3                      # Anzahl Spalten
> (pqMat <- matrix(1:(P*Q), nrow=P, ncol=Q))
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

> (rowMat1 <- row(pqMat))
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    2    2    2

> (colMat1 <- col(pqMat))
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    1    2    3

# manuelle Lösung mit demselben Ergebnis
> rowMat2 <- matrix(rep(1:P, Q),      ncol=Q)
> colMat2 <- matrix(rep(1:Q, each=P), ncol=Q)
```

Die so gewonnenen Matrizen können etwa dazu verwendet werden, eine dreispaltige Matrix zu erstellen, die in einer Spalte alle Elemente der ursprünglichen Matrix besitzt und in den anderen beiden Spalten die zugehörigen Zeilen- und Spaltenindizes enthält (vgl. Abschn. 2.8.5).

```
> cbind(rowIdx=c(rowMat1), colIdx=c(colMat1), val=c(pqMat))
   rowIdx colIdx val
[1,]    1    1    1
[2,]    2    1    2
[3,]    1    2    3
[4,]    2    2    4
[5,]    1    3    5
[6,]    2    3    6
```

³⁵ Eine Matrix kann auch über die `dim()` Funktion in ein eindimensionales Objekt umgewandelt werden, indem `dim()` nur ein einzelner Wert, nämlich die Länge des Vektors, als Dimensionierung zugewiesen wird. Das Ergebnis dieser Umwandlung ist aber ein Objekt der Klasse `array` (vgl. Abschn. 2.10).

Eine andere Anwendungsmöglichkeit besteht in der Erstellung von unteren und oberen Dreiecksmatrizen, die in der linearen Algebra (vgl. Abschn. 2.9.1) häufig zum Einsatz kommen.

```
> mat <- matrix(sample(1:10, 16, replace=TRUE), 4, 4)
> col(mat) >= row(mat)
 [,1]  [,2]  [,3]  [,4]
[1,] TRUE  TRUE  TRUE  TRUE
[2,] FALSE TRUE  TRUE  TRUE
[3,] FALSE FALSE TRUE  TRUE
[4,] FALSE FALSE FALSE TRUE
```

Ein anderer Weg bestünde in der Anwendung der Funktionen `lower.tri()` und `upper.tri()`.

2.8.3 Elemente auswählen und verändern

In einer Matrix ist es ähnlich wie bei einem Vektor möglich, sich einzelne Elemente mit dem `[<Zeile>, <Spalte>]` Operator anzeigen zu lassen. Der erste Index in der eckigen Klammer gibt dabei die Zeile des gewünschten Elements an, der zweite seine Spalte.³⁶

```
> ageMat[2, 2]
[1] 25
```

Analog zum Vorgehen bei Vektoren können auch bei Matrizen einzelne Elemente durch Angabe ihres Zeilen- und Spaltenindex bei der Zuweisung eines Wertes verändert werden:

```
> ageMat[2, 2] <- 24; ageMat[2, 2]
[1] 24
```

Man kann sich Zeilen oder Spalten auch vollständig ausgeben lassen. Dafür wird für die vollständig aufzulistende Dimension kein Index eingetragen, jedoch das Komma trotzdem gesetzt. Es können dabei auch beide Dimensionen weggelassen werden, was für die Ausgabe denselben Effekt wie das Weglassen des `[<Index>]` Operators überhaupt hat.³⁷

```
> ageMat[2, ]
[1] 30 24 21

> ageMat[, 1]
[1] 17 30
```

³⁶ Für Hilfe zu diesem Thema vgl. `?Extract`.

³⁷ Bei Zuweisungen unterscheiden sich jedoch `<Matrix> <- ...` und `<Matrix>[] <- ...` voneinander. Der erste Befehl erstellt ein vollständig neues Objekt, mit dem das alte desselben Namens überschrieben wird. Mit dem zweiten Befehl werden dagegen nur die Elemente der bestehenden Matrix überschrieben, was im Fall sehr großer Matrizen numerisch effizienter ist.

```
> ageMat[ , ]                                # äquivalent: ageMat[]
   [,1] [,2] [,3]
[1,]   17   30   23
[2,]   30   24   21
```

Bei der Ausgabe einer einzelnen Zeile oder Spalte wird diese automatisch in einen Vektor umgewandelt, verliert also eine Dimension. Möchte man dies – wie es häufig der Fall ist – verhindern, kann beim [$\langle\text{Index}\rangle$] Operator als weiteres Argument `drop=FALSE` angegeben werden. Das Ergebnis ist dann eine Matrix mit nur einer Zeile oder Spalte.

```
> ageMat[ , 1, drop=FALSE]
   [,1]
[1,]   17
[2,]   30
```

Analog zum Vorgehen bei Vektoren können auch gleichzeitig mehrere Matrixelemente ausgewählt und verändert werden, indem man etwa eine Sequenz oder einen anderen Vektor als Indexvektor für eine Dimension festlegt.

```
> ageMat[ , 2:3]
   [,1] [,2]
[1,]   30   23
[2,]   24   21

> ageMat[ , c(1, 3)]
   [,1] [,2]
[1,]   17   23
[2,]   30   21

> ageMatNew <- ageMat
> (replaceMat <- matrix(c(11, 21, 12, 22), 2, 2))
   [,1] [,2]
[1,]   11   12
[2,]   21   22

> ageMatNew[ , c(1, 3)] <- replaceMat  # ersetze Spalten 1 und 3
> ageMatNew
   [,1] [,2] [,3]
[1,]   11   30   12
[2,]   21   24   22
```

2.8.4 Weitere Wege, um Elemente auszuwählen und zu verändern

Auf Matrixelemente kann auch zugegriffen werden, wenn nur ein einzelner Index genannt wird. Die Matrix wird dabei implizit in einen Vektor umgewandelt, indem die Spalten untereinander gehängt werden.

```
> ageMat[c(1, 3, 4)]
[1] 17 30 24
```

Weiter ist es möglich, eine zweispaltige Indexmatrix zu verwenden, wobei jede Zeile dieser Matrix ein Element der indizierten Matrix auswählt – der erste Eintrag einer Zeile gibt den Zeilenindex, der zweite den Spaltenindex des auszuwählenden Elements an. In beiden Fällen ist das Ergebnis ein Vektor der ausgewählten Elemente.

```
> (idxMat <- matrix(c(1, 1, 2, 2, 3, 3), 3, 2))
[,1] [,2]
[1,]    1    2
[2,]    1    3
[3,]    2    3

> ageMat[idxMat]
[1] 30 23 21
```

Schließlich können Matrizen auch durch eine logische Matrix derselben Dimensionierung – etwa als Ergebnis eines logischen Vergleichs – indiziert werden, die für jedes Element bestimmt, ob es ausgegeben werden soll. Auch hier ist das Ergebnis ein Vektor der ausgewählten Elemente.

```
> (selMat <- ageMat >= 25)
[,1] [,2] [,3]
[1,] FALSE TRUE FALSE
[2,] TRUE FALSE FALSE

> ageMat[selMat]
[1] 30 30
```

2.8.5 Matrizen verbinden

Eine Möglichkeit, Matrizen aus vorhandenen Daten zu erstellen, besteht mit den Funktionen `cbind(Vektor1, Vektor2, ...)` und `rbind(Vektor1, r(Vektor2), ...)`, die Vektoren zu Matrizen zusammenfügen. Das `c` bei `cbind()` steht für Columns (Spalten), das `r` entsprechend für Rows (Zeilen). Je nachdem, ob die bereits vorhandenen Vektoren als Zeilen oder Spalten dienen sollen, werden sie beim Verbinden untereinander oder nebeneinander angeordnet.

```
> nSubj   <- 10
> age     <- sample( 18:35, nSubj, replace=TRUE)
> weight  <- sample( 65:95, nSubj, replace=TRUE)
> height  <- sample(168:201, nSubj, replace=TRUE)
> rbind(age, weight, height)
[,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
age     19   18   31   20   29   22   20   22   30   23
weight  67   74   90   78   66   69   74   77   91   78
height  170  182  199  177  172  193  172  170  197  199

> varMat <- cbind(age, weight, height); varMat[1:5, ]
age weight height
```

```
[1,] 19 67 170
[2,] 18 74 182
[3,] 31 90 199
[4,] 20 78 177
[5,] 29 66 172
```

2.8.6 Randkennwerte

Die Summe aller Elemente einer Matrix lässt sich mit `sum(<Matrix>)`, die separat über jede Zeile oder jede Spalte gebildeten Summen durch die Funktionen `rowSums(<Matrix>)` bzw. `colSums(<Matrix>)` berechnen. Gleiches gilt für den Mittelwert aller Elemente, der mit `mean(<Matrix>)` ermittelt wird und die mit `rowMeans()` bzw. `colMeans()` separat über jede Zeile oder jede Spalte berechneten Mittelwerte.

```
> ageMat
      [,1] [,2] [,3]
[1,] 17   30   23
[2,] 30   24   21

> sum(ageMat)                                # Summe aller Elemente
[1] 145

> rowSums(ageMat)                            # Summen jeder Zeile
[1] 70 75

> mean(ageMat)                               # Mittelwert aller Elemente
[1] 24.16667

> colMeans(ageMat)                           # Mittelwerte jeder Spalte
[1] 23.5 27.0 22.0
```

2.8.7 Beliebige Funktionen auf Matrizen anwenden

Wenn eine andere Funktion als die Summe oder der Mittelwert separat auf jeweils jede Zeile oder jede Spalte angewendet werden soll, ist dies mit der Funktion `apply()` zu erreichen.

```
> apply(X=<Matrix>, MARGIN=<Nummer>, FUN=<Funktion>, ...)
```

X erwartet die Matrix der zu verarbeitenden Daten. Unter MARGIN wird angegeben, ob die Funktion jeweils von Zeilen (1) oder Spalten (2) Kennwerte berechnet. Für FUN ist die anzuwendende Funktion einzusetzen, die als Argument einen Vektor akzeptieren muss. Gibt sie mehr als einen Wert zurück, ist das Ergebnis eine Matrix mit den Rückgabewerten von FUN in den Spalten. Die drei Punkte ... stehen für optionale, ggf. durch Komma getrennte Argumente für diese Funktion – FUN wertet sie also ihrerseits als Argumente aus.

```

> apply(ageMat, 2, sum)                                # Summen jeder Spalte
[1] 47 54 44

> apply(ageMat, 1, max)                               # Maxima jeder Zeile
[1] 30 30

> apply(ageMat, 2, range)                            # Range jeder Spalte
[,1] [,2] [,3]
[1,]   17   24   21
[2,]   30   30   23

> mat <- matrix(sample(1:100, 120, replace=TRUE), ncol=3)
> apply(mat, 2, mean, trim=0.1)          # gestutzte Mittelwerte jeder Spalte
[1] 48.84375 48.31250 48.00000

```

Im letzten Beispiel wird das für . . . eingesetzte Argument `trim=0.1` an die `mean()` Funktion weitergereicht.

2.8.8 Matrix zeilen- oder spaltenweise mit Kennwerten verrechnen

Zeilen- und Spaltenkennwerte sind häufig Zwischenergebnisse, die für weitere Berechnungen mit einer Matrix nützlich sind. So ist es etwa zum spaltenweisen Zentrieren einer Matrix notwendig, von jeder Spalte den zugehörigen Spaltenmittelwert abzuziehen. Anders gesagt soll die Matrix dergestalt mit einem Vektor verrechnet werden, dass auf jede Spalte dieselbe Operation (hier: Subtraktion), aber mit einem anderen Wert angewendet wird – nämlich mit dem Element des Vektors der Spaltenmittelwerte, das dieselbe Position im Vektor besitzt wie die Spalte in der Matrix. Die genannte Operation lässt sich manuell durchführen, bequemer ist es jedoch, die `sweep()` Funktion zu verwenden.

```
> sweep(x=<Matrix>, MARGIN=<Nummer>, STATS=<Kennwerte>, FUN=<Funktion>, ...)
```

`x` erwartet die Matrix der zu verarbeitenden Daten. Unter `MARGIN` wird angegeben, ob die Funktion jeweils Zeilen (1) oder Spalten (2) mit den Kennwerten verrechnet. An `STATS` sind diese Kennwerte in Form eines Vektors mit so vielen Einträgen zu übergeben, wie `x` Zeilen (`MARGIN=1`) bzw. Spalten (`MARGIN=2`) besitzt. Für `FUN` ist die anzuwendende Funktion einzusetzen, Voreinstellung ist die Subtraktion `"+"`. Die drei Punkte . . . stehen für optionale, ggf. durch Komma getrennte Argumente für diese Funktion – `FUN` wertet sie also ihrerseits als Argumente aus.

Im Beispiel sollen die Daten einer Matrix erst zeilenweise, dann spaltenweise zentriert werden.

```

> rowMs <- rowMeans(ageMat)                         # Mittelwerte der Zeilen
> colMs <- colMeans(ageMat)                         # Mittelwerte der Spalten
> sweep(ageMat, 1, rowMs, "-")                      # zeilenweise zentrieren
[,1]      [,2]      [,3]
[1,] -6.333333 6.666667 -0.3333333
[2,]  5.000000 -1.000000 -4.0000000

```

```
> sweep(ageMat, 2, colMs, "-")                                # spaltenweise zentrieren
   [,1] [,2] [,3]
[1,] -6.5   3    1
[2,]  6.5  -3   -1

> scale(ageMat, center=TRUE, scale=FALSE)                      # Kontrolle mittels scale()
   [,1] [,2] [,3]
[1,] -6.5   3    1
[2,]  6.5  -3   -1

attr("scaled:center")
[1] 23.5 27.0 22.0

# Kontrolle: manuelle Vervielfältigung der Spaltenmittelwerte zur Matrix ...
> ageMat - matrix(rep(colMs, nrow(ageMat)), nrow=nrow(ageMat), byrow=TRUE)
```

2.8.9 Kovarianz- und Korrelationsmatrizen

Zum Erstellen von Kovarianz- und Korrelationsmatrizen können die schon bekannten Funktionen `var(Matrix)`, `cov(Matrix)` und `cor(Matrix)` verwendet werden, wobei die Werte in Form einer Matrix (Variablen in den Spalten) übergeben werden müssen. `var()` und `cov()` liefern beide die Kovarianzmatrix, wobei sie wie bei Vektoren die korrigierten Kennwerte berechnen.

```
> cov(varMat)
      age     weight     height
age    23.15556  23.04444  30.06667
weight 23.04444  74.04444  69.06667
height 30.06667  69.06667 158.32222

> cor(varMat)
      age     weight     height
age   1.0000000 0.5565349 0.4965772
weight 0.5565349 1.0000000 0.6378981
height 0.4965772 0.6378981 1.0000000
```

Zudem existiert die `cov.wt(Matrix)` Funktion, die auch direkt die unkorrigierte Kovarianzmatrix ermitteln kann.

```
> cov.wt(x=Matrix, method=c("unbiased", "ML"))
```

Unter `x` ist die Matrix einzutragen, deren Spalten-Kovarianzen bestimmt werden sollen – zur Berechnung der unkorrigierten Varianz eines einzelnen Vektors ist dieser also zunächst mit `as.matrix(Vektor)` zu konvertieren. Mit `method` kann gewählt werden, ob die korrigierten oder unkorrigierten Kovarianzen sowie Varianzen ausgerechnet werden. `method` ist dabei auf "unbiased" für die korrigierten Kennwerte voreingestellt. Sind die unkorrigierten Kennwerte gewünscht, ist `method` auf "ML" zu setzen, da sie die Maximum-Likelihood-Schätzung der theoretischen Parameter auf Basis einer Stichprobe darstellen.

Das Ergebnis der Funktion ist eine Liste (vgl. Abschn. 3.1), die die Kovarianzmatrix als Komponente `cov`, die Mittelwerte als Komponente `center` und die Anzahl der eingegangenen Fälle als Komponente `n.obs` (Number of Observations) besitzt.

```
> cov.wt(varMat, method="ML")
$cov
    age   weight   height
age   20.84   20.74   27.06
weight 20.74   66.64   62.16
height 27.06   62.16  142.49

$center
    age   weight   height
age   23.4    76.4   183.1

$n.obs
[1] 10
```

Mit `diag(x=(Matrix))` (vgl. Abschn. 2.9.1) lassen sich aus einer Kovarianzmatrix die in der Diagonale stehenden Varianzen extrahieren.

```
> diag(cov(varMat))
    age   weight   height
23.15556 74.04444 158.32222
```

Um gleichzeitig die Kovarianz oder Korrelation einer durch `<Vektor>` gegebenen Variable mit mehreren anderen, spaltenweise zu einer Matrix zusammengefassten Variablen zu berechnen, dient die Syntax `cov(<Matrix>, <Vektor>)` bzw. `cor(<Matrix>, <Vektor>)`. Output ist eine Matrix mit so vielen Zeilen wie das erste Argument Spalten besitzt.

```
> vec <- rnorm(nSubj)
> cor(varMat, vec)
    [,1]
age     0.37395871
weight -0.05100238
height -0.68309930
```

2.8.10 Matrizen sortieren

Die Zeilen von Matrizen können mit Hilfe der `order()` Funktion entsprechend der Reihenfolge der Werte in einer ihrer Spalten sortiert werden. Die `sort()` Funktion ist hier nicht anwendbar, ihr Einsatz ist auf Vektoren beschränkt.

```
> order(<Vektor>, partial, decreasing=FALSE)
```

Für `<Vektor>` ist die Spalte einer Datenmatrix einzutragen, deren Werte in eine Reihenfolge gebracht werden sollen. Unter `decreasing` wird die Sortierreihenfolge eingestellt: per Voreinstellung `FALSE` wird aufsteigend sortiert, auf `TRUE` gesetzt

absteigend. Die Ausgabe ist ein Indexvektor, der die Zeilenindizes der zu ordnenden Matrix in der Reihenfolge der Werte des Sortierkriteriums enthält (vgl. Abschn. 2.1.4).

```
> nSubj <- 5
> sex <- sample(c(0, 1), nSubj, replace=TRUE)
> age <- sample(18:35, nSubj, replace=TRUE)
> rating <- sample(1:3, nSubj, replace=TRUE)
> (mat <- cbind(sex, age, rating))
   sex age rating
[1,] 1   32     1
[2,] 1   18     1
[3,] 0   24     2
[4,] 0   35     1
[5,] 1   22     3

> (rowOrder1 <- order(mat[, 1]))
[1] 3 4 1 2 5
```

Soll die gesamte Matrix entsprechend der Reihenfolge dieser Variable angezeigt werden, ist der von `order()` ausgegebene Indexvektor zum Indizieren der Zeilen der Matrix zu benutzen. Dabei ist der Spaltenindex unter Beibehaltung des Kommas wegzulassen.

```
> mat[rowOrder1, ]
   sex age rating
[1,] 0   24     2
[2,] 0   35     1
[3,] 1   32     1
[4,] 1   18     1
[5,] 1   22     3
```

Mit dem Argument `partial` kann noch eine weitere Matrixspalte eingetragen werden, die dann als sekundäres Sortierkriterium verwendet wird. So kann eine Matrix etwa zunächst hinsichtlich einer die Gruppenzugehörigkeit darstellenden Variable sortiert werden und dann innerhalb jeder Gruppe nach der Reihenfolge der Werte einer anderen Variable.

```
# sortiere primär nach Geschlecht und sekundär nach Alter
> rowOrder2 <- order(mat[, 1], partial=mat[, 2])
> mat[rowOrder2, ]
   sex age rating
[1,] 0   24     2
[2,] 0   35     1
[3,] 1   18     1
[4,] 1   22     3
[5,] 1   32     1
```

Der Argumentname `partial` muss dabei nicht explizit genannt werden – es reicht aus, einfach eine weitere Spalte durch Komma getrennt anzugeben, die dann automatisch als weiteres Sortierkriterium interpretiert wird (etwa `order(mat[, 1], mat[, 2])`). Zudem können noch weitere Sortierkriterien durch Komma getrennt

als Argumente vorhanden sein, es gibt keine Beschränkung auf nur zwei solcher Kriterien.

Das Argument `decreasing` legt global für alle Sortierkriterien fest, ob auf- oder absteigend sortiert wird. Soll die Sortierreihenfolge dagegen zwischen den Kriterien variieren, kann einzelnen Kriterien ein - vorangestellt werden, das als Gegenteil der mit `decreasing` eingestellten Reihenfolge zu verstehen ist.

```
# sortiere aufsteigend nach Geschlecht und absteigend nach Alter
> rowOrder3 <- order(mat[, 1], -mat[, 2])
> mat[rowOrder3, ]
   sex age rating
[1,] 0   35     1
[2,] 0   24     2
[3,] 1   32     1
[4,] 1   22     3
[5,] 1   18     1
```

2.9 Lineare Algebra

Vielen statistischen Auswertungen liegt im Kern das Rechnen mit Matrizen zugrunde, weshalb kurz auf grundlegende Funktionen zur Matrix-Algebra eingegangen werden soll.³⁸ Einem Nutzer der von R bereitgestellten Auswertungsfunktionen bleiben diese Rechnungen meist verborgen, sie sind für das Verständnis der folgenden Abschnitte auch nicht wesentlich. Erst wenn in Kap. 9 zu multivariaten Verfahren Rechnungen manuell nachvollzogen werden, tauchen Matrixoperationen explizit auf. Die Rechentechniken und zugehörigen Konzepte der Linearen Algebra (Fischer, 2008; Härdle und Simar, 2007; Strang, 2003) seien als bekannt vorausgesetzt.

2.9.1 Matrix-Algebra

Eine Matrix X wird mit der `t(x=Matrix)` Funktion transponiert, wodurch die Zeilen von X zu den Spalten der Transponierten X^t und entsprechend die Spalten von X zu Zeilen von X^t werden.

```
> nn <- 4                      # Anzahl Zeilen
> qq <- 2                      # Anzahl Spalten
> (X <- matrix(c(20, 26, 10, 19, 29, 27, 20, 12), nrow=nn, ncol=qq))
```

³⁸ Bei sehr großen Datensätzen können Rechenoperationen für Matrizen die Auswertungszeit entscheidend bestimmen. Ist die Effizienz der numerischen Berechnungen wichtig, empfiehlt es sich daher, auf eine für den im Computer verwendeten Prozessor optimierte Version der sog. *BLAS*-Bibliothek (Basic Linear Algebra Subprograms) von R zurückzugreifen (vgl. Frage 8.2 in Ripley und Murdoch, 2009). Zudem sollten Matrizen nicht dynamisch erweitert, sondern bereits mit der Dimensionierung erstellt werden, die sie später benötigen (vgl. Abschn. 2.1.2, Fußnote 6).

```
[,1] [,2]
[1,] 20 29
[2,] 26 27
[3,] 10 20
[4,] 19 12

> t(X) # Transponierte
[,1] [,2] [,3] [,4]
[1,] 20 26 10 19
[2,] 29 27 20 12
```

Die Diagonalelemente einer Matrix gibt die `diag(x=Matrix)` Funktion in Form eines Vektors aus. Die Umkehrung `diag(x=Vektor)` erzeugt eine Diagonalmatrix, deren Diagonale aus den Elementen von `<Vektor>` besteht. `diag(x=Anzahl)` liefert als besondere Diagonalmatrix eine Einheitsmatrix mit `<Anzahl>` vielen Zeilen und Spalten.

```
# Diagonale der Kovarianzmatrix von X -> Varianzen der Spalten von X
> diag(cov(X))
[1] 43.58333 59.33333

> diag(1:3) # Diagonalmatrix mit Diagonale 1, 2, 3
[,1] [,2] [,3]
[1,] 1 0 0
[2,] 0 2 0
[3,] 0 0 3

> diag(2) # 2x2 Einheitsmatrix
[,1] [,2]
[1,] 1 0
[2,] 0 1
```

Da auch bei Matrizen die herkömmlichen Operatoren elementweise arbeiten, eignet sich für die Addition von Matrizen passender Dimensionierung der `+` Operator mit `<Matrix1> + <Matrix2>` genauso wie etwa `*` für die Multiplikation von Matrizen mit einem Skalar mittels `<Matrix> * <Zahl>`. Auch `<Matrix1> * <Matrix2>` ist als Matrix der elementweisen Produkte von `<Matrix1>` und `<Matrix2>` zu verstehen, die Matrix-Multiplikation von Matrizen kompatibler Dimensionierung lässt sich dagegen mit `<Matrix1> %*% <Matrix2>` formulieren. Dabei existiert für den häufig genutzten Spezialfall aller paarweisen Skalarprodukte der Spalten einer Matrix die `crossprod(x=Matrix)` Funktion.

Die Matrix-Multiplikation ist auch auf Vektoren anwendbar, da diese als Matrizen mit nur einer Zeile bzw. einer Spalte aufgefasst werden können. Gewöhnliche mit `c()` oder `numeric()` erstellte Vektoren gelten dabei als Spaltenvektoren.

Als Beispiel sollen einige wichtige Matrizen zu einer gegebenen Datenmatrix X mit Variablen in den Spalten berechnet werden, etwa die spaltenweise zentrierte Datenmatrix \dot{X} , die *SSCP*-Matrix (Sum of Squares and Cross Products) sowie die

Matrizen, in denen jedes Element durch seinen Zeilen- bzw. Spaltenindex ersetzt wird.³⁹

```
> (Xc <- diag(nn) - (1/nn) * (rep(1, nn) %*% t(rep(1, nn)))) # Zentriermatrix
[1,] [2,] [3,] [4,]
[1,] 0.75 -0.25 -0.25 -0.25
[2,] -0.25 0.75 -0.25 -0.25
[3,] -0.25 -0.25 0.75 -0.25
[4,] -0.25 -0.25 -0.25 0.75

> all((Xc %*% Xc) == Xc)                                # Zentriermatrix ist idempotent
[1] TRUE

> (Xdot <- Xc %*% X)                                     # spaltenweise zentrierte Daten
[1,] [2,]
[1,] 1.25 7
[2,] 7.25 5
[3,] -8.75 -2
[4,] 0.25 -10

> colSums(Xdot)                                         # Kontrolle: Nullvektor
[1] 0 0

> (SSCP <- t(Xdot) %*% Xdot)                            # SSCP-Matrix
[1,] [2,]
[1,] 130.75 60
[2,] 60.00 178

> (1/(nn-1)) * SSCP                                    # korrigierte Kovarianzmatrix
[1,] [2,]
[1,] 43.58333 20.00000
[2,] 20.00000 59.33333

> cov(X)                                                 # Kontrolle ...
> (1:nn) %*% t(rep(1, qq))                            # ersetze Elemente durch Zeilenindex
[1,] [2,]
[1,] 1 1
[2,] 2 2
[3,] 3 3
[4,] 4 4

> rep(1, nn) %*% t(1:qq)                               # ersetze Elemente durch Spaltenindex
[1,] [2,]
[1,] 1 2
[2,] 1 2
[3,] 1 2
[4,] 1 2
```

³⁹ Die Beispiele zeigen nur eine – numerisch eher ineffiziente – Möglichkeit von mehreren, um diese Matrizen zu erzeugen. Vergleiche insbesondere `sweep()` zum spaltenweisen Zentrieren von Matrizen und `row()` bzw. `col()`, um Elemente durch ihren Zeilen- bzw. Spaltenindex zu ersetzen.

2.9.2 Lineare Gleichungssysteme lösen

Für die Berechnung der Inversen A^{-1} einer quadratischen Matrix A mit vollem Rang stellt R keine separate Funktion bereit. Aus diesem Grund ist die allgemeinere Funktion `solve(a=Matrix, b=Matrix)` zu nutzen, die die Lösung x des linearen Gleichungssystems $A \cdot x = b$ liefert, wobei für a eine invertierbare quadratische Matrix und für b ein Vektor oder eine Matrix passender Dimensionierung anzugeben ist.⁴⁰ Fehlt das Argument b , wird als rechte Seite des Gleichungssystems die passende Einheitsmatrix angenommen, womit sich als Lösung für x die Inverse A^{-1} ergibt.⁴¹

```
> mat      <- matrix(c(1, 1, 1, -1), nrow=2)    # zu invertierende Matrix
> (matInv <- solve(mat))                      # ihre Inverse
[,1]  [,2]
[1,]  0.5  0.5
[2,]  0.5 -0.5

> mat %*% matInv                                # Kontrolle: Einheitsmatrix
[,1]  [,2]
[1,]  1     0
[2,]  0     1

# löse lineares Gleichungssystem
> A   <- matrix(c(9, 1, -5, 0), nrow=2)        # Matrix
> b   <- c(5, -3)                               # Ergebnisvektor
> (x <- solve(A, b))                          # Lösung
[1] -3.0 -6.4

> A %*% x                                     # Kontrolle: reproduziere Vektor
[,1]
[1,]  5
[2,] -3
```

2.9.3 Norm und Abstand von Vektoren und Matrizen

Das mit `crossprod(<Vektor>)` ermittelte Skalarprodukt eines Vektors mit sich selbst liefert dessen quadrierte Länge im Sinne der Norm (hier nicht zu verwechseln mit der Anzahl der Elemente des Vektors). Genauso eignet sich `crossprod(<Matrix>)`, um die quadrierten Spaltennormen einer Matrix zu ermitteln, die im Ergebnis in der Diagonale stehen.⁴²

⁴⁰ Für die Pseudoinverse vgl. `ginv()` aus dem MASS Paket (Venables und Ripley, 2002).

⁴¹ Für nicht invertierbare $(p \times p)$ -Matrizen A ($\text{Rang}(A) < p$) ermittelt die `Null()` Funktion aus dem MASS Paket eine Basis des Kerns von A (engl. Null Space bzw. Kernel).

⁴² Für verschiedene Matrixnormen vgl. die `norm()` Funktion aus dem Matrix Paket (Bates und Mächler, 2010).

```

> vec1 <- c(3, 4, 1, 8, 2)
> sqrt(crossprod(vec1))           # Norm (Länge) von vec1
[1,]
[1,] 9.69536

> sqrt(sum(vec1^2))             # Kontrolle ...
> vec2 <- c(6, 9, 10, 8, 7)    # weiterer Vektor
> mat1 <- cbind(vec1, vec2)     # verbinde Vektoren zu Matrix
> sqrt(diag(crossprod(mat1)))   # Spaltennormen der Matrix
vec1      vec2
9.69536 18.16590

> sqrt(colSums(mat1^2))         # Kontrolle ...

```

Der euklidische Abstand zwischen zwei Vektoren ergibt sich als Norm des Differenzvektors, mit `dist()` ist jedoch auch eine Funktion verfügbar, die unterschiedliche Abstandsmaße direkt berechnen kann.

```
> dist(x=<<Matrix>>, method=<<Abstandsmaß>>, diag=FALSE, upper=FALSE, p=2)
```

Das Argument `x` erwartet eine zeilenweise aus Koordinatenvektoren zusammengestellte Matrix. In der Voreinstellung werden alle paarweisen euklidischen Abstände zwischen ihren Zeilen berechnet. Über `method` lassen sich aber auch andere Abstandsmaße wählen, etwa die City-Block-Metrik mit "manhattan" oder die Minkowski- p -Norm mit "minkowski". Ein konkretes p kann in diesem Fall für `p` übergeben werden. Die Ausgabe enthält in der Voreinstellung die untere Dreiecksmatrix der paarweisen Abstände. Soll auch die Diagonale ausgegeben werden, ist `diag=TRUE` zu setzen, ebenso `upper=TRUE` für die obere Dreiecksmatrix.

```

> mat2 <- matrix(sample(-20:20, 12, replace=TRUE), ncol=3)
> dist(mat2, diag=TRUE, upper=TRUE)
      1          2          3          4
1 0.000000 26.095977 29.698485 17.262677
2 26.095977 0.000000 7.681146 30.282008
3 29.698485 7.681146 0.000000 32.954514
4 17.262677 30.282008 32.954514 0.000000

# Kontrolle für den euklidischen Abstand der 1. und 2. Zeile
# Wurzel aus Skalarprodukt des Differenzvektors mit sich selbst
> sqrt(crossprod(mat2[1, ] - mat2[2, ]))
[1,]
[1,] 26.09598

```

Die Mahalanobistransformation ist eine affine Transformation einer multivariaten Variable, durch die das Zentroid der Daten im Ursprung des Koordinatensystems liegt und die Datenmatrix als Kovarianzmatrix die zugehörige Einheitsmatrix besitzt. In diesem Sinne handelt es sich um eine Verallgemeinerung der univariaten

z -Transformation. Ist S die Kovarianzmatrix einer Variable X mit Zentroid \bar{x} ergibt $S^{-1/2} \cdot (x - \bar{x})$ die Mahalanobistransformierte eines konkreten Datenvektors x .⁴³

Im Beispiel seien an mehreren Bewerbern für eine Arbeitsstelle drei Eigenschaften erhoben worden. Zunächst wird die `rmvnorm()` Funktion des `mvtnorm` Pakets (Genz et al., 2009; Hothorn et al., 2001) verwendet, um Zufallsvektoren als simulierte Realisierungen einer multinormalverteilten dreidimensionalen Variable herzustellen. Die Verwendung von `rmvnorm()` gleicht der von `rnorm()`, lediglich muss hier das theoretische Zentroid μ für das Argument `mean` und die theoretische Kovarianzmatrix Σ für `sigma` angegeben werden. Die erzeugten Daten werden dann einer Mahalanobistransformation unterzogen (vgl. Abschn. 2.9.5 für die Berechnung von $S^{-1/2}$ durch Diagonalisieren von S).

```
# theoretische 3x3 Kovarianzmatrix
> sigma <- matrix(c(4,2,-3, 2,16,-1, -3,-1,9), byrow=TRUE, ncol=3)
> mu     <- c(-3, 2, 4)           # theoretisches Zentroid
> nSubj  <- 100                 # Anzahl Bewerber
> library(mvtnorm)              # Zufallsvektoren mit mvtnorm
> X      <- round(rmvnorm(n=nSubj, mean=mu, sigma=sigma)) # Zufallsvektoren
> ctr    <- colMeans(X)         # empirisches Zentroid
> S      <- cov(X)             # empirische Kovarianzmatrix
> Seig   <- eigen(S)            # Eigenwerte und -vektoren von S
> sqrtD <- sqrt(Seig$values)    # Wurzel aus Eigenwerten

# berechne  $S^{-1/2}$  durch Diagonalisieren
> SsqrtInv <- Seig$vectors %*% diag(1/sqrtD) %*% t(Seig$vectors)

# Differenzvektoren zwischen Daten und Zentroid
> Xctr <- sweep(X, 2, ctr, "-")

# Mahalanobistransformation der Daten
> Xm <- t(SsqrtInv %*% t(Xctr))

# Kontrolle: Kovarianzmatrix der transformierten Daten: Einheitsmatrix
> cov(Xm)                      # ...

# Kontrolle: Zentroid der transformierten Daten: Null-Vektor
> colMeans(Xm)                  # ...
```

Die Mahalanobisdistanz zwischen zwei Vektoren x und y bzgl. einer Kovarianzmatrix S ist definiert als $(x - y)^t \cdot S^{-1} \cdot (x - y)$. Ihr Quadrat lässt sich durch `mahalanobis()` berechnen.

```
> mahalanobis(x=(Matrix), center=(Vektor), cov=(Kovarianzmatrix))
```

Das Argument `x` erwartet entweder einen Vektor oder eine zeilenweise aus Koordinatenvektoren zusammengestellte Matrix. Für `center` ist ein Vektor anzugeben,

⁴³ Jede Transformation der Form $A \cdot S^{-1/2} \cdot (x - \bar{x})$ mit A als Orthogonalmatrix ($A^t = A^{-1}$) würde ebenfalls eine multivariate z -Transformation liefern.

dessen jeweilige Distanz zu den Vektoren aus x berechnet wird. Die Kovarianzmatrix, bzgl. der die Transformation durchgeführt werden soll, ist für cov zu nennen. Häufig ist `center` das für die Mahalanobistransformation verwendete Zentroid der Variablen, von denen die Koordinatenvektoren in x stammen und cov die Kovarianzmatrix dieser Variablen. Die Ausgabe ist ein Vektor mit den quadrierten Mahalanobis-Distanzen der Zeilen von x zum Vektor `center`.

Im obigen Beispiel sei der Bewerber zu bevorzugen, der einem Idealprofil, also vorher festgelegten konkreten Ausprägungen für jede Merkmalsdimension, am ehesten entspricht. Bei der Berechnung der Nähe zwischen Bewerber und Profil im Sinne der Mahalanobisdistanz sind dabei Streuungen und Korrelationen der einzelnen Merkmale mit zu berücksichtigen.

```
> ideal <- c(1, 2, 3)           # Idealprofil
> x     <- X[1, ]             # Bewerber 1
> y     <- X[2, ]             # Bewerber 2
> mat   <- rbind(x, y)       # Bewerber als zeilenweise Matrix

# Quadrat der Mahalanobisdistanzen zum Idealprofil
> mahalanobis(mat, ideal, S)
      x          y
11.286151 1.529668

# manuelle Kontrolle
> Sinv <- solve(S)           # Inverse der empirischen Kovarianzmatrix
> t(x-ideal) %*% Sinv %*% (x-ideal)    # quadrierte Distanz 1
      [,1]
[1,] 11.28615

> t(y-ideal) %*% Sinv %*% (y-ideal)    # quadrierte Distanz 2
      [,1]
[1,] 1.529668

# Differenzvektoren zwischen beiden Bewerbern und Idealprofil
> matDiff <- sweep(mat, 2, ideal, "-")
> diag(matDiff %*% Sinv %*% t(matDiff))  # beide quadr. Distanzen simultan
      x          y
11.286151 1.529668
```

Um den Bewerber mit der geringsten Mahalanobisdistanz zum Idealprofil unter allen Bewerbern zu identifizieren, kann auf die `min()` und `which.min()` Funktionen zurückgegriffen werden.

```
> mDist <- mahalanobis(X, ideal, S)      # quadr. Distanz alle Bewerber
> min(mDist)                            # geringste quadrierte Distanz
[1] 0.2095796

> which.min(mDist)                      # zugehöriger Bewerber
[1] 16

> X[idxMin, ]                           # sein Profil
[1] 1 1 2
```

Die Mahalanobisdistanz zweier Vektoren ist gleich deren euklidischer Distanz, nachdem beide derselben Mahalanobistransformation unterzogen wurden.

```
# Mahalanobistransformation des Idealprofils
> idealM <- t(SsqrtInv %*% (ideal - ctr))

# Quadrat der euklidischen Distanz des transformierten ersten
# Bewerbers zum transformierten Idealprofil
> crossprod(Xm[1, ] - t(idealM))
[1,]
[1,] 11.28615

> crossprod(Xm[2, ] - t(idealM))           # zweiter Bewerber
[1,]
[1,] 1.529668

# fasse transformierte Bewerber zeilenweise zu Matrix zusammen
> matM <- rbind(Xm[1, ], Xm[2, ])

# Differenzvektoren zwischen transf. Bewerbern und transf. Idealprofil
> matMdiff <- sweep(matM, 2, idealM, "-")

# simultane Berechnung der quadrierten euklidischen Distanzen
> diag(crossprod(t(matMdiff)))
[1] 11.286151 1.529668
```

2.9.4 Orthogonale Projektion

In vielen statistischen Verfahren spielt die orthogonale Projektion von Daten im durch die Beobachtungsobjekte aufgespannten Personenraum auf bestimmte Unterräume eine entscheidende Rolle. Für jeden Datenpunkt liefert sie den Punkt im betrachteten Unterraum mit dem geringsten euklidischen Abstand. Häufig wird dieses Vorgehen in univariaten Tests (etwa der Varianzanalyse, vgl. Abschn. 8.3 und ?proj) allerdings anders konzeptualisiert und ist deshalb nicht offensichtlich. In multivariaten Verfahren (vgl. Kap. 9) taucht die orthogonale Projektion dagegen auch explizit auf.

Ist A spaltenweise eine Basis des Unterraums V , so sind die Koordinaten des orthogonal auf V projizierten Vektors x bzgl. der Basis A durch $(A^t \cdot A)^{-1} \cdot A^t \cdot x$ gegeben. Die Koordinaten bzgl. der Standardbasis berechnen sich entsprechend durch $A \cdot (A^t \cdot A)^{-1} \cdot A^t \cdot x$. Die Matrix $A \cdot (A^t \cdot A)^{-1} \cdot A^t$ wird auch als orthogonale Projektion P auf V bezeichnet, und es gilt $P^2 = P$ sowie $P^t = P$. Im Fall eines eindimensionalen Unterraums V , dessen Basisvektor a bereits normiert ist, vereinfacht sich die Projektion zu $a \cdot a^t \cdot x$, die Koordinaten der Projektion bzgl. a liefert entsprechend das Skalarprodukt von a und x , also $a^t \cdot x$.

Als Beispiel sollen die im vorangehenden Abschnitt simulierten Daten im durch die Beobachtungsobjekte aufgespannten n -dimensionalen Personenraum auf den eindimensionalen Unterraum projiziert werden, dessen Basisvektor aus lauter 1-

Einträgen besteht. Dies bewirkt, dass alle Werte jeweils durch den Mittelwert der zugehörigen Variable ersetzt werden.

```
# Basisvektor eines eindimensionalen Unterraums: lauter 1-Einträge
> ones <- rep(1, nrow(X))
> P1 <- ones %*% solve(t(ones) %*% ones) %*% t(ones)           # Projektion
> Px1 <- P1 %*% X          # Koordinaten der Projektion
> Px1[1:3, ]                # Werte ersetzt durch Variablen-Mittelwerte
     [,1] [,2] [,3]
[1,] -2.58 2.58 3.66
[2,] -2.58 2.58 3.66
[3,] -2.58 2.58 3.66

> colMeans(X)             # Kontrolle: Spaltenmittel der Daten
[1] -2.58 2.58 3.66

# orthogonale Projektion auf normierten Vektor als Basis des Unterraums
> a <- ones / sqrt(crossprod(ones))      # normiere Basisvektor
> P2 <- a %*% t(a)                      # orthogonale Projektion
> all.equal(P1, P2)                      # Kontrolle: Vergleich mit erster Projektion
[1] TRUE
```

Als zweites Beispiel sollen die Daten im durch die Variablen aufgespannten dreidimensionalen Raum auf den zweidimensionalen Unterraum projiziert werden, dessen Basisvektoren die zwei ersten Vektoren der Standardbasis sind. Dies bewirkt, dass die dritte Komponente jedes Zeilenvektors der Datenmatrix auf 0 gesetzt wird.

```
# Basis eines zweidimensionalen Unterraums: erste 2 Standard-Basisvektoren
> A <- cbind(c(1, 0, 0), c(0, 1, 0))
> P3 <- A %*% solve(t(A) %*% A) %*% t(A) # orthogonale Projektion
> Px3 <- t(P3 %*% t(X))                  # Koordinaten der Projektion
> Px3[1:3, ]                                # 3. Komponente auf 0 gesetzt
     [,1] [,2] [,3]
[1,] -5   -2   0
[2,]  0    4   0
[3,] -7   1   0

> X[1:3, ]                                 # Kontrolle: Originaldaten
     [,1] [,2] [,3]
[1,] -5   -2   11
[2,]  0    4   2
[3,] -7   1   4
```

2.9.5 Kennwerte und Zerlegungen von Matrizen

Die Spur einer Matrix ist als `sum(diag(x=<Matrix>))` zu berechnen, wohingegen für die Determinante einer quadratischen Matrix die eigene Funktion `det(x=<Matrix>)` zur Verfügung steht.

```

> nn <- 4                                # Anzahl Zeilen
> qq <- 2                                # Anzahl Spalten
> X <- matrix(c(20, 26, 10, 19, 29, 27, 20, 12), nrow=nn, ncol=qq)
> Xc <- diag(nn) - (1/nn) * (rep(1, nn) %*% t(rep(1, nn))) # Zentriermatrix
> sum(diag(Xc))                           # Spur der Zentriermatrix
[1] 3

> det(Xc)                                 # Zentriermatrix ist singulär: Determinante = 0
[1] 0

> mat     <- matrix(c(1, 1, 1, -1), nrow=2)    # zu invertierende Matrix
> matInv <- solve(mat)                      # ihre Inverse

# Determinante der Inversen ist gleich dem Kehrwert der Determinante
> all.equal(det(matInv), 1/det(mat))
[1] TRUE

```

Während der Rang einer Matrix X mathematisch klar definiert ist, wirft seine numerische Berechnung für beliebige Matrizen Probleme auf, weil Zahlen im Computer nur mit endlicher Genauigkeit dargestellt werden können (vgl. Abschn. 1.3.6, Fußnote 26). Über die QR-Zerlegung mit `qr(x=Matrix)` lässt sich der Rang meist zuverlässig ermitteln, da sie numerisch wenig störanfällig ist. Die Zerlegung berechnet Matrizen Q und R so, dass $X = Q \cdot R$ gilt, wobei Q eine Orthogonalmatrix der selben Dimensionierung wie X ist ($Q^T = Q^{-1}$) und R eine obere Dreiecksmatrix. Die Ausgabe ist eine Liste, die den Rang von X in der Komponente `rank` enthält (vgl. Abschn. 3.1). Um die Matrizen Q und R zu erhalten, müssen die Hilfsfunktionen `qr.Q(QR-Liste)` bzw. `qr.R(QR-Liste)` auf die von `qr()` ausgegebene Liste angewendet werden.

```

> qrRes <- qr(Xc)                         # QR-Zerlegung
> qrRes$rank                               # Rang
[1] 3

# Rekonstruktion der Zentriermatrix aus Q und R
> qr.Q(qrRes) %*% qr.R(qrRes)
[,1]  [,2]  [,3]  [,4]
[1,]  0.75 -0.25 -0.25 -0.25
[2,] -0.25  0.75 -0.25 -0.25
[3,] -0.25 -0.25  0.75 -0.25
[4,] -0.25 -0.25 -0.25  0.75

```

Eigenwerte und -vektoren einer quadratischen Matrix berechnet `eigen(x=→(Matrix))`. Beide werden als Komponenten einer Liste ausgegeben – die Eigenwerte als Vektor in der Komponente `values`,⁴⁴ die normierten Eigenvektoren als Spalten einer Matrix in der Komponente `vectors`.

⁴⁴ Eigenwerte werden entsprechend ihrer algebraischen Multiplizität ggf. mehrfach aufgeführt. Auch Matrizen mit komplexen Eigenwerten sind zugelassen. Da in der Statistik vor allem Eigenwerte von Kovarianzmatrizen interessant sind, konzentriert sich die Darstellung hier auf den einfacheren Fall symmetrischer Matrizen.

```

> mat  <- matrix(c(1, 0, 0, 2), nrow=2)
> (eig <- eigen(mat))                      # Eigenwerte und -vektoren
$values
[1] 2 1

$vectors
[,1] [,2]
[1,]    0   -1
[2,]    1    0

# mat ist symmetrisch -> Matrix aus Eigenvektoren ist orthogonal
> eig$vectors %*% t(eig$vectors)
[,1] [,2]
[1,]    1    0
[2,]    0    1

# Spur = Summe der Eigenwerte
> all.equal(sum(diag(mat)), sum(eig$values))
[1] TRUE

# Determinante = Produkt der Eigenwerte
> all.equal(det(mat), prod(eig$values))
[1] TRUE

```

Eine symmetrische Matrix X ist mit Eigenwerten und Eigenvektoren als Spektralzerlegung $G \cdot D \cdot G^t$ darstellbar, wenn G die Matrix mit den normierten Eigenvektoren in den Spalten und D die aus den Eigenwerten in zugehöriger Reihenfolge gebildete Diagonalmatrix ist.

```

# stelle mat über Spektralzerlegung dar
> eig$vectors %*% diag(eig$values) %*% t(eig$vectors)
[,1] [,2]
[1,]    1    0
[2,]    0    2

```

Ist eine Matrix X als Spektralzerlegung darstellbar, lassen sich Matrizen finden, durch die X ebenfalls berechnet werden kann, was in verschiedenen Anwendungsfällen erstrebenswert ist. So lässt sich X dann als Quadrat einer Matrix A ($X = A^2 = A \cdot A$) oder als Produkt einer Matrix N mit deren Transponierter darstellen ($X = N \cdot N^t$). Auch die Darstellung als Summe von Matrizen B und C ist möglich ($X = B + C$). Zunächst wird dafür in allen Fällen X wie schon geschehen diagonalisiert, also als Produkt $G \cdot D \cdot G^t$ dargestellt.

```

# Darstellung von mat als Quadrat durch Ziehen der Wurzel
# aus den Diagonalelementen von D und Rücktransformation
> sqrtD <- sqrt(eig$values)
> AA    <- eig$vectors %*% diag(sqrtD) %*% t(eig$vectors)
> AA %*% AA                                # erzeugt mat ...

# Inverse von AA durch Bilden der Kehrwerte der Diagonalelemente
# von D und Rücktransformation

```

```

> AAinv <- eig$vectors %*% diag(1/sqrtD) %*% t(eig$vectors)
> AAinv %*% AA                                # Kontrolle: Einheitsmatrix
[1,] [,1] [,2]
[1,]    1    0
[2,]    0    1

# Darstellung von mat als Produkt N * N^t: Matrix der Eigenvektoren,
# deren Längen gleich der Wurzel aus den zugehörigen Eigenwerten sind
> N <- eig$vectors %*% sqrt(diag(eig$values))      # Matrix N
> N %*% t(N)                                     # erzeugt mat ...

# Darstellung von mat als Summe durch Aufteilen der Diagonalelemente
# von D in zwei separate Vektoren und Rücktransformation
> BB <- eig$vectors %*% diag(c(eig$values[1], 0)) %*% t(eig$vectors)
> CC <- eig$vectors %*% diag(c(0, eig$values[2])) %*% t(eig$vectors)
> BB + CC                                       # erzeugt mat ...

```

Die Kondition κ einer reellen Matrix X ist gleich $\sqrt{\lambda_{\max}/\lambda_{\min}}$ mit λ_{\max} als größtem und λ_{\min} als kleinstem Eigenwert ungleich Null von $X^t \cdot X$. Zur Berechnung dient die `kappa(z=<Matrix>, exact=FALSE)` Funktion, wobei für eine numerisch aufwendigere, aber auch präzisere Bestimmung von κ das Argument `exact=TRUE` gesetzt werden muss.

```

> X <- matrix(c(20, 26, 10, 19, 29, 27, 20, 12), nrow=4, ncol=2)
> kappa(X, exact=TRUE)                         # Kondition kappa
[1] 6.242934

> eigVals <- eigen(t(X) %*% X)$values       # Kontrolle über Eigenwerte
> sqrt(max(eigVals) / min(eigVals))
[1] 6.242934

```

Die Singulärwertzerlegung einer beliebigen Matrix X erhält man mit `svd(x=<Matrix>)` (Singular Value Decomposition). Auch hier erfolgt die Ausgabe als Liste, wobei die Komponente `d` der Vektor der Singulärwerte ist, `u` die in der Zerlegung linke und `v` die in der Zerlegung rechte Matrix. Insgesamt gilt damit $X = u \cdot D \cdot v$, wenn `D` die aus den Singulärwerten gebildete Diagonalmatrix ist.

```

> Xsvd <- svd(X)                             # Singulärwertzerlegung
> Xsvd$u %*% diag(Xsvd$d) %*% Xsvd$v        # rekonstruiere X
[1,] [,1] [,2]
[1,]    20    29
[2,]    26    27
[3,]    10    20
[4,]    19    12

```

Die mit `chol(x=<Matrix>)` durchgeführte Cholesky-Zerlegung einer symmetrischen, positiv-definiten Matrix X berechnet eine obere Dreiecksmatrix R so, dass $X = R^t \cdot R$ gilt.

```

# Cholesky-Zerlegung der Kovarianzmatrix von X
> R <- chol(cov(X))

```

```
> t(R) %*% R                                # rekonstruiere Kovarianzmatrix
      [,1]      [,2]
[1,] 43.58333 20.00000
[2,] 20.00000 59.33333

> cov(X)                                    # Kontrolle ...
```

2.10 Arrays

Das Konzept der Speicherung von Daten in eindimensionalen Vektoren und zweidimensionalen Matrizen lässt sich mit der Klasse `array` auf höhere Dimensionen verallgemeinern. In diesem Sinne sind Vektoren und Matrizen ein- bzw. zweidimensionale Spezialfälle von Arrays, weshalb sich Arrays in allen wesentlichen Funktionen auch wie Matrizen verhalten. So müssen ebenso wie in Vektoren und Matrizen die in einem `array` gespeicherten Daten denselben Datentyp aufweisen (vgl. Abschn. [2.8.1](#)).

```
> array(data=<Vektor>, dim=length(data), dimnames=NULL)
```

Für `data` ist ein Datenvektor mit den Werten anzugeben, die das Array speichern soll. Mit `dim` wird die Dimensionierung sowie die Anzahl der Werte pro Dimension festgelegt. Dies geschieht mit Hilfe eines Vektors, der pro Dimension ein Element beinhaltet, das die Anzahl der zugehörigen Werte spezifiziert. Das Argument `dim=c(2, 3, 4)` würde etwa festlegen, dass das Array zwei Zeilen, drei Spalten und vier Schichten umfassen soll. Ein dreidimensionales Array lässt sich nämlich als Quader vorstellen, der aus mehreren zweidimensionalen Matrizen besteht, die in Schichten hintereinander gereiht sind. Das Argument `dimnames` dient dazu, die einzelnen Stufen in jeder der Dimensionen mit Namen zu versehen. Dies geschieht unter Verwendung einer Liste (vgl. Abschn. [3.1](#)), die für jede Dimension eine Komponente in Form eines Vektors mit Bezeichnungen besitzt.

Als Beispiel soll die Kontingenztafel dreier kategorialer Variablen dienen: Geschlecht mit zwei, Gruppenzugehörigkeit bzgl. einer Variable mit drei und bzgl. einer weiteren Variable mit zwei Stufen. Ein dreidimensionales Array wird durch separate zweidimensionale Matrizen für jede Stufe der dritten Dimension ausgegeben.

```
> (myArr1 <- array(1:12, c(2, 3, 2), dimnames=list(c("f", "m"),
+                                         c("CG", "WL", "T"), c("high", "low"))))
, , high
CG WL T
f  1  3  5
m  2  4  6
, , low
CG WL T
f  7  9 11
m  8 10 12
```

Das Array wird durch die mit dem Vektor `1:12` bereitgestellten Daten in Reihenfolge der Dimensionen aufgefüllt: zunächst alle Zeilen der ersten Spalte der ersten Schicht, dann in diesem Muster alle Spalten der ersten Schicht und zuletzt in diesem Muster alle Schichten. Auf Arrays lassen sich mit `apply()` wie bei Matrizen beliebige Funktionen in Richtung der einzelnen Dimensionen anwenden.

Arrays lassen sich analog zu Matrizen mit dem `[<Index>]` Operator indizieren, wobei die Indizes für die zusätzlichen Dimensionen durch Komma getrennt hinzugefügt werden.

```
> myArr1[1, 3, 2]           # Element in 1. Zeile, 3. Spalte, 2. Schicht
[1] 11

> myArr2 <- myArr1*2
> myArr2[ , , 1]           # zeige nur 1. Schicht
   CG  WL  T
f  2   6  10
m  4   8  12
```

Ähnlich wie sich bei Matrizen durch Transponieren mit `t()` Zeilen und Spalten vertauschen lassen, können mit `aperm()` auch bei Arrays Dimensionen ausgetauscht werden.

```
> aperm(a=<Array>, perm=<Vektor>)
```

Unter `a` ist das zu transformierende Array der Dimension n anzugeben. `perm` legt in Form eines Vektors mit den Elementen $1-n$ fest, welche Dimensionen vertauscht werden sollen. Die Position einer Zahl in `perm` bezieht sich auf die alte Dimension, das Element an dieser Position bestimmt, zu welcher neuen Dimension die alte gemacht wird. Sollen in einem dreidimensionalen Array die Schichten zu Zeilen (und umgekehrt) werden, wäre `perm=c(3, 2, 1)` zu setzen. Das Vertauschen von Zeilen und Spalten wäre mit `perm=c(2, 1, 3)` zu erreichen.

2.11 Häufigkeitsauszählungen

Bei der Analyse kategorialer Variablen besteht ein häufiger Auswertungsschritt darin, die Auftretenshäufigkeiten der Kategorien auszuzählen und relative sowie bedingte relative Häufigkeiten zu berechnen. Wird nur eine Variable betrachtet, ergeben sich einfache Häufigkeitstabellen, bei mehreren Variablen mehrdimensionale Kontingenztafeln der gemeinsamen Häufigkeiten.

2.11.1 Einfache Tabellen absoluter und relativer Häufigkeiten

Eine Tabelle der absoluten Häufigkeiten von Variablenwerten erstellt `table` \rightarrow (`<Faktor>`) und erwartet dafür als Argument ein eindimensionales Objekt, das sich als Faktor interpretieren lässt – häufig ist dies einfach ein Vektor. Das Ergebnis

ist eine Übersicht über die Auftretenshäufigkeit jeder vorkommenden Ausprägung, wobei fehlende Werte ignoriert werden.⁴⁵

```
> (myLetters <- sample(LETTERS[1:5], 12, replace=TRUE))
[1] "C" "D" "A" "D" "E" "D" "C" "E" "E" "B" "E" "E"

> (tab <- table(myLetters))
myLetters
A B C D E
1 1 2 3 5
```

In der oberen Zeile sind die Ausprägungen der Variable, in der zweiten Zeile die jeweils zugehörige Auftretenshäufigkeit aufgeführt. Häufigkeitstabellen eindimensionaler Vektoren verhalten sich wie Vektoren mit benannten Elementen, wobei die Benennungen den vorhandenen Ausprägungen der Variable entsprechen. Die Ausprägungen lassen sich mit dem Befehl `names(<Tabelle>)` separat ausgeben und auch verändern.

```
> names(tab)
[1] "A" "B" "C" "D" "E"

> tab["B"]
B
1

> names(tab)[1] <- "F"; tab
F B C D E
1 1 2 3 5
```

Relative Häufigkeiten ergeben sich durch Division der absoluten Häufigkeiten mit der Gesamtzahl der Beobachtungen, im Beispiel also mit `table(myLetters)/length(myLetters)`. Für diese Rechnung existiert die Funktion `prop.table(<Tabelle>)`, welche die relativen Häufigkeiten ausgibt, wenn als Argument eine Tabelle der absoluten Häufigkeiten übergeben wird.

```
> prop.table(table(myLetters))
myLetters
      A          B          C          D          E 
0.08333333 0.08333333 0.16666667 0.25000000 0.41666667
```

Kommen mögliche Variablenwerte in einem Vektor nicht vor, so tauchen sie auch in einer mit `table()` erstellten Häufigkeitstabelle nicht als ausgezählte Kategorie auf. Um deutlich zu machen, dass Variablen außer den tatsächlich vorhandenen Ausprägungen potentiell auch weitere Werte annehmen, kann auf zweierlei Weise vorgegangen werden: so können die möglichen, tatsächlich aber nicht auftretenden Werte der Häufigkeitstabelle nachträglich mit der Häufigkeit 0 hinzugefügt werden.

⁴⁵ Ist das erste Argument von `table()` ein Vektor, können fehlende Werte über das Argument `exclude=NULL` mit in die Auszählung einbezogen werden. Damit in Faktoren vorkommende fehlende Werte unter einer eigenen Kategorie berücksichtigt werden, muss der Faktor `NA` als eigene Stufe enthalten und deshalb mit `factor(<Vektor>, exclude=NULL)` gebildet werden.

```
> tab["Q"] <- 0; tab
F B C D E Q
1 1 2 3 5 0
```

Beim Hinzufügen von Werten zur Tabelle werden diese ans Ende angefügt. Geht dadurch die natürliche Reihenfolge der Ausprägungen verloren, kann die Tabelle z. B. mittels eines durch `order(names(<Tabelle>))` erstellten Vektors der geordneten Indizes sortiert werden.

```
> tabOrder <- order(names(tab))
> tab[tabOrder]
B C D E F Q
1 2 3 5 1 0
```

Alternativ zur Veränderung der Tabelle selbst können die Daten zunächst in einen Faktor umgewandelt werden. Dem Faktor lässt sich dann der nicht auftretende, aber prinzipiell mögliche Wert als weitere Stufe hinzufügen, die auch hier als letzte in der Tabelle ausgegeben wird.

```
> letFac <- factor(myLetters, levels=LETTERS[1:5])
> levels(letFac) <- c(levels(letFac), "Q"); letFac
[1] C D A D E D C E E B E E
Levels: A B C D E Q

> table(letFac)
letFac
A B C D E Q
1 1 2 3 5 0
```

2.11.2 Häufigkeiten natürlicher Zahlen

Soll die Häufigkeit natürlicher Zahlen ermittelt werden, wobei die Auftretenshäufigkeit nicht nur der in einem Vektor tatsächlich vorkommenden Zahlen von Interesse ist, sondern jeder natürlichen Zahl bis zu einem Höchstwert, so kann dies mit `tabulate()` geschehen. Die Anwendung dieser Funktion ist vor allem für Faktoren interessant, da deren Stufen intern durch natürliche Zahlen repräsentiert sind.

```
> tabulate(bin=<Vektor>, nbins=max(1, bin))
```

Unter `bin` ist ein Vektor mit natürlichen Zahlen anzugeben, deren Häufigkeiten bestimmt werden sollen. Zahlen kleiner als 1 sowie größer als `nbins` werden dabei stillschweigend ignoriert und Dezimalzahlen tranchiert, ihre Dezimalstellen also abgeschnitten. Für `bin` können auch Faktoren übergeben werden, wobei deren numerische Repräsentation der Faktorstufen in der Auszählung Verwendung findet. `nbins` gibt die höchste auszuzählende Kategorie an, Voreinstellung ist das Maximum von `bin`. Das Ergebnis ist ein Vektor der Häufigkeiten ohne Angabe der Kategorien.

```
> tabulate(c(1, 2, 2, 3, 3, 3))
[1] 1 2 3
```

```
# hier werden auch die Häufigkeiten von 3 und 5 gezählt
> tabulate(c(1, 2, 2, 4, 4, 4, 4), nbins=5)
[1] 1 2 0 4 0
```

```
# hier werden die Werte -1, 0 und 6 ignoriert
> tabulate(c(-1, 0, 1, 2, 2, 4, 4, 4, 4, 6), nbins=5)
[1] 1 2 0 4 0
```

2.11.3 Iterationen zählen

Unter einer Iteration innerhalb einer linearen Sequenz von Symbolen ist ein Abschnitt aus der ein- oder mehrfachen Wiederholung desselben Symbols zu verstehen (engl. Run). Iterationen werden begrenzt durch Iterationen eines anderen Symbols oder besitzen kein vorangehendes bzw. auf sie folgendes Symbol. Die Iterationen eines Vektors zählt die `rle(Vektor)` Funktion, deren Ergebnis eine Liste mit zwei Komponenten ist: die erste Komponente `lengths` ist ein Vektor mit der jeweiligen Länge jeder Iteration, die zweite Komponente `values` ein Vektor mit den Symbolen, um die es sich jeweils handelt (vgl. Abschn. 3.1).

```
> (vec <- rep(rep(c("f", "m"), 3), c(1, 3, 2, 4, 1, 2)))
[1] "f" "m" "m" "m" "f" "f" "m" "m" "m" "m" "f" "m" "m" "m"
> (res <- rle(vec))
Run Length Encoding
lengths: int [1:6] 1 3 2 4 1 2
values : chr [1:6] "f" "m" "f" "m" "f" "m"
> length(res$lengths)                                # zähle Anzahl der Iterationen
[1] 6
```

Aus der Länge und dem wiederholten Symbol aller Iterationen lässt sich die ursprüngliche Sequenz wieder eindeutig rekonstruieren. Dies kann durch die `inverse.rle(rle-Ergebnis)` Funktion geschehen, die eine Liste erwartet, wie sie `rle()` als Ergebnis besitzt.

```
> inverse.rle(res)
[1] "f" "m" "m" "m" "f" "f" "m" "m" "m" "m" "f" "m" "m"
```

2.11.4 Absolute, relative und bedingte relative Häufigkeiten in Kreuztabellen

Statt die Häufigkeiten der Werte nur einer einzelnen Variable zu ermitteln, können mit `table(Faktor1, Faktor2, ...)` auch mehrdimensionale Kontingenztafeln erstellt werden. Die Elemente der Faktoren an gleicher Position werden als denselben Beobachtungsobjekten zugehörig interpretiert. Das erste Element von

`<Faktor1>` bezieht sich also auf dieselbe Beobachtung wie das erste Element von `<Faktor2>`, das zweite Element von `<Faktor1>` auf dieselbe Beobachtung wie das zweite Element von `<Faktor2>`, usw. Das Ergebnis ist eine Kreuztabelle mit den gemeinsamen absoluten Häufigkeiten der Merkmale, wobei die Ausprägungen des ersten Faktors in den Zeilen stehen.

Als Beispiel sollen 10 Personen betrachtet werden, die nach ihrem Geschlecht und dem Ort ihres Arbeitsplatzes unterschieden werden.

```
> (work <- factor(sample(c("home", "office"), 10, replace=TRUE)))
[1] home office office home home office office office office
Levels: home office

> (sex <- factor(sample(c("f", "m"), 10, replace=TRUE)))
[1] f m f m m f m f m
Levels: f m

> table(sex, work)
      work
sex   home office
  f     1     3
  m     2     4
```

Um relative Häufigkeiten zu ermitteln, eignet sich auch in Kreuztabellen die `prop.table()` Funktion. Für bedingte relative Häufigkeiten kann sie mit `sweep()` kombiniert werden: sind auf die Zeilen bedingte relative Häufigkeiten zu berechnen, muss jede Zeile durch die zugehörige Zeilensumme dividiert werden, für auf die Spalten bedingte relative Häufigkeiten analog jede Spalte durch die zugehörige Spaltensumme.

```
> (relFreq <- prop.table(table(sex, work)))           # relative Häufigkeiten
      work
sex   home office
  f    0.1    0.3
  m    0.2    0.4

> rSums <- rowSums(relFreq)                         # Zeilensummen
> cSums <- colSums(relFreq)                         # Spaltensummen
> sweep(relFreq, 1, rSums, "/")          # auf Zeilen bedingte rel. Häufigkeiten
      work
sex   home office
  f  0.2500000 0.7500000
  m  0.3333333 0.6666667

> sweep(relFreq, 2, cSums, "/")          # auf Spalten bedingte rel. Häufigkeiten
      work
sex   home office
  f  0.3333333 0.4285714
  m  0.6666667 0.5714286
```

Um Häufigkeitsauszählungen für mehr als zwei Variablen zu berechnen, können beim Aufruf von `table()` einfach weitere Faktoren durch Komma getrennt hinzugefügt werden:

gefügt werden. Die Ausgabe verhält sich dann wie ein Objekt der Klasse `array`. Hierbei werden etwa im Fall von drei Variablen so viele zweidimensionale Kreuztabellen ausgegeben, wie Stufen der dritten Variable vorhanden sind. Soll dagegen auch in diesem Fall eine einzelne Kreuztabelle mit verschachteltem Aufbau erzeugt werden, ist die `ftable()` Funktion (Flat Table) zu nutzen.

```
> ftable(x, row.vars=NULL, col.vars=NULL)
```

Unter `x` kann entweder eine bereits mit `table()` erzeugte Kreuztabelle oder aber eine durch Komma getrennte Reihe von Faktoren eingetragen werden. Die Argumente `row.vars` und `col.vars` kontrollieren, welche Variablen in den Zeilen und welche in den Spalten angeordnet werden. Beide Argumente akzeptieren numerische Vektoren mit den Nummern der entsprechenden Variablen oder aber Vektoren aus Zeichenketten, die den Namen der Faktoren entsprechen.

```
> (group <- factor(sample(c("A", "B"), 10, replace=TRUE)))
[1] B B A B A B A B A A
Levels: A B
```

```
> ftable(work, sex, group, row.vars="work", col.vars=c("sex", "group"))
      sex   f     m
      group A   B   A   B
work
home        0   1   1   1
office       1   2   3   1
```

Beim Erstellen von Kreuztabellen kann auch auf die Funktion `xtabs()` zurückgegriffen werden, insbesondere wenn sich die Variablen in Datensätzen befinden (vgl. Abschn. 3.2).

```
> xtabs(formula= ~ ., data=(Datensatz))
```

Im ersten Argument `formula` erwartet `xtabs()` eine sog. Formel (vgl. Abschn. 5.1). Hier ist dies eine besondere Art der Aufzählung der in der Kontingenztafel zu berücksichtigenden Faktoren, die durch ein + verknüpft hinter der Tilde `~` aufgeführt werden. In der Voreinstellung `~ .` werden alle Faktoren des unter `data` angegebenen Datensatzes einbezogen, d. h. alle möglichen Kombinationen von Faktorstufen gebildet. Stammen die in der Formel genannten Faktoren aus einem Datensatz, muss dieser unter `data` aufgeführt werden. Die Ausprägungen des erstgenannten Faktors bilden die Zeilen der ausgegebenen Kontingenztafel. Bei mehr als zwei betrachteten Variablen werden ohne Benutzung von `ftable()` mehrere zweidimensionale Kreuztabellen ausgegeben – bei drei Variablen etwa für jede Stufe der letztgenannten Variable eine eigene.

```
> (persons <- data.frame(sex, work)) # Faktoren in Datensatz kombinieren
      sex   work
1   f   home
2   m office
3   f office
4   m   home
5   m   home
```

```

6   f  office
7   m  office
8   f  office
9   m  office
10  m  office

> xtabs(~ sex + work, data=persons)
      work
sex home office
  f     1      3
  m     2      4

```

Einen Überblick über die Zahl der in einer Häufigkeitstabelle ausgewerteten Faktoren sowie die Anzahl der zugrundeliegenden Beobachtungen erhält man mit `summary(<Tabelle>)`. Im Fall von Kreuztabellen wird hierbei zusätzlich ein χ^2 -Test auf Unabhängigkeit berechnet (vgl. Abschn. 6.2.1).

```

> summary(table(sex, work))
Number of cases in table: 10
Number of factors: 2
Test for independence of all factors:
Chisq = 4.444, df = 1, p-value = 0.03501
Chi-squared approximation may be incorrect

```

2.11.5 Randkennwerte von Kreuztabellen

Um Randsummen, Randmittelwerte oder ähnliche Kennwerte für eine Kreuztabelle zu berechnen, können alle für Matrizen vorgestellten Funktionen verwendet werden, insbesondere `apply()`, aber etwa auch `rowSums()` und `colSums()` sowie `rowMeans()` und `colMeans()`. Hier nimmt die Tabelle die Rolle der Matrix ein.

```

> apply(xtabs(data=persons), MARGIN=1, FUN=sum)          # Zeilensummen
f  m
4 6

> colMeans(xtabs(data=persons))                          # Spaltenmittel
home office
1.5    3.5

```

`addmargins()` berechnet beliebige Randkennwerte für eine Kreuztabelle A entsprechend der mit dem Argument `FUN` bezeichneten Funktion. Die Funktion operiert separat über jeder der mit dem Vektor `margin` bezeichneten Dimensionen. Die Ergebnisse der Anwendung von `FUN` werden A in der Ausgabe als weitere Zeile und Spalte hinzugefügt.

```

> addmargins(A=<Tabelle>, margin=<Vektor>, FUN=<Funktion>)

> addmargins(xtabs(data=persons), c(1, 2), mean)        # Randmittel
      work

```

```

sex  home  office  mean
f    1.0    3.0    2.0
m    2.0    4.0    3.0
mean 1.5    3.5    2.5

```

2.11.6 Kumulierte relative Häufigkeiten und Prozentrang

Die `ecdf(x=⟨Vektor⟩)` Funktion (Empirical Cumulative Distribution Function) ermittelt die kumulierten relativen Häufigkeiten eines empirischen Datenvektors. Diese geben für einen Wert an, welcher Anteil der Daten nicht größer als dieser ist. Das Ergebnis ist analog zur Verteilungsfunktion quantitativer Zufallsvariablen.

Das Ergebnis von `ecdf()` ist eine Stufenfunktion mit so vielen Sprungstellen, wie es unterschiedliche Werte im Vektor `x` gibt, also `length(unique(x))`. Die Höhe jedes Sprungs entspricht der relativen Häufigkeit des Wertes an der Sprungstelle. Enthält `x` also keine mehrfach vorkommenden Werte, erzeugt `ecdf()` eine Stufenfunktion mit so vielen Sprungstellen, wie `x` Elemente besitzt. Dabei weist jeder Sprung dieselbe Höhe auf – die relative Häufigkeit $1/\text{length}(x)$ jedes Elements von `x`. Tauchen in `x` Werte mehrfach auf, unterscheiden sich die Sprunghöhen dagegen entsprechend den relativen Häufigkeiten.

Der Output von `ecdf()` ist seinerseits eine Funktion, die zunächst einem eigenen Objekt zugewiesen werden muss, ehe sie zur Ermittlung kumulierter relativer Häufigkeiten für bestimmte Werte Verwendung finden kann. Ist `myStep()` diese Stufenfunktion, und möchte man die kumulierten relativen Häufigkeiten der in `x` gespeicherten Werte erhalten, ist `x` selbst als Argument für den Aufruf von `myStep()` einzusetzen. Andere Werte als Argument von `myStep()` sind aber ebenfalls möglich. Auf diese Weise lassen sich empirische, aber auch interpolierte Prozentränge von beliebigen Daten ermitteln, indem `myStep()` für Werte ausgewertet wird, die nicht Element von `x` sind. Mit `ecdf()` erstellte Funktionen lassen sich über `plot()` direkt graphisch abbilden (Abb. 6.1 und 10.21, vgl. Abschn. 10.6.6)

```

> (vec <- round(rnorm(10), 2))
[1] -1.57 2.21 -1.01 0.21 -0.29 -0.61 -0.17 1.90 0.17 0.55

> (myStep <- ecdf(vec))
Empirical CDF
Call: ecdf(vec)
x[1:10] = -1.57, -1.01, -0.61, ..., 1.9, 2.21

> myStep(vec)
[1] 0.1 1.0 0.2 0.7 0.4 0.3 0.5 0.9 0.6 0.8

```

Soll die Ausgabe der kumulierten relativen Häufigkeiten in der richtigen Reihenfolge erfolgen, müssen die Werte mit `sort()` geordnet werden.

```

> myStep(sort(vec))
[1] 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0

```

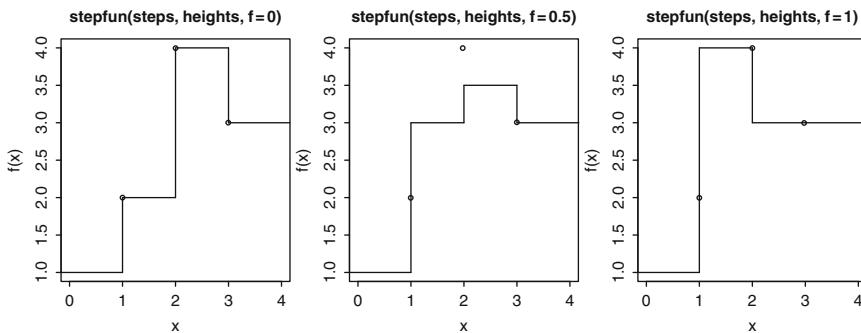


Abb. 2.1 Stufenfunktionen derselben Daten bei Variation des `stepfun()` Parameters f

Die Sprungstellen der von `ecdf()` erstellten Funktion lassen sich mit der `knots()` Funktion extrahieren. Das Ergebnis sind gerade die in `x` enthaltenen unterschiedlichen sortierten Werte.

```
> knots(Fn=(Sprungfunktion))
> knots(myStep)
[1] -1.57 -1.01 -0.61 -0.29 -0.17 0.17 0.21 0.55 1.90 2.21
```

Stufenfunktionen können auch allgemein selbst definiert werden, indem mit Hilfe der `stepfun()` Funktion sowohl die Sprungstellen als auch die Höhe der einzelnen Stufen angegeben werden. Da es bei N Stufen $N - 1$ Sprungstellen zur nächsten Stufe gibt, müssen `stepfun()` zwei Vektoren unterschiedlicher Länge übergeben werden.

```
> stepfun(x=(Sprungstellen), y=(Plateauhöhen), f=(Zahl))
```

Unter `x` werden die Sprungstellen als Vektor angegeben. Für `y` wird ein Vektor eingesetzt, der ein Element mehr als `x` enthält. Seine Werte legen die Höhe der Stufen fest, wobei die erste Stufe jene vor der ersten Sprungstelle und die letzte Stufe jene ab der letzten Sprungstelle ist. Für `f` können Zahlen im Bereich von 0–1 eingesetzt werden. `f` bestimmt die von der resultierenden Funktion vorgenommene lineare Interpolation der Höhe für solche Werte, die sich zwischen zwei Sprungstellen befinden. Mit `f=0` wechselt der Funktionswert an der Sprungstelle n auf die unter `y[n+1]` angegebene Höhe und behält sie für alle Werte bis zur nächsten Sprungstelle bei – der Funktionswert für die jeweils folgende Sprungstelle wird also nicht eher berücksichtigt, als die Sprungstelle erreicht ist. Dies entspricht dem Verhalten von durch `ecdf()` erzeugten Funktionen. Bei `f=1` wechselt der Funktionswert an der Sprungstelle n auf die unter `y[n+2]` angegebene Höhe. Der Funktionswert der jeweils folgenden Sprungstelle bestimmt also vollständig die Höhe der Funktion für Werte, die zwischen dieser und der vorangehenden Sprungstelle liegen. Werte zwischen 0 und 1 kontrollieren entsprechend den linearen Anteil, zu dem der Funktionswert für die jeweils folgende Sprungstelle in der Höhe der Funktion berücksichtigt wird.

Die Sprungstellen der von `stepfun()` erzeugten Funktion lassen sich ebenfalls mit `knots()` extrahieren. Genauso lässt sich die Funktion mit `plot(<Stufenfunktion>)` graphisch anzeigen (Abb. 2.1).

```
> steps    <- 1:3                                # Sprungstellen
> heights <- c(1, 2, 4, 3)                      # Plateauhöhen
> (sFun0  <- stepfun(steps, heights, f=0))
Step function
Call: stepfun(1:3, vec, f = 0)
x[1:3] = 1, 2, 3
4 plateau levels = 1, 2, 4, 3

# Stufenfunktionen derselben Daten unter Variation des Parameters f
> sFun25 <- stepfun(steps, heights, f=0.25)
> sFun50 <- stepfun(steps, heights, f=0.50)
> sFun75 <- stepfun(steps, heights, f=0.75)
> sFun1  <- stepfun(steps, heights, f=1)
> x      <- seq(from=0.5, to=3.5, by=0.5)

# Plateauhöhen der erzeugten Stufenfunktionen
> cbind(x, f0=sFun0(x), f25=sFun25(x), f50=sFun50(x), f75=sFun75(x),
+         f1=sFun1(x))
   x  f0  f25  f50  f75  f1
[1,] 0.5  1  1.00  1.0  1.00  1
[2,] 1.0  2  2.00  2.0  2.00  2
[3,] 1.5  2  2.50  3.0  3.50  4
[4,] 2.0  4  4.00  4.0  4.00  4
[5,] 2.5  4  3.75  3.5  3.25  3
[6,] 3.0  3  3.00  3.0  3.00  3
[7,] 3.5  3  3.00  3.0  3.00  3

# 3 Stufenfunktionen graphisch darstellen
> plot(sFun0); plot(sFun50); plot(sFun1)
```

2.12 Codierung, Identifikation und Behandlung fehlender Werte

Empirische Datensätze besitzen häufig zunächst keine zufriedenstellende Qualität, etwa durch Fehler in der Eingabe von Werten, Ausreißer oder durch unvollständige Daten, wenn also nicht für alle Beobachtungsobjekte Werte von allen erhobenen Variablen vorliegen. So können VPn die Auskunft bzgl. bestimmter Fragen verweigern, Aufgaben übersehen oder in adaptiven Tests aufgrund von Verzweigungen nicht vorgelegt bekommen.

Fehlende Werte (Englisch: Missing Values oder Missing Data) bergen aus versuchsplanerischer Perspektive die Gefahr, dass sie womöglich nicht zufällig, sondern systematisch entstanden sind und so zu verzerrten Ergebnissen führen. Aber auch für die statistische Auswertung bringen sie Probleme mit sich: zum einen sind verschiedene Strategien des Umgangs mit ihnen denkbar, die nicht unbedingt zu gleichen Ergebnissen führen. Zum anderen können fehlende Werte bewirken, dass

nicht wie beabsichtigt in allen Experimentalbedingungen dieselbe Anzahl von Beobachtungen vorliegt. Dies jedoch ist für die Anwendung und Interpretation vieler üblicher Verfahren relevant, deren Einsatz durch das Vorliegen fehlender Werte problematisch werden könnte.⁴⁶

Für die Behandlung fehlender Werte in statistischen Tests vgl. Abschn. 5.3. Um fehlende Werte in Indexvektoren zu vermeiden, wo sie meist zu nicht intendierten Konsequenzen führen (vgl. Abschn. 2.2.2), sollten logische Indexvektoren mit `which()` in numerische konvertiert werden.

2.12.1 Fehlende Werte codieren und ihr Vorhandensein prüfen

Wenn ein Datensatz eingegeben wird und fehlende Werte vorliegen, so dürfen diese nicht einfach weggelassen, sondern müssen unabhängig vom Datentyp mit der Konstante NA (Not Available) codiert werden – auch bei character Vektoren ist sie nicht in Anführungszeichen zu setzen. Auf diese Weise kann die wichtige Anzahl der Beobachtungsobjekte richtig gezählt werden, auch wenn weniger eigentliche Werte vorliegen.

```
> (vec1 <- c(10, 20, NA, 40, 50, NA))
[1] 10 20 NA 40 50 NA
```

```
> length(vec1)
[1] 6
```

Ob in einem Vektor fehlende Werte vorhanden sind, wird mit der Funktion `is.na(x=Vektor)` ermittelt.⁴⁷ Als Output liefert sie einen logischen Vektor, der für jede Position angibt, ob das Element ein fehlender Wert ist. Im Fall einer Datenmatrix liefert `is.na()` eine Matrix aus Wahrheitswerten, die für jedes Matrixelement angibt, ob es sich um einen fehlenden Wert handelt.

```
> is.na(vec1)
FALSE FALSE TRUE FALSE FALSE TRUE

> vec2 <- c(NA, 7, 9, 10, 1, 8)
> (matNA <- rbind(vec1, vec2))
 [,1] [,2] [,3] [,4] [,5] [,6]
vec1    10    20    NA    40    50    NA
vec2     NA      7      9     10      1      8
```

⁴⁶ Neben den hier beschriebenen Methoden zur Behandlung fehlender Werte existieren auch andere Versuche, mit diesem Problem umzugehen. Das als Multiple Imputation bezeichnete Verfahren ersetzt fehlende Werte dabei durch solche Zahlen, die unter Berücksichtigung bestimmter Rahmenbedingungen generiert wurden und dabei Eigenschaften der tatsächlich vorhandenen Daten berücksichtigen sollen. Multiple Imputation wird in R u. a. durch die Pakete `Hmisc`, `Amelia II` (Honaker et al., 2009) und `mice` (van Buuren und Groothuis-Oudshoorn, 2010) unterstützt.

⁴⁷ Der `==` Operator eignet sich nicht zur Prüfung auf fehlende Werte, da das Ergebnis von `<Wert> == NA` selbst NA ist (vgl. Abschn. 2.12.3).

```
> is.na(matNA)
     [,1]  [,2]  [,3]  [,4]  [,5]  [,6]
vec1 FALSE FALSE  TRUE FALSE FALSE  TRUE
vec2  TRUE FALSE FALSE FALSE FALSE FALSE
```

Bei einem großen Datensatz ist es mühselig, die Ausgabe von `is.na()` manuell nach TRUE Werten zu durchsuchen. Daher bietet sich zunächst `any()` an, um zu erfahren, ob überhaupt fehlende Werte vorliegen und `sum()`, um deren Anzahl zu ermitteln. `which()` fragt nach der Position der TRUE Werte, hier also nach der Position der fehlenden Werte.

```
> any(is.na(vec1))                      # gibt es fehlende Werte?
[1] TRUE

> sum(is.na(vec1))                     # wie viele?
[1] 2

> which(is.na(vec1))                  # an welcher Position im Vektor?
[1] 3 6
```

2.12.2 Fehlende Werte ersetzen oder umcodieren

Fehlende Werte werden bei der Dateneingabe in anderen Programmen oft mit Zahlen codiert, die keine mögliche Ausprägung einer Variable sind, z. B. mit 999. Bisweilen ist diese Codierung auch nicht einheitlich, sondern verwendet verschiedene Zahlen, etwa wenn Daten aus unterschiedlichen Quellen zusammengeführt werden. Bei der Verarbeitung von aus anderen Programmen übernommenen Datensätzen in R muss die Codierung fehlender Werte also ggf. angepasst werden (vgl. Abschn. 4.2).

Die Identifikation der zu ersetzenen Werte kann über mehrere mit ODER verknüpfte Kriterien zur Erstellung eines logischen Indexvektors erfolgen. Dieser logische Vektor kann alternativ auch mit `(Vektor) %in% (Menge)` erstellt werden. Mit Hilfe des Indexvektors lassen sich dann die Werte über eine Zuweisung auf NA setzen. Das Vorgehen bei Matrizen ist analog.

```
# fehlende Werte sind zunächst mit 99 und 999 codiert
> vecOrg <- c(30, 25, 23, 21, 99, 999)          # Vektor mit fehlenden Werten
> vecNA  <- vecOrg                            # Kopie des Originalvektors
> (idx   <- (vecOrg == 99) | (vecOrg == 999))    # finde Missings
[1] FALSE FALSE FALSE FALSE TRUE TRUE

> idx       <- vecOrg %in% c(99, 999)          # alternative Möglichkeit
> vecNA[idx] <- NA; vecNA                      # ersetze durch NA
[1] 30 25 23 21 NA NA

> (matOrg <- matrix(vecOrg, 2, 3))            # Matrix mit fehlenden Werten
     [,1]  [,2]  [,3]
[1,]    30    23    99
[2,]    25    21   999
```

```
> matNA      <- matOrg          # Kopie der Originalmatrix
> idx        <- (matOrg == 99) | (matOrg == 999)    # finde Missings
> matNA[idx] <- NA; matNA      # ersetze durch NA
[1,]  [2,]  [3]
[1,] 30   23   NA
[2,] 25   21   NA
```

2.12.3 Behandlung fehlender Werte bei der Berechnung einfacher Kennwerte

Wenn in einem Vektor oder einer Matrix fehlende Werte vorhanden sind, muss Funktionen zur Berechnung statistischer Kennwerte über ein Argument angegeben werden, wie mit ihnen zu verfahren ist. Andernfalls kann der Kennwert nicht berechnet werden, und der Output der Funktion ist seinerseits NA.⁴⁸ Allerdings können NA Einträge zunächst manuell entfernt werden, ehe die Daten an eine Funktion übergeben werden. Zu diesem Zweck existiert auch die `na.omit(object=⟨Vektor⟩)` Funktion, die `object` um fehlende Werte bereinigt ausgibt.

```
> goodIdx <- !is.na(vecNA)      # Indizes der nicht fehlenden Werte
> mean(vecNA[goodIdx])         # um NA bereinigten Vektor übergeben
[1] 24.33333

> sd(na.omit(vecNA))          # alternative Möglichkeit
[1] 5.125102
```

Sind fehlende Werte Teil der an eine Funktion übergebenen Daten, lässt sich deren Behandlung über ein Argument steuern – für viele Funktionen lautet dies `na.rm`. In der Voreinstellung `FALSE` sorgt es dafür, dass fehlende Werte nicht stillschweigend bei der Berechnung des Kennwertes ausgelassen werden, sondern das Ergebnis NA ist. Soll der Kennwert dagegen auf Basis der vorhandenen Werte berechnet werden, muss das Argument `na.rm=TRUE` gesetzt werden.

```
> sum(vecNA)
[1] NA

> sum(vecNA, na.rm=TRUE)
[1] 146

> apply(matNA, 1, mean)
[1] NA NA

> apply(matNA, 1, mean, na.rm=TRUE)
[1] 23.33333 25.33333
```

⁴⁸ Allgemein ist das Ergebnis aller Rechnungen NA, sofern der fehlende Wert für das Ergebnis relevant ist. Ist das Ergebnis auch ohne den fehlenden Wert eindeutig bestimmt, wird es ausgegeben – so erzeugt `TRUE | NA` die Ausgabe `TRUE`, da sich bei einem logischen ODER das zweite Argument nicht auf das Ergebnis auswirkt, wenn das erste WAHR ist.

Auf die dargestellte Weise lassen sich fehlende Werte u. a. in den Funktionen `sum()`, `prod()`, `range()`, `mean()`, `median()`, `quantile()`, `var()`, `sd()`, `cov()` und `cor()` behandeln.

2.12.4 Behandlung fehlender Werte in Matrizen

Bei den Funktionen `cov(<Vektor1>, <Vektor2>)` und `cor(<Vektor1>, <Vektor2>)` bewirkt das Argument `na.rm=TRUE`, dass ein aus zugehörigen Werten von `<Vektor1>` und `<Vektor2>` gebildetes Wertepaar nicht in die Berechnung von Kovarianz oder Korrelation eingeht, wenn wenigstens einer der beiden Werte NA ist.

Zur Behandlung fehlender Werte stehen bei `cov()` und `cor()` außer dem Argument `na.rm=TRUE` weitere Möglichkeiten zur Verfügung, die aber erst bei der Erstellung von Kovarianz- bzw. Korrelationsmatrizen auf Basis von Matrizen mit mehr als zwei Spalten relevant sind. Mit `use="complete.obs"` werden fehlende Werte fallweise (zeilenweise) ausgeschlossen, der paarweise Fallausschluss erfolgt mit `use="pairwise.complete.obs"`.

2.12.4.1 Zeilenweiser (fallweiser) Fallausschluss

Beim zeilenweisen Fallausschluss werden die Zeilen einer Matrix, in denen NA Werte auftauchen, komplett entfernt, ehe die Matrix für Berechnungen herangezogen wird. Weil eine Zeile oft allen an einem Beobachtungsobjekt erhobenen Daten entspricht, wird dies auch als fallweiser Fallausschluss bezeichnet. Dazu bietet sich zunächst die Möglichkeit, alle Zeilen mit fehlenden Werten anhand von `apply(is.na(<Matrix>), 1, any)` selbst festzustellen.⁴⁹ Mit dem resultierenden logischen Indexvektor lassen sich die fraglichen Zeilen von weiteren Berechnungen ausschließen.

```
> ageNA <- c(18, NA, 27, 22)
> DV1 <- c(NA, 1, 5, -3)
> DV2 <- c(9, 4, 2, 7)
> (matNA <- cbind(ageNA, DV1, DV2))
   ageNA DV1 DV2
[1,]    18   NA    9
[2,]    NA    1    4
[3,]    27    5    2
[4,]    22   -3    7

> (rowNAidx <- apply(is.na(matNA), 1, any))      # Zeilen mit NA feststellen
[1] TRUE TRUE FALSE FALSE

> matNA[!rowNAidx, ]                                # Zeilen mit NA entfernen
   ageNA DV1 DV2
```

⁴⁹ Bei numerischen Matrizen rechnerisch effizient auch mit `is.na(<Matrix>%*% rep(1, ncol(<Matrix>)))`. Dies liefert einen Spaltenvektor zurück.

```
[1,]    27    5    2
[2,]    22   -3    7
```

Mit `na.omit(Matrix)` lässt sich eine Matrix mit fehlenden Werten aber auch in einem Schritt so verändern, dass Zeilen mit fehlenden Werten vollständig entfernt werden. Die Indizes der dabei ausgeschlossenen Zeilen werden als Attribute mit aufgeführt.

Mit der so gebildeten Matrix fließen im Beispiel die Zeilen 1 und 2 nicht mit in Berechnungen ein.

```
> na.omit(matNA)
  ageNA DV1 DV2
[1,]    27    5    2
[2,]    22   -3    7

attr("na.action")
[1] 2 1

attr("class")
[1] "omit"

> colMeans(na.omit(matNA))
ageNA DV1 DV2
24.5 1.0 4.5
```

Bei den Funktionen `cov()` und `cor()` bewirkt bei der Berechnung von Kovarianz- und Korrelationsmatrizen mit mehr als zwei Variablen das Argument `use="complete.obs"` den fallweisen Ausschluss fehlender Werte. Seine Verwendung hat denselben Effekt wie die vorherige Reduktion der Matrix um Zeilen, in denen fehlende Werte auftauchen.

```
> cov(matNA, use="complete.obs")
  age DV1 DV2
age  12.5 20 -12.5
DV1  20.0 32 -20.0
DV2 -12.5 -20  12.5

# beide Arten des fallweisen Ausschlusses erzielen dasselbe Ergebnis
> all(cov(matNA, use="complete.obs") == cov(na.omit(matNA)))
[1] TRUE
```

2.12.4.2 Paarweiser Fallausschluss

Beim paarweisen Fallausschluss werden die Werte einer auch NA beinhaltenden Zeile soweit als möglich in Berechnungen berücksichtigt, die Zeile also nicht vollständig ausgeschlossen. Welche Werte einer Zeile Verwendung finden, hängt von der konkreten Auswertung ab. Der paarweise Fallausschluss wird im Fall der Berechnung der Summe oder des Mittelwertes über Zeilen oder Spalten mit dem Argument `na.rm=TRUE` realisiert, das alle Werte außer NA einfließen lässt.

```
> rowMeans(matNA)
[1] NA NA 11.333333 8.666667

> rowMeans(mat, na.rm=TRUE)
[1] 13.500000 2.500000 11.333333 8.666667
```

Bei der Berechnung von Kovarianz- und Korrelationsmatrizen für mehr als zwei Variablen mit `cov()` und `cor()` bewirkt das Argument `use="pairwise.complete.obs"` den paarweisen Ausschluss fehlender Werte. Es wird dann bei der Berechnung jeder Kovarianz geprüft, ob pro Zeile in den zugehörigen beiden Spalten ein gültiges Wertepaar existiert und dieses ggf. verwendet. Anders als beim fallweisen Ausschluss geschieht dies also auch dann, wenn in derselben Zeile Werte anderer Variablen fehlen, die für die zu berechnende Kovarianz aber irrelevant sind.

Während im Beispiel also beim fallweisen Ausschluss das von VP 1 gelieferte Wertepaar nicht in die Berechnung der Kovarianz von `ageNA` und `DV2` einfließt (weil der Wert für `DV1` bei dieser VP fehlt), werden diese Werte beim paarweisen Ausschluss berücksichtigt. Lediglich bei der Berechnung der Kovarianz von `DV1` und `DV2` werden keine Daten der ersten VP verwendet, weil ein Wert für `DV1` von ihr fehlt.

```
> cov(matNA, use="pairwise.complete.obs")
            ageNA   DV1       DV2
ageNA  20.33333  20 -16.000000
DV1    20.00000  16 -10.000000
DV2   -16.00000 -10  9.666667
```

Ob fehlende Werte fall- oder paarweise ausgeschlossen werden sollen, hängt u. a. von den Ursachen ab, warum manche Untersuchungseinheiten unvollständige Daten geliefert haben und andere nicht. Insbesondere stellt sich die Frage, ob Untersuchungseinheiten mit fehlenden Werten systematisch andere Eigenschaften haben, so dass von ihren Daten generell kein Gebrauch gemacht werden sollte.

2.12.5 Behandlung fehlender Werte beim Sortieren von Daten

Beim Sortieren von Daten mit `sort()` und `order()` wird die Behandlung fehlender Werte mit dem Argument `na.last` kontrolliert, das auf `NA`, `TRUE` oder `FALSE` gesetzt werden kann. Bei `sort()` ist `na.last` per Voreinstellung auf `NA` gesetzt und sorgt dafür, dass fehlende Werte entfernt werden. Bei `order()` ist die Voreinstellung `TRUE`, wodurch fehlende Werte ans Ende plaziert werden. Auf `FALSE` gesetzt bewirkt `na.last` die Plazierung fehlender Werte am Anfang.

2.13 Zeichenketten verarbeiten

Obwohl Zeichenketten bei der numerischen Auswertung von Daten oft eine Nebenrolle spielen und zuvorderst in Form von Bezeichnungen für Variablen oder

Gruppen in Erscheinung treten, ist es bisweilen hilfreich, sie flexibel erstellen, manipulieren und ausgeben zu können.

2.13.1 Objekte in Zeichenketten umwandeln

Mit der `toString(x=<Objekt>)` Funktion lassen sich Ergebnisse beliebiger Berechnungen in Zeichenketten umwandeln. Als Argument `x` wird ein Objekt erwartet – typischerweise die Ausgabe einer Funktion. Das Ergebnis ist eine einzelne Zeichenkette, die als Inhalt die normalerweise auf der Konsole erscheinende Ausgabe von `<Objekt>` hat. Dabei werden einzelne Elemente der Ausgabe innerhalb der Zeichenkette durch Komma mit folgendem Leerzeichen getrennt. Komplexere Objekte (z. B. Matrizen) werden so konvertiert, dass ihre Ausgabe in Form eines Vektors erfolgt.

```
> randVals <- round(rnorm(5), 2)
> toString(randVals)
[1] "-0.03, 1.01, -0.52, -1.03, 0.18"
```

Die `formatC()` Funktion ist spezialisiert auf die Umwandlung von Zahlen in Zeichenketten und bietet sich vor allem für die formatierte Ausgabe von Dezimalzahlen an.

```
> formatC(x=<Zahl>, digits=<Dezimalstellen>, width=<Breite>,
+           format=<Zahlentyp>)"")
```

Ist `x` eine Dezimalzahl, wird sie mit `digits` vielen Stellen ausgegeben. Die Angabe von `digits` fügt ganzen Zahlen keine Dezimalstellen hinzu, allerdings verbreitert sich die ausgegebene Zeichenkette auf `digits` viele Zeichen, indem `x` entsprechend viele Leerzeichen vorangestellt werden. Die Länge der Zeichenkette lässt sich auch unabhängig von der Zahl der Dezimalstellen mit dem Argument `width` kontrollieren. Schließlich ermöglicht `format` die Angabe, was für ein Zahlentyp bei `x` vorliegt, insbesondere ob es eine ganze Zahl ("d") oder eine Dezimalzahl ist. Im letzten Fall kann die Ausgabeform etwa mit "f" wie gewohnt erfolgen (z. B. "1.234") oder mit "e" in wissenschaftlicher Notation (z. B. "1.23e+03") – für weitere Möglichkeiten vgl. `?formatC`.

```
> formatC(c(1, 2.345), width=5, format="f")
[1] "1.0000" "2.3450"
```

2.13.2 Zeichenketten erstellen und ausgeben

Die einfachste Möglichkeit zum Erstellen eigener Zeichenketten ist ihre manuelle Eingabe auf der Konsole oder im Editor. Diese Methode stößt jedoch dort schnell an ihre Grenzen, wo Zeichenketten von Berechnungen abhängen sollen oder viele Zeichenketten nach demselben Muster erzeugt werden müssen. Die Funktionen `paste()` und `sprintf()` sind hier geeignete Alternativen.

Mit der `paste()` Funktion lassen sich Zeichenketten mit einem bestimmten Aufbau erzeugen, indem verschiedene Komponenten aneinandergehängt werden, die etwa aus einem gemeinsamen Präfix und unterschiedlicher laufender Nummer bestehen können.

```
> paste(<Objekt1>, <Objekt2>, ..., sep = " ")
```

Die ersten Argumente von `paste()` sind Objekte, deren Elemente jeweils die Bestandteile der zu erstellenden Zeichenketten ausmachen und zu diesem Zweck aneinandergesetzt werden. Das erste Element des ersten Objekts wird dazu mit dem ersten Element des zweiten Objekts verbunden, ebenso die jeweils zweiten und folgenden Elemente. Das Argument `sep` kontrolliert, welche Zeichen jeweils zwischen Elementen aufeinander folgender Objekte einzufügen sind – in der Voreinstellung ist dies das Leerzeichen. Das Ergebnis ist ein Vektor aus Zeichenketten, wobei jedes seiner Elemente aus der Kombination jeweils eines Elements aus jedem übergebenen Objekt besteht. Hierbei werden unterschiedlich lange Vektoren ggf. zyklisch verlängert (vgl. Abschn. 2.1.7.1).

```
> paste("group", LETTERS[1:5], sep = "_")
[1] "group_A" "group_B" "group_C" "group_D" "group_E"
```

```
# Farben der Default-Farbpalette
> paste(1:5, palette()[1:5], sep = ": ")
[1] "1: black" "2: red" "3: green3" "4: blue" "5: cyan"
```

Die an die gleichnamige Funktion der Programmiersprache C angelehnte Funktion `sprintf()` erzeugt Zeichenketten, deren Aufbau durch zwei Komponenten bestimmt wird: einerseits durch einen die Formatierung und feste Elemente definierenden Teil (den sog. Format-String), andererseits durch eine Reihe von Objekten, deren Werte an festgelegten Stellen des Format-Strings einzufügen sind.

```
> sprintf(fmt = "<Format-String>", <Objekt1>, <Objekt2>, ...)
```

Das Argument `fmt` erwartet eine Zeichenkette aus festen und variablen Elementen. Gewöhnliche Zeichen werden als feste Elemente interpretiert und tauchen unverändert in der erzeugten Zeichenkette auf. Variable Elemente werden durch das Prozentzeichen `%` eingeleitet, auf das ein Buchstabe folgen muss, der die Art des hier einzufügenden Wertes definiert. So gibt etwa `%d` an, dass hier ein ganzzahliger Wert einzufügen ist, `%f` dagegen weist auf eine Dezimalzahl hin. Das Prozentzeichen selbst wird durch `%%` ausgegeben, doppelte Anführungszeichen durch `\"`,⁵⁰ Tabulatoren durch `\t`.

Für jedes durch ein Prozentzeichen definierte Feld muss nach `fmt` ein passendes Objekt genannt werden, dessen Wert an der durch `%` bezeichneten Stelle eingefügt wird. Die Entsprechung zwischen variablen Feldern und Objekten wird über deren

⁵⁰ Alternativ kann `fmt` in einfache Anführungszeichen '`(Format-String)`' gesetzt werden, innerhalb derer sich dann auch doppelte Anführungszeichen ohne voranstehendes `\` Symbol befinden können.

Reihenfolge hergestellt, das erste Feld wird also mit dem Wert des ersten Objekts gefüllt, etc.

```
> nSubj <- 20
> gName <- "A"
> mVal <- 14.2
> sprintf("For the %d participants in group %s, the mean was %f",
+         nSubj, gName, mVal)
[1] "For the 20 participants in group A, the mean was 14.200000"
```

Format-Strings erlauben eine weitergehende Formatierung der Ausgabe, indem zwischen dem % und dem folgenden Buchstaben Angaben gemacht werden, die sich z. B. auf die Anzahl der auszugebenden Dezimalstellen beziehen kann. Für detaillierte Informationen vgl. ?sprintf.

```
> sprintf("%.3f", 1.23456)      # begrenze Ausgabe auf 3 Dezimalstellen
[1] "1.234"
```

Um mehrere Zeichenketten kombiniert als eine einzige Zeichenkette auszugeben, kann die cat() (Concatenate) Funktion verwendet werden, die auch eine gewisse Formatierung erlaubt – etwa in Form von Zeilenumbrüchen durch die sog. Escape-Sequenz \n oder \t für Tabulatoren (vgl. ?Quotes).

```
> cat("<Zeichenkette1>", "<Zeichenkette2>", ..., sep=" ")
```

cat() kombiniert die übergebenen Zeichenketten durch Verkettung zunächst zu einer einzelnen, wobei zwischen den Zeichenketten das unter sep genannte Trennzeichen eingefügt wird. Numerische Variablen werden hierbei automatisch in Zeichenketten konvertiert. Anders als in der Ausgabe von Werten durch print() erscheint in der Ausgabe von cat() nicht automatisch eine Anzeige zu Beginn jeder Zeile, die über die laufende Nummer des zu Zeilenbeginn stehenden Wertes informiert, etwa [1].

```
> cVar <- "A string"
> cat(cVar, "with\n", 4, "\nwords\n", sep="+")
A string+with
+4+
words
```

2.13.3 Zeichenketten manipulieren

Die Funktionen tolower(x="*Zeichenkette*") und toupper(x="*Zeichenkette*") konvertieren die für x übergebenen (Vektoren von) Zeichenketten in Klein- bzw. Großbuchstaben.

```
> tolower(c("A", "BC", "DEF"))
[1] "a" "bc" "def"

> toupper(c("ghi", "jk", "i"))
[1] "GHI" "JK" "I"
```

Aus Zeichenketten lassen sich mit substring() konsekutive Teilstücke von Zeichen extrahieren.

```
> substring(text="{Zeichenkette}", first=<Beginn>, last=<Ende>)
```

Aus den für `text` übergebenen Zeichenketten wird jeweils jene Zeichenfolge extrahiert, die beim Buchstaben an der Stelle `first` beginnt und mit dem Buchstaben an der Stelle `last` endet. Sollte eine Zeichenkette weniger als `first` oder `last` Buchstaben umfassen, werden nur so viele ausgeben, wie tatsächlich vorhanden sind – ggf. eine leere Zeichenkette.

```
> substring(c("ABCDEF", "GHIJK", "LMNO", "PQR"), 4, 5)
[1] "DE" "JK" "O" ""
```

Soll die durch `first` und `last` bezeichnete Zeichenfolge in den Elementen von `text` ersetzt werden, ist dem Ergebnis von `substring()` eine Zeichenkette zuzuweisen. Dabei ist es notwendig, dass für `text` ein bereits bestehendes Objekt übergeben wird, das dann der Änderung unterliegt.

```
> charVec <- c("ABCDEF", "GHIJK", "LMNO", "PQR")
> substring(charVec, 4, 5) <- c(..", "xx", "++", "***"); charVec
[1] "ABC..F" "GHIxx" "LMN+" "PQR"
```

Mit der `strsplit()` Funktion (String Split) ist es möglich, eine einzelne Zeichenkette in mehrere Teile zu zerlegen.

```
> strsplit(x="{Zeichenkette}", split="{Zeichenkette}", fixed=FALSE)
```

Die für `x` übergebene Zeichenkette wird dafür nach Vorkommen der unter `split` genannten Zeichenkette durchsucht, die als Trennzeichen interpretiert wird. Die Zeichenfolgen links und rechts von `split` machen also die Komponenten der Ausgabe aus, die aus einer Liste von Vektoren von Zeichenketten besteht – eine Komponente für jedes Element des Vektors von Zeichenketten `x`.⁵¹ Die `strsplit()` Funktion ist damit die Umkehrung von `paste()`. Das Argument `fixed` bestimmt, ob `split` im Sinne eines sog. Regulären Ausdrucks interpretiert werden soll (Voreinstellung `FALSE`, vgl. Abschn. 2.13.4) oder als exakt die übergebene Zeichenfolge selbst (`TRUE`).

```
> strsplit(c("abc_def_ghi", "jkl_mno"), "_")
[[1]]
[1] "abc" "def" "ghi"

[[2]]
[1] "jkl" "mno"
```

Mit der in der Hilfeseite von `strsplit()` definierten `strReverse("`**{Zeichenkette}**`")` Funktion (String Reverse, vgl. Abschn. 11.1) wird die Reihenfolge der Zeichen innerhalb einer Zeichenkette umgekehrt. Dies ist deswegen nicht mit `rev(<Vektor>)` möglich, weil eine Zeichenkette ein einzelnes Element eines Vektors darstellt.

⁵¹ Die Ausgabe ist ggf. mit `unlist()` in einen Vektor umzuwandeln, vgl. Abschn. 3.1, insbesondere 3.1.2.

```
# Definition der strReverse() Funktion
> strReverse <- function(x) {
+   sapply(lapply(strsplit(x, NULL), rev), paste, collapse="")
}

> strReverse(c("Lorem", "ipsum", "dolor", "sit"))
[1] "meroL" "muspi" "rolod" "tis"
```

2.13.4 Zeichenfolgen finden

Die Suche nach bestimmten Zeichenfolgen innerhalb von Zeichenketten ist mit den Funktionen `match()`, `pmatch()` und `grep()` möglich. Soll geprüft werden, ob die in einem Vektor `x` enthaltenen Elemente jeweils eine exakte Übereinstimmung in den Elementen eines Vektors `table` besitzen, ist `match()` anzuwenden. Beide Objekte müssen nicht unbedingt Zeichenketten sein, werden aber intern zu solchen konvertiert.

```
> match(x=<gesuchte Werte>, table=<Objekt>)
```

Die Ausgabe gibt für jedes Element von `x` die erste Position im Objekt `table` an, an der es dort ebenfalls vorhanden ist. Enthält `table` kein mit `x` übereinstimmendes Element, ist die Ausgabe an dieser Stelle NA.

Die fast identische `pmatch()` Funktion unterscheidet sich darin, dass die Elemente von `table` nicht nur auf exakte Übereinstimmung getestet werden: findet sich für ein Element von `x` ein identisches Element in `table`, ist der zugehörige Index das Ergebnis. Andernfalls wird in `table` nach teilweisen Übereinstimmungen in dem Sinne gesucht, dass auch eine Zeichenkette zu einem Treffer führt, wenn sie mit jener aus `x` beginnt, sofern es nur eine einzige solche Zeichenkette in `table` gibt.

```
> match(c("abc", "de", "f", "h"), c("abcde", "abc", "de", "fg", "ih"))
[1] 2 3 NA NA
```

```
> pmatch(c("abc", "de", "f", "h"), c("abcde", "abc", "de", "fg", "ih"))
[1] 2 3 4 NA
```

Die `grep()` Funktion ähnelt dem gleichlautenden POSIX-Befehl Unix-artiger Betriebssysteme und bietet stark erweiterte Suchmöglichkeiten.

```
> grep(pattern=<Suchmuster>, x=<Zeichenkette>)
```

Unter `pattern` ist ein Muster anzugeben, das die zu suchende Zeichenfolge definiert. Obwohl hier auch einfach eine bestimmte Zeichenfolge übergeben werden kann, liegt die Besonderheit darin, dass `pattern` sog. Reguläre Ausdrücke akzeptiert. Mit Regulären Ausdrücken lassen sich auch Muster von Zeichenfolgen charakterisieren wie „ein A gefolgt von einem B oder C und einem Leerzeichen“: `"A [BC] [[:blank:]]"` (vgl. ?`regex` und Friedl, 2006). Die zu durchsuchende Zeichenkette bzw. der Vektor von Zeichenketten wird unter `x` genannt.

Die Ausgabe besteht in einem Vektor von Indizes derjenigen Elemente von `x`, die das gesuchte Muster enthalten. Alternativ gibt die ansonsten genauso zu verwendende Funktion `grep1()` einen logischen Indexvektor aus, der für jedes Element von `x` angibt, ob es `pattern` enthält.

```
> grep("A[BC] [[:blank:]]", c("AB ", "AB", "AC ", "A "))
[1] 1 3
```

```
> grep1("A[BC] [[:blank:]]", c("AB ", "AB", "AC ", "A "))
[1] TRUE FALSE TRUE FALSE
```

Da die Syntax Regulärer Ausdrücke recht komplex ist, stellt den nicht mit der Materie vertrauten Anwender u. U. die Suche schon nach einfachen Mustern vor Schwierigkeiten. Eine Vereinfachung bietet die Funktion `glob2rx()`, mit der Muster von Zeichenfolgen mit Hilfe gebräuchlicherer Platzhalter (sog. Wildcards bzw. Globbing-Muster) beschrieben und in einen Regulären Ausdruck umgewandelt werden können. So steht z. B. der Platzhalter `?` für ein beliebiges einzelnes Zeichen, `*` für eine beliebige Zeichenkette.

```
> glob2rx(pattern = "(Muster mit Platzhaltern)")
```

Das Argument `pattern` akzeptiert eine Zeichenfolge aus Buchstaben und Platzaltern, die Ausgabe besteht in einem Regulären Ausdruck, wie er z. B. in der `grep()` Funktion angewendet werden kann.

```
> glob2rx("asdf*.txt")      # Namen, die mit asdf beginnen und .txt enden
[1] "^asdf.*\\.txt$"
```

2.13.5 Zeichenfolgen ersetzen

Wenn in Zeichenketten nach bestimmten Zeichenfolgen gesucht wird, dann häufig, um sie durch andere zu ersetzen. Neben der Möglichkeit, mit `substring()` identifizierte Zeichenfolgen durch Zuweisung zu ändern, dienen die `sub()` (Substitute) und `gsub()` Funktionen diesem Zweck.

```
> sub(pattern = "(Suchmuster)", replacement = "(Ersatz)",
+      x = "(Zeichenkette)")
```

Für `pattern` kann ein Regulärer Ausdruck übergeben werden, dessen Vorkommen in der Zeichenkette `x` durch die unter `replacement` genannte Zeichenfolge ersetzt werden. Wenn `pattern` in einem Element von `x` mehrfach vorkommt, wird es nur beim ersten Auftreten ersetzt.

```
> sub("em", "XX", "Lorem ipsum dolor sit Lorem ipsum")
[1] "LorXX ipsum dolor sit LorXX ipsum"
```

Im Unterschied zu `sub()` ersetzt die ansonsten gleich zu verwendende `gsub()` Funktion `pattern` nicht nur beim ersten Auftreten in `x` durch `replacement`, sondern überall.

```
> gsub("em", "XX", "Lorem ipsum dolor sit Lorem ipsum")
[1] "LorXX ipsum dolor sit LorXX ipsum"
```

2.13.6 Zeichenketten als Befehl ausführen

Durch die Kombination der Funktionen `parse()` und `eval()` lassen sich Zeichenketten als Befehle interpretieren und wie über die Konsole übergebene Befehle ausführen. Dieses Zusammenspiel ermöglicht es, in Abhängigkeit von vorherigen Auswertungen einen nachfolgend benötigten Befehl zunächst als Zeichenkette zu erstellen und dann auszuführen.

```
> parse(file="(Pfad und Dateiname)", text="(Zeichenkette)")
```

Hierfür ist zunächst mit `parse()` eine für das Argument `text` zu übergebende Zeichenkette in ein weiter interpretierbares Objekt umzuwandeln.⁵² Ist `text` ein Vektor von Zeichenketten, wird jedes Element als ein Befehl verstanden. Alternativ kann mit `file` eine Datei oder sonstige Quelle genannt werden, die eine solche Zeichenkette enthält (vgl. Abschn. 4.2.3).

```
> obj1 <- parse(text="3 + 4")
> obj2 <- parse(text=c("vec <- c(1, 2, 3)", "vec^2"))
```

Das Ausführen eines mit `parse()` erstellten Objekts geschieht mit `eval(expr= ~>(Objekt))`.

```
> eval(obj1)
[1] 7

> eval(obj2)
[1] 1 4 9
```

2.14 Datum und Uhrzeit

Insbesondere bei der Analyse von Zeitreihen ist es sinnvoll, Zeit- und Datumsangaben in einer Form zu speichern, die es erlaubt, solche Werte in natürlicher Art für Berechnungen zu nutzen – etwa um über die Differenz zweier Uhrzeiten die zwischen ihnen verstrichene Zeit ebenso zu ermitteln wie die zwischen zwei Datumsangaben liegende Anzahl von Tagen. R bietet solche Möglichkeiten mit eigens eingerichteten Klassen.⁵³

⁵² Solcherart erstellte Objekte können mit `deparse(expr=Objekt)` wieder in Zeichenketten umgewandelt werden.

⁵³ Als Startpunkt für die Auswertung von Zeitreihen vgl. den Abschnitt `TimeSeries` der Task Views, (R Development Core Team, 2009a). Für eine einführende Behandlung der vielen für Zeitangaben existierenden Subtilitäten vgl. Ripley und Hornik (2001) sowie `?DateTimeClasses`. Das Paket `timeDate` (Würtz und Chalabi, 2009) enthält viele weiterführende Funktionen zur Behandlung von Zeitangaben.

2.14.1 Datum

Objekte der Klasse `Date` codieren ein Datum mittels der seit einem Stichtag (meist der 1. Januar 1970) verstrichenen Anzahl von Tagen und können Tag, Monat und Jahr eines Zeitpunkts ausgeben. Das aktuelle Datum nennt `Sys.Date()`; um selbst ein Datum zu erstellen, dient `as.Date()`.

```
> as.Date(x=<Datumsangabe>, format=<Format-String>)
```

Die Datumsangabe für `x` ist eine Zeichenkette, die ein Datum in einem Format nennt, das unter `format` als sog. Format-String zu spezifizieren ist. In einer solchen Zeichenkette stehen `%<Buchstabe>` Kombinationen als Platzhalter für den einzusetzenden Teil einer Datumsangabe, sonstige Zeichen i. d. R. für sich selbst. Voreinstellung ist `"%Y-%m-%d"`, wobei `%Y` für die vierstellige Jahreszahl, `%m` für die zweistellige Zahl des Monats und `%d` für die zweistellige Zahl der Tage steht.⁵⁴ In diesem Format erfolgt auch die Ausgabe, die sich jedoch mit `format(x=<Date-Objekt>, format=<Format-String>)` kontrollieren lässt.

```
> Sys.Date()
[1] "2009-02-09"
> (myDate <- as.Date("01.11.1974", format="%d.%m.%Y"))
[1] "1974-11-01"

> format(myDate, format="%d.%m.%Y")
[1] "01.11.1974"
```

`Date` Objekte verhalten sich in arithmetischen Kontexten in der eingangs skizzierten natürlichen Weise: eine ihnen hinzugefügte Zahl wird als Anzahl von Tagen interpretiert; das Ergebnis ist ein Datum, das entsprechend viele Tage vom `Date` Objekt abweicht. Die Differenz zweier `Date` Objekte besitzt die Klasse `difftime` und wird als Anzahl der Tage ausgegeben, die vom zweiten zum ersten Datum vergehen. Hierbei ergeben sich negative Zahlen, wenn das erste Datum zeitlich vor dem zweiten liegt.⁵⁵

```
> myDate + 365
[1] "1975-11-01"

> (diffDate <- as.Date("1976-06-19") - myDate)
Time difference of 596 days
```

Ebenso wie Zahlen lassen sich auch `difftime` Objekte zu `Date` Objekten addieren.

```
> myDate + diffDate
[1] "1976-06-19"
```

⁵⁴ Vergleiche Abschn. 2.13.2 und `?strptime` für weitere mögliche Elemente des Format-Strings.

⁵⁵ Die Zeiteinheit der im `difftime` Objekt gespeicherten Werte (etwa Tage oder Minuten), hängt davon ab, aus welchen Datumsangaben das Objekt entstanden ist. Alternativ bestimmt das Argument `units` von `difftime(<Datum1>, <Datum2>, units)`, um welche Einheit es sich handeln soll.

2.14.2 Uhrzeit

Objekte der Klasse `POSIXct` (Calendar Time) repräsentieren neben dem Datum gleichzeitig die Uhrzeit eines Zeitpunkts als Anzahl der Sekunden, die seit einem Stichtag (meist der 1. Januar 1970) verstrichen ist, besitzen also eine Genauigkeit von einer Sekunde. Objekte der Klasse `POSIXlt` (Local Time) speichern dieselbe Information in Form einer Liste mit benannten Komponenten: dies sind numerische Vektoren u. a. für die Sekunden (`sec`), Minuten (`min`) und Stunden (`hour`) der Uhrzeit sowie für Tag (`mday`), Monat (`mon`) und Jahr des Datums. Zeichenketten lassen sich analog zu `as.Date()` mit `as.POSIXct()` bzw. mit `as.POSIXlt()` in entsprechende Objekte konvertieren, `strptime()` erzeugt ebenfalls ein `POSIXlt` Objekt.

```
> as.POSIXct(x = "Datum und Uhrzeit", format = "(Format-String)")
> as.POSIXlt(x = "Datum und Uhrzeit", format = "(Format-String)")
> strptime(x = "Datum und Uhrzeit", format = "(Format-String)")
```

Voreinstellung für den Format-String bei `as.POSIXlt()` und bei `as.POSIXct()` ist `%Y-%m-%d %H %M %S`, wobei `%H` für die zweistellige Zahl der Stunden im 24 h-Format, `%M` für die zweistellige Zahl der Minuten und `%S` für die zweistellige Zahl der Sekunden des Datums stehen (vgl. Fußnote 54).

```
> (myTime <- as.POSIXct("2009-02-07 09:23:02"))
[1] "2009-02-07 09:23:02 CET"

> charDates <- c("05.08.1972, 03:37", "31.03.1981, 12:44")
> (lDates <- strptime(charDates, format = "%d.%m.%Y, %H:%M"))
[1] "1972-08-05 03:37:00" "1981-03-31 12:44:00"

> lDates$mday
[1] 5 31
```

Ebenso wie Objekte der Klasse `Date` lassen sich `POSIXlt` und `POSIXct` Objekte in bestimmten arithmetischen Kontexten verwenden: ihnen hinzu addierte Zahlen werden als Sekunden interpretiert. Aus der Differenz zweier `POSIXlt` oder zweier `POSIXct` Objekte entsteht ein Objekt der Klasse `difftime`. Die Addition von `difftime` und `POSIXlt` oder `POSIXct` Objekten ist ebenfalls definiert.

```
> lDates + 120                                # 2 Minuten später
[1] "1972-08-05 03:39:00 CET" "1981-03-31 12:46:00 CEST"

> (diff21 <- lDates[2] - lDates[1])
Time difference of 3160.338 days

> lDates[1] + diff21
[1] "1981-03-31 12:44:00 CEST"
```


Kapitel 3

Datensätze

Vektoren, Matrizen und Arrays unterliegen der Beschränkung, gleichzeitig nur Werte desselben Datentyps aufnehmen zu können. Da in empirischen Erhebungssituationen meist Daten unterschiedlichen Typs – etwa numerische Variablen, Faktoren und Zeichenketten – anfallen, sind sie deshalb nicht unmittelbar geeignet, Datensätze in Gänze zu speichern. Objekte der Klasse `list` und `data.frame` sind in dieser Hinsicht flexibler. So eignen sich Listen zur Repräsentation heterogener Sammlungen von Daten und werden deshalb von vielen Funktionen genutzt, die ihren Output in Form einer Liste zurückgeben. Listen sind darüber hinaus die allgemeine Grundform von Datensätzen (Klasse `data.frame`), der gewöhnlich am besten geeigneten Struktur für empirische Daten.

3.1 Listen

Listen werden mit dem Befehl `list(<Komponente1>, <Komponente2>, ...)` erzeugt. Komponenten einer Liste können Objekte jeglicher Klasse, also auch selbst wieder Listen sein, die wiederum Werte beliebigen Datentyps beinhalten können. Die erste Komponente könnte also z. B. ein numerischer Vektor, die zweite ein Vektor von Zeichenketten und die dritte eine Matrix aus Wahrheitswerten sein. Als Länge einer Liste gilt die Anzahl ihrer Komponenten auf oberster Ebene. Die Anzahl der Komponenten einer Liste gibt `length(<Liste>)` aus.

3.1.1 Komponenten auswählen und verändern

Um auf Komponenten einer Liste zuzugreifen, kann der `[[<Index>]]` Operator benutzt werden, der als Argument die Position der zu extrahierenden Komponente in der Liste benötigt. Ist z. B. die zweite Komponente einer Liste `myList1` ein numerischer Vektor `c(12, 8, 29, 5)`, so könnte dieser Vektor so ausgelesen werden:

```
> myList1 <- list(c(1, 3), c(12, 8, 29, 5)) # Liste erstellen  
> length(myList1)  
[1] 2
```

```
> myList1[[2]]
[1] 12 8 29 5
```

Einzelne Werte einer aus mehreren Werten bestehenden Komponente können auch direkt abgefragt werden, etwa das dritte Element des obigen Vektors. Dazu wird der für Vektoren genutzte `[[Index]]` Operator an den Listenindex `[[[Index]]]` angehängt, weil die Auswertung des Befehls `myList1[[2]]` zuerst erfolgt und den im zweiten Schritt zu indizierenden Vektor zurückliefert:

```
> myList1[[2]][3]
[1] 29
```

Stellt man sich vor, dass der die zweite Komponente von `myList1` darstellende Vektor eine Variable mit Daten unterschiedlicher VPn repräsentiert, weicht die Reihenfolge des Aufrufs beim `[[Index]]` Operator damit von jener bei Matrizen ab: bei Matrizen ist die Reihenfolge `[[Zeile], [Spalte]]` und entspricht in der üblichen Anordnung von Variablen als Spalten `[[VpNr], [Variable]]`. Bei Listen entspricht die Reihenfolge dagegen `[[Variable]] [[VpNr]]`. Bei Listen und Datensätzen ist deshalb darauf zu achten, dass die Reihenfolge der Indizes zur beabsichtigten Auswahl von Einzelementen führt. Oft existieren bei Datensätzen jedoch auch andere, weniger fehlerträchtige Möglichkeiten zur Indizierung (vgl. Abschn. 3.2.2).

Beispiel sei eine Liste aus drei Komponenten. Die erste soll ein Vektor sein, die zweite eine aus je zwei Zeilen und Spalten bestehende Matrix, die dritte ein Vektor aus Zeichenketten.

```
> (myList2 <- list(c(1:4), matrix(1:4, 2, 2), c("Lorem", "ipsum")))
[[1]]
[1] 1 2 3 4

[[2]]
[,1] [,2]
[1,]    1    3
[2,]    2    4

[[3]]
[1] "Lorem" "ipsum"
```

Das Element in der ersten Zeile und zweiten Spalte der Matrix, die ihrerseits die zweite Komponente der Liste darstellt, wäre dann so aufzurufen:

```
> myList2[[2]][1, 2]
[1] 3
```

Die Schreibweise kann auch zu `[[c(IndexKmp), IndexLi]]` kombiniert werden, dabei bezeichnen `IdxKmp` den Index der Komponente und `IdxLi` den Index des Elements der Komponente. Nach dem zweiten Element der dritten Komponente, also des aus Zeichenketten bestehenden Vektors, kann daher so gefragt werden:¹

¹ Eine Matrix kann in dieser Syntax nur wie ein Vektor indiziert werden, d. h. mittels eines einzelnen Index.

```
> myList2[[c(3, 2)]]
[1] "ipsum"
```

Auch bei Listen kann der `[<Index>]` Operator verwendet werden. Im Unterschied zu `[[<Index>]]` gibt er allerdings nicht die Komponente selbst zurück, sondern eine Liste, die wiederum als einzige Komponente das gewünschte Objekt besitzt – also gewissermaßen eine Teilliste ist. Während `myList2[[2]]` also eine Matrix als Output liefert, ist das Ergebnis von `myList2[2]` eine Liste, deren einzige Komponente die Matrix ist.

```
> myList2[[2]]
[,1] [,2]
[1,]    1    3
[2,]    2    4

> class(myList2[[2]])
[1] "matrix"

> myList2[2]
[[1]]
[,1] [,2]
[1,]    1    3
[2,]    2    4

> class(myList2[2])
[1] "list"
```

Wie auch bei Vektoren können die Komponenten einer Liste benannt sein und über ihren Namen ausgewählt werden.

```
> (myList3 <- list(c(1:5), c(8:4), word="dolor"))
[[1]]
[1] 1 2 3 4 5

[[2]]
[1] 8 7 6 5 4

$word
[1] "dolor"

> myList3[["word"]]
[1] "dolor"
```

Wenn man auf die benannte Komponente zugreifen will, kann dies also mit numerischen Indizes, mit Hilfe von `[[<Variablenname>]]`, aber auch über den `<Liste>$<Variablenname>` Operator geschehen.² Dieser bietet den Vorteil, dass er ohne Klammern und numerische Indizes auskommt und damit recht übersichtlich ist. Nur wenn der Name Leerzeichen enthält, wie es bisweilen bei von R zurückgegebenen Objekten der Fall ist, muss er zudem in Anführungszeichen gesetzt

² Obwohl es nicht empfehlenswert ist, reicht es beim \$ Operator bereits aus, den unvollständigen Anfang von `<Variablenname>` zu nennen, sofern dieser bereits eindeutig ist.

werden. Welche Namen die Komponenten tragen, erfährt man mit der `names(<Liste>)` Funktion.

```
> myList3$word
[1] "dolor"

> mat      <- cbind(1:10, sample(-10:10, 10, replace=FALSE))
> retList <- cov.wt(mat, method="ML")      # unkorrigierte Kovarianzmatrix
> names(retList)                          # Komponenten der Liste
[1] "cov" "center" "n.obs"

> retList$cov                           # Kovarianzmatrix selbst
[,1] [,2]
[1,] 8.25  6.20
[2,] 6.20  28.44

> retList$center                         # Spaltenmittel
[1] 5.5 3.4

> retList$n.obs                          # Anzahl Beobachtungen
[1] 10
```

3.1.2 Listen mit mehreren Ebenen

Da Komponenten einer Liste Objekte verschiedener Klassen und auch selbst Listen sein können, ergibt sich die Möglichkeit, Listen zur Repräsentation hierarchisch organisierter Daten unterschiedlicher Art zu verwenden. Derartige Objekte können auch als Baum mit mehreren Verästelungsebenen betrachtet werden. Soll aus einem solchen Objekt ein bestimmter Wert extrahiert werden, muss man sich zunächst mit `str(<Liste>)` einen Überblick über die Organisation der Liste verschaffen und sich ggf. sukzessive von Ebene zu Ebene zum gewünschten Element vorarbeiten.

Im Beispiel soll aus einer Liste mit letztlich drei Ebenen ein Wert aus einer Matrix extrahiert werden, die sich in der dritten Verschachtelungsebene befindet.

```
# Elemente der dritten Verschachtelungsebene
> myListAA <- list(AAA=c(1, 2), AAB=c("AAB1", "AAB2", "AAB3"))
> myMatAB <- matrix(1:8, nrow=2)
> myListBA <- list(BAA=matrix(rnorm(10), ncol=2), BAB=c("BAB1", "BAB2"))
> myVecBB <- sample(1:10, 5)
# Elemente der zweiten Verschachtelungsebene
> myListA <- list(AA=myListAA, AB=myMatAB)
> myListB <- list(BA=myListBA, BB=myVecBB)

# Elemente der ersten Verschachtelungsebene
> myList4 <- list(A=myListA, B=myListB)
> str(myList4)                         # Gesamtstruktur
List of 2
$ A:List of 2
..$ AA:List of 2
```

```

... .$. AAA: num [1:2] 1 2
... .$. AAB: chr [1:3] "AAB1" "AAB2" "AAB3"
..$. AB: int [1:2, 1:4] 1 2 3 4 5 6 7 8

$ B:List of 2
..$. BA:List of 2
... .$. BAA: num [1:5, 1:2] -0.618 -1.059 -1.150 0.919 -1.146 ...
... .$. BAB: chr [1:2] "BAB1" "BAB2"
..$. BB: int [1:5] 3 5 7 2 9

> str(myList4$B)                      # Struktur der 2. Verschachtelungsebene
List of 2
$ BA:List of 2
..$. BAA: num [1:5, 1:2] -0.618 -1.059 -1.150 0.919 -1.146 ...
..$. BAB: chr [1:2] "BAB1" "BAB2"
$ BB: int [1:5] 3 5 7 2 9

> str(myList4$B$BA)                  # Struktur der 3. Verschachtelungsebene
List of 2
$ BAA: num [1:5, 1:2] -0.618 -1.059 -1.150 0.919 -1.146 ...
$ BAB: chr [1:2] "BAB1" "BAB2"

> myList4$B$BA$BAA[4, 2]           # Element der 3. Verschachtelungsebene
[1] 0.1446770

```

Die hierarchische Struktur einer Liste kann mit dem Befehl `unlist(<Liste>)` aufgelöst werden. Der Effekt besteht darin, dass alle Komponenten (in der Voreinstellung `recursive=TRUE` rekursiv, d. h. einschließlich aller Ebenen) in denselben Datentyp umgewandelt und seriell in einem Vektor zusammengefügt werden. Als Datentyp wird jener gewählt, der alle Einzelwerte ohne Informationsverlust speichern kann (vgl. Abschn. 1.3.5).

```

> myList5 <- list(c(1, 2, 3), c("A", "B"), matrix(5:12, 2))
> unlist(myList5)
[1] "1" "2" "3" "A" "B" "5" "6" "7" "8" "9" "10" "11" "12"

```

3.2 Datensätze

Ein Datensatz ist eine spezielle Liste und besitzt die Klasse `data.frame`. Ein Datensatz erbt damit die Grundeigenschaften einer Liste, besitzt aber bestimmte einschränkende Merkmale – so müssen seine Komponenten alle dieselbe Länge besitzen. Die in einem Datensatz zusammengefassten Objekte können von unterschiedlicher Klasse sein und Werte unterschiedlichen Datentyps beinhalten. Dies entspricht der empirischen Situation, dass Werte verschiedener Variablen an derselben Menge von Beobachtungsobjekten erhoben wurden. Anders gesagt enthält jede einzelne Variable Werte einer festen Menge von Beobachtungsobjekten, die auch die Werte für die übrigen Variablen liefert haben. Werte auf unterschiedlichen Variablen lassen sich somit einander hinsichtlich des Beobachtungsobjekts, von dem sie stammen, zuordnen. Da Datensätze normale Objekte sind, ist es im Gegensatz

zu anderen Statistikprogrammen möglich, mit mehreren von ihnen gleichzeitig zu arbeiten.

Die Basisinstallation von R beinhaltet bereits viele vorbereitete Datensätze, an denen sich statistische Auswertungsverfahren erproben lassen, vgl. `data()` für eine Liste. Weitere Datensätze werden durch Zusatzpakete bereitgestellt – hervorzuheben ist etwa das Paket DAAG (Maindonald und Braun, 2009) – und lassen sich mit `data(<Datensatz>, package=<Paketname>)` laden. Die Dokumentation eines Datensatzes gibt `help(<Datensatz>)` aus.

Objekte der Klasse `data.frame` sind die bevorzugte Organisationsweise für empirische Datensätze.³ Listenkomponenten spielen dabei die Rolle von Variablen und ähneln damit den Spalten einer Datenmatrix. Werden Datensätze in R ausgegeben, stehen die Variablen als Komponenten in den Spalten, während jede Zeile i. d. R. für ein Beobachtungsobjekt steht. Datensätze werden mit der Funktion `data.frame(<Objekt1>, <Objekt2>, ...)` aus mehreren einzelnen Objekten erzeugt, die typischerweise Vektoren oder Faktoren sind. Matrizen werden dabei wie separate, durch die Spalten der Matrix gebildete Vektoren behandelt.⁴

Als Beispiel seien 12 VPn betrachtet, die zufällig auf drei Untersuchungsgruppen (Kontrollgruppe CG, Wartelisten-Gruppe WL, Treatment-Gruppe T) verteilt werden. Als Abhängige Variablen (AVn) werden demographische Daten, Ratings und der IQ-Wert simuliert. Zudem soll die fortlaufende Nummer jeder VP gespeichert werden.⁵

```
> nSubj    <- 12
> sex      <- sample(c("f", "m"), nSubj, replace=TRUE)
> group    <- sample(rep(c("CG", "WL", "T"), 4), nSubj, replace=FALSE)
> age      <- sample(18:35, nSubj, replace=TRUE)
> IQ        <- round(rnorm(nSubj, mean=100, sd=15))
> rating   <- round(runif(nSubj, min=0, max=6))
> (myDf1 <- data.frame(id=1:nSubj, sex, group, age, IQ, rating))
   id sex group age  IQ rating
1  1   f     T  26 112      1
2  2   m    CG  30 122      3
3  3   m    CG  25  95      5
4  4   m     T  34 102      5
5  5   m    WL  22  82      2
6  6   f    CG  24 113      0
7  7   m     T  28  92      3
8  8   m    WL  35  90      2
9  9   m    WL  23  88      3
10 10   m    WL  29  81      5
```

³ Außer bei sehr großen Datensätzen, die sich effizienter als Matrix verarbeiten lassen.

⁴ Gleiches gilt für Listen – hier werden die Komponenten als separate Vektoren gewertet. Soll dieses Verhalten verhindert werden, um eine Liste als eine einzelne Variable des Datensatzes zu erhalten, muss sie in `I()` eingeschlossen werden: `data.frame(I(<Liste>), <Objekt2>, ...)`.

⁵ Für die automatisierte Simulation von Datensätzen nach vorgegebenen Kriterien, etwa hinsichtlich der UV-Effekte, vgl. die `sim.<Typ>()` Funktionen des psych Pakets.

```
11 11   m    CG   20   92      1
12 12   f     T   21   98      1
```

Die Anzahl von Beobachtungen (Zeilen) und Variablen (Spalten) kann wie bei Matrizen mit den Funktionen `dim(<Datensatz>)`, `nrow(<Datensatz>)` und `ncol(<Datensatz>)` ausgegeben werden. Die mit `length()` ermittelte Länge eines Datensatzes ist die Anzahl der in ihm gespeicherten Variablen, also Spalten. Eine Übersicht aller Variablen eines Datensatzes erhält man durch `summary(<Datensatz>)`.

```
> dim(myDf1)
[1] 12 6

> nrow(myDf1)
[1] 12

> ncol(myDf1)
[1] 6

> summary(myDf1)
   id   sex group        age          IQ       rating
Min. : 1.00 m:9 CG:4  Min. :20.00 Min. : 81.00 Min. :0.000
1st Qu.: 3.75 w:3 T :4  1st Qu.:22.75 1st Qu.: 89.50 1st Qu.:1.000
Median : 6.50 WL:4 Median :25.50 Median : 93.50 Median :2.500
Mean   : 6.50          Mean :26.42 Mean   : 97.25 Mean   :2.583
3rd Qu.: 9.25          3rd Qu.:29.25 3rd Qu.:104.50 3rd Qu.:3.500
Max.   :12.00          Max. :35.00 Max.   :122.00 Max. :5.000
```

Will man sich einen Überblick über die in einem Datensatz x gespeicherten Werte verschaffen, können die Funktionen `head(x=<Datensatz>, n=<Anzahl>)` und `tail(x=<Datensatz>, n=<Anzahl>)` verwendet werden, die die ersten bzw. letzten n Zeilen von x anzeigen.

3.2.1 Datentypen in Datensätzen

Mit der Funktion `str(<Datensatz>)` kann die interne Struktur des Datensatzes erfragt werden, d.h. aus welchen Gruppierungsfaktoren und wie vielen Beobachtungen an welchen Variablen er besteht.

```
> str(myDf1)
'data.frame': 12 obs. of 6 variables:
 $ id    : int 1 2 3 4 5 6 7 8 9 10 ...
 $ sex   : Factor w/ 2 levels "f", "m": 1 2 2 2 2 1 2 2 2 2 ...
 $ group : Factor w/ 3 levels "CG", "T", "WL": 2 1 1 2 3 1 2 ...
 $ age   : int 26 30 25 34 22 24 28 35 23 29 ...
 $ IQ    : num 112 122 95 102 82 113 92 90 88 81 ...
 $ rating: num 1 3 5 5 2 0 3 2 3 5 ...
```

In der Struktur ist zu erkennen, dass die in den Datensatz aufgenommenen character Vektoren automatisch in Objekte der Klasse `factor` umgewandelt

werden. Eine Umwandlung von Zeichenketten in `factor` Objekte ist nicht immer gewünscht, denn Zeichenketten codieren nicht immer eine Gruppenzugehörigkeit, sondern können auch normale AVn darstellen. Wird ein Zeichenketten-Vektor in `I(<Vektor>)` eingeschlossen, verhindert dies die automatische Umwandlung. Für alle `character` Vektoren gleichzeitig kann dies auch über das `stringsAsFactors=FALSE` Argument von `data.frame()` erreicht werden.

```
> fac    <- c("CG", "T1", "T2")
> DV1   <- c(14, 22, 18)
> DV2   <- c("red", "blue", "blue")
> myDf2 <- data.frame(fac, DV1, DV2, stringsAsFactors=FALSE)
> str(myDf2)
'data.frame': 3 obs. of 3 variables:
$ fac: chr "CG" "T1" "T2"
$ DV1: num 14 22 18
$ DV2: chr "red" "blue" "blue"
```

Dabei wird allerdings auch der Vektor `fac` nicht mehr als Faktor interpretiert. Stellt nur einer von mehreren `character` Vektoren eine Gruppierungsvariable dar, so kann das Argument `stringsAsFactors=FALSE` zwar verwendet werden, die Gruppierungsvariable ist dann aber vor oder nach der Zusammenstellung des Datensatzes manuell mit `as.factor(<Vektor>)` zu konvertieren.

```
> fac    <- as.factor(fac)
> myDf3 <- data.frame(fac, DV1, DV2, stringsAsFactors=FALSE)
> str(myDf3)
'data.frame': 3 obs. of 3 variables:
$ fac: Factor w/ 3 levels "CG",...: 1 2 3
$ DV1: num 14 22 18
$ DV2: chr "red" "blue" "blue"
```

Matrizen und Vektoren können mit der `as.data.frame(<Objekt>)` Funktion in einen Datensatz umgewandelt werden, wobei hier die in Matrizen und Vektoren notwendigerweise identischen Datentypen nachträglich zu konvertieren sind, wenn sie eigentlich unterschiedliche Variabtentypen repräsentieren. Dies ist z. B. dann der Fall, wenn numerische Vektoren und Zeichenketten in Matrizen zusammengefasst und so alle Elemente zu Zeichenketten gemacht wurden.

Listen können in Datensätze umgewandelt werden, wenn ihre Komponenten alle dieselbe Länge besitzen. Umgekehrt können Datensätze mit `data.matrix(<Datensatz>)` und auch `as.matrix(<Datensatz>)` zu Matrizen gemacht, wobei alle Werte in denselben Datentyp umgewandelt werden. Bei der Umwandlung in eine Liste mit `as.list(<Datensatz>)` ist dies dagegen nicht notwendig.

3.2.2 Elemente auswählen und verändern

Um einzelne Elemente anzeigen zu lassen und diese zu ändern, können dieselben Befehle wie bei Listen verwendet werden.

```
> myDf1[[3]][2]                                # 2. Element der 3. Variable
[1] CG
Levels: CG T WL

> myDf1$rating                                 # Variable rating
[1] 1 3 5 5 2 0 3 2 3 5 1 1

> myDf1$age[4]                                  # Alter der 4. VP
[1] 34

> myDf1$IQ[10:12] <- c(99, 110, 89)          # ändere IQ-Werte für VPn 10-12
```

Als Besonderheit können Datensätze analog zu Matrizen mit dem [$\langle\text{Index}\rangle$ → Element der Komponente], $\langle\text{Index der Komponente}\rangle$] Operator indiziert werden. Bei dieser Variante bleibt die gewohnte Reihenfolge von Zeile (entspricht einer VP) – Spalte (entspricht einer Variable) erhalten, ist also in vielen Fällen dem [[$\langle\text{Index}\rangle$]] Operator vorzuziehen.

```
> myDf1[3, 4]                                  # 3. Element der 4. Variable
[1] 25

> myDf1[4, 3]                                  # 4. Element der 3. Variable
[1] T
Levels: CG T WL
```

Wie bei Matrizen gilt das Weglassen eines Index unter Beibehaltung des Kommas als Anweisung, alle Werte der ausgelassenen Dimension anzuzeigen.⁶ Auch hier kann das Argument drop=FALSE verwendet werden, wenn das Ergebnis bei der Ausgabe nur einer Spalte weiterhin ein Datensatz sein soll.

```
> myDf1[2, ]                                     # alle Werte der 2. VP
   id sex group age  IQ rating
2   2   m    CG  30 122      3

> myDf1[ , 4]                                    # alle Elemente der 4. Variable
[1] 26 30 25 34 22 24 28 35 23 29 20 21

> myDf1[1:5, 4, drop=FALSE]                      # Elemente als Datensatz
  age
1  26
2  30
3  25
4  34
5  22
```

⁶ Das Komma ist von Bedeutung: so würde etwa $\langle\text{Datensatz}\rangle[3]$ wie in Listen nicht einfach die dritte Variable von $\langle\text{Datensatz}\rangle$ zurückgeben, sondern einen Datensatz, dessen einzige Spalte diese Variable ist.

3.2.3 Datensätze in den Suchpfad einfügen

Die in einem Datensatz vorhandenen Variablen sind außerhalb des Datensatzes unbekannt. Enthält ein Datensatz `myDf` die Variable `var`, so kann auf diese nur mit `myDf$var`, nicht aber einfach mit `var` zugegriffen werden. Nun kann es bequem sein, die Variablennamen auch ohne das wiederholte Aufführen von `myDf$` zu verwenden. Eine temporär wirkende Möglichkeit hierzu bietet die Funktion `with()`.

```
> with(data=<Datensatz>, expr=<R-Befehle>)
```

Innerhalb der unter `expr` angegebenen Befehle sind die Variablennamen des unter `data` genannten Datensatzes bekannt, außerhalb von `with()` jedoch nicht. Der Datensatz selbst kann innerhalb von `with()` nicht verändert, sondern nur gelesen werden.

```
> with(myDf1, tapply(IQ, group, mean))
   KG      T      WL
105.50 101.00 85.25
```

Demselben Zweck dient in vielen Funktionen das Argument `data=<Datensatz>`, das es erlaubt, in anderen Argumenten der Funktion Variablen von `data` zu verwenden.

```
> xtabs(~ sex + group, data=myDf1)
       group
sex CG T WL
  f  1 2 0
  m  3 2 4
```

Mit der Funktion `attach(<Datensatz>)` ist es möglich, einen Datensatz in den Suchpfad einzuhängen und die Namen seiner Variablen so auch permanent ohne wiederholtes Voranstellen von `<Datensatz>$` verfügbar zu machen. Dies wird etwa an der Ausgabe von `search()` deutlich, die den Datensatz nach Einhängen in den Suchpfad mit aufführt (vgl. Abschn. 1.3.1).

```
> IQ[3]
Fehler: Objekt "IQ" nicht gefunden

> attach(myDf1)
> IQ[3]
[1] 95

> search()[1:4]
[1] ".GlobalEnv" "myDf1" "package:grDevices" "package:datasets"
```

Wichtig ist, dass durch `attach()` im Workspace Kopien aller Variablen des Datensatzes angelegt werden.⁷ Greift man auf eine Variable `var` direkt, d.h. ohne

⁷ Bei sehr großen Datensätzen empfiehlt es sich daher aus Gründen der Speichernutzung, nur eine geeignete Teilmenge von Fällen mit `attach()` verfügbar zu machen, vgl. Abschn. 3.2.6.2.

Nennung von `myDf$` zu, so verwendet man die Kopie. Insbesondere werden in diesem Fall Änderungen nur an der Kopie im Workspace, nicht aber am eigentlichen Datensatz vorgenommen. Datensätze sollten daher erst dann in den Suchpfad eingefügt werden, wenn alle Modifikationen an ihm abgeschlossen sind. Mit dem Befehl `detach(<Datensatz>)` kann der Datensatz wieder aus dem Suchpfad entfernt werden, wenn nicht mehr auf seine Variablen zugegriffen werden muss. Dies sollte nicht vergessen werden, sonst besteht das Risiko, mit einem neuerlichen Aufruf von `attach()` denselben Datensatz mehrfach verfügbar zu machen, was für Verwirrung sorgen kann.

```
> IQ[3] <- 130; IQ[3]
[1] 130

> myDf1$IQ[3]
[1] 95

> detach(myDf1)
> IQ
Fehler: Objekt "IQ" nicht gefunden
```

3.2.4 Namen von Variablen und Beobachtungen

Um die Namen eines Datensatzes auf beiden Dimensionen (Zeilen und Spalten, also meist VPn und Variablen) zu erfragen und auch zu ändern, dient die von Arrays bekannte Funktion `dimnames(<Datensatz>)`.⁸ Sie gibt eine Liste aus, in der die einzelnen Namen für jede der beiden Dimensionen als Komponenten enthalten sind. Wurden die Zeilen nicht benannt, werden ihre Nummern ausgegeben.

```
> dimnames(myDf1)
[[1]]
[1] "1" "2" "3" "4" "5" "6" "7" "8" "9" "10" "11" "12"

[[2]]
[1] "id" "sex" "group" "age" "IQ" "rating"
```

Die Namen der Variablen in einem Datensatz können mit der `names(<Datensatz>)` Funktion erfragt werden. Diese gibt nur die Variablennamen aus und ist damit gleichwertig zur Funktion `colnames(<Datensatz>)`, welche die Spaltennamen liefert.

```
> names(myDf1)
[1] "id" "sex" "group" "age" "IQ" "rating"
```

Variablennamen lassen sich verändern, indem der entsprechende Variablenname im obigen Output-Vektor ausgewählt und über eine Zuweisung umbenannt wird. Dabei

⁸ Namen werden als Attribut gespeichert und sind mit `attributes(<Datensatz>)` sichtbar, vgl. Abschn. 1.3.

kann entweder die Position direkt angegeben oder durch Vergleich mit dem gesuchten Variablennamen implizit ein logischer Indexvektor verwendet werden.

```
> names(myDf1)[3]
[1] "group"

> names(myDf1)[3] <- "fac"; names(myDf1)
[1] "id" "sex" "fac" "age" "IQ" "rating"

> names(myDf1)[names(myDf1) == "fac"] <- "group"; names(myDf1)
[1] "id" "sex" "group" "age" "IQ" "rating"
```

Die Bezeichnungen der Zeilen können mit den Funktionen `rownames`(`<Datensatz>`) bzw. `row.names`(`<Datensatz>`) ausgegeben und auch geändert werden. Der Vorgang ist analog zu jenem bei `names`(`<Datensatz>`).

```
> (rows <- paste("Z", 1:12, sep=""))
[1] "Z1" "Z2" "Z3" "Z4" "Z5" "Z6" "Z7" "Z8" "Z9" "Z10" "Z11" "Z12"

> rownames(myDf1) <- rows; myDf1[1:5, ]
   id sex group age IQ rating
Z1  1   f      T  26 112     1
Z2  2   m     CG  30 122     3
Z3  3   m     CG  25  95     5
Z4  4   m      T  34 102     5
Z5  5   m    WL  22  82     2
```

3.2.5 Variablen einem Datensatz hinzufügen oder aus diesem entfernen

Einem bestehenden Datensatz können neue Variablen mit den `<Datensatz>$<neue Variable>` und `<Datensatz>["<neue Variable>"]` Operatoren hinzugefügt werden. Analog zum Vorgehen bei Matrizen kann an einen Datensatz auch mit `cbind(<Datensatz>, <Vektor>, ...)` eine weitere Variable als Spalte angehängt werden, sofern der zugehörige Vektor passend viele Elemente umfasst.⁹

Im Beispiel soll der Beziehungsstatus der VPn dem Datensatz hinzugefügt werden.

```
> isSingle <- sample(c(TRUE, FALSE), nrow(myDf1), replace=TRUE)
> myDf2      <- myDf1                                # erstelle Kopie
> myDf2$isSingle1 <- isSingle                         # Möglichkeit 1
> myDf2["isSingle2"] <- isSingle                      # Möglichkeit 2
> myDf3      <- cbind(myDf1, isSingle); myDf3[1:5, ]  # Möglichkeit 3
   id sex group age IQ rating isSingle
```

⁹ Dagegen ist das Ergebnis von `cbind(<Vektor1>, <Vektor2>)` eine Matrix. Dies ist insbesondere wichtig, wenn numerische Daten und Zeichenketten zusammengefügt werden – in einer Matrix würden die numerischen Werte automatisch in Zeichenketten konvertiert.

```
Z1 1 f T 26 112 1 TRUE
Z2 2 m CG 30 122 3 TRUE
Z3 3 m CG 25 95 5 TRUE
Z4 4 m T 34 102 5 FALSE
Z5 5 m WL 22 82 2 TRUE
```

Alternativ kann auch die `transform()` Funktion Verwendung finden.

```
> transform(<Datensatz>, <Variablenname>=<Ausdruck>)
```

Unter `<Ausdruck>` ist anzugeben, wie sich die Werte der neuen Variable ergeben, die unter `<Variablenname>` gespeichert und an `<Datensatz>` angehängt wird. Im Beispiel soll das Quadrat des Ratings angefügt werden. Die Variablennamen des Datensatzes sind innerhalb der `transform()` Funktion bekannt.

```
> myDf4 <- transform(myDf1, rSq=rating^2); myDf4[1:3, ]
   id sex group age IQ rating rSq
Z1 1 f T 26 112 1 1
Z2 2 m CG 30 122 3 9
Z3 3 m CG 25 95 5 25
```

Variablen eines Datensatzes werden gelöscht, indem ihnen die leere Menge `NULL` zugewiesen wird – im Fall mehrerer Variablen in Form einer Liste mit so vielen Komponenten `NULL`, wie Variablen entfernt werden sollen.¹⁰

```
> dfTemp <- myDf1 # Kopie erstellen
> dfTemp$group <- NULL; dfTemp[1:5, ] # eine Variable löschen
   id sex age IQ rating
Z1 1 w 26 112 1
Z2 2 m 30 122 3
Z3 3 m 25 95 5
Z4 4 m 34 102 5
Z5 5 m 22 82 2

> delVars <- c("sex", "IQ") # mehrere Variablen löschen
> nullList <- vector("list", length(delVars))
> dfTemp[delVars] <- nullList; dfTemp[1:5, ]
   id age rating
Z1 1 26 1
Z2 2 30 3
Z3 3 25 5
Z4 4 34 5
Z5 5 22 2
```

¹⁰ Das genannte Vorgehen wirft die Frage auf, wie sich allen Elementen einer Variable gleichzeitig der Wert `NULL` zuweisen lässt, statt die Variable zu löschen. Dies ist durch `<Datensatz>$<Variable><-list(NULL)` bzw. `<Datensatz>[["<Variable>"]]<-list(NULL)` möglich.

3.2.6 Teilmengen von Daten auswählen

Bei der Analyse eines Datensatzes möchte man häufig eine Auswahl der Daten treffen, etwa VPn aus nur einer bestimmten Gruppe oder über einem bestimmten Testwert separat untersuchen, oder aber die Auswertung auf eine Teilgruppe von Variablen beschränken. Die Auswahl von Beobachtungsobjekten auf der einen und Variablen auf der anderen Seite unterscheidet sich in R dabei nicht konzeptuell von einander, vielmehr kommen in beiden Fällen die bereits bekannten Strategien zur Indizierung von Objekten zum Tragen.

3.2.6.1 Fälle und Variablen mit logischem Indexvektor auswählen

Beziehen sich Auswahlkriterien auf die Beobachtungsobjekte, ist das Ziel, eine Teilmenge der Zeilen des Datensatzes auszuwählen. Welche Beobachtungsobjekte einem Kriterium entsprechen, ergibt sich dabei durch den Vergleich einer bestimmten Variable mit einem Referenzwert – dieser Vergleich liefert einen logischen Indexvektor, der zum Indizieren der Zeilen des Datensatzes verwendet werden kann. Enthält der Datensatz möglicherweise fehlende Werte, sollten die Wahrheitswerte mit `which()` in numerische Indizes konvertiert werden (vgl. Abschn. 2.2.2).

```
> (idxLog <- myDf1$sex == "f")                      # Indizes der weiblichen VPn
[1] TRUE FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE TRUE

> (idxNum <- which(idxLog))                          # numerische Indizes
[1] 1 6 12

> myDf1[idxNum, ]
   id sex group age  IQ rating
Z1  1   f      T  26 112     1
Z6  6   f     CG  24 113     0
Z12 12   f      T  21  98     1
```

Möchte man nun Fälle hinsichtlich mehrerer Kriterien auswählen, kann es der Fall sein, dass mehrere Bedingungen gleichzeitig erfüllt sein müssen (logisches UND, `&`) oder es ausreicht, wenn bereits eines von mehreren Kriterien erfüllt ist (logisches ODER, `|`).

```
# alle männlichen VPn mit einem Rating größer als 2
> (idx2 <- (myDf1$sex == "m") & (myDf1$rating > 2))
[1] FALSE TRUE TRUE TRUE FALSE FALSE TRUE FALSE TRUE FALSE FALSE

> myDf1[which(idx2), ]
   id sex group age  IQ rating
Z2  2   m     CG  30 122     3
Z3  3   m     CG  25  95     5
Z4  4   m      T  34 102     5
Z7  7   m      T  28  92     3
Z9  9   m     WL  23  88     3
Z10 10   m     WL  29  81     5

# alle VPn mit einem eher hohen ODER eher niedrigen IQ-Wert
> (idx3 <- (myDf1$IQ < 90) | (myDf1$IQ > 110))
```

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

> myDf1[which(idx3), ]
   id sex group age  IQ rating
Z1  1   f     T  26 112      1
Z2  2   m    CG  30 122      3
Z5  5   m    WL  22  82      2
Z6  6   f    CG  24 113      0
Z9  9   m    WL  23  88      3
Z10 10   m    WL  29  81      5
```

Auf analoge Weise lässt sich ein Datensatz auf eine Teilmenge seiner Variablen beschränken: hier dient ein sich auf die Spalten beziehender Vektor dazu, nur bestimmte Spalten anhand ihrer Namen oder numerischen Indizes auszuwählen.

```
> myDf1[1:4, c("group", "IQ")]  # Werte 1 bis 4 von group und IQ
   group  IQ
Z1     T 112
Z2    CG 122
Z3    CG  95
Z4     T 102

> myDf1[1:4, 2:4]                # Werte 1 bis 4 der Variablen 2 bis 4
   sex group age
Z1   f     T  26
Z2   m    CG  30
Z3   m    CG  25
Z4   m     T  34
```

Um nach Namensmustern unter den Spaltenbezeichnungen zu suchen, können die in Abschn. 2.13.4 vorgestellten Funktionen zur Suche nach Zeichenfolgen dienen.

```
> dfTemp <- myDf1           # kopiere myDf1

# neue Variablennamen, so dass sich zwei Gruppen von Variablen ergeben
> (names(dfTemp) <- paste(rep(c("A", "B"), each=3), 100:102, sep=""))
[1] "A100" "A101" "A102" "B100" "B101" "B102"

> (colIdx <- grep("^B.*$", names(dfTemp)))  # mit B beginnende Variablen
[1] 4 5 6

> dfTemp[1:4, colIdx]        # Werte 1 bis 4 dieser Variablen
   B100 B101 B102
Z1   26   112    1
Z2   30   122    3
Z3   25   95     5
Z4   34   102    5
```

3.2.6.2 Fälle mit `subset()` auswählen

Eine etwas übersichtlichere Methode zur Auswahl von Fällen bietet die Funktion `subset()`, die einen Datensatz mit den passenden Fällen zurückgibt. Sie eignet sich damit in Situationen, in denen eine andere Auswertungsfunktion einen Datensatz als

Argument erwartet – etwa wenn ein inferenzstatistischer Test nur mit einem Teil der Daten durchgeführt werden soll.

```
> subset(x=<Datensatz>, subset=<Indexvektor>, select=<Spalten>)
```

Das Argument `subset` ist ein logischer Indexvektor zum Indizieren der Zeilen, der sich häufig aus einem Vergleich der Werte einer Variable mit einem Kriterium ergibt.¹¹ Innerhalb von `subset()` sind die Variablennamen des Datensatzes bekannt, ihnen muss also im Ausdruck zur Bildung eines Indexvektors nicht `<Datensatz>$` vorangestellt werden.

```
> subset(myDf1, sex == "f")           # Daten der weiblichen VPn
   id sex group age  IQ rating
Z1  1   f     T  26 112      1
Z6  6   f     CG 24 113      0
Z12 12  f     T  21  98      1
```

Da im Beispiel die Ausprägung der Variable `sex` zur Bedingung gemacht wurde, könnte diese für alle ausgegebenen Zeilen konstante Variable im Output auch weggelassen werden. Dazu dient das Argument `select`, das einen Vektor mit auszugebenden Spaltenindizes oder -namen benötigt. Um die Ausgabe der Variable `sex` (Spalte 2) zu unterdrücken, eignet sich wie gewohnt ein negatives Vorzeichen vor dem Index.

```
> subset(myDf1, sex == "f", select=-2)
   id group age  IQ rating
Z1  1     T  26 112      1
Z6  6    CG 24 113      0
Z12 12    T  21  98      1
```

Mit `subset()` ist auch die Auswahl nach mehreren Bedingungen gleichzeitig möglich, indem `<Indexvektor>` durch entsprechend erweiterte logische Ausdrücke gebildet wird.

```
# männliche VPn mit einem Rating > 2
> subset(myDf1, (myDf1$sex == "m") & (myDf1$rating > 2))      # ...

# VPn mit eher niedrigem ODER eher hohem IQ
> subset(myDf1, (myDf1$IQ < 90) | (myDf1$IQ > 110))          # ...
```

Für die Auswahl von Fällen, deren Wert auf einer Variable aus einer Menge bestimmter Werte stammen soll (logisches ODER), gibt es eine weitere Möglichkeit: mit dem `<Menge1> %in% <Menge2>` Operator als Kurzform der `is.element()` Funktion kann ebenfalls ein logischer Indexvektor zur Verwendung in `subset()` gebildet werden. Dabei prüft `<Menge1> %in% <Menge2>` für jedes Element von `<Menge1>`, ob es auch in `<Menge2>` vorhanden ist und gibt einen logischen Vektor mit den einzelnen Ergebnissen aus.

¹¹ Da fehlende Werte innerhalb von `subset` als `FALSE` behandelt werden, ist es nicht notwendig, die logischen Indizes mit `which()` in numerische umzuwandeln.

```
# VPn aus Wartelisten- ODER Kontrollgruppe
> subset(myDf1, group %in% c("CG", "WL"))
   id sex group age  IQ rating
Z2  2   m    CG  30 122     3
Z3  3   m    CG  25  95     5
Z5  5   m    WL  22  82     2
Z6  6   f    CG  24 113     0
Z8  8   m    WL  35  90     2
Z9  9   m    WL  23  88     3
Z10 10  m    WL  29  81     5
Z11 11  m    CG  20  92     1
```

3.2.7 Organisationsform von Daten ändern

Bisweilen weicht der Aufbau eines Datensatzes von dem beschriebenen ab, etwa wenn sich Daten derselben, in verschiedenen Bedingungen erhobenen AV in unterschiedlichen Spalten befinden, die mit den Bedingungen korrespondieren. Wurde in jeder Bedingung eine andere Menge von VPn beobachtet, enthält eine Zeile dann nicht mehr die Daten einer einzelnen VP. Um diese Organisationsform in die übliche zu überführen, dient die `stack()` Funktion: Ziel ist es also, eine Zeile pro VP, eine Spalte für die Werte der AV in allen Bedingungen und eine Spalte für die zu jedem Wert gehörende Stufe der UV zu erhalten.

```
> stack(x=<Liste>, select=<Spalten>)
```

Unter `x` ist eine Liste mit benannten Komponenten (meist ein Datensatz) anzugeben, deren zu reorganisierende Variablen über das Argument `select` bestimmt werden. Das Argument erwartet einen Vektor mit Spaltenindizes oder Variablennamen. Das Ergebnis ist ein Datensatz, in dessen erster Spalte mit dem Namen `values` sich alle Werte der AV befinden. Diese kommt zustande, indem die ursprünglichen Spalten wie durch den Befehl `c(<Spalte1>, <Spalte2>, ...)` aneinander gehängt werden. Die zweite Spalte des Datensatzes mit dem Namen `ind` ist ein Faktor, dessen Stufen codieren, aus welcher Spalte von `x` ein einzelner Wert der nun ersten Spalte stammt. Hierzu werden die Variablennamen von `x` herangezogen.

```
> nSubj  <- 4
> vec1  <- sample(1:10, nSubj, replace=TRUE)
> vec2  <- sample(1:10, nSubj, replace=TRUE)
> vec3  <- sample(1:10, nSubj, replace=TRUE)
> dfTemp <- data.frame(cond1=vec1, cond2=vec2, cond3=vec3)
> (res  <- stack(dfTemp, select=c("cond1", "cond3")))
   values   ind
1      3 cond1
2      2 cond1
3      4 cond1
4      3 cond1
5      6 cond3
6      8 cond3
```

```
7      7  cond3
8      3  cond3
```

```
> str(res)
'data.frame': 8 obs. of 2 variables:
$ values: int 3 2 4 3 6 8 7 3
$ ind   : Factor w/ 2 levels "cond1","cond3": 1 1 1 1 2 2 2 2
```

Das Ergebnis von `stack()` wird durch `unstack()` umgekehrt. Diese Funktion transformiert also einen Datensatz, der aus einer Variable mit den Werten der AV und einer Variable mit den zugehörigen Faktorstufen besteht, in einen Datensatz mit so vielen Spalten wie Faktorstufen. Dabei beinhaltet jede Spalte die zu einer Stufe gehörenden Werte der AV. Das Ergebnis von `unstack()` ist nur dann ein Datensatz, wenn alle Faktorstufen gleich häufig vorkommen, wenn also die resultierenden Variablen dieselbe Länge aufweisen. Andernfalls ist das Ergebnis eine Liste.

```
> unstack(x=<Datensatz>, form=<Formel>)

> unstack(res)
  cond1 cond3
1     3     6
2     2     8
3     4     7
4     3     3
```

Kommen im Datensatz x mehrere AVn und UVn vor, so kann über eine an das Argument `form` zu übergebende sog. Formel $\langle \text{AV} \rangle \sim \langle \text{UV} \rangle$ festgelegt werden, welche AV ausgewählt und nach welcher UV die Trennung der Werte der AV vorgenommen werden soll (vgl. Abschn. 5.1).

```
> res$newIV <- factor(sample(rep(c("A", "B"), c(4, 4), 8, replace=FALSE)))
> res$newDV <- sample(100:200, 8)
> unstack(res, newDV ~ newIV)
  A   B
1 183 193
2 129 115
3 142 140
4 194 134
```

Die Organisationsformen eines Datensatzes, zwischen denen mit `stack()` und `unstack()` gewechselt werden kann, werden im Kontext von Daten aus Messwiederholungen auch als Long-Format und Wide-Format bezeichnet. Häufig sind die Datensätze dann jedoch zu komplex, um noch mit diesen Funktionen bearbeitet werden zu können. Für solche Situationen ist die im folgenden Abschnitt beschriebene `reshape()` Funktion geeignet.

3.2.8 Organisationsform eines Datensatzes ändern

Wurden an denselben VPn zu verschiedenen Messzeitpunkten Daten derselben AV erhoben, können die Werte jeweils eines Messzeitpunkts als zu einer Variable gehörend betrachtet werden. In einem Datensatz findet sich jede dieser Variablen dann

in einer separaten Spalte. Diese Organisationsform folgt dem bisher dargestellten Prinzip, dass pro Zeile die Daten jeweils einer VP aus verschiedenen Variablen stehen. Eine solche Strukturierung wird auch als Wide-Format bezeichnet, weil der Datensatz durch mehr Messzeitpunkte an Spalten gewinnt, also breiter wird. Das Wide-Format entspricht einer multivariaten Betrachtungsweise von Daten aus Messwiederholungen.

Für die univariate Analyse von Daten, die aus abhängigen Messungen hervorgehen, ist jedoch oft eine andere Organisationsform notwendig – das sog. Long-Format. Die zu verschiedenen Messzeitpunkten gehörenden Werte einer VP stehen hier in separaten Zeilen. Auf diese Weise beinhalten mehrere Zeilen Daten derselben VP. Der Name des Long-Formats leitet sich daraus ab, dass mehr Messzeitpunkte zu mehr Zeilen führen, der Datensatz also länger wird. Wichtig bei Verwendung dieses Formats ist zum einen das Vorhandensein eines Faktors, der codiert, von welcher VP eine Beobachtung stammt. Diese Variable ist dann jeweils über so viele Zeilen konstant, wie es Messzeitpunkte gibt. Zum anderen muss ein Faktor existieren, der den Messzeitpunkt codiert.

Im Beispiel sei an vier VPn eine AV zu jeweils drei Messzeitpunkten erhoben worden. Bei zwei der VPn sei dies in Bedingung A, bei den anderen beiden in Bedingung B einer Manipulation geschehen. Damit liegen zwei UVn vor, zum einen als Intra-Gruppen Faktor der Messzeitpunkt, zum anderen ein Zwischen-Gruppen Faktor (sog. Split-Plot Design, vgl. Abschn. 8.7). Zunächst soll demonstriert werden, wie sich das Long-Format manuell aus gegebenen Vektoren herstellen lässt. Soll mit einem solchen Datensatz etwa eine univariate Varianzanalyse mit Messwiederholung gerechnet werden, muss sowohl die VP- bzw. Blockzugehörigkeit eines Messwertes als auch der Messzeitpunkt jeweils in einem Objekt der Klasse `factor` gespeichert sein.

```

> nSubj    <- 2                                # Zellbesetzung
> nGroups  <- 2                                # Anzahl Gruppen
> id       <- 1:(nGroups*nSubj)                 # VP / Blockzugehörigkeit
> group    <- factor(rep(c("A", "B"), nSubj))  # Gruppenzugehörigkeit
> DV_t1    <- round(rnorm(nGroups*nSubj, -1, 1), 2)      # AV zu t1
> DV_t2    <- round(rnorm(nGroups*nSubj, 0, 1), 2)      # AV zu t2
> DV_t3    <- round(rnorm(nGroups*nSubj, 1, 1), 2)      # AV zu t3

# Datensatz im Wide-Format
> (dfWide <- data.frame(id, group, DV_t1, DV_t2, DV_t3))
   id group DV_t1 DV_t2 DV_t3
1  1     A -1.64  0.01  1.31
2  2     B -0.81 -1.23  1.59
3  3     A -1.43 -0.80  0.68
4  4     B -1.79 -0.13 -0.14

# Variablen für Long-Format
> DVL     <- c(DV_t1, DV_t2, DV_t3)           # alle Daten in einem Vektor
> idL     <- factor(rep(id, 3))                # zugehöriger VP-Code
> groupL  <- rep(group, 3)                     # Gruppenzugehörigkeit
> timeFac <- factor(rep(1:3, each=nSubj*nGroups)) # Messzeitpunkt

```

```
# Datensatz im Long-Format
> dfLong <- data.frame(id=idL, group=groupL, varTime=timeFac, DV=DVL)
> dfLong[order(dfLong$id), ] # sortierte Ausgabe
   id group varTime   DV
1   1     A      1 -1.64
5   1     A      2  0.01
9   1     A      3  1.31
2   2     B      1 -0.81
6   2     B      2 -1.23
10  2     B      3  1.59
3   3     A      1 -1.43
7   3     A      2 -0.80
11  3     A      3  0.68
4   4     B      1 -1.79
8   4     B      2 -0.13
12  4     B      3 -0.14
```

Die `reshape()` Funktion bietet die Möglichkeit, einen Datensatz ohne manuelle Zwischenschritte zwischen Wide- und Long-Format zu transformieren.¹²

```
> reshape(data=Datensatz, varying, timevar="time",
+          idvar="id", direction=c("wide", "long"), v.names="(Name)")
```

Die Argumente haben folgende Bedeutung:

- Zunächst wird der Datensatz unter `data` eingefügt. Um ihn vom Wide- ins Long-Format zu transformieren, muss das Argument `direction="long"` gesetzt werden.
- Daneben ist unter `varying` anzugeben, welche Variablen im Wide-Format die-
selbe AV zu unterschiedlichen Messzeitpunkten repräsentieren. Die Variablen werden im Long-Format über unterschiedliche Ausprägungen der neu gebil-
deten Variable `time` identifiziert, deren Name über das Argument `timevar` auch
selbst festgelegt werden kann. `varying` benötigt eine Liste, deren Komponenten Vektoren mit Variablennamen oder Spaltenindizes sind. Jeder Vektor gibt dabei
eine Gruppe von Variablen an, die jeweils zu einer AV gehören.¹³
- Besitzt der Datensatz eine Variable, die codiert, von welcher VP ein Wert stammt,
kann sie im Argument `idvar` genannt werden. Andernfalls wird eine solche Va-
riable auf Basis des Zeilenindex gebildet und trägt den Namen `id`. Auch andere
Variablen von `data`, die pro Messzeitpunkt zwischen den VPn variieren, gelten
als `idvar`, dies trifft etwa auf die Ausprägung von Zwischen-Gruppen Faktoren
zu. Im Fall mehrerer solcher Variablen sind diese als Vektor von Variablennamen
anzugeben.

¹² Das Paket `reshape` (Wickham, 2007) stellt weitere spezialisierte Möglichkeiten zur Transfor-
mation zwischen beiden Organisationsformen bereit.

¹³ Im Fall zweier AVn, für die jeweils eine Gruppe von zwei Spalten im Wide-Format vor-
handen ist, könnte das Argument `varying=list(c("DV1_t1", "DV1_t2"), c("DV2_t1",
→"DV2_t2"))` lauten.

- Der Name der Variable im Long-Format mit den Werten der AV kann über das Argument `v.names` bestimmt werden.

```
> resLong <- reshape(dfWide, varying=list(c("DV_t1", "DV_t2", "DV_t3")),
+                     direction="long", idvar=c("id", "group"), v.names="DV")

> resLong[order(resLong$id), ] # sortierte Ausgabe
   id group time    DV
1.1  1     A    1 -1.64
1.2  1     A    2  0.01
1.3  1     A    3  1.31
2.1  2     B    1 -0.81
2.2  2     B    2 -1.23
2.3  2     B    3  1.59
3.1  3     A    1 -1.43
3.2  3     A    2 -0.80
3.3  3     A    3  0.68
4.1  4     B    1 -1.79
4.2  4     B    2 -0.13
4.3  4     B    3 -0.14
```

Die Variablen in der Rolle von `idvar` und `time` sollten Objekte der Klasse `factor` sein. Da `reshape()` die Variablen aber als numerische Vektoren generiert, müssen sie ggf. manuell mit `<- factor >`` umgewandelt werden.

Ist der Datensatz vom Long- ins Wide-Format zu transformieren, muss das `direction="wide"` Argument gesetzt werden. Für das Argument `v.names` wird jene Variable genannt, die die Werte der AV im Long-Format über alle Messzeitpunkte hinweg speichert. Diese Variable wird im Wide-Format auf mehrere Spalten aufgeteilt, die den Messzeitpunkten entsprechen. Dafür ist unter `timevar` anzugeben, welche Variable den Messzeitpunkt codiert. Unter `idvar` sind jene Variablen zu nennen, deren Werte die Daten der AV einer VP zuordnen bzw. pro Messzeitpunkt über die VPn variieren, etwa weil sie die Ausprägung von Zwischen-Gruppen Faktoren darstellen.

```
> reshape(dfLong, v.names="DV", timevar="varTime",
+          idvar=c("id", "group"), direction="wide")
   id group DV.1 DV.2 DV.3
1  1     A -1.64  0.01  1.31
2  2     B -0.81 -1.23  1.59
3  3     A -1.43 -0.80  0.68
4  4     B -1.79 -0.13 -0.14
```

Ist eine AV an denselben VPn mehrfach in den kombinierten Bedingungen zweier Intra-Gruppen Faktoren erhoben worden, so muss `reshape()` zweimal angewandt werden, um die Daten vom Wide- ins Long-Format zu transformieren. Im ersten Schritt werden die zu unterschiedlichen Bedingungen des ersten Faktors gehörenden Spalten zusammengefasst, im zweiten Schritt diejenigen des zweiten.

Im Beispiel seien an vier VPn in jeder Stufenkombination der UV1 mit drei und der UV2 mit zwei Messzeitpunkten Werte einer AV erhoben worden.

```

> nSubj     <- 4                                # Zellbesetzung
> id        <- 1:nSubj                          # VP-Code
> t_11      <- round(rnorm(nSubj, 8, 2),2)      # AV zu t1-1
> t_12      <- round(rnorm(nSubj, 10, 2),2)      # AV zu t1-2
> t_21      <- round(rnorm(nSubj, 13, 2),2)      # AV zu t2-1
> t_22      <- round(rnorm(nSubj, 15, 2),2)      # AV zu t2-2
> t_31      <- round(rnorm(nSubj, 13, 2),2)      # AV zu t3-1
> t_32      <- round(rnorm(nSubj, 15, 2),2)      # AV zu t3-2
> dfWide    <- data.frame(id, t_11, t_12, t_21, t_22, t_31, t_32)
> (dfLong1 <- reshape(dfWide, varying=list(c("t_11", "t_21", "t_31"),
+                                         c("t_12", "t_22", "t_32")),
+                     direction="long", timevar="IV1", idvar="id",
+                     v.names=c("IV2-1", "IV2-2")))
   IV1  IV2-1  IV2-2  id
1.1   1  11.59  10.27  1
2.1   1   7.28  10.63  2
3.1   1   8.43   9.74  3
4.1   1   5.05  12.58  4
1.2   2  11.96  15.27  1
2.2   2  14.97  11.47  2
3.2   2  11.96  11.39  3
4.2   2  19.12  15.45  4
1.3   3  14.40  16.18  1
2.3   3  13.48  12.06  2
3.3   3  17.34  11.62  3
4.3   3  12.55  14.76  4

```

Da IV1 nun wie id pro Messzeitpunkt von IV2 über die VPn variiert, muss die Variable im zweiten Schritt ebenfalls unter idvar genannt werden.

```

> dfLong2 <- reshape(dfLong1, varying=list(c("IV2-1", "IV2-2")),
+                      direction="long", timevar="IV2",
+                      idvar=c("id", "IV1"), v.names="DV")

> head(dfLong2)
  id  IV1  IV2      DV
1.1.1  1    1    1  9.26
2.1.1  2    1    1  8.93
3.1.1  3    1    1  8.12
4.1.1  4    1    1  3.57
1.2.1  1    2    1 13.78
2.2.1  2    2    1 15.85

```

3.2.9 Datensätze teilen

Wenn die Beobachtungen (Zeilen) eines Datensatzes in durch die Stufen eines Faktors festgelegte Gruppen aufgeteilt werden sollen, kann dies mit `split()` geschehen.

```
> split(x=<Datensatz>, f=<Faktor>)
```

Für jede Zeile des Datensatzes `x` muss der Faktor `f` die Gruppenzugehörigkeit angeben und deshalb die Länge `nrow(x)` besitzen. Auch wenn `f` Teil des Datensatzes ist, muss der Faktor vollständig mit `<Datensatz>$Faktor` angegeben werden. Der Output ist eine Liste, die für jede sich aus den Stufen von `f` ergebende Gruppe eine Komponente in Form eines Datensatzes besitzt.

Durch `split()` können die Daten getrennt nach Bedingungen verarbeitet werden. Sind jedoch nur getrennt nach Gruppen zu berechnende Kennwerte einzelner Variablen von Interesse, ist meist `tapply()` das einfachere Mittel.

```
> (splitRes <- split(myDf1, myDf1$group))
$CG
  id sex group age IQ rating
Z2  2   m    CG  30 122     3
Z3  3   m    CG  25  95     5
Z6  6   f    CG  24 113     0
Z11 11  m    CG  20  92     1

$T
  id sex group age IQ rating
Z1  1   f    T   26 112     1
Z4  4   m    T   34 102     5
Z7  7   m    T   28  92     3
Z12 12  f    T   21  98     1

$WL
  id sex group age IQ rating
Z5  5   m    WL  22  82     2
Z8  8   m    WL  35  90     2
Z9  9   m    WL  23  88     3
Z10 10  m    WL  29  81     5

> myDf1CG <- splitRes$CG
> class(myDf1CG)
[1] "data.frame"

# berechne Mittelwerte getrennt nach Gruppen
# einfacher wäre hier: tapply(myDf1$IQ, myDf1$group, mean)
> (meanIQT  <- mean(splitRes$T$IQ))
[1] 101

> (meanIQWL <- mean(splitRes$WL$IQ))
[1] 85.25

> (meanIQCG <- mean(splitRes$CG$IQ))
[1] 105.5
```

3.2.10 Datensätze zusammenfügen

Wenn zwei oder mehr Datensätze vorliegen, die hinsichtlich ihrer Variablen identisch sind, so fügt die Funktion `rbind(<Datensatz1>, <Datensatz2>, ...)` die

Datensätze analog zum Vorgehen bei Matrizen zusammen, indem sie untereinander anordnet. Auf diese Weise könnten z. B. die Daten mehrerer Teilstichproben kombiniert werden, an denen dieselben Variablen erhoben wurden. Die Bezeichnungen der Zeilen müssen danach ggf. über `rownames()` angepasst werden.

```
> (dfNew <- data.frame(id=13:15, group=c("CG", "WL", "T"),
+                         sex=c("f", "f", "m"), age=c(18, 31, 21),
+                         IQ=c(116, 101, 99), rating=c(4, 4, 1)))
   id sex group age  IQ rating
1  13    f     CG  18 116      4
2  14    f     WL  31 101      4
3  15    m      T  21  99      1

> dfComb <- rbind(myDf1, dfNew); dfComb[11:15, ]
   id sex group age  IQ rating
Z11 11    m     CG  20  92      1
Z12 12    f      T  21  98      1
13 13    f     CG  18 116      4
14 14    f     WL  31 101      4
15 15    m      T  21  99      1
```

Beim Zusammenfügen mehrerer Datensätze besteht die Gefahr, Fälle doppelt aufzunehmen, wenn es Überschneidungen zwischen den Datensätzen hinsichtlich der Beobachtungsobjekte gibt. Die Funktionen `duplicated()` und `unique()` eignen sich dazu, eindeutige bzw. mehrfach vorkommende Zeilen eines Datensatzes zu identifizieren (vgl. Abschn. 2.3.1).

Liegen von denselben Beobachtungsobjekten zwei Datensätze `df1` und `df2` aus unterschiedlichen Variablen vor, können diese analog zum Anhängen einzelner Variablen an einen Datensatz mit `cbind(df1, df2)` oder `data.frame(df1, df2)` so kombiniert werden, dass die Variablen nebeneinander angeordnet sind.

Um Datensätze zusammenzuführen, die sich nur teilweise in den Variablen oder in den Beobachtungsobjekten entsprechen, kann die flexible `merge()` Funktion zum Einsatz kommen.

```
> merge(x=<Datensatz1>, y=<Datensatz2>)
```

Zunächst sei die Situation betrachtet, dass die unter `x` und `y` angegebenen Datensätze Daten derselben Beobachtungsobjekte beinhalten. Dabei sollen einige Variablen bereits in beiden, andere Variablen hingegen nur in jeweils einem der beiden Datensätze vorhanden sein. Die in beiden Datensätzen gleichzeitig enthaltenen Variablen sind dann identisch, da die Daten von denselben Beobachtungsobjekten stammen. Ohne weitere Argumente ist das Ergebnis von `merge()` ein Datensatz, der jede der in `x` und `y` vorkommenden Variablen nur einmal enthält, identische Spalten werden also nur einmal aufgenommen. Zur Identifizierung gleicher Variablen werden die Spaltennamen mittels `intersect(names(x), names(y))` herangezogen.¹⁴

¹⁴ Insbesondere bei Gruppierungsfaktoren ist es wichtig, dass diese Variablen auch in beiden Datensätzen Objekte derselben Klasse (i. d. R. `factor`, dann auch mit denselben Stufen) sind.

```
> (dfa <- data.frame(id=1:4, initials=c("AB", "CD", "EF", "GH"),
+                      IV1=c("-", "-", "+", "+"), DV1=c(10, 19, 11, 14)))
  id initials IV1 DV1
1 1      AB   - 10
2 2      CD   - 19
3 3      EF   + 11
4 4      GH   + 14

> (dfb <- data.frame(id=1:4, initials=c("AB", "CD", "EF", "GH"),
+                      IV2=c("A", "B", "A", "B"), DV2=c(91, 89, 92, 79)))
  id initials IV2 DV2
1 1      AB     A 91
2 2      CD     B 89
3 3      EF     A 92
4 4      GH     B 79

> merge(dfa, dfb)
  id initials IV1 DV1 IV2 DV2
1 1      AB   - 10     A 91
2 2      CD   - 19     B 89
3 3      EF   + 11     A 92
4 4      GH   + 14     B 79
```

Über die Argumente `by.x` und `by.y` kann manuell festgelegt werden, welche Variablen von `x` bzw. von `y` mit Variablen im jeweils anderen Datensatz übereinstimmen und deshalb im Output nur einmal aufgenommen werden sollen. Als Wert für `by.x` und `by.y` muss jeweils ein Vektor mit Namen oder Indizes der übereinstimmenden Spalten eingesetzt werden. Alternativ kann dies auch ein logischer Vektor sein, der für jede Spalte von `x` (im Fall von `by.x`) bzw. jede Spalte von `y` (im Fall von `by.y`) angibt, ob sie eine auch im jeweils anderen Datensatz vorkommende Variable darstellt. Beide Argumente `by.x` und `by.y` müssen gleichzeitig verwendet werden und müssen dieselbe Anzahl von gleichen Spalten bezeichnen. Haben beide Datensätze insgesamt dieselbe Anzahl von Spalten, kann auch auf das Argument `by` zurückgegriffen werden, das sich dann auf die Spalten beider Datensätze gleichzeitig bezieht.

Im Beispiel soll die Spalte der VP-Initialen künstlich als nicht übereinstimmende Variable gekennzeichnet werden, indem das zweite Element des an `by.x` und `by.y` übergebenen Vektors auf `FALSE` gesetzt wird.

```
> merge(dfa, dfb, by.x=c(TRUE, FALSE, FALSE, FALSE),
+        by.y=c(TRUE, FALSE, FALSE, FALSE))
  id initials.x IV1 DV1 initials.y IV2 DV2
1 1      AB   - 10      AB     A 91
2 2      CD   - 19      CD     B 89
3 3      EF   + 11      EF     A 92
4 4      GH   + 14      GH     B 79
```

Berücksichtigt werden bei dieser Art des Zusammenfügens nur jene Zeilen, bei denen die in beiden Datensätzen vorkommenden Variablen identische Werte aufweisen. Hier werden also alle Zeilen entfernt, deren Werte für die gemeinsamen Variablen zwischen den Datensätzen abweichen. Werden mit `by` Spalten als über-

einstimmend gekennzeichnet, für die tatsächlich aber keine Zeile identische Werte aufweist, ist das Ergebnis deshalb ein leerer Datensatz.

```
> (dfC <- data.frame(id=3:6, initials=c("EF", "GH", "IJ", "KL"),
+                      IV2=c("A", "B", "A", "B"), DV2=c(92, 79, 101, 81)))
  id initials IV2 DV2
1 3      EF     A  92
2 4      GH     B  79
3 5      IJ     A 101
4 6      KL     B  81

> merge(dfA, dfC)
  id initials IV1 DV1 IV2 DV2
1 3      EF     +  11   A  92
2 4      GH     +  14   B  79
```

Um das Weglassen solcher Zeilen zu verhindern, können die Argumente `all.x` und `all.y` auf `TRUE` gesetzt werden. `all.x` bewirkt, dass all jene Zeilen in `x`, die auf den übereinstimmenden Variablen andere Werte als in `y` haben, ins Ergebnis aufgenommen werden. Die in `y` (aber nicht in `x`) enthaltenen Variablen werden für diese Zeilen auf `NA` gesetzt. Für das Argument `all.y` gilt dies analog. Das Argument `all=TRUE` steht kurz für `all.x=TRUE` in Kombination mit `all.y=TRUE`.

Im Beispiel sind die Werte bzgl. der übereinstimmenden Variablen in den ersten beiden Zeilen von `dfC` identisch mit jenen in `dfA`. Darüber hinaus enthält `dfC` jedoch auch zwei Zeilen mit Werten, die sich auf den übereinstimmenden Variablen von jenen in `dfA` unterscheiden. Um diese Zeilen im Ergebnis von `merge()` einzuschließen, muss deshalb `all.y=TRUE` gesetzt werden.

```
> merge(dfA, dfC, all.y=TRUE)
  id initials IV1 DV1 IV2 DV2
1 3      EF     +  11   A  92
2 4      GH     +  14   B  79
3 5      IJ    <NA>  NA   A 101
4 6      KL    <NA>  NA   B  81
```

Analoges gilt für das Einschließen der ersten beiden Zeilen von `dfA` im Ergebnis. Diese beiden Zeilen haben andere Werte auf den übereinstimmenden Variablen als die Zeilen in `dfC`. Damit sie im Ergebnis auftauchen, muss `all.x=TRUE` gesetzt werden.

```
> merge(dfA, dfC, all.x=TRUE, all.y=TRUE)
  id initials IV1 DV1 IV2 DV2
1 1      AB     -  10 <NA>  NA
2 2      CD     -  19 <NA>  NA
3 3      EF     +  11   A  92
4 4      GH     +  14   B  79
5 5      IJ    <NA>  NA   A 101
6 6      KL    <NA>  NA   B  81
```

3.2.11 Funktionen auf Variablen anwenden

Wenn separat von jeder Variable eines Datensatzes derselbe Kennwert berechnet werden soll, so ist dies wie bei Matrizen mit `apply()` möglich.

```
> apply(X=<Datensatz>, MARGIN=<Nummer>, FUN=<Funktion>, ...)
```

Für `MARGIN` ist hier der Wert 2 für die Spalten zu vergeben. Zwar lassen sich bei Datensätzen auch über die Zeilen (`MARGIN=1`) Kennwerte berechnen, allerdings ist dies oft nicht ohne weiteres sinnvoll: die Daten einer VP können nämlich von unterschiedlichen, auch nicht numerischen Variablen stammen. Bei gleichzeitigem Vorhandensein von numerischen Werten wie von Zeichenketten in einer Zeile würde `apply()` numerische Werte in Zeichenketten umwandeln, so dass statistische Kennwerte nicht zu berechnen wären. Über die Einschränkung auf numerische Variablen mit `X=<Datensatz>[, <Indizes>]` wäre das Problem aber vermeidbar.

```
> lapply(X=<Liste>, FUN=<Funktion>, ...)
```

`lapply()` (List Apply) verallgemeinert die Funktionsweise von `apply()` auch auf solche Listen, die im Gegensatz zu Objekten der Klasse `data.frame` Komponenten unterschiedlicher Länge besitzen. Hier entfällt die Angabe, ob die Funktion auf Zeilen oder Spalten angewendet werden soll – es sind immer die Spalten, respektive die Variablen. Das Ergebnis von `lapply()` ist eine Liste mit ebenso vielen Komponenten wie sie die Liste `X` enthält.

```
# Mittelwert der numerischen Variablen
> (myList <- lapply(myDf1[ , c(1, 4, 5, 6)], mean))
$id
[1] 6.5

$age
[1] 26.41667

$IQ
[1] 97.25

$rating
[1] 2.583333
```

Die Funktion `sapply()` (Simplified Apply) arbeitet wie `lapply()`, formatiert die Ausgabe der Werte aber etwas übersichtlicher – insbesondere gibt sie keine Liste, sondern einen einfacher zu verarbeitenden Vektor mit benannten Elementen aus. Gibt `FUN` pro Aufruf mehr als einen Wert zurück, ist das Ergebnis eine Matrix, deren Spalten aus diesen Werten gebildet sind.

```
# range der numerische Variablen
> sapply(myDf1[ , c(1, 4, 5, 6)], range)
      id   age    IQ rating
[1,]  1   20   82      0
[2,] 12   35  122      5
```

Durch die Ausgabe eines Vektors eignet sich `sapply()` z. B. dazu, aus einem Datensatz jene Variablen zu extrahieren, die eine bestimmte Bedingung erfüllen – etwa einen numerischen Datentyp besitzen. Der Ergebnisvektor kann später zur Indizierung der Spalten Verwendung finden.¹⁵

```
> sapply(myDf1, is.numeric)
  id  sex group age   IQ rating
TRUE FALSE FALSE TRUE TRUE    TRUE

> dataNum <- myDf1[ , sapply(myDf1, is.numeric)]; dataNum[1:5, ]
  id age   IQ rating
1  1  26 112      1
2  2  30 122      3
3  3  25  95      5
4  4  34 102      5
5  5  22  82      2
```

Eine vereinfachte Form von `sapply()` ist als `replicate()` Funktion verfügbar, die lediglich dafür sorgt, dass derselbe Ausdruck `expr` mehrfach (n mal) wiederholt wird. Die Ausgabe erfolgt als Matrix mit n Spalten und so vielen Zeilen, wie `expr` pro Ausführung Werte erzeugt.

```
> replicate(n=<Anzahl>, expr=<Befehl>)

> replicate(6, round(rnorm(4), 2))
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,] -0.31 -0.17 0.35 -0.46 -1.23 -2.39
[2,]  0.73 -1.12 0.60 -0.61 -0.07  0.58
[3,] -0.16  0.64 0.46 -1.21  0.20 -0.81
[4,]  0.87  0.49 0.04 -1.38  0.05  1.83
```

Mit `do.call()` ist eine etwas andere automatisierte Anwendung von Funktionen auf die Komponenten einer Liste bzw. Variablen eines Datensatzes möglich. Während `apply()` eine Funktion so häufig aufruft, wie Variablen vorhanden sind und dabei jeweils eine Variable als Argument übergibt, geschieht dies bei `do.call()` nur einmal, dafür aber mit mehreren Argumenten.

```
> do.call(what=<Funktionsname>, args=<Liste>)
```

Unter `what` ist die aufzurufende Funktion zu nennen, unter `args` deren Argumente in Form einer Liste, wobei jede Komponente von `args` ein Funktionsargument liefert. Ist von vornherein bekannt, welche und wie viele Argumente `what` erhalten soll, könnte `do.call()` auch durch einen einfachen Aufruf von `<Funktion>(<Liste>[[1]], <Liste>[[2]], ...)` ersetzt werden, nachdem die Argumente als Liste zusammengestellt wurden. Der Vorteil der Konstruktion eines

¹⁵ `sapply()` ist auch für jene Fälle nützlich, in denen auf jedes Element eines Vektors eine Funktion angewendet werden soll, diese Funktion aber nicht vektorisiert ist – d. h. für ein Argument nur einzelne Werte, nicht aber Vektoren akzeptiert. In diesem Fall betrachtet `sapply()` jedes Element des Vektors als eigene Variable, die nur einen Wert beinhaltet.

Funktionsaufrufs aus Funktionsname einerseits und Argumenten andererseits tritt jedoch dann zutage, wenn sich Art oder Anzahl der Argumente erst zur Laufzeit der Befehle herausstellen, etwa weil die Liste selbst erst mit vorangehenden Befehlen dynamisch erzeugt wurde.

Aus einer von `lapply()` zurückgegebenen Liste ließe sich damit wie folgt ein Vektor machen, wie ihn auch `sapply()` zurückgibt:

```
# äquivalent zu
# c(id=myList[[1]], age=myList[[2]], IQ=myList[[3]], rating=myList[[4]])
> do.call("c", myList)
  id      age      IQ   rating
6.500000 26.416667 97.250000 2.583333
```

Sind die Komponenten von `args` benannt, behalten sie ihren Namen bei der Verwendung als Argument für die unter `what` genannte Funktion bei. Damit lassen sich beliebige Funktionsaufrufe samt zu übergebender Daten und weiterer Optionen konstruieren: alle späteren Argumente werden dafür als Komponenten in einer Liste gespeichert, wobei die Komponenten die Namen erhalten, die die Argumente von `what` tragen.

```
> work <- factor(sample(c("home", "office"), 20, replace=TRUE))
> hiLo <- factor(sample(c("hi", "lo"), 20, replace=TRUE))
> group <- factor(sample(c("A", "B"), 20, replace=TRUE))

# Kreuztabelle der Faktoren mit ftable => lege fest, welche in
# Zeilen (row.vars), welche in Spalten (col.vars) stehen sollen
> argLst <- list(fac1=work, fac2=hiLo, fac3=group,
+                  row.vars="fac1", col.vars=c("fac2", "fac3"))

> do.call("ftable", argLst)
    fac2  hi     lo
    fac3 A  B     A  B
fac1
home       2  3    3  1
office      2  4    1  4
```

3.2.12 Funktionen für mehrere Variablen anwenden

`lapply()` und `sapply()` wenden Funktionen auf Variablen eines Datensatzes an, die nur die Daten jeweils einer Variable als Basis für Berechnungen nutzen. Die `mapply()` Funktion verallgemeinert dieses Prinzip auf Funktionen, die aus mehr als einer einzelnen Variable Kennwerte berechnen. Dies ist insbesondere für viele inferenzstatistische Tests der Fall, die etwa in zwei Variablen vorliegende Daten aus zwei Stichproben hinsichtlich verschiedener Kriterien vergleichen.

```
> mapply(FUN=<Funktion>, <Datensatz1>, <Datensatz2>, ...,
+         MoreArgs=<Liste mit Optionen für FUN>)
```

Die anzuwendende Funktion ist als erstes Argument `FUN` zu nennen. Es folgen so viele Datensätze, wie `FUN` Eingangsgrößen benötigt. Im Beispiel einer Funktion für

zwei Variablen verrechnet die Funktion schrittweise zunächst die erste Variable des ersten zusammen mit der ersten Variable des zweiten Datensatzes, dann die zweite Variable des ersten zusammen mit der zweiten Variable des zweiten Datensatzes, etc. Sollen an FUN weitere Argumente übergeben werden, kann dies mit dem Argument MoreArgs in Form einer Liste geschehen.

Im Beispiel soll ein *t*-Test für zwei unabhängige Stichproben für jeweils alle Variablen-Paare zweier Datensätze berechnet werden (vgl. Abschn. 8.2). Dabei soll im *t*-Test eine gerichtete Hypothese getestet (`alternative="less"`) und von Varianzhomogenität ausgegangen werden (`var.equal=TRUE`). Die Ausgabe wird hier verkürzt dargestellt.

```
> x1    <- rnorm(100, 10, 10)          # Variablen für ersten Datensatz
> y1    <- rnorm(100, 10, 10)
> x2    <- x1 + rnorm(100, 5, 4)      # Variablen für zweiten Datensatz
> y2    <- y1 + rnorm(100, 10, 4)
> myDf2 <- data.frame(cbind(x1, y1))
> myDf3 <- data.frame(cbind(x2, y2))
> mapply(t.test, myDf2, myDf3, MoreArgs=list(alternative="less",
+                                              var.equal=TRUE))
      x1
statistic -1.925841
parameter 198
p.value 0.02777827
alternative "less"
method     "Two Sample t-test"          # ...

      y1
statistic -33.75330
parameter 198
p.value 2.291449e-84
alternative "less"
method     "Two Sample t-test"          # ...
```

3.2.13 Funktionen getrennt nach Gruppen anwenden

Um für Variablen eines Datensatzes Kennwerte nicht nur über alle Beobachtungen hinweg, sondern getrennt nach Gruppen zu berechnen, sind die Funktionen `aggregate()` und `by()` vorhanden.

```
> aggregate(x=<Datensatz>, by=<Liste mit Faktoren>, FUN=<Funktion>, ...)
> by(data=<Datensatz>, INDICES=<Liste mit Faktoren>, FUN=<Funktion>, ...)
```

Dabei wird ein Kennwert für die Variablen des unter `x` bzw. `data` anzugebenden Datensatzes berechnet. Die Argumente `by` bzw. `INDICES` kontrollieren, in welche Gruppen die Beobachtungen dabei eingeteilt werden. Anzugeben ist eine Liste, die als Komponenten Gruppierungsfaktoren der Länge `nrow(x)` bzw. `nrow(data)` enthält. Mit dem Argument `FUN` wird die Funktion spezifiziert, die auf die gebildeten Gruppen in jeder Variable angewendet werden soll. `aggregate()` und `by()` ähneln der `tapply()` Funktion, unterscheiden sich jedoch von ihr durch die Gruppenbildung und anschließende Funktionsanwendung auf mehrere Variablen gleichzeitig.

FUN wird auf alle Variablen des übergebenen Datensatzes pro Gruppe angewendet, auch wenn für einzelne von ihnen, etwa Faktoren, die Berechnung numerischer Kennwerte nicht möglich oder nicht sinnvoll ist. Um dies von vornherein zu verhindern, ist der Datensatz auf eine geeignete Teilmenge seiner Variablen zu beschränken.

```
# pro Bedingungskombination: Mittelwert für Alter, IQ und Rating
> aggregate(myDf1[, 4:6], list(myDf1$sex, myDf1$group), mean)
  Group.1 Group.2    age    IQ rating
1       m      CG 25.00 103.00     3
2       f      CG 24.00 113.00     0
3       m        T 31.00  97.00     4
4       f        T 23.50 105.00     1
5       m       WL 27.25  85.25     3
```

Während das Ergebnis von `aggregate()` ein Objekt der Klasse `data.frame` ist, erfolgt die Ausgabe von `by()` als Objekt der Klasse `by`, das im Fall eines einzelnen Gruppierungsfaktors eine Liste, sonst ein Array ist. Der Output wird hier verkürzt dargestellt.

```
# pro Bedingungskombination: Mittelwert für Alter, IQ und Rating
> by(myDf1[, 4:6], list(myDf1$sex, myDf1$group), FUN=mean)
: m, CG
  age    IQ rating
  25   103     3

: f, CG
  age    IQ rating
  24   113     0

: m, T
  age    IQ rating
  31   97     4

: f, T
  age    IQ rating
  3.5 105.0   1.0

: m, WL
  age    IQ rating
 27.25 85.25   3.00
: f, WL
NULL
```

3.2.14 Doppelte und fehlende Werte

Doppelte Werte können in Datensätzen etwa auftreten, nachdem sich teilweise überschneidende Daten aus unterschiedlichen Quellen in einem Datensatz integriert wurden. Mehrfach auftretende Zeilen werden durch `duplicated(⟨Datensatz⟩)` identifiziert und durch `unique(⟨Datensatz⟩)` ausgeschlossen (vgl. Abschn. 2.3.1).

```
# Datensatz mit doppelten Werten herstellen
> myDfDouble <- rbind(myDf1, myDf1[sample(1:nrow(myDf1), 4), ])
> duplicated(myDfDouble) # doppelte Zeilen identifizieren
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[11] FALSE FALSE TRUE TRUE TRUE TRUE
> myDfUnique <- unique(myDfDouble) # doppelte Zeilen ausschließen
```

Fehlende Werte werden in Datensätzen weitgehend wie in Matrizen behandelt (vgl. Abschn. 2.12.4). Statt eines direkten Abgleichs der Variablenwerte mit NA über den == Operator muss auch hier die `is.na()` Funktion benutzt werden, um das Vorhandensein fehlender Werte zu prüfen.

```
> myDfNA <- myDf1
> myDfNA$IQ[4] <- NA
> myDfNA$rating[5] <- NA
> is.na(myDfNA)[1:5, 4:6]
   age   IQ rating
1 FALSE FALSE FALSE
2 FALSE FALSE FALSE
3 FALSE FALSE FALSE
4 FALSE TRUE FALSE
5 FALSE FALSE TRUE

# prüfe jede Variable, ob sie mindestens ein NA enthält
> apply(is.na(myDfNA), 2, any)
   id sex group age IQ rating
FALSE FALSE FALSE TRUE TRUE
```

Eine weitere Funktion zur Behandlung fehlender Werte ist `complete.cases` →(`(Datensatz)`). Sie liefert einen logischen Indexvektor zurück, der für jedes Beobachtungsobjekt (jede Zeile) angibt, ob fehlende Werte vorliegen.¹⁶ Die Fälle mit fehlenden Werten können mit `subset()` ausgegeben werden, wobei der Indexvektor aus der Negation des Ergebnisses von `complete.cases()` gebildet wird.

```
> complete.cases(myDfNA)
[1] TRUE TRUE TRUE FALSE FALSE TRUE TRUE TRUE TRUE TRUE TRUE

> subset(myDfNA, !complete.cases(myDfNA))
   id sex group age IQ rating
4  4   m     T  34 NA      5
5  5   m    WL  22 82     NA
```

Als Alternative können wie bei Matrizen alle Zeilen mit der `na.omit()` Funktion gelöscht werden, in denen nicht vorhandene Werte existieren.

```
> na.omit(myDfNA)[1:5, ]
   id sex group age IQ rating
1  1   f     T  26 112      1
```

¹⁶ Auch `apply(Datensatz, 1, function(x){ all(!is.na(x))})` wäre möglich (vgl. Abschn. 11.1).

2	2	m	CG	30	122	3
3	3	m	CG	25	95	5
6	6	f	CG	24	113	0
7	7	m	T	28	92	3

3.2.15 Datensätze sortieren

Datensätze werden ebenso wie Matrizen mit der `order()` Funktion sortiert (vgl. Abschn. 2.8.10).

```
> order(<Vektor>, na.last=TRUE, decreasing=FALSE)
```

Unter `<Vektor>` ist die Variable (Spalte) eines Datensatzes anzugeben, die in eine Reihenfolge zu bringen ist. `na.last` ist per Voreinstellung auf `TRUE` gesetzt und sorgt ggf. dafür, dass Indizes fehlender Werte zum Schluss ausgegeben werden. Die Voreinstellung `decreasing=FALSE` bewirkt eine aufsteigende Reihenfolge. Zeichenketten werden in alphabetischer Reihenfolge sortiert. Die Reihenfolge bei Faktoren wird dagegen von der Reihenfolge der Bezeichnungen der Stufen bestimmt, die ihrerseits nicht mit der alphabetischen Reihenfolge der Labels übereinstimmen muss (vgl. Abschn. 2.7.4).

`order()` gibt einen Indexvektor aus, der die Zeilenindizes des Datensatzes in der Reihenfolge der zu ordnenden Variablenwerte enthält. Soll der gesamte Datensatz entsprechend der Reihenfolge dieser Variable angezeigt werden, ist der ausgegebene Indexvektor zum Indizieren der Zeilen des Datensatzes zu benutzen.

```
# sortiere myDf1 hinsichtlich der Größe von rating
> (idx1 <- order(myDf1$rating))
[1] 6 1 11 12 5 8 2 7 9 3 4 10
```

```
> myDf1[idx1, ]
   id sex group age  IQ rating
Z6   6   f    CG  24 113     0
Z1   1   f      T  26 112     1
Z11 11   m    CG  20  92     1
Z12 12   f      T  21  98     1
Z5   5   m    WL  22  82     2
Z8   8   m    WL  35  90     2
Z2   2   m    CG  30 122     3
Z7   7   m      T  28  92     3
Z9   9   m    WL  23  88     3
Z3   3   m    CG  25  95     5
Z4   4   m      T  34 102     5
Z10 10   m    WL  29  81     5
```

Soll nach zwei Kriterien sortiert werden, weil die Reihenfolge durch eine Variable noch nicht vollständig festgelegt ist, können weitere Datenvektoren in der Rolle von Sortierkriterien als Argumente an `order()` angegeben werden.

```
# sortiere myDf1 primär nach group und innerhalb jeder Gruppe nach IQ
> (idx2 <- order(myDf1$group, myDf1$IQ))
[1] 11 3 6 2 7 12 4 1 10 5 9 8
```

```
> myDf1[idx2, ]  
   id sex group age  IQ rating  
Z11 11   m    CG  20  92     1  
Z3  3   m    CG  25  95     5  
Z6  6   f    CG  24 113     0  
Z2  2   m    CG  30 122     3  
Z7  7   m    T   28  92     3  
Z12 12   f    T   21  98     1  
Z4  4   m    T   34 102     5  
Z1  1   f    T   26 112     1  
Z10 10   m    WL  29  81     5  
Z5  5   m    WL  22  82     2  
Z9  9   m    WL  23  88     3  
Z8  8   m    WL  35  90     2
```

Kapitel 4

Befehle und Daten verwalten

Für Datenanalysen, die über wenige Teilschritte hinausgehen, ist die interaktive Arbeitsweise, in der sich eingegebene Befehle mit dem von R erzeugten Output auf der Konsole abwechseln, meist nicht sinnvoll. Stattdessen lässt sich die Auswertung automatisieren, indem alle Befehle zunächst zeilenweise in eine als Skript bezeichnete Textdatei geschrieben werden, die dann ihrerseits von R komplett oder in Teilen ausgeführt wird. Analoges gilt für die Verwaltung empirischer Daten: gewöhnlich werden diese nicht von Hand auf der Konsole eingegeben, sondern in separaten Dateien gespeichert – sei es in R, in Programmen zur Tabellenkalkulation oder in anderen Statistikpaketen.

Für viele Befehle ist die Pfadangabes zu einer Datei notwendig, die es zu öffnen oder zu speichern gilt. Obwohl unter Windows üblicherweise der Backslash \ als Verzeichnistrenner in Pfaden Verwendung findet, kann er in R nicht verwendet werden. Stattdessen ist bei allen Pfadangaben bevorzugt der Forward Slash / zu benutzen, etwa "c:/work/r/datei.txt". Als Alternative ist weiterhin der doppelte Backslash \\ möglich: "c:\\work\\r\\datei.txt".

4.1 Befehlssequenzen im Editor bearbeiten

Zum Erstellen eines Befehlsskripts beinhaltet R einen einfachen Texteditor, der sich aus dem Programmfenster mit Datei: Neues Skript öffnen lässt und daraufhin bereit für die Eingabe von Befehlszeilen ist.¹ Mit der Tastenkombination Strg+R wird der Befehl in derjenigen Zeile von R ausgeführt, in der sich der Cursor gerade befindet. Um in einem Schritt mehrere Befehlszeilen auswerten zu lassen, markiert man diese und führt sie ebenfalls mit Strg+R aus. Um das komplette Skript in Gänze ausführen zu lassen, sind demnach alle Zeilen mit Strg+A zu markieren und anschließend mit Strg+R auszuführen. Verursacht einer der auszuführenden Befehle eine Fehlermeldung, unterbricht dies die Verarbeitung, ggf. folgende Befehle

¹ Natürlich können auch andere Texteditoren zum Erstellen einer Skript-Datei verwendet werden. Nur manchen von ihnen stehen aber auch die Funktionen zum zeilenweisen Ausführen der Befehle zur Verfügung, vgl. Abschn. 1.1.1.

werden dann also nicht ausgeführt. Warnungen werden gesammelt am Schluss aller Ausgaben genannt. Befehlsskripte lassen sich im Editor über Datei: Speichern unter speichern und im Programmfenster über Datei: Öffne Skript laden.

In externen Dateien gespeicherte Skripte lassen sich in der Konsole mit `source("<Dateiname>")` einlesen, wobei R die enthaltenen Befehle ausführt. Befindet sich die Skriptdatei nicht im aktiven Arbeitsverzeichnis, muss der vollständige Pfad zum Skript zusätzlich zu seinem Namen mit angegeben werden, z.B. `source("c:/work/r/skript.r").2` Wird das Argument `echo=TRUE` gesetzt, werden die im Skript enthaltenen Befehle selbst mit auf der Konsole ausgegeben, andernfalls erscheint nur die Ausgabe dieser Befehle.

Mit in Textdateien gespeicherten Skripten zu arbeiten bietet u.a. folgende Vorteile:

- Der Überblick über alle auszuführenden Befehle wird erleichtert, zudem können die Auswertungsschritte gedanklich nachvollzogen werden.
- Komplexe Auswertungen lassen sich in kleinere Teilschritte zerlegen. Diese können einzeln nacheinander oder in Teilsequenzen ausgeführt werden, um Zwischenergebnisse auf ihre Richtigkeit zu prüfen.
- Man kann ein einmal erstelltes Skript immer wieder ausführen lassen. Dies ist insbesondere bei der Fehlersuche und bei nachträglichen Veränderungswünschen, etwa an Graphiken, hilfreich: anders als z.B. in Programmen zur Tabellenkalkulation müssen so im Fall von anderen Überschriften oder Achsenkalierungen nicht viele schon bestehende Graphiken mit immer denselben Schritten einzeln geändert werden. Stattdessen reicht es, das Skript einmal zu ändern und neu auszuführen, um die angepassten Graphiken zu erhalten.
- Ein für die Auswertung eines Datensatzes erstelltes Skript lässt sich häufig mit nur geringen Änderungen für die Analyse anderer Datensätze anpassen. Diese Wiederverwendbarkeit einmal geleisteter Arbeit ist bei rein graphisch zu bedienenden Programmen nicht gegeben und spart auf längere Sicht Zeit. Zudem vermeidet eine solche Vorgehensweise Fehler, wenn geprüfte und bewährte Befehlssequenzen kopiert werden können.
- Skripte lassen sich zusammen mit dem Datensatz, für dessen Auswertung sie gedacht sind, an Dritte weitergeben. Neben dem Aspekt der so möglichen Arbeitsteilung kann der Auswertungsvorgang von anderen Personen auf diese Weise genau nachvollzogen und kontrolliert werden. Dies ist im Sinne einer größeren Auswertungsobjektivität sinnvoll.³

² Wird das einzulesende Skript nicht gefunden, ist zunächst mit `dir()` zu prüfen, ob das von R durchsuchte Verzeichnis (ohne Pfadangabe das mit `getwd()` angezeigte Arbeitsverzeichnis) auch tatsächlich das Skript enthält. Es ist nicht notwendig, R im interaktiven Modus zu starten, um ein Befehlsskript ausführen zu lassen, dafür ist auch der sog. Batch-Modus ausreichend: befindet man sich unter Windows in der Eingabeaufforderung im Verzeichnis mit der ausführbaren Datei `Rterm.exe`, lautet dazu der Befehl `Rterm.exe --no-restore --no-save < R-Skriptdatei.r > Ausgabedatei.txt`.

³ Da R mit `shell()` auch auf Funktionen des Betriebssystems zugreifen kann, sollten aus Sicherheitsgründen nur geprüfte Skripte aus vertrauenswürdiger Quelle ausgeführt werden.

- Zudem lassen sich von R nicht als Befehl interpretierte Kommentare einfügen, z. B. um die Bedeutung der Befehlssequenzen zu erläutern. Kommentare sind dabei alle Zeilen bzw. Teile von Zeilen, die mit einem # Symbol beginnen. Ihre Verwendung ist empfehlenswert, damit auch andere Personen schnell erfassen können, was Befehle bedeuten oder bewirken sollen. Aber auch für den Autor des Skripts selbst sind Kommentare hilfreich, wenn es längere Zeit nach Erstellen geprüft oder für eine andere Situation angepasst werden soll.

4.2 Daten importieren und exportieren

Empirische Daten können auf verschiedenen Wegen in R verfügbar gemacht werden. Zunächst ist es möglich, Werte durch Zuweisungen etwa in Vektoren zu speichern und diese dann zu Datensätzen zusammenzufügen. Bequemer und übersichtlicher ist die Benutzung des in R integrierten Dateneditors (vgl. Abschn. 4.2.2). Häufig liegen Datensätze aber auch in Form von mit anderen Programmen erstellten Dateien vor (vgl. Abschn. 4.2.5). R bietet die Möglichkeit, auf verschiedenste Datenformate zuzugreifen und in diesen auch wieder Daten abzulegen. Immer sollte dabei überprüft werden, ob die Daten auch tatsächlich korrekt transferiert wurden. Zudem empfiehlt es sich, nie mit den Originaldaten selbst zu arbeiten. Stattdessen sollten immer nur Kopien des Referenz-Datensatzes verwendet werden, um diesen gegen unbeabsichtigte Veränderungen zu schützen. Dateneingabe sowie der Datenaustausch mit anderen Programmen werden vertieft im Manual „R Data Import/Export“ (R Development Core Team, 2009c) sowie von Muenchen (2008) behandelt.

4.2.1 Daten in der Konsole einlesen

Während es mit der bisher verwendeten Methode, mittels `c(<Wert1>, <Wert2>, ...)` Vektoren aus Werten zu bilden, zwar möglich ist, ganze Datensätze in R einzugeben, ist dieses Vorgehen aufgrund der ebenfalls einzutippenden Kommata nicht sehr effizient. Etwas schneller ist die Dateneingabe mit dem `scan()` Befehl, bei dem nur das Leerzeichen als Trennzeichen zwischen den Daten vorhanden sein muss.

```
> scan(file="", what="numeric", na.strings="NA", dec=".")
```

Sollen Daten manuell auf der Konsole eingegeben werden, ist das Argument `file` bei der Voreinstellung "" zu belassen. Mit `scan()` können auch Dateien eingelesen werden, allerdings ist dies bequemer über andere Funktionen möglich (vgl. Abschn. 4.2.3). Das Argument `what` benötigt eine Angabe der Form "logical", "numeric" oder "character", die Auskunft über den Datentyp der folgenden Werte gibt. Zeichenketten müssen durch die Angabe "character" nicht mehr in Anführungszeichen eingegeben werden, es sei denn sie beinhalten Leerzeichen. Mit `na.strings` wird festgelegt, auf welche Weise fehlende Werte codiert sind.

Das Dezimaltrennzeichen der folgenden Werte kann über das Argument `dec` definiert werden.

Beim Aufruf ist das Ergebnis von `scan()` einem Objekt zuzuweisen, damit die folgenden Daten auch gespeichert werden. Auf den Befehlaufruf `scan()` hin erscheint in der Konsole eine neue Zeile als Signal dafür, dass nun durch Leerzeichen getrennt Werte eingegeben werden können. Eine neue Zeile wird dabei durch Drücken der Return Taste begonnen und zeigt in der ersten Spalte an, der wievielte Wert folgt. Die Eingabe der Werte gilt als abgeschlossen, wenn in einer leeren Zeile die Return Taste gedrückt wird.

```
> vec <- scan()
1: 123 456 789
4:
Read 3 items

> charVec <- scan(what="character")
1: as df ej kl
5:
Read 4 items
```

4.2.2 Daten im Editor eingeben

Bereits bestehende Datensätze oder einzelne Variablen können über den in R integrierten Dateneditor geändert werden, der mit `edit(<Objekt>)` aufgerufen wird. Innerhalb des Editors können Zellen mit der Maus ausgewählt und dann durch entsprechende Tastatureingaben mit Werten gefüllt werden – eine leere Zelle steht dabei für einen fehlenden Wert. Ebenso lassen sich durch einen Klick auf die Spaltenköpfe Name und Datentyp der Variablen neu festlegen. Während der Dateneditor geöffnet ist, bleibt die Konsole für Eingaben blockiert. Beim Schließen des Dateneditors liefert `edit()` als Rückgabewert das Objekt mit allen ggf. geänderten Werten. Wichtig ist, dieses zurückgelieferte Objekt dem zu ändernden Objekt beim Aufruf zuzuweisen, damit die Änderungen auch gespeichert werden. Wurde dies vergessen, ist der geänderte Datensatz noch als `.Last.value` vorhanden, solange kein neuer Output erzeugt wird.

Der Befehl `fix(<Objekt>)` ähnelt `edit()`, fasst das Speichern des bearbeiteten Objekts aber bereits mit ein und ist daher meist zu bevorzugen. Aus diesem Grund lassen sich mit `fix()` auch keine neuen Datensätze erstellen, wie dies mit `edit(data.frame())` über den Umweg eines verschachtelt im Aufruf erzeugten leeren Datensatzes möglich ist.

```
> myDf  <- data.frame(IV=factor(rep(c("A", "B"), 5)), DV=rnorm(10))
> myDf  <- edit(myDf)          # Zuweisung des Ergebnisses erforderlich
> fix(myDf)                  # Zuweisung nicht erforderlich
> newDf <- edit(data.frame()) # erzeuge leeren Datensatz
```

4.2.3 Im Textformat gespeicherte Daten

Für jeden Import in R sollten Daten so organisiert sein, dass sich die Variablen in den Spalten und die Werte jeweils eines Beobachtungsobjekts in den Zeilen befinden. Für alle Variablen sollten gleich viele Beobachtungen vorliegen, so dass sich insgesamt eine rechteckige Datenmatrix ergibt. Bei fehlenden Werten ist es am günstigsten, sie konsistent mit einem expliziten Code zu kennzeichnen, der selbst kein möglicher Wert ist. Weiter ist darauf zu achten, dass Variablennamen den R-Konventionen entsprechen und beispielsweise kein #, %, oder Leerzeichen enthalten (vgl. Abschn. 1.3.1).

Mit der Funktion `read.table()` werden in Textform vorliegende Daten geladen und in einem Objekt der Klasse `data.frame` ausgegeben. Wichtige Argumente von `read.table()` sind in Tabelle 4.1 dargestellt.

Für das Argument `file` können nicht nur lokal gespeicherte Dateien angegeben werden: die Funktion liest mit `file="clipboard"` auch Werte aus der Zwi-

Tabelle 4.1 Wichtige Argumente von `read.table()`

Argument	Bedeutung
<code>file</code>	(ggf. Pfad und) Name der einzulesenden Quelle bzw. des zu schreibenden Ziels (meist eine Datei), in Anführungszeichen gesetzt ⁴
<code>header</code>	Wenn in der einzulesenden Quelle Spaltennamen vorhanden sind, muss <code>header=TRUE</code> gesetzt werden (Voreinstellung ist <code>FALSE</code>)
<code>sep</code>	Trennzeichen zwischen zwei Spalten in <code>file</code> . Voreinstellung sind ein oder mehrere aufeinander folgende nicht gedruckte Zeichen (sog. Whitespace: Leerzeichen und Tabulatorzeichen). Andere häufig verwendete Werte sind das Komma (<code>sep=","</code>) oder das Tabulatorzeichen (<code>sep="\t"</code>)
<code>dec</code>	Das in der Datei verwendete Dezimaltrennzeichen, Voreinstellung ist der Punkt (<code>dec=". "</code>)
<code>colClasses</code>	Vektor, der für jede Spalte der einzulesenden Quelle den Datentyp angibt, z. B. <code>c("numeric", "logical", ...)</code> . Mit <code>NULL</code> können Spalten auch übersprungen, also vom Import ausgeschlossen werden, es müssen aber für alle Spalten Angaben vorhanden sein. Ohne Angabe von <code>colClasses</code> bestimmt R selbst die Datentypen, was bei großen Datenmengen langsamer ist
<code>na.strings</code>	Vektor mit den zur Codierung fehlender Werte verwendeten Zeichenketten. Voreinstellung ist <code>"NA"</code>
<code>stringsAsFactors</code>	Variablen mit Zeichenketten als Werten werden automatisch in Gruppierungs faktoren (<code>factor</code>) konvertiert (Voreinstellung <code>TRUE</code>). Sollen solche Variablen als <code>character</code> Vektoren gespeichert werden, ist das Argument auf <code>FALSE</code> zu setzen (vgl. Abschn. 3.2.1)

⁴ Werden die einzulesenden Daten von R nicht gefunden, ist zunächst mit `dir()` zu prüfen, ob das von R durchsuchte Verzeichnis (ohne Pfadangabe das mit `getwd()` angezeigte Arbeitsverzeichnis) auch jenes ist, das die Datei enthält.

schenablage, die dort in einem anderen Programm etwa mit Strg+C hineinkopiert wurden. Ebenso liest sie wie `scan()` Daten von der Konsole, wenn `file=stdin()` verwendet wird. Online verfügbare Dateien können mit `file="\(URL\)"` direkt von einem Webserver geladen werden. Anders als in Webbrowsersn muss dabei der Protokollteil der Adresse (`http://` oder `ftp://`) explizit genannt werden, also z. B. `file="http://www.server.de/datei.txt"`.⁵

```
> (xDf <- read.table(file=stdin(), header=TRUE))
0: id group rating
1: 1 A 3
2: 2 A 1
3: 3 B 5
4:
  id group rating
1 1     A     3
2 2     A     1
3 3     B     5
```

Zum Speichern von Objekten in Textdateien dient die Funktion `write.table()`.

```
> write.table(x=(Objekt), file="\(Dateiname\)", sep=" ", dec=".",
+               row.names=TRUE, col.names=TRUE, quote=TRUE)
```

Die Funktion akzeptiert als Argumente u. a. `file`, `sep` und `dec` mit derselben Bedeutung wie bei `read.table()`. Statt in eine Datei kann `write.table()` mit `file="clipboard"` Daten auch in die Zwischenablage schreiben, woraufhin sie in anderen Programmen mit Strg+V eingefügt werden können. Über die Argumente `row.names` und `col.names` wird festgelegt, ob Zeilen- und Spaltennamen mit in die Datei geschrieben werden sollen (Voreinstellung für beide ist `TRUE`). Zeichenketten werden in der Ausgabe in Anführungszeichen gesetzt, sofern nicht das Argument `quote=FALSE` gesetzt wird.

Wenn z. B. der Datensatz `myDf` im aktuellen Arbeitsverzeichnis in Textform abgespeichert und später wieder eingelesen werden soll, so lauten die Befehle:

```
> write.table(myDf, file="data.txt")
> myDf <- read.table("data.txt", header=TRUE)
> str(myDf)
'data.frame': 10 obs. of 2 variables:
 $ IV: Factor w/ 2 levels "A","B": 1 2 1 2 1 2 1 2 1 2
 $ DV: num 0.425 -1.224 -0.572 -0.738 -1.753 ...
```

Das von `read.table()` ausgegebene Objekt besitzt die Klasse `data.frame`, selbst wenn mit `write.table()` eine Matrix gespeichert wurde. Ist dies unerwünscht, muss der Datensatz explizit z. B. mit `as.matrix(Objekt)` in eine andere Klasse konvertiert werden.

⁵ Statt einer Datei akzeptieren die meisten Funktionen für das Argument `file` allgemein eine sog. Connection, bei der es sich etwa auch um die Verbindung zu einem Vektor von Zeichenketten handeln kann, vgl. `?textConnection` und `?connections`.

4.2.4 R-Objekte

Eine andere Möglichkeit zur Verwaltung von Objekten in externen Dateien bieten die Funktionen `save(<Daten>, file=<Dateiname>)` zum Speichern und `load(file=<Dateiname>)` zum Öffnen. Unter `<Daten>` können dabei verschiedene Objekte durch Komma getrennt angegeben werden. Die Daten werden in einem R-spezifischen Format gespeichert, bei dem Namen und Klassen der gespeicherten Objekte erhalten bleiben. Deshalb ist es nicht notwendig, das Ergebnis von `load()` einem Objekt zuzuweisen; die gespeicherten Objekte werden unter ihrem Namen wiederhergestellt. Die `save.image(file=".RData")` Funktion speichert alle Objekte des aktuellen Workspaces.

```
> save(myDf, file="data.RData")      # speichere myDf im Arbeitsverzeichnis
> load("data.RData")              # lese myDf wieder ein
```

Ähnlich wie `save()` Objekte in einem binären Format speichert, schreibt `dump("<Objekt>", file=<Dateiname>)` die Inhalte von Objekten in eine Textdatei, die sich auch durch gewöhnliche Texteditoren bearbeiten lässt. Auf diese Weise erzeugte Dateien lassen sich mit `source(file=<Dateiname>)` einlesen. Im Gegensatz zur universelleren `save()` Funktion ist die Anwendung von `dump()` auf Objekte einfacher Klassen beschränkt.

```
> dump("myDf", file="dumpMyDf.txt")
> source("dumpMyDf.txt")
```

4.2.5 Daten mit anderen Programmen austauschen

Wenn Daten mit anderen Programmen ausgetauscht werden sollen – etwa weil die Dateneingabe nicht in R stattgefunden hat, so ist der Datentransfer oft in Form von reinen Textdateien möglich. Diese Methode ist auch recht sicher, da sich die Daten jederzeit mit einem Texteditor inspizieren lassen und der korrekte Transfer in allen Stufen kontrolliert werden kann. In diesem Fall kommen in R meist die Funktionen `read.table()` und `write.table()` zum Einsatz.

Beim Import- und Export von Daten in Dateiformate kommerzieller Programme besteht dagegen oft die Schwierigkeit, dass die Dateiformate nicht öffentlich dokumentiert und auch versionsabhängigen Änderungen unterworfen sind. Wie genau Daten aus diesen Formaten gelesen und geschrieben werden können, ist deshalb mitunter für die Entwickler der entsprechenden R-Funktionen nicht mit Sicherheit zu ermitteln. Beim Austausch von Daten über proprietäre Formate ist aus diesem Grund Vorsicht geboten – bevorzugt sollten einfach strukturierte Datensätze verwendet werden.

4.2.5.1 Programme zur Tabellenkalkulation

Wurde ein Programm zur Tabellenkalkulation (etwa Microsoft Excel oder Open-Office Calc) zur Dateneingabe benutzt, so ist der Datentransfer am einfachsten, wenn die Daten von dort in eine Textdatei exportiert werden, wobei als Spalten-

Trennzeichen der Tabulator verwendet wird.⁶ Dezimaltrennzeichen ist in Programmen zur Tabellenkalkulation für Deutschland das Komma.⁷ Um eine mit diesen Einstellungen exportierte Datei mit Spaltennamen in der ersten Zeile in R zu laden, wäre ein geeigneter Aufruf von `read.table()`:

```
> myDf <- read.table(file="Dateiname", header=TRUE, sep="\t", dec=",")
```

Programme zur Tabellenkalkulation verwenden in der Voreinstellung meist den Tabulator als Spaltentrennzeichen, ein Austausch kleinerer Datenmengen ohne Umweg über eine externe Datei ist also auch wie folgt möglich: zunächst wird in der Tabellenkalkulation der gewünschte Datenbereich inkl. der Variablennamen in der ersten Zeile markiert und mit Strg+C in die Zwischenablage kopiert. In R können die Daten dann so eingelesen werden:

```
> myDf <- read.table(file="clipboard", header=TRUE, sep="\t", dec=",")
```

Um einen Datensatz aus R heraus wieder einem anderen Programm verfügbar zu machen, wird er in demselben Format gespeichert – entweder in einer Datei oder in der Zwischenablage. Im anderen Programm können die Daten aus der Zwischenablage dann mit Strg+V eingefügt werden.

```
> write.table(x=Datensatz, file="clipboard", sep="\t", dec=",",
+               row.names=FALSE)
```

Für Excel stellt RExcel ein sog. Add-In zur Verfügung, das dafür sorgt, dass R-Funktionen direkt aus Excel heraus benutzbar sind (Heiberger und Neuwirth, 2009). Zudem ermöglicht es einen Datenaustausch ohne Umweg eines Exports ins Textformat, indem es in R Funktionen zum Lesen und Schreiben von [Dateiname.xls](#) Dateien bereitstellt. Das Add-In unterstützt Excel 2007 und ist in R über das Paket RExcelInstaller (Neuwirth et al., 2010) installierbar. Um Excel-Dateien in R zu verwenden, eignet sich auch das Paket xlsReadWrite (Suter, 2010), das die Funktionen `read.xls()` und `write.xls()` enthält, das Dateiformat `xlsx` aus Excel 2007 derzeit aber nicht unterstützt.

Mit R.matlab (Bengtsson und Riedy, 2009) existiert ein ähnliches Paket mit Funktionen, die es erlauben, MATLAB-Dateien im MAT Format zu lesen wie auch zu schreiben.

4.2.5.2 SPSS, Stata und SAS

Ab Version 16 verfügt SPSS über das sog. SPSS Statistics-R Integration Plugin, mit dem R-Befehle direkt in SPSS ausgewertet werden können. Auf diese Weise lassen sich dort nicht nur in R verwaltete Datensätze nutzen, sondern auch ganze

⁶ Auf diese Weise können bei Microsoft Excel-Versionen älter als Office 2007 maximal 256 Spalten exportiert werden. Darüber hinaus ist der Transfer von Datumsangaben mit dieser Methode fehlerträchtig.

⁷ Sofern dies nicht in den Ländereinstellungen der Windows Systemsteuerung geändert wurde.

Auswertungsschritte bis hin zur Erstellung von Diagrammen (ab Version 17) in R-Syntax durchführen. Genauso erlaubt es das Plugin, mit SPSS erstellte Datensätze im R-Format zu exportieren. Einige der im folgenden beschriebenen Einschränkungen, die andere Methoden des Datenaustauschs mit sich bringen, existieren für das Plugin nicht, was es zur bevorzugten Methode der gemeinsamen Verwendung von SPSS und R macht.⁸

SPSS-Datensätze können in R mit Funktionen gelesen und geschrieben werden, die das Paket `foreign` (R core members et al., 2010) bereitstellt. Hierfür ist es empfehlenswert, Datensätze durch SPSS im sog. Portable Format in Dateien der Endung `.por` abzuspeichern – der Import dieses Dateiformats ist mitunter unproblematischer als jener von `<Dateiname>.sav` Dateien. Mit der `read.spss()` Funktion können beide Dateitypen eingelesen werden.⁹

```
> read.spss(file="Dateiname", to.data.frame=FALSE, trim.factor.names=FALSE)
```

In der Voreinstellung ist das Ergebnis ein Objekt der Klasse `list`, was durch das Argument `to.data.frame=TRUE` geändert werden kann. Mit `trim.factor.names=TRUE` wird erreicht, dass Bezeichnungen von Faktorstufen auf ihre tatsächliche Länge gekürzt werden – andernfalls würden sie jeweils 256 Zeichen umfassen. Wurden in SPSS auch Labels für die Variablen vergeben, tauchen diese nach dem Import als Vektor im Attribut `variable.labels` des erstellten Objekts auf und können etwa über `attr(<Objekt>, "variable.labels")` gelesen und verändert werden.

Sollen in R bearbeitete Datensätze SPSS verfügbar gemacht werden, ist die durch das `foreign` Paket bereitgestellte `write.foreign()` Funktion zu benutzen.

```
> write.foreign(df=<Datensatz>, datafile="Dateiname",  
+                 codefile="Dateiname", package="SPSS")
```

Hierbei ist unter `datafile` der Name der Textdatei anzugeben, in der sich die eigentlichen Daten befinden sollen, während der für `codefile` einzutragende Name die SPSS Syntax-Datei mit der Endung `sps` mit jenen Befehlen benennt, die in SPSS zum Einlesen dieser Daten dienen. Der erste von R in diese Datei geschriebene Befehl bezeichnet dabei den Namen der Daten-Datei – häufig empfiehlt es sich, ihm im Syntax-Editor von SPSS den vollständigen Dateipfad voranzustellen, damit SPSS die Datei in jedem Fall findet. Zudem kann SPSS Textdateien einlesen, wie sie mit `write.table(..., row.names=FALSE, sep="\t", dec=",")` erstellt werden.

Beim Import von Daten in SPSS ist darauf zu achten, dass die Variablen letztlich das richtige Format (numerisch oder Text) sowie den richtigen Skalentyp (nominal, ordinal oder metrisch) erhalten. Weiterhin orientiert sich SPSS ab Version 16 in seiner Wahl des Dezimaltrennzeichens nicht mehr an den

⁸ Für einen detaillierten Vergleich der Arbeit mit R, SAS und SPSS vgl. Muenchen (2008), in dem auch der Datenaustausch zwischen den Programmen behandelt wird.

⁹ Die Funktion `spss.get()` aus dem Paket `Hmisc` verwendet `read.spss()` mit geeigneteren Voreinstellungen, verbessert außerdem den Import von Datumsangaben und Variablen-Labels.

Windows-Ländereinstellungen, sondern macht es von einer internen Ländereinstellung (sog. LOCALE) abhängig. Nach Möglichkeit sollten Daten also bereits mit jenem Dezimaltrennzeichen exportiert werden, das SPSS erwartet. Wo dies nicht möglich ist, lässt sich die Ländereinstellung in SPSS vor dem Import mit dem Befehl `SET LOCALE='German'`. so umschalten, dass ein Komma als Dezimaltrennzeichen gilt und mit `SET LOCALE='English'`. so, dass dies der Punkt ist. Sollten Umlaute beim Import Probleme bereiten, könnten sie bereits in R mit Befehlen wie `<Variable> <- gsub("ä", "ae", (Variable))` ersetzt werden. Andernfalls erläutert die SPSS Hilfe zu den allgemeinen Optionen, wie Daten- und Syntaxdateien in unterschiedlichen Zeichencodierungen eingelesen werden können.

Neben dem Datenaustausch mit SPSS gibt es auch die Möglichkeit, Daten mit Stata zu teilen, wofür ebenfalls Funktionen aus dem `foreign` Paket dienen: `read.dta()` liest `(Dateiname).dta` Dateien, `write.foreign(..., package="Stata")` schreibt sie. Der Austausch mit SAS geschieht analog über `read.xport()` und `write.foreign(..., package="SAS")`. Das `Hmisc` Paket stellt mit `stata.get()` und `sasxport.get()` Funktionen mit geeigneteren Voreinstellungen zum Lesen von Dateien dieser Programme bereit.

4.2.5.3 Datenbanken

In R lassen sich Daten aus Datenbanken vieler verschiedener Formate direkt lesen. Dabei muss zunächst eine Verbindung zur Datenbank hergestellt werden, woraufhin sich SQL-Kommandos wie `fetch` und `query` in der üblichen Syntax anwenden lassen. Dies bietet sich besonders bei extrem großen Datensätzen an, die zuviel Arbeitsspeicher belegen würden, wenn man sie als Ganzes in R öffnen wollte (vgl. Abschn. 1.1.1, Fußnote 3). Geeignete Funktionen werden für Datenbanken im ODBC-Format (dies schließt Excel-Dateien im Format Office 97-2007 mit ein¹⁰) vom Paket `RODBC` bereitgestellt (Ripley, 2009). Auch für MySQL, Oracle, SQLite Datenbanken und die Java Datenbankschnittstelle existieren ähnliche Pakete (Ripley, 2001).

Im Beispiel soll eine Datenbankverbindung zur Excel-Datei `data.xls` geöffnet werden, die u. a. das Tabellenblatt `table1` enthält. In diesem Tabellenblatt befinden sich drei Spalten mit einem Variablenamen in der ersten und Daten von fünf Beobachtungen in den folgenden Zeilen.

```
> library(RODBC)
> xlsCon <- odbcConnectExcel("data.xls")      # Datenbankverbindung öffnen
> odbcGetInfo(xlsCon)                         # Verbindungsinformationen ...
> sqlTables(xlsCon)                           # Tabellenblätter auflisten (Spalte TABLE_NAME)
   TABLE_CAT TABLE_SCHEMA TABLE_NAME TABLE_TYPE REMARKS
1 (Pfad xls Datei)\data          <NA>       table1$    SYSTEM TABLE  <NA>
2 (Pfad xls Datei)\data          <NA>       table2$    SYSTEM TABLE  <NA>
3 (Pfad xls Datei)\data          <NA>       table3$    SYSTEM TABLE  <NA>

> (myDfXls <- sqlFetch(xlsCon, "table1"))      # table1 speichern
```

¹⁰ Diese Möglichkeit setzt voraus, dass ein ODBC-Treiber unter Windows installiert ist.

Kapitel 5

Hilfsmittel für die Inferenzstatistik

Bevor in den kommenden Kapiteln Funktionen zur inferenzstatistischen Datenanalyse vorgestellt werden, ist es notwendig Hilfsmittel bereitzustellen, auf die viele dieser Funktionen zurückgreifen. Dies sind im wesentlichen die Syntax zur Formulierung linearer Modelle sowie einige Familien statistischer Verteilungen von Zufallsvariablen, die bereits bei der Erstellung zufälliger Werte in Erscheinung getreten sind (vgl. Abschn. 2.5.2).

5.1 Lineare Modelle formulieren

Manche Funktionen in R erwarten als Argument die symbolische Formulierung eines linearen statistischen Modells, dessen Passung für die zu analysierenden Daten getestet werden soll. Ein solches Modell besitzt die Klasse `formula` und beschreibt, wie der systematische Anteil von Werten einer AV aus Werten einer oder mehrerer UVn theoretisch hervorgeht.¹ Die Annahme der prinzipiellen Gültigkeit eines linearen Modells über das Zustandekommen von Variablenwerten steht hinter vielen statistischen Verfahren, etwa der linearen Regression oder Varianzanalysen. Ein Formelobjekt wird wie folgt erstellt:

```
> (modellierte Variable) ~ (lineares Modell)
```

Vor der Tilde `~` steht die Variable, deren systematischer Anteil sich laut Modellvorstellung aus anderen Variablen ergeben soll. Die Variable wird im konkreten Fall durch einen Vektor mit Werten dieser Variable angegeben. Die modellierenden Variablen werden in Form einzelner Terme hinter der `~` aufgeführt. Im konkreten Fall werden für diese Terme Datenvektoren oder Faktoren derselben Länge wie der des Vektors der modellierten Variable eingesetzt.

¹ Im Sinne des Allgemeinen Linearen Modells beschreibt die Formel die Spalten der Designmatrix, die `model.matrix(lm(<Formel>))` für ein konkretes Modell ausgibt. Bei einem multivariaten Modell können auch mehrere AVn vorhanden sein (vgl. Kap. 9).

Im Modell einer einfachen linearen Regression (vgl. Kap. 7) sollen sich etwa die Werte des Kriteriums aus Werten des quantitativen Prädiktors ergeben, hier hat die Formel also die Form $\langle \text{Kriterium} \rangle \sim \langle \text{Prädiktor} \rangle$. In der Varianzanalyse (vgl. Abschn. 8.3) hat die AV die Rolle der modellierten und die kategorialen UVn die Rolle der modellierenden Variablen. Hier hat die Formel die Form $\langle \text{AV} \rangle \sim \langle \text{UV} \rangle$. Um R die Möglichkeit zu geben, beide Fälle zu unterscheiden, müssen im Fall der Regression die Prädiktoren numerische Vektoren sein, die UVn in der Varianzanalyse dagegen Objekte der Klasse `factor`.

Es können mehrere, in der Formel durch + getrennte Vorhersageterme als systematische Varianzquellen der AV in ein statistisches Modell eingehen. Ein einzelner Vorhersageterm kann dabei entweder aus einer Variable oder aber aus der Kombination von Variablen im Sinne eines statistischen Interaktionsterms bestehen. Die Beziehung zwischen den zu berücksichtigenden Variablen wird durch Symbole ausgedrückt, die sonst numerische Operatoren darstellen, in einer Formel aber eine andere Bedeutung tragen.² An Möglichkeiten, Variablen in einem Modell zu berücksichtigen, gibt es u. a. die in Tabelle 5.1 aufgeführten.

Als Beispiel gebe es in einer multiplen linearen Regression ein Kriterium Y und drei Prädiktoren X_1, X_2, X_3 . In der Regression können neben den additiven Effekten der Prädiktoren auch ihre Interaktionsterme berücksichtigt werden. Zudem beinhaltet das Modell i. d. R. einen absoluten Term (den y-Achsenabschnitt der Vorhersagegeraden im Fall der einfachen linearen Regression), der aber auch

Tabelle 5.1 Formelnotation für lineare Modelle

Operator	übliche Bedeutung	Bedeutung in einer Formel
+	Addition	den folgenden Vorhersageterm hinzufügen
-	Subtraktion	den folgenden Vorhersageterm ausschließen
$\langle A \rangle : \langle B \rangle$	Sequenz	Interaktion von $\langle A \rangle$ und $\langle B \rangle$ als Vorhersageterm
$\langle A \rangle * \langle B \rangle$	Multiplikation	alle additiven Effekte und Interaktionseffekt(e) von $\langle A \rangle$ und $\langle B \rangle$
\wedge	potenzieren	Begrenzung des Grads zu berücksichtigender Interaktionen
$\langle A \rangle \%in\% \langle B \rangle$	Element von Menge	Verschachtelung von $\langle A \rangle$ in $\langle B \rangle$ (genestetes Design): Interaktion $\langle A \rangle : \langle B \rangle$ als Vorhersageterm
$\langle A \rangle / \langle B \rangle$	Division	Verschachtelung von $\langle A \rangle$ in $\langle B \rangle$ (genestetes Design): $\langle A \rangle$ und $\langle A \rangle : \langle B \rangle$ als Vorhersageterme
1	1	absoluter Term (Gesamterwartungswert). Implizit vorhanden, wenn nicht durch -1 ausgeschlossen
.		bei Veränderung: alle bisherigen Terme

² Ihre Bedeutung folgt der sog. Wilkinson-Rogers-Notation.

unterdrückt werden kann. Im folgenden Beispiel seien alle Variablen numerische Vektoren.

```
> Y ~ X1 - 1          # einfache lineare Regression von Y auf X1
# ohne absoluten Term (y-Achsenabschnitt)
> Y ~ X1 + X2 + X3  # multiple Regression von Y auf X1, X2 und X3
> Y ~ X1 + X2 + X1:X2 # Regression von Y auf X1 und X2 sowie den
# Interaktionsterm von X1 und X2
> Y ~ X1*X2          # dieselbe Bedeutung
> Y ~ X1*X2*X3      # Regression von Y auf alle additiven Effekte
# sowie alle Interaktionseffekte, also:
# Y ~ X1+X2+X3 + X1:X2 + X1:X3 + X2:X3 + X1:X2:X3
> Y ~ (X1 + X2 + X3)^2 # Regression von Y auf alle additiven Effekte
# sowie alle Interaktionseffekte bis zum 2. Grad:
# Y ~ X1 + X2 + X3 + X1:X2 + X1:X3 + X2:X3
```

Die sich ergebenden Vorhersageterme einer Formel sowie weitere Informationen zum Modell können mit der Funktion `terms(<Formel>)` erfragt werden, deren Output hier gekürzt folgt.

```
> terms(Y ~ X1*X2*X3)          # ...
[1] "X1" "X2" "X3" "X1:X2" "X1:X3" "X2:X3" "X1:X2:X3"
> terms(Y ~ (X1 + X2 + X3)^2 - X1 - X2:X3)          # ...
[1] "X2" "X3" "X1:X2" "X1:X3"
```

Innerhalb einer Formel können die Terme selbst das Ergebnis der Anwendung von Funktionen auf Variablen sein. Soll etwa nicht Y als Kriterium durch X als Prädiktor vorhergesagt werden, sondern der Logarithmus von Y durch den Betrag von X , lautet die Formel $\log(Y) \sim \text{abs}(X)$. Sollen hierbei innerhalb einer Formel Operatoren in ihrer arithmetischen Bedeutung zur Transformation von Variablen verwendet werden, muss der entsprechende Term in die Funktion `I(<Transformation>)` eingeschlossen werden. Um etwa das Doppelte von X als Prädiktor für Y zu verwenden, lautet die Formel damit $Y \sim I(2*X)$.

5.2 Funktionen von Zufallsvariablen

Mit **R** lassen sich die Werte von häufig benötigten Dichte- bzw. Wahrscheinlichkeitsfunktionen, Verteilungsfunktionen und deren Umkehrfunktionen an beliebiger Stelle bestimmen. Dies erübrigt es u. a., etwa für p -Werte oder kritische Werte Tabellen konsultieren zu müssen und bei nicht tabellierten Werten für Wahrscheinlichkeiten oder Freiheitsgrade zwischen angegebenen Werten zu interpolieren. Tabelle 5.2 gibt Auskunft über einige der hierfür verfügbaren Funktionsfamilien sowie ihre Argumente und deren Voreinstellungen. Für ihre Verwendung zur Erzeugung von Zufallszahlen vgl. Abschn. 2.5.2.

Tabelle 5.2 Vordefinierte Funktionen von Zufallsvariablen

Familienname	Funktion	Argumente mit Voreinstellung
binom	Binomialverteilung	size, prob
chisq	χ^2 -Verteilung	df, ncp=0
exp	Exponentielle Funktion	rate=1
f	F-Verteilung	df1, df2, ncp=0
gamma	Γ -Funktion	shape, rate=1, scale=1/rate
hyper	Hypergeometrische Verteilung	m, n, k
logis	Logistische Funktion	location=0, scale=1
norm	Normalverteilung ³	mean=0, sd=1
pois	Poisson-Verteilung	lambda
signrank	Wilcoxon-Vorzeichen-Rangverteilung	x, n
t	t-Verteilung	df, ncp=0
unif	Gleichverteilung	min=0, max=1
weibull	Weibull-Funktion	shape, scale=1
wilcox	Wilcoxon-Rangsummenverteilung	m, n

5.2.1 Dichtefunktionen

Mit Funktionen, deren Namen nach dem Muster `d(Funktionsfamilie)` aufgebaut sind, lassen sich die Werte der Dichtefunktionen⁴ der o. g. Funktionsfamilien bestimmen. Mit dem Argument `x` wird angegeben, für welche Stelle der Wert der Dichtefunktion berechnet werden soll. Dies kann auch ein Vektor sein – dann wird für jedes Element von `x` der Wert der Dichtefunktion bestimmt. Die Bedeutung der übrigen Argumente ist identisch zu jener bei den zugehörigen Funktionen zum Generieren von Zufallszahlen (vgl. Abschn. 2.5.2).

```
> dbinom(x, size, prob)                      # Binomialverteilung
> dnorm(x, mean=0, sd=1)                     # Normalverteilung
> dchisq(x, df,      ncp=0)                  # chi^2-Verteilung
> dt(x,   df,      ncp=0)                   # t-Verteilung
> df(x,   df1, df2, ncp=0)                  # F-Verteilung
```

Die Wahrscheinlichkeit, beim zehnfachen Werfen einer fairen Münze genau siebenmal Kopf als Ergebnis zu erhalten, ergibt sich beispielsweise so:

```
> dbinom(7, 10, 0.5)
[1] 0.1171875

> choose(10, 7) * 0.5^7 * (1-0.5)^(10-7)    # manuelle Kontrolle
[1] 0.1171875
```

³ Für multivariate t- und Normalverteilungen vgl. das Paket `mvtnorm`.

⁴ Im Fall diskreter (z. B. binomialverteilter) Variablen die Wahrscheinlichkeitsfunktion.

5.2.2 Verteilungsfunktionen

Die Werte der zu einer Dichte- bzw. Wahrscheinlichkeitsfunktion gehörenden Verteilungsfunktion lassen sich mit Funktionen berechnen, deren Namen nach dem Muster `p(Funktionsfamilie)` aufgebaut sind.⁵ Mit dem Argument `q` wird angegeben, für welche Stelle der Wert der Verteilungsfunktion berechnet werden soll. In der Voreinstellung sorgt das Argument `lower.tail=TRUE` dafür, dass der Rückgabewert an einer Stelle q die Wahrscheinlichkeit angibt, dass die zugehörige Zufallsvariable Werte $\leq q$ annimmt.⁶ Die Gegenwahrscheinlichkeit (Werte $> q$) wird mit dem Argument `lower.tail=FALSE` berechnet – dies entspricht dem von inferenzstatistischen Tests ausgegebenen p -Wert.

```

> pbinom(q, size, prob,           lower.tail=TRUE) # Binomialverteilung
> pnorm(q, mean=0, sd=1,         lower.tail=TRUE) # Normalverteilung
> pchisq(q, df,                ncp=0, lower.tail=TRUE) # chi^2-Verteilung
> pt(q, df,                   ncp=0, lower.tail=TRUE) # t-Verteilung
> pf(q, df1, df2, ncp=0, lower.tail=TRUE) # F-Verteilung

> pbinom(7, size=10, prob=0.5)      # Verteilungsfunktion Binomialverteilung
[1] 0.9453125

> sum(dbinom(0:7, size=10, prob=0.5))          # Kontrolle über W-Funktion
[1] 0.9453125

> pnorm(c(-Inf, 0, Inf), mean=0, sd=1)        # Standardnormalverteilung
[1] 0.0 0.5 1.0

# Standardnormalverteilung: Fläche unter Dichtefunktion rechts von 1.645
> pnorm(1.645, mean=0, sd=1, lower.tail=FALSE)
[1] 0.04998491

# äquivalent: 1-(Fläche unter Dichtefunktion links von 1.645)
> 1-pnorm(1.645, mean=0, sd=1, lower.tail=TRUE)
[1] 0.04998491

```

Mit der Verteilungsfunktion lässt sich auch die Wahrscheinlichkeit dafür berechnen, dass die zugehörige Variable Werte innerhalb eines bestimmten Intervalls annimmt, indem der Wert der unteren Intervallgrenze von jenem der oberen subtrahiert wird.

```
# Standardnormalverteilung: Wkt. für Werte im Intervall mu +- sd
> m <- 100                                # Erwartungswert
```

⁵ Zusätzlich existiert mit `pbirthday()` eine Funktion zur Berechnung der Wahrscheinlichkeit, dass in einer Menge mit n Elementen `coincident` viele denselben Wert auf einer kategorialen Variable mit `classes` vielen, gleich wahrscheinlichen Stufen haben. Mit `classes=365` und `coincident=2` entspricht dies der Wahrscheinlichkeit, dass zwei Personen am selben Tag Geburtstag haben.

⁶ Bei der diskreten Binomialverteilung ist es bedeutsam, dass die Grenze q mit eingeschlossen ist.

```
> s <- 15                                # Standardabweichung
> pnorm(m+s, mean=m, sd=s) - pnorm(m-s, mean=m, sd=s)
[1] 0.6826895
```

Nützlich ist die Verteilungsfunktion insbesondere für die manuelle Berechnung des p -Wertes, sowie der Power eines Tests, vgl. Abschn. 8.9.

5.2.3 Quantilfunktionen

Die Werte der zu einer Dichte- bzw. Wahrscheinlichkeitsfunktion gehörenden Quantilfunktion lassen sich mit Funktionen berechnen, deren Namen nach dem Muster $q\langle$ Funktionsfamilie \rangle aufgebaut sind. Mit dem Argument p wird angegeben, für welche Wahrscheinlichkeit der Quantilwert berechnet werden soll. Das Ergebnis ist die Zahl, die in der zugehörigen Dichtefunktion die Fläche p links (Argument `lower.tail=TRUE`) bzw. rechts (`lower.tail=FALSE`) abschneidet. Anders formuliert ist das Ergebnis der Wert, für den die zugehörige Verteilungsfunktion den Wert p annimmt.⁷ Die Quantilfunktion ist also die Umkehrfunktion der Verteilungsfunktion.

```
> qbinom(p, size, prob,      lower.tail=TRUE)  # Binomialverteilung
> qnorm(p, mean=0, sd=1,     lower.tail=TRUE)  # Normalverteilung
> qchisq(p, df,            ncp=0,  lower.tail=TRUE) # chi^2-Verteilung
> qt(p, df,               ncp=0,  lower.tail=TRUE) # t-Verteilung
> qf(p, df1, df2,         ncp=0,  lower.tail=TRUE) # F-Verteilung
```

Die Quantilfunktion lässt sich nutzen, um kritische Werte für inferenzstatistische Tests zu bestimmen. Dies erübrigt es Tabellen zu konsultieren und die damit verbundene Notwendigkeit zur Interpolation bei nicht tabellierten Werten für Wahrscheinlichkeiten oder Freiheitsgrade. Wenn die empirische Teststatistik in Richtung der Hypothese jenseits des kritischen Wertes liegt, entspricht dies einem signifikanten Testergebnis.

```
> qnorm(pnorm(0))          # qnorm() ist Umkehrfunktion von pnorm()
[1] 0

> qnorm(1-(0.05/2), 0, 1) # krit. Wert zweiseitiger z-Test, alpha=0.05
[1] 1.959964

> qt(1-0.01, 18, 0)       # krit. Wert einseitiger t-Test, alpha=0.01, df=18
[1] 2.552380
```

⁷ Bei der diskreten Binomialverteilung ist das Ergebnis bei `lower.tail=TRUE` der kleinste Wert, der in der zugehörigen Wahrscheinlichkeitsfunktion mindestens p links abschneidet. Bei `lower.tail=FALSE` ist das Ergebnis entsprechend der größte Wert, der mindestens p rechts abschneidet.

5.3 Behandlung fehlender Werte in inferenzstatistischen Tests

Viele Funktionen zur Berechnung statistischer Tests besitzen das Argument `na.action`, das festlegt, wie mit fehlenden Werten zu verfahren ist. Mögliche Werte sind u.a. die Namen der `na.omit()` und `na.fail()` Funktionen, die sich auch direkt auf Daten anwenden lassen (vgl. Abschn. 2.12.3 und 3.2.14). `na.action=na.omit` bewirkt den fallweisen Ausschluss (vgl. Abschn. 2.12.4), mit `na.action=na.fail` wird eine Auswertung bei fehlenden Werten abgebrochen und eine Fehlermeldung ausgegeben. Global kann dieses Verhalten mit `options(na.action = <Wert>)` geändert werden (vgl. `?na.action`).

Kapitel 6

Nonparametrische Methoden

Wenn inferenzstatistische Tests zur Datenauswertung herangezogen werden sollen, aber davon ausgegangen werden muss, dass strenge Anforderungen an die Art und Qualität der erhobenen Daten nicht erfüllt sind, kommen viele konventionelle Tests womöglich nicht in Betracht. Für solche Situationen hält der Bereich der nonparametrischen Statistik Methoden bereit, deren Voraussetzungen gewöhnlich weniger restriktiv sind und die auch bei kleinen Stichproben als Auswertungsverfahren in Frage kommen (Agresti, 2007; Bortz et al., 2008; Büning und Trenkler, 1994). Insbesondere für (gemeinsame) Häufigkeiten kategorialer Variablen und Rangdaten¹ sind viele der im folgenden angeführten Methoden geeignet.²

6.1 Anpassungstests

Viele Tests setzen voraus, dass die Verteilung der AV in den untersuchten Bedingungen bekannt ist und bestimmte Voraussetzungen erfüllt – oft muss es sich etwa um eine Normalverteilung handeln. Ob die erhobenen Werte in einer bestimmten Stichprobe mit einer solchen Annahme verträglich sind, kann mit verschiedenen Anpassungstests geprüft werden.

Häufig werden Anpassungstests mit der Nullhypothese durchgeführt, dass eine bestimmte Verteilung vorliegt und dieses Vorliegen auch den gewünschten Zustand beschreibt. Da hier die Alternativhypothese gegen die fälschliche Nicht-Annahme abzusichern ist, muss der β -Fehler kontrolliert werden. Dies kann angestrebt werden, indem das α -Niveau höher als üblich gewählt wird, auch wenn sich der

¹ Auf Rangdaten basierende Tests machen häufig die Voraussetzung, dass die Ränge eindeutig bestimmbar sind, also keine gleichen Werte (sog. Bindungen, engl. ties) auftauchen. Für den Fall, dass dennoch Bindungen vorhanden sind, existieren unterschiedliche Strategien, wobei die von R-Funktionen gewählte häufig in der zugehörigen Hilfe erwähnt wird.

² Einige Tests approximieren die Verteilung der verwendeten Teststatistik durch eine mit wachsender Stichprobengröße asymptotisch gültige Verteilung. Für exakte Pendants sowie für Monte-Carlo-Approximationen vgl. das Paket `coin` (Hothorn et al., 2008). Für Bootstrap-Verfahren vgl. das Paket `boot` (Canty und Ripley, 2009; Davison und Hinkley, 1997).

β -Fehler so nicht exakt begrenzt lässt. Per Konvention werden in solchen Fällen häufig Werte in der Größenordnung von 0.2 oder 0.25 für α verwendet.

6.1.1 Binomialtest

Der Binomialtest ist auf dichotome Daten anzuwenden, d. h. auf Werte solcher Variablen, die nur zwei Ausprägungen als Ergebnis eines Bernoulli-Zufallsexperiments annehmen können. Entsprechend der üblichen Terminologie soll eine Ausprägung der Variable als *Treffer* bzw. *Erfolg* bezeichnet werden. Der Test prüft, ob die empirische Auftretenshäufigkeit eines Treffers in einer Stichprobe verträglich mit der Nullhypothese einer bestimmten Trefferwahrscheinlichkeit ist.

```
> binom.test(x=<Erfolge>, n=<Stichprobengröße>, p=0.5, conf.level=0.95,
+             alternative=c("two.sided", "less", "greater"))
```

Unter x ist die beobachtete Anzahl der Erfolge anzugeben. n steht für die Stichprobengröße, d. h. die Anzahl der Wiederholungen des Zufallsexperiments. Alternativ zur Angabe von x und n kann als erstes Argument ein Vektor mit zwei Elementen übergeben werden, dessen Einträge die Anzahl der Erfolge und Misserfolge sind – etwa das Ergebnis einer Häufigkeitsauszählung mit `table()`. Unter p ist die Wahrscheinlichkeit eines Erfolgs unter der Nullhypothese einzutragen. Das Argument `alternative` bestimmt, ob zweiseitig ("two.sided"), links- ("less") oder rechtsseitig ("greater") getestet wird. Die Aussage bezieht sich dabei auf die Reihenfolge Alternativhypothese "less" bzw. "greater" als die Nullhypothese. Mit dem Argument `conf.level` wird die Breite des Konfidenzintervalls für die Trefferwahrscheinlichkeit festgelegt. Im Fall einer zweiseitigen Fragestellung wird das zweiseitige, im Fall einer einseitigen Fragestellung entsprechend das passende einseitige Konfidenzintervall gebildet.

Als Beispiel sollen aus einer (unendlich großen) Urne zufällig Lose gezogen werden, wobei die Grundwahrscheinlichkeit eines Gewinns 0.25 beträgt. Hier sei nach der Wahrscheinlichkeit gefragt, bei mindestens 5 von insgesamt 7 Ziehungen einen Gewinn zu ziehen. Die einzelnen Wahrscheinlichkeiten der Fälle mit 5, 6 oder 7 Gewinnen werden summiert unter `p-value` ausgegeben.

```
> draws <- 7                      # Stichprobenumfang
> hits <- 5                       # Anzahl Treffer
> pH0 <- 0.25                     # Trefferwahrscheinlichkeit unter H0
> binom.test(hits, draws, p=pH0, alternative="greater", conf.level=0.95)
Exact binomial test
data: hits and draws
number of successes = 5, number of trials = 7, p-value = 0.01288
alternative hypothesis: true probability of success is greater than 0.25
95 percent confidence interval:
0.3412614 1.0000000
sample estimates:
probability of success
0.7142857
```

Das Ergebnis `(Objekt)` von `binom.test()` ist wie das Ergebnis aller Tests in R eine Liste, deren Komponenten mit `names((Objekt))` und `str((Objekt))` abgefragt werden können. Das Ergebnis enthält neben einer Zusammenfassung der eingegebenen Daten (`number of successes` und `number of trials`, `probability of success`) mit `p-value` die Wahrscheinlichkeit, die beobachtete Trefferhäufigkeit oder extremere Trefferzahlen unter Gültigkeit der Nullhypothese zu erhalten. Dabei wird berücksichtigt, ob es sich um eine ein- oder zweiseitige Alternativhypothese handelt, beim zweiseitigen Test muss `p-value` also nicht mit dem halbierten Signifikanzniveau verglichen werden.³ Liegt `p-value` unter dem festgesetzten Signifikanzniveau, kann die Nullhypothese verworfen werden. Schließlich wird je nach Fragestellung das zwei-, links- oder rechtsseitige Konfidenzintervall für die Trefferwahrscheinlichkeit⁴ in der gewünschten Breite ausgegeben.⁵

Der ausgegebene p -Wert kann manuell mit Hilfe der Verteilungsfunktion der Binomialverteilung verifiziert werden. Hier ist bei der Verwendung von `pbinom(q, size, prob)` zu beachten, dass die Funktion die Wahrscheinlichkeit dafür berechnet, dass die zugehörige Zufallsvariable Werte $\leq q$ annimmt – die Grenze q also mit eingeschlossen ist. Für die Berechnung der Wahrscheinlichkeit, dass die Variable Werte $\geq q$ annimmt (rechtsseitiger Test), ist als Argument für `1-pbinom()` deshalb $q - 1$ zu übergeben, andernfalls würde nur die Wahrscheinlichkeit für Werte $> q$ bestimmt.

```
> (pVal <- 1-pbinom(hits-1, draws, pH0))
[1] 0.01287842
```

Der kritische Wert kann mit `qbinom(p, size, prob)` bestimmt werden. Dabei ist zu beachten, dass die Nullhypothese dann verworfen wird, wenn die beobachtete Trefferzahl größer als der kritische Wert ist.⁶

```
> (critB <- qbinom(1-0.05, draws, pH0))
[1] 4
```

³ Für den zweiseitigen Binomialtest existieren verschiedene Definitionen des p -Wertes: R summiert die Wahrscheinlichkeiten unter Gültigkeit der Nullhypothese für alle Ereignisse mit höchstens der Wahrscheinlichkeit des eingetretenen Ereignisses. So entspricht etwa der von `binom.test(10, 20, p=0.25, alternative="two.sided")` ausgegebene p -Wert `sum(dbinom(10:20, 20, 0.25))+ sum(dbinom(0, 20, 0.25))`. Eine andere Definition bestünde in der Verdoppelung des kleineren der p -Werte der einseitigen Tests. Wenn unter der Nullhypothese mit $p = 0.5$ eine symmetrische Binomialverteilung vorliegt, führen die Definitionen zum selben Ergebnis.

⁴ Das Intervall ist jenes nach Clopper-Pearson. Für die Berechnung u. a. nach Wilson, Agresti-Coull und Jeffreys vgl. etwa das `binom` Paket (Dorai-Raj, 2009).

⁵ Auch bei anderen statistischen Tests werden p -Wert und Konfidenzintervall entsprechend der Richtung der Fragestellung berechnet.

⁶ Oft wird auch der mindestens zu erreichende Wert als der kritische bezeichnet, also der erste Wert des Ablehnungsbereichs der Nullhypothese. Hier soll dagegen der kritische Wert immer der zu überschreitende sein. Dieser Unterschied in der Bezeichnungskonvention ist nur für diskrete Verteilungen relevant.

Bei einer asymmetrischen Verteilung wie im Beispiel ließe sich die Frage formulieren, wie wahrscheinlich unter Gültigkeit der Nullhypothese das beobachtete Ergebnis oder mindestens ebenso extreme sind. In diesem Fall müssten die p -Werte für jede der beiden einseitigen Fragestellungen einzeln bestimmt und dann addiert werden. Dafür muss nicht nur die Wahrscheinlichkeit von 5 oder mehr Gewinnen unter der Nullhypothese berechnet werden, sondern auch die von 2 (also 7–5) oder weniger. Durch die Addition der beiden Wahrscheinlichkeiten ergibt sich die zweiseitige Wahrscheinlichkeit, die mit dem Signifikanzniveau zu vergleichen ist.

```
# rechtsseitiger Test
> resG <- binom.test(hits, draws, p=pH0, alternative="greater")
> resG$p.value                                # p-Wert
[1] 0.01287842

# linksseitiger Test
> resL <- binom.test(draws-hits, draws, p=pH0, alternative="less")
> resL$p.value                                # p-Wert
[1] 0.7564087

> resG$p.value + resL$p.value                # Gesamt-p-Wert
[1] 0.7692871
```

Bei der Berechnung der kritischen Werte für den zweiseitigen Test stellt sich die Frage, wie das gesamte α -Niveau auf beide Seiten der diskreten und möglicherweise asymmetrischen Verteilung zu verteilen ist. Nach einer gängigen Konvention ist auf jeder Seite nicht mehr als $\alpha/2$ abzuschneiden, auch wenn bei Überschreitung von $\alpha/2$ auf einer Seite dennoch insgesamt weniger als α abgeschnitten würde.

Hypothesen über die jeweiligen Trefferwahrscheinlichkeiten einer dichotomen Variable in zwei Stichproben lassen sich mit Fishers exaktem Test (vgl. Abschn. 6.2.5), in mehr als zwei Stichproben mit einem χ^2 -Test prüfen (vgl. Abschn. 6.2.3).

6.1.2 Test auf Zufälligkeit (Runs Test)

Eine zufällige Reihenfolge von N Ausprägungen einer dichotomen Variable sollte sich dann ergeben, wenn die Erfolgswahrscheinlichkeit für jede Realisierung $1/2$ beträgt und die Realisierungen unabhängig voneinander zustande kommen. In diesem Fall besitzt jedes feste Wertemuster dieselbe Auftretenswahrscheinlichkeit $1/2^N$. Mit dem Test auf Zufälligkeit kann geprüft werden, ob eine empirische Datenreihe mit der Annahme von Zufälligkeit verträglich ist. Teststatistik R ist die Anzahl der Iterationen (Runs, vgl. 2.11.3), wobei eine sehr geringe als auch sehr hohe Anzahl gegen die Nullhypothese spricht.⁷

⁷ Für den Wald-Wolfowitz-Test für zwei Stichproben vgl. Abschn. 6.4.3.

Eine spezielle Funktion für den Test auf Zufälligkeit ist nicht im Basisumfang von R enthalten, eine manuelle Durchführung jedoch möglich. Als Beispiel diene jenes aus Bortz et al. (2008, p. 548 ff.): Bei einer Warteschlange aus 8 Personen (5 Frauen, f und 3 Männer, m) wird das Geschlecht des an jeder Position Wartenden erhoben.

```
> queue <- c("f", "m", "m", "f", "m", "f", "f", "f")           # Daten
> Ns      <- table(queue)                                     # Gruppengrößen
> (runs <- rle(queue))                                       # Iterationen
Run Length Encoding
lengths: int [1:5] 1 2 1 1 3
values: chr [1:5] "f" "m" "f" "m" "f"

> (rr <- length(runs$lengths))                                # Gesamtzahl Iterationen
[1] 5

> (rr1 <- sum(runs$values == names(Ns)[1]))     # Iterationen Gruppe 1
[1] 3

> (rr2 <- sum(runs$values == names(Ns)[2]))     # Iterationen Gruppe 2
[1] 2
```

Für den p -Wert des beobachteten Wertes von R sind die Punktwahrscheinlichkeiten für alle Fälle aufzuaddieren, die dieses R oder extremere Werte ergeben. Die Berechnung der Punktwahrscheinlichkeiten geschieht hier mit einer eigens erstellten Funktion (vgl. Abschn. 11.1). Für die Ermittlung des p -Wertes ist zu beachten, dass ein ungerades R generell auf zwei Arten zustande kommen kann: entweder ist die Anzahl der Iterationen der ersten Gruppe um 1 größer als die der zweiten oder umgekehrt. Im konkreten Fall sind also die Fälle für $R = 5, 6, 7$ zu berücksichtigen – der Obergrenze möglicher Iterationen. Der Fall $R = 7$ kann sich hier nur auf eine Weise ergeben, da nur 3 Männer vorhanden sind, so dass insgesamt 4 Punktwahrscheinlichkeiten zu addieren sind.

```
# Funktion, um Punktwahrscheinlichkeit für Anzahl von Iterationen der
# Gruppe 1 (r1), Gruppe 2 (r2), mit Gruppengrößen n1 und n2 zu berechnen
> getP <- function(r1, r2, n1, n2) {
+   # Punktwahrscheinlichkeit für r1+r2 ungerade
+   p <- (choose(n1-1, r1-1) * choose(n2-1, r2-1)) / choose(n1+n2, n1)
+
+   # Punktwahrscheinlichkeit für r1+r2 gerade: das doppelte von ungerade
+   if((r1+r2) %% 2) == 0) { 2*p } else { p }
+ }

> nn1   <- Ns[1]                                              # Größe Gruppe 1
> nn2   <- Ns[2]                                              # Größe Gruppe 2
> N     <- sum(Ns)                                            # Gesamt-N
> rMin <- 2                                                    # Untergrenze für Anzahl der Iterationen

# Obergrenze Anzahl Iterationen. Fallunterscheidung: nn1 == nn2?
> (rMax <- ifelse(nn1 == nn2, N, 2*min(nn1, nn2) + 1))
[1] 7
```

```
# addiere Punktwahrscheinlichkeiten für R=beobachtet und größer
> p3.2 <- getP(3, 2, nn1, nn2)                                # r1=3, r2=2 -> R=5
> p2.3 <- getP(2, 3, nn1, nn2)                                # r1=2, r2=3 -> R=5
> p3.3 <- getP(3, 3, nn1, nn2)                                # r1=3, r2=3 -> R=6
> p4.3 <- getP(4, 3, nn1, nn2)                                # r1=4, r2=3 -> R=7
> (pValGr <- p3.2 + p2.3 + p3.3 + p4.3)                  # p-Wert einseitig
0.5714286

# Punktwahrscheinlichkeit aller anderen Fälle
> p2.2 <- getP(2, 2, nn1, nn2)                                # r1=2, r2=2 -> R=4
> p1.2 <- getP(1, 2, nn1, nn2)                                # r1=1, r2=2 -> R=3
> p2.1 <- getP(2, 1, nn1, nn2)                                # r1=2, r2=1 -> R=3
> p1.1 <- getP(1, 1, nn1, nn2)                                # r1=1, r2=1 -> R=2
> (pValLe <- p2.2 + p1.2 + p2.1 + p1.1)
0.4285714

> (pValGr + pValLe)           # Kontrolle: Summe beider p-Werte muss 1 sein
1
```

Aus R lässt sich eine mit wachsender Stichprobengröße asymptotisch standardnormalverteilte Teststatistik berechnen, deren Verwendung ab einer Anzahl von ca. 30 Beobachtungen nur zu geringen Fehlern führen sollte.

```
> muR   <- 1 + ((2*nn1*nn2) / N)                         # Erwartungswert von R
> varR  <- (2*nn1*nn2*(2*nn1*nn2 - N)) / (N^2 * (N-1))    # Varianz von R
> rZ    <- (rr-muR) / sqrt(varR)                           # z-Transformierte von R
> (pVal <- 1-pnorm(rZ))                                    # p-Wert einseitig
0.4184066
```

6.1.3 Kolmogorov-Smirnov-Anpassungstest

Der Kolmogorov-Smirnov-Test auf eine feste Verteilung ist als exakter Test auch bei kleinen Stichproben anwendbar und vergleicht die kumulierten relativen Häufigkeiten von Daten einer stetigen Variable mit einer frei wählbaren Verteilungsfunktion – etwa der einer bestimmten Normalverteilung. Gegen die Nullhypothese, dass die Verteilungsfunktion der angegebenen entspricht, kann eine ungerichtete wie gerichtete Alternativhypothese getestet werden. Der Test lässt sich durch eine visuell-explorative Analyse mittels eines Quantil-Quantil-Diagramms ergänzen (vgl. Abschn. 10.6.5).

```
> ks.test(x=<Vektor>, y="(Name der Verteilungsfunktion)", ...,
+          alternative=c("two.sided", "less", "greater"))
```

Unter x ist der Datenvektor einzugeben und unter y die Verteilungsfunktion der Variable unter der Nullhypothese.⁸ Um durch Komma getrennte Argumente an diese Verteilungsfunktion zu ihrer genauen Spezifikation übergeben zu können, dienen die ... Auslassungspunkte. Mit $alternative$ wird die Art der Fragestellung definiert,

⁸ Für den Zweistichprobenfall vgl. Abschn. 6.4.4.

wobei sich "less" und "greater" darauf beziehen, ob y stochastisch kleiner oder größer als x ist.⁹

Im Beispiel soll zum Test die Verteilungsfunktion der Normalverteilung mit Erwartungswert $\mu = 1$ und Streuung $\sigma = 2$ herangezogen werden.

```
> vec <- rnorm(8, mean=1, sd=2) # Daten
> ks.test(vec, "pnorm", mean=1, sd=2, alternative="two.sided")
One-sample Kolmogorov-Smirnov test
data: vec
D = 0.2326, p-value = 0.6981
alternative hypothesis: two-sided
```

Die Ausgabe umfasst den empirischen Wert der zweiseitigen Teststatistik (D) sowie den zugehörigen p -Wert (p-value). Im folgenden wird die Teststatistik sowohl für den ungerichteten wie für beide gerichteten Tests manuell berechnet, zudem sollen die hierfür relevanten Differenzen zwischen empirischer und angenommener Verteilung graphisch veranschaulicht werden (Abb. 6.1).

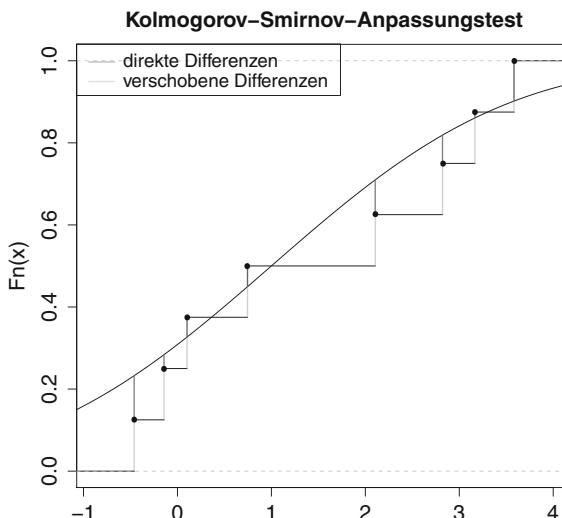


Abb. 6.1 Kolmogorov-Smirnov-Anpassungstest: Abweichungen zwischen kumulierten relativen Häufigkeiten der Daten und der Verteilungsfunktion der Normalverteilung $N(\mu = 1, \sigma = 2)$

```
> myStep <- ecdf(vec) # Funktion für kumulierte rel. Häufigkeiten
> sortVec <- sort(vec) # sortierte Daten
> emp <- myStep(sortVec) # kumulierte relative Häufigkeiten
# theoretische Werte: Verteilungsfkt. der Normalverteilung N(mu=1, sigma=2)
> theo <- pnorm(sortVec, mean=1, sd=2)
```

⁹ Bei zwei Zufallsvariablen X und Y ist Y dann stochastisch größer als X , wenn die Verteilungsfunktion von Y an jeder Stelle unter der von X liegt. Besitzen X und Y etwa Verteilungen derselben Form, ist dies der Fall, wenn die Dichte- bzw. Wahrscheinlichkeitsfunktion von Y eine nach rechts verschobene Version der von X darstellt.

```

> diff1 <- emp-theo           # direkte Differenzen
> diff2 <- c(0, emp[1:(length(emp)-1)]) - theo    # verschobene Differ.

# Teststatistik für zweiseitigen Test: maximale absolute Abweichung
> (DtwoS <- max(abs(c(diff1, diff2))))
[1] 0.2325919

# Teststatistik für "less": Betrag der stärksten Abweichung nach unten
> (Dless <- abs(min(c(diff1, diff2))))
[1] 0.2325919

# Teststatistik für "greater": Betrag der stärksten Abweichung nach oben
> (Dgreat <- abs(max(c(diff1, diff2))))
[1] 0.09828234

# Kontrolle über Ausgabe von ks.test()
> ks.test(vec, "pnorm", mean=1, sd=2, alternative="less")$statistic
D^-
0.2325919

> ks.test(vec, "pnorm", mean=1, sd=2, alternative="greater")$statistic
D^+
0.09828234

# graphische Darstellung der direkten und verschobenen Differenzen
> plot(myStep, main="Kolmogorov-Smirnov-Anpassungstest", xlab=NA)
> myX <- seq(-6, 6, length.out=200)                      # x-Koordinaten
> points(myX, pnorm(myX, mean=1, sd=2), type="l")      # Verteilungsfunktion

# direkte Abweichungen
> matlines(rbind(sortVec, sortVec), rbind(emp, theo),
+           col=rgb(0, 0, 1, 0.7), lty=1, lwd=2)

# verschobene Abweichungen
> matlines(rbind(sortVec, sortVec), rbind(c(0, emp[1:(length(emp)-1)]),
+           theo), col=rgb(1, 0, 0, 0.5), lty=1, lwd=2)

> legend(x="topleft", legend=c("direkte Differenzen",
+                               "verschobene Differenzen"), col=c("blue", "red"), lwd=2)

```

Der beschriebene Kolmogorov-Smirnov-Test setzt voraus, dass die theoretische Verteilungsfunktion vollständig festgelegt ist, also keine zusammengesetzte Nullhypothese vorliegt, die nur die Verteilungsklasse benennt. Für einen korrekten Test dürfen die Parameter also nicht auf Basis der Daten in `x` geschätzt werden. Das Paket `nortest` (Groß, 2008) stellt mit `lillie.test()` die Abwandlung mit Lilliefors-Schranken sowie mit `ad.test()` den Anderson-Darling-Test zur Verfügung, die diese Voraussetzungen nicht machen.

Der über `shapiro.test()` aufzurufende Shapiro-Wilk-Test auf Normalverteiltheit ist eine Alternative, die auch für den Test auf gemeinsame Normalverteilung multivariater Daten existiert. Sie wird von der `mshapiro()` Funktion des `mvnortest` Pakets (Jarek, 2009) umgesetzt.

6.1.4 χ^2 -Test auf eine feste Verteilung

Der χ^2 -Test auf eine feste Verteilung prüft kategoriale Daten daraufhin, ob die empirischen Auftretenshäufigkeiten der einzelnen Kategorien verträglich mit einer theoretischen Verteilung sind, die in der Nullhypothese in Form von Klassenwahrscheinlichkeiten formuliert wird.¹⁰ Als ungerichtete Alternativhypothese ergibt sich, dass die tatsächliche Verteilung nicht der unter der Nullhypothese genannten entspricht.

```
> chisq.test(x=(Häufigkeiten), p=(Wahrscheinlichkeiten),
+             simulate.p.value=FALSE)
```

Unter x ist der Vektor anzugeben, der die empirischen absoluten Auftretenshäufigkeiten der Kategorien beinhaltet. Dies kann z. B. eine einfache Häufigkeitstabelle als Ergebnis von `table()` sein. Unter p ist ein Vektor einzutragen, der die Wahrscheinlichkeiten für das Auftreten der Kategorien unter der Nullhypothese enthält und dementsprechend dieselbe Länge wie x haben muss. Treten erwartete Häufigkeiten von Kategorien als Produkt der Klassenwahrscheinlichkeiten mit der Stichprobengröße kleiner als 5 auf, sollte das Argument `simulate.p.value=TRUE` gesetzt werden. Andernfalls gibt R in einer solchen Situation die Warnung aus, dass die χ^2 -Approximation noch unzulänglich sein kann.

Als Beispiel soll ein empirischer Würfel daraufhin getestet werden, ob er fair ist. Die Auftretenswahrscheinlichkeit jeder Augenzahl unter der Nullhypothese beträgt 1/6.

```
> nToss    <- 50                                # Anzahl Ziehungen
> nCateg   <- 6                                # Anzahl Kategorien
> pH0      <- rep(1/nCateg, nCateg)            # Verteilung unter H0
> myData   <- sample(1:nCateg, nToss, replace=TRUE) # simulierte Daten
> (tab     <- table(myData))                   # Häufigkeiten
myData
 1 2 3 4 5 6
 8 8 4 14 3 13

> chisq.test(tab, p=pH0)
Chi-squared test for given probabilities
data: tab
X-squared = 12.16, df = 5, p-value = 0.03266
```

Die Ausgabe umfasst den Wert der mit wachsender Stichprobengröße asymptotisch χ^2 -verteilten Teststatistik (X-squared) und ihre Freiheitsgrade (df, nur wenn `simulate.p.value=FALSE` ist) samt zugehörigem p-Wert (p-value). Das Ergebnis lässt sich manuell prüfen:

```
> expected    <- pH0 * nToss                  # erwartete Häufigkeiten unter H0
> (statChisq <- sum((tab-expected)^2 / expected))    # Teststatistik
[1] 12.16
```

¹⁰ Dieser Test ist auch bei quantitativen Variablen durchführbar, wobei zunächst eine Einteilung der Werte in disjunkte Klassen vorzunehmen ist. Die getestete Nullhypothese ist dann in dem Sinne schwächer, dass alle Verteilungen äquivalent sind, die zu gleichen Klassenwahrscheinlichkeiten führen.

```
# p-Wert: Fläche rechts von statChisq unter Dichtefunktion unter H0
> (pVal <- 1-pchisq(statChisq, nCateg-1))
[1] 0.03265998
```

6.1.5 χ^2 -Test auf eine Verteilungsklasse

Als Spezialfall des χ^2 -Tests auf Gleichheit von Verteilungen (vgl. Abschn. 6.2.2) kann die Verträglichkeit der Daten mit der Nullhypothese getestet werden, dass die zugehörige theoretische Verteilung etwa aus der Familie der Normalverteilungen stammt. Die Alternativhypothese ist ungerichtet, der Test asymptotisch korrekt bei wachsender Stichprobengröße und kann durch eine visuell-explorative Begutachtung eines Quantil-Quantil-Diagramms ergänzt werden (vgl. Abschn. 10.6.5).

Die Daten sind zunächst in disjunkte Klassen einzuteilen, deren empirische Auftretenshäufigkeiten mit den unter der Nullhypothese erwarteten verglichen werden.¹¹ Die Einteilung der Variable in Kategorien ist so vorzunehmen, dass sich unter Gültigkeit der Nullhypothese erwartete Häufigkeiten von mindestens 5 ergeben.

Die Parameter der konkreten Normalverteilung unter der Nullhypothese folgen entweder aus theoretischen Erwägungen oder werden auf Basis der Daten geschätzt, häufig anhand des Mittelwertes und der korrigierten Stichprobenstreuung.¹² Durch eine solche Schätzung der zwei Parameter aus den Daten reduziert sich die Anzahl der Freiheitsgrade beim Test um 2. Dies wird beim Aufruf von `chisq.test()` nicht berücksichtigt, d. h. der ausgegebene p -Wert ist für diesen Test nicht der richtige. Ist `<Objekt>` das Ergebnis von `chisq.test()`, muss die Bestimmung des richtigen p -Wertes deshalb manuell mit dem empirischen χ^2 -Wert (`<Objekt>$statistic`), den richtigen Freiheitsgraden (`<Objekt>$parameter - 2`) und der Verteilungsfunktion `pchisq(<Quantil>, <Freiheitsgrade>)` erfolgen.

Im Beispiel sollen IQ-Werte daraufhin geprüft werden, ob sie mit der Annahme einer Normalverteiltheit verträglich sind. Die gewählten Klassengrenzen werden zunächst entsprechend des Mittelwertes und der Streuung der Daten z -transformiert, um einen Vergleich mit den unter der Standardnormalverteilung erwarteten Häufigkeiten zu ermöglichen. Die Bestimmung der erwarteten Häufigkeiten beinhaltet als Schritte die Berechnung der Werte der Verteilungsfunktion für die Klassengrenzen (`pnorm(<Grenzen>)`), die Berechnung der geschätzten Wahrscheinlichkeiten der einzelnen Intervalle durch Subtrahieren des Wertes der jeweils unteren Klassengrenze von dem der oberen (`diff(<Vektor>)`) und die Multiplikation mit der Stichprobengröße (`<Vektor> * length(<Daten>)`).

¹¹ Die Klassenbildung führt dazu, dass statt der Verträglichkeit mit einer bestimmten Normalverteilung die schwächere Nullhypothese einer Verträglichkeit mit allen Verteilungen getestet wird, die zu denselben erwarteten Häufigkeiten führen.

¹² Für eine korrekte Testkonstruktion wäre eigentlich eine Schätzung von μ und σ notwendig, die die gewählte Klasseneinteilung berücksichtigt. Dies wäre bei einer gruppierten Maximum-Likelihood- oder einer Minimum- χ^2 -Schätzung der Fall.

```

> myData    <- rnorm(100, mean=100, sd=15)           # simulierte Daten
> limits    <- c(80, 90, 100, 110, 120)            # innere Int.grenzen
> dataCut   <- cut(myData, c(-Inf, limits, Inf))    # + äußere Int.grenzen
> (dataFreq <- table(dataCut))                      # beobachtete Häufigk.
(-Inf,80] (80,90] (90,100] (100,110] (110,120] (120, Inf]
6         19        19       28       19        9

> zLimits   <- (limits-mean(myData))/sd(myData)      # standardis. Grenzen
> zLimits   <- c(-Inf, zLimits, Inf)                  # + äußere Int.grenzen

# erwartete Häufigkeiten
> (freqExp  <- diff(pnorm(zLimits))*length(myData))
[1] 7.102769 14.618496 24.490381 26.076912 17.648505 10.062937

# Test mit den falschen Freiheitsgraden
> (resChisq <- chisq.test(dataFreq, p=diff(pnorm(zLimits))))
Pearson's Chi-squared test
data: rbind(dataFreq, freqExp)
X-squared = 3.0729, df = 5, p-value = 0.6887

# Test mit den richtigen Freiheitsgraden
> statChisq <- resChisq$statistic                 # Teststatistik chi^2
> realDf     <- resChisq$parameter - 2             # korrig. Freiheitsgr.
> (realPval  <- 1-pchisq(statChisq, realDf))       # korrigierter p-Wert
0.38052

```

Das Paket nortest enthält mit `pearson.test()` eine spezialisierte Funktion für den χ^2 -Test auf Normalverteiltheit, über die sowohl die Klasseneinteilung wie auch die Korrektur der Freiheitsgrade per Argument gesteuert werden können.

```
> pearson.test(x=<Vektor>, n.classes=<Anzahl>, adjust=TRUE)
```

Der Datenvektor wird der Funktion als Argument `x` übergeben, `n.classes` legt die Zahl der zu bildenden Klassen fest. Die genaue Lage der Klassengrenzen wird von der Funktion selbst so gewählt, dass alle Klassen unter der Nullhypothese dieselbe Wahrscheinlichkeit besitzen. Über `adjust` wird angegeben, ob die Zahl der Freiheitsgrade in Folge einer Schätzung der Parameter der Normalverteilung unter der Nullhypothese aus den Daten um 2 nach unten zu korrigieren ist.

6.2 Analyse von gemeinsamen Häufigkeiten kategorialer Variablen

Inwieweit die empirischen Auftretenshäufigkeiten der Ausprägungen kategorialer Variablen theoretischen Vorstellungen entsprechen, kann durch folgende Tests geprüft werden, die sich auf unterschiedliche versuchsplanerische Situationen beziehen.

6.2.1 χ^2 -Test auf Unabhängigkeit

Beim zweidimensionalen χ^2 -Test auf Unabhängigkeit wird die empirische Kontingenztafel von zwei an derselben Stichprobe erhobenen kategorialen Variablen daraufhin geprüft, ob sie verträglich mit der Nullhypothese ist, dass beide Variablen unabhängig sind.¹³ Die Alternativhypothese ist ungerichtet, der Test asymptotisch korrekt bei wachsender Stichprobengröße.

```
> chisq.test(x, y=NULL, simulate.p.value=FALSE)
```

Unter x kann eine zweidimensionale Kontingenztafel eingegeben werden – etwa als Ergebnis von `table(<Faktor1>, <Faktor2>)` oder als Matrix mit den Häufigkeiten der Stufenkombinationen zweier Variablen. Alternativ kann x ein Objekt der Klasse `factor` mit den Ausprägungen der ersten Variable sein. In diesem Fall muss auch y angegeben werden, das dann ebenfalls ein Objekt der Klasse `factor` derselben Länge wie x mit an denselben Beobachtungsobjekten erhobenen Daten zu sein hat. Unter der Nullhypothese ergeben sich die Zellwahrscheinlichkeiten jeweils als Produkt der zugehörigen Randwahrscheinlichkeiten, die über die relativen Randhäufigkeiten geschätzt werden. Das Argument `simulate.p.value=TRUE` sollte gesetzt werden, wenn erwartete Zellhäufigkeiten kleiner als 5 auftreten. Andernfalls gibt R in einer solchen Situation die Warnung aus, dass die χ^2 -Approximation noch unzulänglich sein kann.

Als Beispiel sei eine Stichprobe von Studenten betrachtet, die angeben, ob sie rauchen und wie viele Geschwister sie haben.

```
> smokes    <- factor(sample(c("no", "yes"), 50, replace=TRUE))
> siblings  <- factor(round(abs(rnorm(50, 1, 0.5))))
> cTab      <- table(smokes, siblings)           # Kontingenztafel
> addmargins(cTab)                            # Randsummen
   siblings
smokes 0 1 2 Sum
  no  6 18 1 25
  yes 2 18 5 25
  Sum 8 36 6 50

> chisq.test(cTab)
Pearson's Chi-squared test
data: cTab
X-squared = 4.6667, df = 2, p-value = 0.09697

Warning message:
In chisq.test(cTab) : Chi-squared approximation may be incorrect
```

¹³ Dieser Test ist auch bei quantitativen Variablen durchführbar, wobei zunächst eine Einteilung der Werte in disjunkte Klassen vorzunehmen ist. Die getestete Nullhypothese ist dann in dem Sinne schwächer, dass nur die Unabhängigkeit bzgl. der vorgenommenen Klasseneinteilung getestet wird.

Das Ergebnis lässt sich manuell kontrollieren:

```
> J <- nlevels(smokes)                                # Anzahl smokes Kategorien
> K <- nlevels(siblings)                            # Anzahl siblings Kategorien
# erwartete Häufigkeiten -> alle Produkte der Randhäufigkeiten
# relativiert an der Summe von Beobachtungen
> expected   <- outer(rowSums(cTab), colSums(cTab)) / sum(cTab)
> (statChisq <- sum((cTab-expected)^2 / expected))  # Teststatistik
[1] 4.666667

# p-Wert: Fläche rechts von statChisq unter Dichtefunktion unter H0
> (pVal <- 1-pchisq(statChisq, (J-1)*(K-1)))
[1] 0.09697197
```

6.2.2 χ^2 -Test auf Gleichheit von Verteilungen

Beim χ^2 -Test auf Gleichheit von Verteilungen werden die empirischen eindimensionalen Häufigkeitsverteilungen einer kategorialen Variable in unterschiedlichen Stichproben daraufhin geprüft, ob sie mit der Nullhypothese verträglich sind, dass ihre Stufen in allen Bedingungen jeweils dieselben Auftretenswahrscheinlichkeiten besitzen.¹⁴ Die Alternativhypothese ist ungerichtet, der Test asymptotisch korrekt bei wachsender Stichprobengröße.

Als Beispiel soll die politische Orientierung von 100 Studenten aus zwei Studiengängen getestet werden. AV sei die gewählte von fünf Parteien.

```
> voteX <- rep(LETTERS[1:5], c(3, 8, 12, 19, 8))      # Ergebnisse Fach X
> voteY <- rep(LETTERS[1:5], c(8, 17, 16, 7, 2))      # Ergebnisse Fach Y
> vote   <- c(voteX, voteY)                            # beide Ergebnisse

# entsprechender Faktor mit Gruppenzugehörigkeiten
> studies <- factor(rep(c("X", "Y"), c(length(voteX), length(voteY))))
> cTab     <- table(studies, vote)                      # Kontingenztafel
> addmargins(cTab)                                     # Randsummen

          vote
studies A   B   C   D   E   Sum
      X  3   8  12  19  8   50
      Y  8  17  16   7  2   50
      Sum 11  25  28  26 10  100

> chisq.test(cTab, simulate.p.value=TRUE)
Pearson's Chi-squared test with simulated p-value (based on 2000 replicates)
```

¹⁴ Dieser Test ist auch bei quantitativen Variablen durchführbar, wobei zunächst eine Einteilung der Werte in disjunkte Klassen vorzunehmen ist. Die getestete Nullhypothese ist dann in dem Sinne schwächer, dass nur die Gleichheit der Verteilungen bzgl. der vorgenommenen Klasseneinteilung getestet wird.

```
data: cTab
X-squared = 15.2226, df = NA, p-value = 0.005497
```

6.2.3 χ^2 -Test für mehrere Auftretenswahrscheinlichkeiten

Für eine in mehr als einer Bedingung erhobene dichotome Variable prüft `prop.test()`, ob die Trefferwahrscheinlichkeit in allen Bedingungen dieselbe ist. Es handelt sich also um eine Hypothese zur Gleichheit von Verteilungen einer Variable in mehreren Bedingungen. Im Gegensatz zum Binomialtest und zu Fishers exaktem Test (vgl. Abschn. 6.2.5) ist `prop.test()` nur asymptotisch korrekt bei wachsender Stichprobengröße. Die Alternativhypothese ist ungerichtet.

```
> prop.test(x=(Erfolge), n=(Stichprobenumfänge), p=NULL)
```

Die Argumente `x`, `n` und `p` beziehen sich auf die Anzahl der Treffer, die Stichprobengrößen und die Trefferwahrscheinlichkeiten in den Bedingungen unter der Nullhypothese. Für sie können Vektoren gleicher Länge mit Einträgen für jede Gruppe angegeben werden. Anstelle von `x` und `n` kann auch eine Matrix mit zwei Spalten übergeben werden, die in jeder Zeile die Zahl der Treffer und Nieten für jeweils eine Gruppe enthält. Ohne konkrete Angabe für `p` testet `prop.test()` die Hypothese, dass die Trefferwahrscheinlichkeit in allen Bedingungen dieselbe ist.

Als Beispiel soll die Nullhypothese getestet werden, dass in drei Gruppen, hier gleicher Größe, jeweils dieselbe Trefferwahrscheinlichkeit vorliegt.

```
> total <- c(5000, 5000, 5000)                                # Gruppengrößen
> hits  <- c(585, 610, 539)                               # Anzahl Treffer
> prop.test(hits, total)
3-sample test for equality of proportions without continuity correction
data: hits out of total
X-squared = 5.0745, df = 2, p-value = 0.07908
alternative hypothesis: two.sided
sample estimates:
prop 1 prop 2 prop 3
0.1170 0.1220 0.1078
```

Die Ausgabe beinhaltet den Wert der χ^2 -Teststatistik (`X-squared`) samt des zugehörigen p -Wertes (`p-value`) zusammen mit den Freiheitsgraden (`df`) und schließlich die relativen Erfolgshäufigkeiten in den Gruppen (`sample estimates`). Das selbe Ergebnis lässt sich auch durch geeignete Anwendung des χ^2 -Tests auf Gleichheit von Verteilungen erzielen. Hierfür müssen zunächst für jede der drei Stichproben auch die Auftretenshäufigkeiten der zweiten Kategorie berechnet und zusammen mit jenen der ersten Kategorie in einer Matrix zusammengestellt werden.

```
> (mat <- cbind(hits, total-hits))    # Matrix mit Treffern und Nieten
 [,1] [,2]
[1,]  585 4415
[2,]  610 4390
[3,]  539 4461
```

```
> chisq.test(mat)
Pearson's Chi-squared test
data: mat
X-squared = 5.0745, df = 2, p-value = 0.07908
```

Ergeben sich die Gruppen durch Ausprägungen einer ordinalen Variable, ist mit der Funktion `prop.trend.test()` ebenfalls über eine χ^2 -Teststatistik die spezialisiertere Prüfung möglich, ob die Erfolgswahrscheinlichkeiten der dichotomen Variable einem Trend bzgl. der ordinalen Variable folgen.¹⁵

6.2.4 Fishers exakter Test auf Unabhängigkeit

Werden zwei dichotome Variablen in einer Stichprobe erhoben, kann mit Fishers exaktem Test die Nullhypothese geprüft werden, dass beide Variablen unabhängig sind.¹⁶ Anders als beim χ^2 -Test handelt es sich um einen exakten Test, der auch bei kleinen Stichproben anwendbar ist. Zudem sind hier auch gerichtete Alternativhypthesen über die Richtung des Zusammenhangs möglich.

```
> fisher.test(x, y=NULL, or=1, conf.level=0.95,
+               alternative=c("two.sided", "less", "greater"))
```

Unter `x` ist entweder die 2×2 Kontingenztafel zweier dichotomer Variablen oder ein Objekt der Klasse `factor` mit zwei Stufen anzugeben, das die Ausprägungen der ersten dichotomen Variable enthält. In diesem Fall muss auch ein Faktor `y` mit zwei Stufen und derselben Länge wie `x` angegeben werden, der Daten derselben Beobachtungsobjekte beinhaltet. Das Argument `alternative` bestimmt, ob zweiseitig, links- oder rechtsseitig getestet werden soll. Die Richtung der Fragestellung bezieht sich dabei auf die Größe der Odds Ratio in der theoretischen Kontingenztafel. Linksseitiges Testen bedeutet, dass die Odds Ratio unter der Alternativhypothese kleiner als 1, rechtsseitiges Testen entsprechend, dass sie größer als 1 ist. Mit dem Argument `conf.level` wird die Breite des Konfidenzintervalls für die Odds Ratio festgelegt.

Im Beispiel soll geprüft werden, ob das Ergebnis eines diagnostischen Instruments für eine bestimmte Krankheit wie gewünscht positiv mit dem Vorliegen dieser Krankheit zusammenhängt.

```
# Gruppenzugehörigkeit: 10 Gesunde, 5 Kranke
> disease <- factor(rep(c("no", "yes"), c(10, 5)))
> diagN <- rep(c("isHealthy", "isIll"), c(8, 2))    # Diagnose für Gesunde
> diagY <- rep(c("isHealthy", "isIll"), c(1, 4))    # Diagnose für Kranke
> diagT <- factor(c(diagN, diagY))                 # alle Diagnosen
> contT1 <- table(disease, diagT)                   # Kontingenztafel
```

¹⁵ Für den Cochran-Armitage-Trend-Test vgl. `independence_test()` aus dem `coin` Paket.

¹⁶ Die Nullhypothese ist äquivalent zur Hypothese, dass die Odds Ratio der Kontingenztafel beider Variablen gleich 1 ist. Dabei ist die Odds Ratio der Quotient der Zellen $(a/b)/(c/d)$, anders geschrieben $(a \cdot d)/(b \cdot c)$. Die Zelle a soll hierbei diejenige links oben sein, b die rechts daneben liegende, c und d in gleicher Reihenfolge die darunter. Der Test lässt sich auf Variablen mit mehr als zwei Stufen verallgemeinern, vgl. `?fisher.test`.

```
> addmargins(contT1)                                     # Randsummen
      diagT
disease isHealthy isIll Sum
  no       8     2   10
  yes      1     4   5
  Sum      9     6   15

> fisher.test(contT1, alternative="greater")
Fisher's Exact Test for Count Data
data: contT1
p-value = 0.04695
alternative hypothesis: true odds ratio is greater than 1
95 percent confidence interval:
1.031491 Inf
sample estimates:
odds ratio
12.49706
```

Die Ausgabe enthält neben dem p -Wert (`p-value`) das Konfidenzintervall für die Odds Ratio in der gewünschten Breite sowie die Maximum-Likelihood Schätzung der Odds Ratio gegeben die Randhäufigkeiten (`sample estimates`). Der p -Wert lässt sich manuell nachprüfen: hierfür müssen die Punktwahrscheinlichkeiten für die vorliegende sowie für (im Sinne der Alternativhypothese) extremere Kontingenztafeln bei gleichen Randhäufigkeiten mit der Hypergeometrischen Verteilung ermittelt und summiert werden. Im gegebenen Fall besteht nur eine Möglichkeit, die Kontingenztafel extremer zu machen, ohne die Randhäufigkeiten zu ändern.

```
# Punktwahrscheinlichkeit für die gegebene Kontingenztafel
> (p1 <- choose(8+1, 8) * choose(2+4, 2) / choose(10+5, 10))
[1] 0.04495504

# Kontrolle über Wahrscheinlichkeitsfkt. der Hypergeometrischen Verteilung
> dhyper(8, 8+2, 1+4, 8+1)                                # ...

# Punktwahrscheinlichkeit für extremere Kontingenztafel
> (p2 <- choose(9+0, 9) * choose(1+5, 1) / choose(10+5, 10))
[1] 0.001998002

# Kontrolle über Wahrscheinlichkeitsfkt. der Hypergeometrischen Verteilung
> dhyper(9, 9+1, 0+5, 9+0)                                # ...
> (pVal <- p1+p2)    # Summe der Punktwahrscheinlichkeiten -> p-Wert
[1] 0.04695305
```

6.2.5 Fishers exakter Test auf Gleichheit von Verteilungen

Für Daten einer dichotomen Variable aus zwei unabhängigen Stichproben prüft Fishers exakter Test die Nullhypothese, dass die Variablen in beiden Bedingungen identische Erfolgswahrscheinlichkeit besitzen. Anders als beim χ^2 -Test handelt es sich um einen exakten Test, der auch bei kleinen Stichproben anwendbar ist. Zudem sind hier gerichtete Alternativhypthesen möglich.

Der Aufruf von `fisher.test()` ist identisch zu jenem beim Test auf Unabhängigkeit (vgl. auch Fußnote 16). Unter `x` ist hier entweder die 2×2 Kontingenztafel einer dichotomen Zufallsvariable und eines Gruppierungsfaktors mit zwei Ausprägungen anzugeben oder ein Objekt der Klasse `factor` mit zwei Stufen, das die Ausprägungen der Variable in der ersten Stichprobe enthält. In letzterem Fall muss auch ein Faktor `y` mit zwei Stufen und derselben Länge wie `x` angegeben werden, der Daten derselben Variable aus einer zweiten Stichprobe beinhaltet. Der Test lässt sich auch für Nullhypotesen anwenden, die nicht in einer Gleichheit von Verteilungen bestehen. Für diesen Fall kann das Argument `or=<Zahl1>` auf den Wert gesetzt werden, den die Odds Ratio unter Gültigkeit der Nullhypothese besitzen soll.

Im Beispiel sei an je einer Stichprobe aus der Population der Frauen und der Männer erhoben worden, ob die Person raucht. Geprüft wird die Nullhypothese, dass der Anteil der Raucher in beiden Populationen gleich ist, wobei als Alternativhypothese hier vermutet wird, dass Frauen generell häufiger rauchen. Der Aufbau der Kontingenztafel verlangt nach einem linksseitigen Test.

```
> smokesFem <- rbinom(20, size=1, p=0.6)           # Stichprobe Frauen
> smokesMale <- rbinom(20, size=1, p=0.4)          # Stichprobe Männer

# Faktoren, die Rauchverhalten und Geschlecht codieren
> smokes   <- factor(c(smokesFem, smokesMale), labels=c("no", "yes"))
> sex       <- factor(rep(c("f", "m"), c(20, 20)))      # Kontingenztafel
> contT2   <- table(sex, smokes)                   # Randsummen
> addmargins(contT2)
  smokes
sex no yes Sum
  f   8 12 20
  m 16  4 20
Sum 24 16 40

> fisher.test(contT2, alternative="less")
Fisher's Exact Test for Count Data
data: contT2
p-value = 0.01124
alternative hypothesis: true odds ratio is less than 1
95 percent confidence interval:
0.0000000 0.6668953
sample estimates:
odds ratio
0.1751986
```

6.2.6 Kennwerte für 2×2 Kontingenztafeln

Bei der Analyse von 2×2 Kontingenztafeln als Ergebnis der Anwendung einer dichotomen Klassifikation sind mitunter einige Kennwerte von Interesse, die verschiedene Eigenschaften der Klassifikation beschreiben. Hierzu zählen die Sensitivität, Spezifität und Relevanz bzw. Positiver Vorhersagewert. R stellt zu ihrer Ermittlung

keine eigenen Funktionen bereit, eine manuelle Berechnung ist jedoch unkompliziert.¹⁷

Als Beispiel seien die Daten herangezogen, die bereits für Fishers exakten Test auf Unabhängigkeit verwendet wurden. Dabei sollen die Abkürzungen TP (True Positive, Hit) für richtig positive, TN (True Negative) für richtig negative, FP (False Positive) für falsch positive und FN (False Negative, Miss) für falsch negative Diagnosen stehen.

```
> addmargins(contT1)
      diagT
disease isHealthy isIll Sum
  no        8      2   10
  yes       1      4    5
  Sum       9      6   15

> TP <- contT1[2, 2]                      # True Positive / Hit
> TN <- contT1[1, 1]                      # True Negative
> FP <- contT1[1, 2]                      # False Positive
> FN <- contT1[2, 1]                      # False Negative / Miss
```

Die Prävalenz der Krankheit ist der Anteil der Kranken an der Gesamtzahl von Beobachtungen. Im Kontext des Satzes von Bayes entspricht dies der a-priori Wahrscheinlichkeit eines Merkmals.

```
> (prevalence <- sum(contT1[2, ]) / sum(contT1))
[1] 0.3333333
```

Die Sensitivität, in anderem Kontext auch Recall genannt, ist der Quotient aus TP und der Summe von TP und FN, also das Verhältnis von richtig entdeckten zu allen zu entdeckenden Elementen. In der Sprechweise inferenzstatistischer Tests wäre dies auf theoretischer Ebene die Power eines Tests. Entsprechend wäre der Fehler zweiter Art (β) gleich $1 - \text{Sensitivität}$.

```
> (sensitivity <- recall <- TP / (TP+FN))
[1] 0.8
```

Die Spezifität ist der Quotient aus TN und der Summe von TN und FP, also hier das Verhältnis von richtig als gesund Eingestuften zu allen Gesunden. In der Sprechweise inferenzstatistischer Tests wäre $1 - \text{Spezifität}$ auf theoretischer Ebene gleich dem Fehler erster Art (α), dem somit $FP/(TN + FP)$ entspräche.

```
> (specificity <- TN / (TN+FP))
[1] 0.8
```

Die Relevanz, je nach Kontext auch als Präzision oder Positiver Vorhersagewert bezeichnet, ist der Quotient aus TP und der Summe von TP und FP. Er gibt damit hier an, welcher Anteil der als krank Diagnostizierten tatsächlich krank ist. Im Kontext des Satzes von Bayes entspricht dies der a-posteriori Wahrscheinlichkeit eines Merkmals gegeben die positive Diagnose.

¹⁷ Das Paket **ROCR** (Sing et al., 2009) ermöglicht weitergehende Analysen, etwa die Berechnung von ROC-Kurven im Kontext von Auswertungen nach der Signalentdeckungstheorie.

```
> (relevance <- precision <- TP / (TP+FP))
[1] 0.6666667
```

Der Anteil richtiger Diagnosen an allen Diagnosen wird auch als Rate der korrekten Klassifikation bezeichnet und ist der Quotient aus der Summe der Diagonalelemente und der Summe aller Elemente.

```
> (probT <- sum(diag(contT1)) / sum(contT1))
[1] 0.8
```

Der sog. *F*-Wert als harmonisches Mittel von Präzision und Recall wir bisweilen als integriertes Gütemaß für eine Klassifikation herangezogen. Er ist nicht mit Werten einer *F*-Verteilung zu verwechseln.

```
> (Fval <- 2 * (precision*recall) / (precision+recall))
[1] 0.7272727
```

6.3 Maße für Zusammenhang und Übereinstimmung kategorialer Daten

6.3.1 Zusammenhang ordinaler Variablen: Spearmans ρ und Kendalls τ

Zur Berechnung der Kovarianz und Korrelation zweier ordinaler Variablen nach Spearman und nach Kendall dienen die `cov()` und `cor()` Funktionen, für die als Argument `method="spearman"` bzw. `method="kendall"` anzugeben ist. Spearmans ρ entspricht dabei der gewöhnlichen Pearsonschen Korrelation beider Variablen, nachdem ihre Werte durch den zugehörigen Rang ersetzt wurden.

```
> vec1 <- c(100, 76, 56, 99, 50, 62, 36, 69, 55, 17)
> vec2 <- c(42, 74, 22, 99, 73, 44, 10, 68, 19, -34)
> cor(vec1, vec2, method="spearman")           # Spearmans rho
[1] 0.6727273

> cor(rank(vec1), rank(vec2))                 # Korrelation der Ränge
[1] 0.6727273
```

Für Kendalls τ ist die Differenz der Anzahl konkordanter und diskonkordanter Paare von je zwei abhängigen Messwerten zu bilden und an der Gesamtzahl möglicher Paare zu relativieren. Enthalten dafür X und Y die Daten aus den abhängigen Stichproben, ist ein Paar $\{(x_i, y_i), (x_j, y_j)\}$ mit $i \neq j$ dann konkordant, wenn x_i und x_j dieselbe Rangordnung aufweisen wie y_i und y_j . Identische Werte jeweils innerhalb von X und Y (Bindungen) seien dabei ausgeschlossen.

```
> cor(vec1, vec2, method="kendall")            # Kendalls tau
[1] 0.6

# Matrizen der paarweisen Rangvergleiche jeweils in X und Y
> cmpMat1 <- outer(vec1, vec1, ">")          # Vergleiche in X
```

```

> cmpMat2 <- outer(vec2, vec2, ">")           # Vergleiche in Y
> selMat   <- upper.tri(cmpMat1)      # relevant nur obere Dreiecksmatrix

# Anzahl konkordanter Paare -> Rangordnung in X und Y identisch
> nCP <- sum((cmpMat1 == cmpMat2)[selMat])
# Anzahl diskonkordanter Paare -> Rangordnung in X und Y verschieden
> nDP    <- sum((cmpMat1 != cmpMat2)[selMat])
> nSubj  <- length(vec1)                      # Anzahl Versuchspersonen
> nPairs <- choose(nSubj, 2)                   # Anzahl möglicher Paare
> (tau   <- (nCP-nDP) / nPairs)               # Kendalls tau
[1] 0.6

```

Spearmans und Kendalls empirische Maße des Zusammenhangs zweier Variablen lassen sich mit `cor.test()` inferenzstatistisch daraufhin prüfen, ob sie mit der Nullhypothese verträglich sind, dass deren theoretischer Zusammenhang 0 ist.

```

> cor.test(x=<Vektor1>, y=<Vektor2>, alternative=c("two.sided", "less",
+           "greater"), method=c("pearson", "kendall", "spearman"), use)

```

Die Daten beider Variablen sind als Vektoren derselben Länge über die Argumente `x` und `y` anzugeben. Ob die Alternativhypothese zwei- ("`two.sided`"), links- (negativer Zusammenhang, "`less`") oder rechtsseitig (positiver Zusammenhang, "`greater`") ist, legt das Argument `alternative` fest. Für den Test von Rangdaten nach Spearman und Kendall ist das Argument `method="spearman"` bzw. `method="kendall"` zu setzen. Ebenfalls können über das Argument `use` verschiedene Strategien zur Behandlung fehlender Werte ausgewählt werden (vgl. Abschn. 2.12.4).

```

> cor.test(vec1, vec2, method="spearman")
Spearman's rank correlation rho
data: vec1 and vec2
S = 54, p-value = 0.03938
alternative hypothesis: true rho is not equal to 0
sample estimates:
rho
0.6727273

```

Die Ausgabe des Spearman-Tests beinhaltet den Wert der Hotelling-Pabst-Teststatistik (`S`) nebst zugehörigem p -Wert (`p-value`) und Spearmans ρ (`rho`). `S` berechnet sich als Summe der quadrierten Differenzen zwischen den Rängen zugehöriger Messwerte in beiden Stichproben.¹⁸

```

> sum((rank(vec1)-rank(vec2))^2)                 # Teststatistik S
[1] 54

> cor.test(vec1, vec2, method="kendall")          # Test nach Kendall
Kendall's rank correlation tau
data: vec1 and vec2

```

¹⁸ Die Berechnung des zugehörigen p -Wertes ist nur über eine intern definierte Funktion möglich, die Verteilungsfunktion der Teststatistik ist nicht direkt als R-Funktion vorhanden.

```
T = 36, p-value = 0.01667
alternative hypothesis: true tau is not equal to 0
sample estimates:
tau
0.6
```

Die Ausgabe des Kendall-Tests beinhaltet den Wert der Teststatistik (T) gemeinsam mit dem zugehörigen p -Wert (p-value) sowie Kendalls τ (tau). T ist hier als Anzahl konkordanter Paare definiert – die Differenz der Anzahl konkordanter und diskonkordanter Paare wäre gleichermaßen geeignet, da sich beide zur festen Anzahl möglicher Paare addieren, sofern keine Bindungen vorliegen (vgl. Fußnote 18).

6.3.2 Weitere Zusammenhangsmaße: φ , Cramérs V, Kontingenzkoeffizient, Goodman und Kruskals γ , Somers' d, ICC, r_{WG}

Als weitere Maße des Zusammenhangs zweier kategorialer Variablen dienen der φ -Koeffizient (bei dichotomen Variablen gleich deren Korrelation), dessen Verallgemeinerung Cramérs V (identisch mit φ für dichotome Variablen) und der Kontingenzkoeffizient CC. Diese Kennwerte basieren auf dem χ^2 -Wert der Kontingenztafel der gemeinsamen Häufigkeiten beider Variablen. Beruht die Kontingenztafel auf N Beobachtungen, und ist L der kleinere Wert der Anzahl ihrer Zeilen und Spalten, gelten folgende Zusammenhänge: $N \cdot (L - 1)$ ist der größtmögliche χ^2 -Wert. Weiterhin gilt $\varphi = \sqrt{\chi^2/N}$, $V = \sqrt{\chi^2/(N \cdot (L - 1))}$, $CC = \sqrt{\chi^2/(N + \chi^2)}$. Alle Kennwerte lassen sich mit der assocstats() Funktion aus dem vcd Paket (Meyer et al., 2010) oder manuell ermitteln.

```
> vec1 <- c(100, 76, 56, 99, 50, 62, 36, 69, 55, 17)
> vec2 <- c(42, 74, 22, 99, 73, 44, 10, 68, 19, -34)
> cTab <- table(vec1, vec2)           # Kontingenztafel
> N    <- sum(cTab)                 # Anzahl Beobachtungen
> library(vcd)                     # Auswertung mit vcd Paket
> assocstats(cTab)
          X^2 df P(> X^2)
Likelihood Ratio 46.052 81 0.99938
Pearson         90.000 81  0.23134

Phi-Coefficient   : 3
Contingency Coeff. : 0.949
Cramer's V        : 1

# erwartete Zellhäufigkeiten
> expected <- outer(rowSums(cTab), colSums(cTab)) / N
> (chisqVal <- sum((cTab-expected)^2 / expected))      # chi^2-Wert
[1] 90
```

```

> chisq.test(cTab)$statistic          # Kontrolle: chi^2 aus Test
X-squared
 90
> (phiVal <- sqrt(chisqVal / N))    # phi-Koeffizient
[1] 3

> L       <- min(nrow(cTab), ncol(cTab))
> phisqMax <- L-1                  # maximaler phi^2-Wert
> chisqMax <- N*phisqMax          # maximaler chi^2-Wert
> (CrV     <- sqrt(chisqVal / chisqMax))      # Cramérs V
[1] 1

# Kontingenzkoeffizient
> (contCoeff <- sqrt(chisqVal / (N+chisqVal)))
[1] 0.9486833

```

Goodman und Kruskals γ als Maß für den Zusammenhang ordinaler Variablen lässt sich über die `rcorr.cens()` Funktion aus dem `Hmisc` Paket berechnen, wobei das Argument `outx=TRUE` zu setzen ist. Ebenfalls aus dem `Hmisc` Paket stammt die Funktion `somers2()`, die Somers' d , also die Rangkorrelation einer quantitativen und einer dichotomen Variable ausgibt. Die Intra-Klassen-Korrelation zumindest intervallskalierter Variablen wird von der `icc()` Funktion aus dem `irr` Paket (Gamer et al., 2009) oder von der `ICC()` Funktion des `psych` Pakets ermittelt. Verschiedene r_{WG} Koeffizienten lassen sich mit gleichlautenden Funktionen des `multilevel` Pakets (Bliese, 2008) berechnen. Auch Kennwerte, die als Maß der Übereinstimmung mehrerer Beurteiler dienen, können als Zusammenhangsmaß von kategorialen Variablen verstanden werden, vgl. Abschn. 6.3.3.

Die Nullhypothese, dass zwei dichotome Variablen in mehreren, durch eine dritte kategoriale Variable gebildeten Bedingungen jeweils unabhängig sind, prüft der Cochran-Mantel-Haenszel-Test, für den die Funktionen `mantelhaen.test()` und aus dem `coin` Paket `cmh_test()` existieren. Beide Funktionen sind auch im allgemeineren Fall anwendbar, dass die Unabhängigkeit kategorialer Variablen mit mehr als zwei Stufen zu testen ist.

6.3.3 Inter-Rater-Übereinstimmung: Cohens κ , Fleiss' κ , Kendalls W

Wenn mehrere Personen dieselben Objekte in Kategorien einordnen, ist häufig der Grad der Übereinstimmung der Urteile relevant, bei dem man auch von Inter-Rater-Reliabilität spricht. Hierbei lassen sich zum einen Fälle unterscheiden, bei denen entweder nur zwei oder auch mehr Rater vorhanden sind, zum anderen Kategorien, die Ausprägungen einer nur nominalen, oder aber einer ordinalen Variable darstellen. Soweit nicht anders vermerkt, stammen die folgend genannten Funktionen aus dem `irr` Paket, das neben den beschriebenen auch noch weitere Funktionen zur Analyse der Inter-Rater-Übereinstimmung mitbringt.

6.3.3.1 Prozentuale Übereinstimmung

Die prozentuale Übereinstimmung der Urteile zweier oder mehr Personen, die dieselben Beobachtungsobjekte in mehrere Kategorien einteilen, kann mit der `agree()` Funktion oder manuell mittels `table()` berechnet werden: übereinstimmend vergebene Kategorien stehen in der resultierenden Kontingenztafel der gemeinsamen Häufigkeiten in der Diagonale, sofern Zeilen und Spalten dieselben Kategorien in derselben Reihenfolge umfassen. Als Beispiel diene jenes aus Bortz et al. (2008, p. 458 ff.), in dem zwei Rater 100 Beobachtungsobjekte in drei Kategorien einteilen.

```
> categ <- c("V", "N", "P")                                # Rating-Kategorien
> rater1 <- rep(categ, c(60, 30, 10))                      # Urteile Rater 1

# Urteile Rater 2
> rater2 <- c(rep(categ, c(53, 5, 2)), rep(categ, c(11, 14, 5)),
+             rep(categ, c(1, 6, 3)))

> r1fac <- factor(rater1, levels=categ)                      # Urteile als Faktor
> r2fac <- factor(rater2, levels=categ)                      # Urteile als Faktor
> cTab <- table(r1fac, r2fac)                                # Kontingenztafel
> addmargins(cTab)                                           # Randsummen
  r2fac
r1fac   V   N   P Sum
  V    53   5   2  60
  N    11  14   5  30
  P     1   6   3  10
  Sum  65  25  10 100

> library(irr)                                              # Auswertung mit Paket irr
> agree(cbind(rater1, rater2))
Percentage agreement (Tolerance=0)
Subjects = 100
  Raters = 2
  %-agree = 70

# manuelle Berechnung: Summe der Diagonalelemente in der
# Kontingenztafel beider Rater / Anzahl Beobachtungen
> (f0bs <- sum(diag(cTab)) / sum(cTab))
[1] 0.7

# Daten Rater 3
> rater3 <- c(rep(categ, c(48, 8, 3)), rep(categ, c(15, 10, 7)),
+             rep(categ, c(3, 4, 2)))

> library(irr)                                              # Auswertung mit Paket irr
> agree(cbind(rater1, rater2, rater3))
Percentage agreement (Tolerance=0)
Subjects = 100
  Raters = 3
  %-agree = 60
```

```
> r3fac <- factor(rater3, levels=categ)           # Urteile als Faktor

# 3D-Kontingenztafel der Urteile aller 3 Beobachter
> cTab3 <- table(r1fac, r2fac, r3fac)
# Summe der Diagonalelemente / Anzahl Beobachtungen
> sum(diag(apply(cTab3, 3, diag))) / sum(cTab3)
[1] 0.6
```

6.3.3.2 Cohens κ

Der Grad, mit dem die Urteile zweier Personen übereinstimmen, die dieselben Beobachtungsobjekte in mehrere ungeordnete Kategorien einteilen, kann auch mit Cohens κ -Koeffizient ausgedrückt werden. Er zielt darauf ab, den Anteil beobachteter Übereinstimmungen mit dem Anteil auch zufällig erwartbarer Übereinstimmungen in Beziehung zu setzen. Cohens κ lässt sich mit der kappa2() Funktion berechnen, die als Argument eine spaltenweise aus den Urteilen beider Rater zusammengesetzte Matrix erwartet.

```
> library(irr)                                # Auswertung mit Paket irr
> kappa2(cbind(rater1, rater2))
Cohen's Kappa for 2 Raters (Weights: unweighted)
Subjects = 100
Raters = 2
Kappa = 0.429
z = 5.46
p-value = 4.79e-08

# beobachteter Anteil an Übereinstimmungen (Diagonalelemente)
> f0bs <- sum(diag(cTab)) / sum(cTab)

# zufällig erwartbarer Anteil an Übereinstimmungen
> fExp    <- sum(rowSums(cTab) * colSums(cTab)) / sum(cTab)^2
> (Ckappa <- (f0bs-fExp) / (1-fExp))          # Cohens kappa
[1] 0.4285714
```

6.3.3.3 Fleiss' κ

Die Funktion kappam.fleiss() ermittelt für das Ausmaß der Übereinstimmung von mehr als zwei Urteilern bzgl. der Einteilung in mehrere ungeordnete Kategorien den κ -Koeffizienten nach Fleiss. Als Beispiel diene jenes aus Bortz et al. (2008, p. 460 ff.), in dem sechs Rater 30 Begriffe einer von fünf Kategorien (a–e) zuordnen sollen.

```
> rater1 <- letters[c(4,2,2,5,2, 1,3,1,1,5, 1,1,2,1,2, 3,1,1,2,1,
+                   5,2,2,1,1, 2,1,2,1,5)]
> rater2 <- letters[c(4,2,3,5,2, 1,3,1,1,5, 4,2,2,4,2, 3,1,1,2,3,
+                   5,4,2,1,4, 2,1,2,3,5)]
> rater3 <- letters[c(4,2,3,5,2, 3,3,3,4,5, 4,4,2,4,4, 3,1,1,4,3,
+                   5,4,4,4,4, 2,1,4,3,5)]
```

```

> rater4 <- letters[c(4,5,3,5,4, 3,3,3,4,5, 4,4,3,4,4, 3,4,1,4,5,
+                      5,4,5,4,4, 2,1,4,3,5)]
> rater5 <- letters[c(4,5,3,5,4, 3,5,3,4,5, 4,4,3,4,4, 3,5,1,4,5,
+                      5,4,5,4,4, 2,5,4,3,5)]
> rater6 <- letters[c(4,5,5,5,4, 3,5,4,4,5, 4,4,3,4,5, 5,5,2,4,5,
+                      5,4,5,4,5, 4,5,4,3,5)]

# Matrix der Ratings: Rater in den Spalten, Objekte in den Zeilen
> rateMat <- cbind(rater1, rater2, rater3, rater4, rater5, rater6)
> library(irr)                                     # Auswertung mit Paket irr
> kappa.fleiss(rateMat)
Fleiss' Kappa for m Raters
Subjects = 30
Raters = 6
Kappa = 0.43
z = 17.7
p-value = 0

# manuelle Berechnung
> nRater <- ncol(rateMat)                         # Anzahl Rater
> nObs    <- nrow(rateMat)                         # Anzahl Objekte

# Vektor aller Urteile
> ratings <- c(rater1, rater2, rater3, rater4, rater5, rater6)

# zugehöriger Faktor, welches Objekt beurteilt wurde
> obsFac <- factor(rep(1:nObs, nRater))

# Kreuztabelle der Häufigkeiten, wie oft jedes Objekt (Zeilen)
# einer Kategorie (Spalten) zugeordnet wurde
> cTab <- xtabs(~ obsFac + ratings)

# relative Häufigkeit jeder Beurteilungskategorie
> (rateTab <- prop.table(table(ratings)))
ratings
      a          b          c          d          e
0.14444444 0.14444444 0.16666667 0.30555556 0.23888889

# per Zufall erwartbarer Anteil an Übereinstimmungen
> (fExp <- sum(rateTab^2))
[1] 0.2199383

# beobachtete Übereinstimmung der Rater pro Objekt
> fObsAll <- apply(cTab, 1, function(x) {
+   sum(x*(x-1)) / (nRater*(nRater-1)) } )

> (fObs <- mean(fObsAll))      # mittlere beobachtete Übereinstimmung
[1] 0.5555556

> (Fkappa <- (fObs-fExp) / (1-fExp))           # Fleiss' kappa
[1] 0.4302445

```

6.3.3.4 Gewichtetes Cohens κ

Liegen geordnete Kategorien vor, kann nicht nur berücksichtigt werden, wenn keine Übereinstimmung vorliegt, sondern auch, in welchem Ausmaß dies der Fall ist bzw. wie sehr die Urteile auseinander liegen. Für die Situation mit zwei Ratern stehen in der `kappa2()` Funktion mehrere Gewichtungen zur Verfügung: wird das Argument `weight="equal"` gesetzt, gilt jede Kategorie als im selben Maße unterschiedlich zu den jeweiligen Nachbarkategorien – es soll also von gleichabständigen Kategorien ausgegangen werden. Als Beispiel diene jenes aus Bortz et al. (2008, p. 484 ff.), in dem zwei Rater die zu erwartende Einschaltquote von 100 Fernsehsendungen einschätzen.

```
# Beurteilungskategorien
> categ <- c("<10%", "11-20%", "21-30%", "31-40%", "41-50%", ">50%")
> lvls  <- factor(categ, levels=categ)           # Kategorien als Faktor
> tv1   <- rep(lvls, c(22, 21, 23, 16, 10, 8))    # Urteile Rater 1

# Urteile Rater2
> tv2 <- rep(rep(lvls, nlevels(lvls)), c(5,8,1,2,4,2, 3,5,3,5,5,0,
+                                         1,2,6,11,2,1, 0,1,5,4,3,3, 0,0,1,2,5,2, 0,0,1, 2,1,4))

> cTab <- table(tv1, tv2)                         # Kontingenztafel der Rater
> addmargins(cTab)                                # Randsummen
      tv2
tv1   <10% 11-20% 21-30% 31-40% 41-50% >50% Sum
<10%     5     8     1     2     4     2   22
11-20%    3     5     3     5     5     0   21
21-30%    1     2     6    11     2     1   23
31-40%    0     1     5     4     3     3   16
41-50%    0     0     1     2     5     2   10
>50%     0     0     1     2     1     4    8
Sum       9    16    17    26    20    12  100

> library(irr)                                    # Auswertung mit Paket irr
> kappa2(cbind(tv1, tv2), weight="equal")        # gewichtetes kappa
Cohen's Kappa for 2 Raters (Weights: equal)
Subjects = 100
Raters = 2
Kappa = 0.316
z = 5.38
p-value = 7.3e-08
```

Für die manuelle Berechnung muss für jede Zelle der Kontingenztafel beider Rater ein Gewicht angegeben werden, dass der Unterschiedlichkeit der von den Ratern verwendeten Kategorien entspricht. Übereinstimmende Kategorien (Diagonale) haben dabei eine Unterschiedlichkeit von 0, die maximale Unterschiedlichkeit ist 1. Ist von Gleichabständigkeit der Kategorien auszugehen, sind die Gewichte dazwischen gleichabständig zu wählen.

```
> P <- ncol(cTab)                                # Anzahl Kategorien
```

```
# bei Unabhängigkeit erwartete Häufigkeiten
> (expected <- outer(rowSums(cTab), colSums(cTab)))
   <10% 11-20% 21-30% 31-40% 41-50% >50%
<10%    1.98    3.52    3.74    5.72    4.4  2.64
11-20%   1.89    3.36    3.57    5.46    4.2  2.52
21-30%   2.07    3.68    3.91    5.98    4.6  2.76
31-40%   1.44    2.56    2.72    4.16    3.2  1.92
41-50%   0.90    1.60    1.70    2.60    2.0  1.20
>50%    0.72    1.28    1.36    2.08    1.6  0.96

# gleichabständige Gewichtungsfaktoren für nicht übereinstimmende Kategorien
> (myWeights <- seq(0, 1, length.out=P))
[1] 0.0 0.2 0.4 0.6 0.8 1.0

> weightsMat <- matrix(0, nrow=P, ncol=P)      # Matrix der Gewichte
> for (i in 1:P) {
+   weightsMat[i, ] <- c(rev(myWeights), myWeights[2:P])[((P-(i-1)):(2*P-i))]
+ }

> weightsMat
 [,1] [,2] [,3] [,4] [,5] [,6]
[1,] 0.0  0.2  0.4  0.6  0.8  1.0
[2,] 0.2  0.0  0.2  0.4  0.6  0.8
[3,] 0.4  0.2  0.0  0.2  0.4  0.6
[4,] 0.6  0.4  0.2  0.0  0.2  0.4
[5,] 0.8  0.6  0.4  0.2  0.0  0.2
[6,] 1.0  0.8  0.6  0.4  0.2  0.0

# Summe der gewichteten beobachteten relativen Häufigkeiten
> wfObs <- sum(cTab * (1-weightsMat)) / sum(cTab)

# Summe der gewichteten erwarteten relativen Häufigkeiten
> wfExp <- sum(expected * (1-weightsMat)) / sum(cTab)

# gewichtetes Cohens kappa
> (wKappa <- (wfObs-wfExp) / (1-wfExp))
[1] 0.3156685
```

6.3.3.5 Kendalls W

Für ordinale Urteile von mehr als zwei Ratern kann Kendalls Konkordanzkoeffizient W von der `kendall()` Funktion ermittelt werden, die als Argument eine spaltenweise aus den durch die Rater vergebenen Urteilen zusammengesetzte Matrix erwartet. Als Beispiel diene jenes aus Bortz et al. (2008, p. 466 ff.), in dem drei Rater sechs Beobachtungsobjekte entsprechend der Ausprägung eines Merkmals in eine Rangreihe bringen.

```
> rater1 <- c(1, 6, 3, 2, 5, 4)          # Urteile Rater 1
> rater2 <- c(1, 5, 6, 2, 4, 3)          # Urteile Rater 2
> rater3 <- c(2, 3, 6, 5, 4, 1)          # Urteile Rater 3
> ratings <- cbind(rater1, rater2, rater3) # Matrix der Urteile
```

```
> library(irr)                                # Auswertung mit Paket irr
> kendall(cbind(rater1, rater2, rater3))      # Kendalls W
Kendall's coefficient of concordance W
Subjects = 6
Raters = 3
W = 0.568
Chisq(5) = 8.52
p-value = 0.130
```

Für die manuelle Berechnung wird zunächst die Matrix der Beurteiler-weisen Ränge benötigt, die im konkreten Beispiel bereits vorliegt, da Rangurteile abgegeben wurden. Bei anderen ordinalen Urteilen wäre sie aus der Matrix der Ratings mit `apply(<Rating>, 2, rank)` zu bilden.

```
> rankMat <- ratings                         # Matrix der Ränge
> nObj      <- nrow(rankMat)                 # Anzahl Objekte
> nRater    <- ncol(rankMat)                 # Anzahl Rater
> (rankSum <- apply(rankMat, 1, sum))        # Rangsumme pro Objekt
[1] 4 14 15 9 13 8

# Summe der quadrierten Abweichungen der Rangsummen vom Erwartungswert
> (S <- sum((rankSum - (nRater*(nObj+1)/2))^2))
[1] 89.5

> (W <- 12*S / (nRater^2 *nObj*(nObj^2 - 1)))  # Kendalls W
[1] 0.568254

> (chisqVal <- nRater * (nObj-1) * W)          # Teststatistik
[1] 8.52381

> (pVal <- 1-pchisq(chisqVal, nObj-1))         # p-Wert
[1] 0.1296329
```

6.3.3.6 Krippendorffs α

Zur Berechnung von Krippendorffs α für Urteile von mehreren Ratern über mehrere Objekte dient die `kripp.alpha()` Funktion, die als Argument eine zeilenweise aus den durch die Rater vergebenen Urteilen zusammengesetzte Matrix erwartet. Die Berechnung von α hängt vom angenommenen Skalenniveau der Urteile ab, das mit dem Argument `method` anzugeben ist. Als Beispiel sei wieder auf jenes aus Bortz et al. (2008, p. 484 ff.) zurückgegriffen. Die Beurteilung der Einschaltquote soll hier einmal nur als nominalskaliert und einmal als intervallskaliert angenommen werden.

```
# bei Annahme nominalskaliert Urteile
> kripp.alpha(rbind(tv1, tv2), method="nominal")
Krippendorff's alpha
Subjects = 100
Raters = 2
alpha = 0.144

# bei Annahme intervallskaliert Urteile
> kripp.alpha(rbind(tv1, tv2), method="interval")
```

```
Krippendorff's alpha
Subjects = 100
Raters = 2
alpha = 0.384
```

Für den Stuart-Maxwell-Test, ob zwei Rater die Kategorien einer mehrstufigen Variable mit denselben Grundwahrscheinlichkeiten verwenden vgl. Abschn. 6.4.12.

6.4 Tests auf Übereinstimmung von Verteilungen

Die im folgenden aufgeführten Tests lassen sich mit Werten geordneter kategorialer Variablen, also etwa mit Rangdaten durchführen und testen Hypothesen darüber, ob die zugehörigen Verteilungen identisch sind. Dies ist oft gleichbedeutend mit der Frage, ob die Lageparameter der Verteilungen übereinstimmen.

6.4.1 Vorzeichen-Test

Der Vorzeichen-Test für den Median ist anwendbar, wenn eine Variable eine bestimmte Verteilung unbekannter Form besitzt. Es wird die Nullhypothese getestet, dass der Median der Verteilung gleich einem bestimmten Wert m_0 ist.¹⁹ Sowohl gerichtete wie ungerichtete Alternativhypothesen sind möglich.

R stellt für die Durchführung des Tests keine eigene Funktion bereit, eine manuelle Rechnung ist jedoch unkompliziert.²⁰ Zunächst werden dazu aus der Stichprobe diejenigen Werte eliminiert, die gleich m_0 sind.²¹ Teststatistik ist die Anzahl der beobachteten Werte größer als m_0 – bei intervallskalierten Daten wird also die Anzahl positiver Differenzen der Werte zum Median unter der Nullhypothese gezählt. Diese Anzahl besitzt unter der Nullhypothese eine Binomialverteilung mit der Trefferwahrscheinlichkeit 0.5.

```
> m0      <- 30                      # Median unter H0
> myData <- sample(0:100, 20, replace=TRUE)    # simulierte Daten
> myData <- myData[myData != m0]            # Werte gleich m0 eliminieren
> N       <- length(myData)              # Anzahl an Beobachtungen
> (obs   <- sum(myData > m0))           # Teststatistik
[1] 15
```

¹⁹ Bei Variablen mit symmetrischer Verteilung und eindeutig bestimmtem Median ist dieser gleich dem Erwartungswert.

²⁰ Als Alternative vgl. die Funktion `median_test()` des `coin` Pakets.

²¹ Auch andere Vorgehensweisen werden diskutiert, insbesondere wenn es viele Werte gleich m_0 gibt. So können im Fall geradzahlig vieler Werte gleich m_0 die Hälfte dieser Werte als kleiner m_0 , die andere Hälfte als größer m_0 codiert werden. Im Fall ungeradzahlig vieler Werte gleich m_0 wird ein Wert eliminiert und dann wie für geradzahlig viele Werte beschrieben verfahren.

Um den p -Wert der Teststatistik zu berechnen, dient die `pbinom(q, size, prob)` Funktion für die Verteilungsfunktion der Binomialverteilung.²²

```
> (pGreater <- 1-pbinom(obs-1, N, 0.5))           # rechtsseitiger Test
[1] 0.02069473

> (pLess <- 1-pbinom(N-obs-1, N, 0.5))          # linksseitig
[1] 0.994091
```

Der p -Wert des zweiseitigen Binomialtests kann als das Doppelte des kleineren der p -Werte der einseitigen Tests definiert werden (vgl. Abschn. 6.1.1, Fußnote 3).

```
> (pTwoSided <- 2 * (1-pbinom(max(obs, N-obs)-1, N, 0.5)))
[1] 0.04138947
```

Der Vorzeichentest lässt sich ebenso auf Daten einer Variable aus zwei abhängigen Stichproben anwenden, deren zugehörige Verteilungen darauf getestet werden sollen, ob ihre Lageparameter identisch sind. Zu diesem Zweck werden die paarweisen Differenzen gebildet und wie beschrieben mit $m_0 = 0$ getestet. Besitzen die Daten aus zwei abhängigen Stichproben nur ein ordinates Messniveau, lässt sich die Teststatistik auch als Anzahl der paarweisen Beobachtungen definieren, bei denen der zweite Wert größer als der erste ist.

6.4.2 Wilcoxon-Vorzeichen-Rang-Test für eine Stichprobe

Der Wilcoxon-Vorzeichen-Rang-Test für eine Stichprobe prüft, ob die in einer Stichprobe ermittelten Werte einer Variable mit symmetrischer Verteilung mit der Nullhypothese verträglich sind, dass der Median der Verteilung einen bestimmten Wert besitzt. Anders als beim t -Test für eine Stichprobe (vgl. Abschn. 8.2.1) wird nicht vorausgesetzt, dass die Variable normalverteilt ist. Im Vergleich zum Vorzeichen-Test für den Median wird neben der Anzahl der Werte größer dem Median unter der Nullhypothese auch das Ausmaß ihrer Differenz zum Median berücksichtigt.

```
> wilcox.test(x=(Vektor), alternative=c("two.sided", "less", "greater"),
+               mu=0)
```

Unter `x` ist der Datenvektor einzutragen. Mit `alternative` wird festgelegt, ob die Alternativhypothese bzgl. des Vergleichs mit dem unter der Nullhypothese angenommenen Median `mu` gerichtet oder ungerichtet ist. `"less"` und `"greater"` beziehen sich dabei auf die Reihenfolge Median unter der Alternativhypothese `"less"` bzw. `"greater"` Median unter der Nullhypothese.

²² Dabei ist zu beachten, dass die Funktion die Wahrscheinlichkeit dafür berechnet, dass die zugehörige Variable Werte $\leq q$ annimmt – die Grenze q also mit eingeschlossen ist. Für die Berechnung der Wahrscheinlichkeit, dass die Variable Werte $\geq q$ annimmt (rechtsseitiger Test), ist als Argument für `1-pbinom()` deshalb $q - 1$ zu übergeben, andernfalls würde nur die Wahrscheinlichkeit für Werte $> q$ bestimmt.

Als Beispiel diene jenes aus Büning und Trenkler (1994, p. 90 ff.): an Studenten einer Fachrichtung sei der IQ-Wert erhoben worden. Diese Werte sollen daraufhin geprüft werden, ob sie mit der Nullhypothese verträglich sind, dass der theoretische Median 110 beträgt. Die Alternativhypothese sei, dass der Median größer ist.

```
> IQ      <- c(99, 131, 118, 112, 128, 136, 120, 107, 134, 122)
> medH0 <- 110                                # Median unter H0
> wilcox.test(IQ, alternative="greater", mu=medH0)
Wilcoxon signed rank test
data: IQ
V = 48, p-value = 0.01855
alternative hypothesis: true location is greater than 110
```

Die Teststatistik liefert die Ausgabe unter V , den empirischen p -Wert unter $p\text{-value}$. Das Ergebnis lässt sich auch manuell nachvollziehen. Die diskret verteilte Teststatistik berechnet sich dabei als Summe der Ränge der absoluten Differenzen zum Median unter H_0 , wobei nur die Ränge von Werten aufsummiert werden, die größer als dieser Median sind.²³ Um den p -Wert der Teststatistik zu erhalten, dient `psignrank(q, n)` als Verteilungsfunktion der Teststatistik.²⁴

```
> (diffIQ <- IQ-medH0)                      # Differenzen zum Median unter H0
[1] -11 21 8 2 18 26 10 -3 24 12

> (idx <- diffIQ > 0)                      # Wert > Median unter H0?
[1] FALSE TRUE TRUE TRUE TRUE TRUE FALSE TRUE

# Ränge der absoluten Differenzen zum Median unter H0
> (rankDiff <- rank(abs(diffIQ)))
[1] 5 8 3 1 7 10 4 2 9 6

# Teststatistik: Summe über die Ränge von Werten > Median unter H0
> (V <- sum(rankDiff[idx]))
[1] 48

> (pVal <- 1-psignrank(V-1, n=length(IQ)))    # p-Wert einseitig
[1] 0.01855469
```

Für kleine Stichprobengrößen N kann die exakte Verteilung von V in Form der Punktwahrscheinlichkeiten auch durch Auszählen aller möglichen Fälle berechnet werden, die zu einem bestimmten V führen. Hierfür ist es zunächst notwendig, eine Matrix mit allen möglichen Ergebnissen davon zu erstellen, ob jeder einzelne der bereits nach ihrem Rang geordneten Werte über dem Median unter H_0 liegt.

²³ Das Beispiel wurde so gewählt, dass die Ränge eindeutig sind, also keine Bindungen auftreten, wodurch mehrere Wege zur Berechnung der Teststatistik denkbar wären.

²⁴ Dabei ist zu beachten, dass die Funktion die Wahrscheinlichkeit dafür berechnet, dass die Teststatistik Werte $\leq q$ annimmt – die Grenze q also mit eingeschlossen ist. Für die Berechnung der Wahrscheinlichkeit, dass die Variable Werte $\geq q$ annimmt (rechtsseitiger Test), ist als Argument für `1-psignrank()` deshalb $q - 1$ zu übergeben, andernfalls würde nur die Wahrscheinlichkeit für Werte $> q$ bestimmt.

Hierfür gibt es 2^N Möglichkeiten, die unter H_0 alle dieselbe Wahrscheinlichkeit $1/2^N$ besitzen, da die Wahrscheinlichkeit für jeden einzelnen Wert $1/2$ ist und von unabhängigen Realisierungen der Werte ausgegangen wird.

```
> NN    <- 3                                # Stichprobengröße
> l1W   <- lapply(numeric(NN), function(x) { c(0, 1) } )
> (mat <- as.matrix(expand.grid(l1W)))
  Var1 Var2 Var3
[1,]   0   0   0
[2,]   1   0   0
[3,]   0   1   0
[4,]   1   1   0
[5,]   0   0   1
[6,]   1   0   1
[7,]   0   1   1
[8,]   1   1   1
```

Im nächsten Schritt lässt sich die Matrix mit allen 2^N möglichen zu zählenden Rängen bestimmen und für jeden Fall die Rangsumme berechnen. Die Wahrscheinlichkeit einer bestimmten Rangsumme ergibt sich dann als Division der Anzahl der Ergebnisse, die sie hervorbringt, mit der Anzahl aller möglichen Ergebnisse.

```
> rMat <- col(mat)      # Matrix aus den Spaltennummern (=Rängen) von mat
> (vMat <- rMat * mat) # Matrix mit allen Fällen zu zählender Ränge
  Var1 Var2 Var3
[1,]   0   0   0
[2,]   1   0   0
[3,]   0   2   0
[4,]   1   2   0
[5,]   0   0   3
[6,]   1   0   3
[7,]   0   2   3
[8,]   1   2   3

> (rS   <- rowSums(vMat))      # alle auftretenden Rangsummen
> (dTab <- table(rS))        # Anzahl der Fälle jeder Rangsumme
0 1 2 3 4 5 6
1 1 1 2 1 1 1

> (pTab <- dTab / (2^NN))      # Wahrscheinlichkeit jeder Rangsumme
0     1     2     3     4     5     6
0.125 0.125 0.125 0.250 0.125 0.125 0.125

> sum(pTab)      # Kontrolle: Summe der Wahrscheinlichkeiten muss 1 sein
[1] 1

> dsigrank(0:sum(1:NN), NN)    # Kontrolle: Punktwahrscheinlichkeiten
[1] 0.125 0.125 0.125 0.250 0.125 0.125 0.125
```

6.4.3 Wald-Wolfowitz-Test für zwei Stichproben

Mit dem Test auf Zufälligkeit (vgl. Abschn. 6.1.2) lässt sich auch testen, ob zwei Stichproben ähnlicher Größe aus derselben Grundgesamtheit stammen bzw. ob die zu ihnen gehörenden Variablen dieselbe Verteilung besitzen. Dazu werden die Daten beider Stichproben gemeinsam ihrer Größe nach geordnet. Anschließend ist jeder Wert durch die Angabe zu ersetzen, aus welcher Stichprobe er stammt und der Test wie jener auf Zufälligkeit durchzuführen. Dieses Verfahren wird auch als Wald-Wolfowitz-Test bezeichnet.

```

> DV1 <- c(87, 55, 118, 106, 113)                      # Daten Gruppe 1
> DV2 <- c(91, 103, 102, 102, 130)                    # Daten Gruppe 2
> DV  <- c(DV1, DV2)                                    # kombinierte Daten

# Vektor der Gruppenzugehörigkeiten
> IV      <- rep(c("A", "B"), c(length(DV1), length(DV2)))
> (DVtst <- IV[order(DV)])                            # zu testende Daten
[1] "A" "A" "B" "B" "B" "A" "A" "A" "B"

> Ns      <- table(DVtst)                             # Gruppengrößen
> (runs <- rle(DVtst))                                # Iterationen
Run Length Encoding
lengths: int [1:4] 2 4 3 1
values: chr [1:4] "A" "B" "A" "B"

> (rr <- length(runs$lengths))                         # Gesamtzahl Iterationen
[1] 4

> nn1 <- Ns[1]                                         # Größe Gruppe 1
> nn2 <- Ns[2]                                         # Größe Gruppe 2
> N    <- sum(Ns)                                       # Gesamt-N

# Test über Approximation durch Normalverteilung
> muR   <- 1 + ((2*nn1*nn2) / N)
> varR  <- (2*nn1*nn2*(2*nn1*nn2 - N)) / (N^2 * (N-1))
> rZ    <- (rr-muR) / sqrt(varR)
> (pVal <- 1-pnorm(rZ))                               # p-Wert
0.9101438

```

6.4.4 Kolmogorov-Smirnov-Test für zwei Stichproben

Der Kolmogorov-Smirnov-Test (vgl. Abschn. 6.1.3) lässt sich auch verwenden, um die Daten einer stetigen Variable aus zwei unabhängigen Stichproben daraufhin zu prüfen, ob sie mit der Nullhypothese verträglich sind, dass die Variable in beiden Bedingungen dieselbe Verteilung besitzt. Dazu wird `ks.test()` so aufgerufen, dass sowohl für `x` als auch für `y` ein Datenvektor angegeben ist. Über das Argument `alternative` sind sowohl ungerichtete wie gerichtete Alternativhypthesen

prüfbare, wobei sich letztere darauf beziehen, ob y stochastisch kleiner ("less") oder größer ("greater") als x ist (vgl. Fußnote 9).

```

> DV1 <- round(rnorm(8, mean=1, sd=2), 2)           # Daten Stichprobe 1
> DV2 <- round(rnorm(8, mean=3, sd=2), 2)           # Daten Stichprobe 2
> ks.test(DV1, DV2, alternative="greater")
Two-sample Kolmogorov-Smirnov test
data: DV1 and DV2
D^+ = 0.5, p-value = 0.1353
alternative hypothesis: the CDF of x lies above that of y

```

Die Teststatistiken werden wie im Anpassungstest gebildet, wobei alle möglichen absoluten Differenzen zwischen den kumulierten relativen Häufigkeiten der Daten beider Stichproben in die Teststatistiken einfließen (Abb. 6.2).

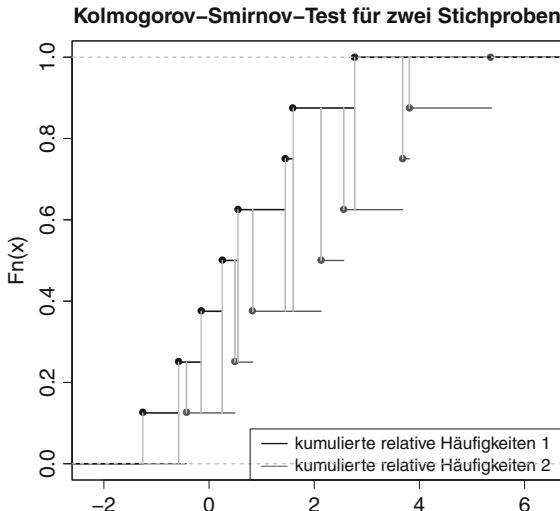


Abb. 6.2 Kolmogorov-Smirnov-Test für zwei Stichproben: Abweichungen zwischen kumulierten relativen Häufigkeiten der Daten aus beiden Bedingungen

```

> sortDV1    <- sort(DV1)                      # sortierte Daten Stichprobe 1
> sortDV2    <- sort(DV2)                      # sortierte Daten Stichprobe 2
> sortBoth   <- sort(c(DV1, DV2))            # kombinierte sortierte Daten
> both1      <- ecdf(DV1)(sortBoth)           # kumulierte rel. Häufigkeiten 1
> both2      <- ecdf(DV2)(sortBoth)           # kumulierte rel. Häufigkeiten 2
> diff1      <- both1-both2                   # Differenzen 1
> diff2      <- c(0, both1[1:(length(both1)-1)]) - both2  # Differenzen 2
> diffBoth   <- c(diff1, diff2)                # kombinierte Differenzen
> (DtwoS     <- max(abs(diffBoth)))          # Teststatistik zweiseitig
[1] 0.5

> (Dless     <- abs(min(diffBoth)))          # Teststatistik "less"
[1] 0

> (Dgreat    <- abs(max(diffBoth)))          # Teststatistik "greater"
[1] 0.5

```

```
# Kontrolle über Ausgabe von ks.test()
> ks.test(DV1, DV2, alternative="less")$statistic
D^-
 0

> ks.test(DV1, DV2, alternative="two.sided")$statistic
D
0.5

# graphische Darstellung
> xRange <- c(min(sortBoth)-1, max(sortBoth)+1) # Wertebereich x-Achse

# zeichne kumulierte relative Häufigkeiten 1 ein
> plot(ecdf(DV1), xlim=xRange, main="Kolmogorov-Smirnov-Test für zwei
+      Stichproben", xlab=NA, col.points="blue", col.hor="blue", lwd=2)

> par(new=TRUE)          # füge kumulierte relative Häufigkeiten 2 hinzu
> plot(ecdf(DV2), xlim=xRange, main=NA, xaxt="n", xlab=NA, ylab=NA,
+       lwd=2, col.points="red", col.hor="red")

# zeichne alle möglichen Abweichungen ein
> X   <- rbind(sortBoth, sortBoth)
> Y1  <- rbind(both1, both2)
> Y2  <- rbind(c(0, both1[1:(length(both1)-1)]), both2)
> matlines(X, Y1, col="darkgray", lty=1, lwd=2)
> matlines(X, Y2, col="darkgray", lty=1, lwd=2)
> legend(x="bottomright", legend=c("kumulierte relative Häufigkeiten 1",
+      "kumulierte relative Häufigkeiten 2"), col=c("blue", "red"), lwd=2)
```

6.4.5 Wilcoxon-Rangsummen-Test/Mann-Whitney-U-Test für zwei unabhängige Stichproben

Im Wilcoxon-Rangsummen-Test für unabhängige Stichproben werden die in zwei Stichproben ermittelten Werte einer Variable daraufhin miteinander verglichen, ob sie mit der Nullhypothese verträglich sind, dass die Verteilungen der Variable in den zugehörigen Bedingungen identisch sind. Anders als im analogen *t*-Test (vgl. Abschn. 8.2.2) wird nicht vorausgesetzt, dass die Variable in den Gruppen normalverteilt ist. Gefordert wird jedoch, dass die Verteilungen in ihrer Form in beiden Bedingungen übereinstimmen, d. h. lediglich ggf. horizontal verschobene Versionen voneinander darstellen.²⁵ Es kann sowohl gegen eine ungerichtete wie gerichtete Alternativhypothese getestet werden, die sich auf den Lageparameter der Verteilungen (etwa den Median) bezieht.

```
> wilcox.test(x=<Vektor>, y=<Vektor>, paired=FALSE,
+               alternative=c("two.sided", "less", "greater"))
```

²⁵ Für den Wald-Wolfowitz-Test, ob zwei Stichproben aus derselben Grundgesamtheit stammen, vgl. Abschn. 6.1.2.

Unter x sind die Daten der ersten Stichprobe einzutragen, unter y entsprechend die der zweiten. Alternativ kann statt x und y auch das `formula` Argument verwendet und eine Formel $\langle AV \rangle \sim \langle UV \rangle$ angegeben werden, wobei $\langle UV \rangle$ ein Gruppierungsfaktor derselben Länge wie $\langle AV \rangle$ ist und für jede Beobachtung in $\langle AV \rangle$ die zugehörige UV-Stufe angibt. Geschieht dies mit Variablennamen, die nur innerhalb eines Datensatzes bekannt sind, muss dieser unter `data` eingetragen werden. Mit `alternative` wird festgelegt, ob die Alternativhypothese gerichtet oder ungerichtet ist. "less" und "greater" beziehen sich dabei auf den Lageparameter in der Reihenfolge x "less" bzw. "greater" y . Mit dem Argument `paired` wird angegeben, ob es sich um unabhängige (FALSE) oder abhängige (TRUE) Stichproben handelt.

Als Beispiel diene jenes aus Bortz et al. (2008, p. 201 ff.): Es wird vermutet, dass sich durch die Einnahme eines Medikaments die Reaktionszeit im Vergleich zu einer Kontrollgruppe verkürzen lässt.

```
> rtCntrl <- c(85,106,118,81,138,90,112,119,107,95,88,103)
> rtDrug <- c(96,105,104,108,86,84,99,101,78,124,121,97,129,87,109)
> wilcox.test(rtCntrl, rtDrug, alternative="greater")
Wilcoxon rank sum test
data: rtCntrl and rtDrug
W = 94, p-value = 0.4333
alternative hypothesis: true location shift is greater than 0
```

Die Ausgabe nennt den Wert der Teststatistik (W) gefolgt vom empirischen p -Wert ($p\text{-value}$). Das Ergebnis lässt sich auch manuell nachvollziehen. Dazu muss für die Reaktionszeiten aus der ersten Stichprobe die Summe ihrer Ränge in der Gesamtstichprobe ermittelt werden. Die diskret verteilte Teststatistik ergibt sich, indem von dieser Rangsumme ihr theoretisches Minimum $\sum_{i=1}^{n_1} i$ abgezogen wird.²⁶ Um den p -Wert zu erhalten, dient `pwilcox(q, m, n)` als Verteilungsfunktion der Teststatistik.²⁷

```
> n1 <- length(rtCntrl)                                # Größe Gruppe 1
> n2 <- length(rtDrug)                                 # Größe Gruppe 2

# Faktor der Gruppenzugehörigkeiten
> IV          <- factor(rep(1:2, c(n1, n2)), labels=c("control", "drug"))
> rtAll        <- c(rtCntrl, rtDrug)                  # kombinierte Daten
> rankAll     <- rank(rtAll)                          # Ränge Gesamtstichprobe
> (W <- sum(rankAll[IV == "control"]) - sum(1:n1))   # Teststatistik
[1] 94
> (pVal <- 1-pwilcox(W-1, m=n1, n=n2))            # p-Wert einseitig
[1] 0.433342
```

²⁶ Das Beispiel wurde so gewählt, dass die Ränge eindeutig sind, also keine Bindungen auftreten, wodurch mehrere Wege zur Berechnung der Teststatistik denkbar wären.

²⁷ Dabei ist zu beachten, dass die Funktion die Wahrscheinlichkeit dafür berechnet, dass die Teststatistik Werte $\leq q$ annimmt – die Grenze q also mit eingeschlossen ist. Für die Berechnung der Wahrscheinlichkeit, dass die Variable Werte $\geq q$ annimmt (rechtsseitiger Test), ist als Argument für `1-pwilcox()` deshalb $q - 1$ zu übergeben, andernfalls würde nur die Wahrscheinlichkeit für Werte $> q$ bestimmt.

Der Mann-Whitney-*U*-Test zählt für jeden Wert der ersten Stichprobe, wie viele Werte der zweiten Stichprobe kleiner als er sind. Die Teststatistik *U* ist die Summe dieser Häufigkeiten und führt zum selben Wert wie die oben definierte Teststatistik des Wilcoxon-Tests, besitzt also auch dieselbe Verteilung.

```
> (U <- sum(outer(rtCntrl, rtDrug, ">=")))
[1] 94
```

6.4.6 Wilcoxon-Test für zwei abhängige Stichproben

Der Wilcoxon-Test für abhängige Stichproben wird wie jener für unabhängige Stichproben durchgeführt, jedoch ist hier das Argument *paired*=TRUE von *wilcox.test()* zu verwenden. Der Test setzt voraus, dass sich die in *x* und *y* angegebenen Daten einander paarweise zuordnen lassen, weshalb *x* und *y* dieselbe Länge besitzen müssen. Nach Bildung der paarweisen Differenzen einander zugeordneter Werte wird im Wilcoxon-Vorzeichen-Rang-Test für eine Stichprobe die Nullhypothese getestet, dass der theoretische Median der Differenzwerte gleich 0 ist.

6.4.7 Kruskal-Wallis-*H*-Test für unabhängige Stichproben

Der Kruskal-Wallis-*H*-Test verallgemeinert die Fragestellung eines Wilcoxon-Tests für zwei unabhängige Stichproben auf Situationen, in denen Werte einer Variable in mehr als zwei unabhängigen Stichproben ermittelt wurden. Unter der Nullhypothese sind die Verteilungen der Variable in den zugehörigen Bedingungen identisch. Die Alternativhypothese ist unspezifisch und besagt, dass sich mindestens zwei Lageparameter unterscheiden. Anders als in der einfaktoriellen Varianzanalyse (vgl. Abschn. 8.3) wird nicht vorausgesetzt, dass die Variable in den Bedingungen normalverteilt ist. Gefordert wird jedoch, dass die Form der Verteilungen in allen Bedingungen übereinstimmt, die Verteilungen also lediglich ggf. horizontal verschobene Versionen voneinander darstellen.

```
> kruskal.test(formula=(Formel), data=(Datensatz), subset=(Indexvektor))
```

Unter *formula* sind Daten und Gruppierungsvariable als Formel $\langle AV \rangle \sim \langle UV \rangle$ zu nennen, wobei $\langle UV \rangle$ ein Gruppierungsfaktor derselben Länge wie $\langle AV \rangle$ ist und für jede Beobachtung in $\langle AV \rangle$ die zugehörige UV-Stufe angibt. Geschieht dies mit Variablennamen, die nur innerhalb eines Datensatzes bekannt sind, muss dieser unter *data* eingetragen werden. Das Argument *subset* erlaubt es, nur eine Teilmenge der Beobachtungen einfließen zu lassen – es erwartet einen entsprechenden Indexvektor, der sich auf die Zeilen des Datensatzes bezieht.

Als Beispiel diene jenes aus Büning und Trenkler (1994, p. 183 ff.): Es wird vermutet, dass der IQ-Wert in vier Studiengängen einen unterschiedlichen Erwartungswert besitzt.

```
# IQ-Werte in den einzelnen Studiengängen
> IQ1    <- c(99, 131, 118, 112, 128, 136, 120, 107, 134, 122)
> IQ2    <- c(134, 103, 127, 121, 139, 114, 121, 132)
> IQ3    <- c(120, 133, 110, 141, 118, 124, 111, 138, 120)
> IQ4    <- c(117, 125, 140, 109, 128, 137, 110, 138, 127, 141, 119, 148)
> IQall <- c(IQ1, IQ2, IQ3, IQ4)                                # kombinierte Daten

# Stichprobengrößen
> Ni <- c(length(IQ1), length(IQ2), length(IQ3), length(IQ4))
> N  <- sum(Ni)                                              # Gesamt-N

# Faktor der Gruppenzugehörigkeiten
> IV   <- factor(rep(1:4, Ni), labels=c("I", "II", "III", "IV"))
> KWdf <- data.frame(id=1:N, IV, DV=IQall)                  # Datensatz
> kruskal.test(DV ~ IV, data=KWdf)
Kruskal-Wallis rank sum test
data: DV by IV
Kruskal-Wallis chi-squared = 1.7574, df = 3, p-value = 0.6242
```

Die Ausgabe nennt den Wert der asymptotisch χ^2 -verteilten H -Teststatistik gefolgt von den Freiheitsgraden (df) und dem empirischen p -Wert (p-value). Das Ergebnis lässt sich auch manuell nachvollziehen, wobei das zentrale Element der Teststatistik das Quadrat der pro Gruppe gebildeten Summe der Ränge in der Gesamtstichprobe ist, das jeweils an der zugehörigen Gruppengröße relativiert wird.²⁸

```
> rankAll    <- rank(IQall)                                     # Ränge Gesamtstichprobe
> (rankSumI <- tapply(rankAll, IV, sum))                      # Rangsumme pro Gruppe
     I      II      III      IV
168.5 160.0 173.0 278.5

> (H <- (12 / (N*(N+1))) * sum(rankSumI^2/Ni) - 3*(N+1))  # Teststatistik
[1] 1.75531

> (pVal <- 1-pchisq(H, nlevels(IV)-1))                         # p-Wert
[1] 0.624708
```

6.4.8 Friedman-Rangsummen-Test für abhängige Stichproben

Der Rangsummentest nach Friedman dient der Analyse von Daten einer kategorialen Variable, die in p abhängigen Stichproben erhoben wurde. Jede Menge von p abhängigen Beobachtungen (eine aus jeder Bedingung) wird dabei als Block bezeichnet und stammt entweder vom selben Beobachtungsobjekt (Messwiederholung) oder von mehreren homogenen, also gematchten Beobachtungsobjekten. Wie im Kruskal-Wallis- H -Test wird die Nullhypothese geprüft, dass die Vertei-

²⁸ Im gewählten Beispiel sind die Ränge nicht eindeutig, es treten also Bindungen auf. Für diesen Fall gibt `rank()` in der Voreinstellung mittlere Ränge aus, was vom Vorgehen in `kruskal.test()` abweicht. Die Teststatistiken stimmen deshalb nicht exakt überein.

lung der Variable in allen Bedingungen identisch ist. Die Alternativhypothese ist unspezifisch und besagt, dass sich mindestens zwei Lageparameter unterscheiden. Anders als in der einfaktoriellen Varianzanalyse für abhängige Gruppen (vgl. Abschn. 8.4) wird nicht vorausgesetzt, dass die blockweise als Vektor zusammengefasste Variable gemeinsam normalverteilt ist.

```
> friedman.test(y=Daten, groups=Faktor, blocks=Faktor)
```

Unter `y` ist der Vektor aller Daten anzugeben. Als zweites Argument wird für `groups` ein Faktor derselben Länge wie `y` übergeben, der die Gruppenzugehörigkeit jeder Beobachtung in `y` codiert. `blocks` ist ebenfalls ein Faktor derselben Länge wie `y` und gibt für jede Beobachtung in `y` an, zu welchem Block sie gehört, z. B. von welcher VP sie stammt, wenn Messwiederholung vorliegt.

Als Beispiel diene jenes aus Bortz et al. (2008, p. 269 ff.): An Ratten wird die Auswirkung von zentral erregenden Präparaten erhoben, wobei die Anzahl der pro Zeiteinheit gemessenen Umdrehungen in einem Laufrad die Abhängige Variable ist. Aus jeweils vier hinsichtlich verschiedener Störvariablen homogenen Ratten werden fünf Blöcke gebildet und jeder Block in allen vier Bedingungen beobachtet.

```
# Umdrehungen in den einzelnen Bedingungen
> rpmCaff <- c(14, 13, 12, 11, 10)          # Koffein
> rpmDS  <- c(11, 12, 13, 14, 15)          # Medikament einfache Dosis
> rpmDD  <- c(16, 15, 14, 13, 12)          # Medikament doppelte Dosis
> rpmPlac <- c(13, 12, 11, 10, 9)          # Placebo
> rpmAll  <- c(rpmCaff, rpmDS, rpmDD, rpmPlac)    # kombinierte Daten
> nBl    <- length(rpmCaff)                  # Anzahl Blöcke
> P      <- 4                                # Anzahl Bedingungen

# Faktor Gruppenzugehörigkeit
> IV <- factor(rep(1:P, each=nBl),
+               labels=c("Caffeine", "Single", "Double", "Placebo"))

> blocks <- factor(rep(1:nBl, P))          # Faktor Blockzugehörigkeit
> friedman.test(rpmAll, IV, blocks)
Friedman rank sum test
data: rpmAll, IV and blocks
Friedman chi-squared = 8.2653, df = 3, p-value = 0.04084
```

Die Ausgabe nennt den Wert der asymptotisch χ^2 -verteilten Teststatistik gefolgt von den Freiheitsgraden (`df`) und dem empirischen *p*-Wert (`p-value`). Das Ergebnis lässt sich auch manuell nachvollziehen, wobei das zentrale Element der Teststatistik das Quadrat der pro Gruppe gebildeten Summe der Ränge innerhalb jedes Blocks ist.²⁹

```
> (rpmMat <- cbind(rpmCaff, rpmDS, rpmDD, rpmPlac))  # Datenmatrix
  rpmCaff  rpmDS  rpmDD  rpmPlac
```

²⁹ Im gewählten Beispiel sind die Ränge im zweiten Block nicht eindeutig, es treten also Bindungen auf. Deshalb wird die Teststatistik *S* in `friedman.test()` weiter korrigiert und stimmt nicht exakt mit der hier berechneten überein.

```
[1,]    14    11    16    13
[2,]    13    12    15    12
[3,]    12    13    14    11
[4,]    11    14    13    10
[5,]    10    15    12     9

> (rankMat <- t(apply(rpmMat, 1, rank)))    # Matrix blockweise Ränge
      rpmCaff rpmDS rpmDD rpmPlac
[1,]    3   1.0    4   2.0
[2,]    3   1.5    4   1.5
[3,]    2   3.0    4   1.0
[4,]    2   4.0    3   1.0
[5,]    2   4.0    3   1.0

> (rankSumJ <- apply(rankMat, 1, sum))      # gruppenweise Rangsummen
rpmCaff rpmDS rpmDD rpmPlac
12.0 13.5 18.0   6.5
# Teststatistik
> (S <- (12 / (nBl*p*(P+1))) * sum(rankSumJ^2) - 3*nBl*(P+1))
[1] 8.1

> (pVal <- 1-pchisq(S, P-1))                # p-Wert
[1] 0.04398959
```

6.4.9 Cochran-*Q*-Test für abhängige Stichproben

Der *Q*-Test von Cochran lässt sich als Spezialfall des Friedman-Tests für den Fall auffassen, dass die Abhängige Variable dichotom ist. Auch hier liegen abhängige Daten aus p Bedingungen vor: dabei liefert entweder jedes Beobachtungsobjekt Werte aus jeder Bedingung (Messwiederholung), oder aber p homogene (gematchte) Beobachtungsobjekte werden zu einem Block zusammengefasst, der dann p abhängige Werte aus den Bedingungen liefert. Der *Q*-Test wird mit der *symmetry_test()* Funktion aus dem Paket *coin* durchgeführt.

```
> symmetry_test(formula=<Formel>, data=<Datensatz>, subset=NULL,
+                 teststat="quad")
```

Unter *formula* ist eine Formel der Form $\langle AV \rangle \sim \langle UV \rangle | \langle BlockVar \rangle$ zu nennen, wobei $\langle UV \rangle$ ein Faktor derselben Länge wie $\langle AV \rangle$ ist und für jede Beobachtung in $\langle AV \rangle$ die zugehörige UV-Stufe angibt. $\langle BlockVar \rangle$ ist ebenfalls ein Faktor derselben Länge wie $\langle AV \rangle$ und enthält die Blockzugehörigkeit jedes Wertes. Sind die Variablennamen nur innerhalb eines Datensatzes bekannt, muss dieser unter *data* eingetragen werden. Das Argument *subset* erlaubt es, nur eine Teilmenge der Beobachtungen einfließen zu lassen, es erwartet einen entsprechenden Indexvektor, der sich auf die Zeilen des Datensatzes bezieht. Für den *Q*-Test ist zudem das Argument *teststat="quad"* zu setzen, da es *symmetry_test()* erlaubt, verschiedene Testverfahren für dieselbe Hypothese anzuwenden.

Als Beispiel diene jenes aus Büning und Trenkler (1994, p. 208 ff.): Über fünf Jahre hinweg geben dieselben zehn Wahlberechtigten als dichotomes Präferenzurteil an, ob sie eine bestimmte Partei den anderen vorziehen.

```
# Präferenzurteile der 10 Blöcke in den 5 Jahren
> pref <- c(1,1,0,1,0, 0,1,0,0,1, 1,0,1,0,0, 1,1,1,1,1, 0,1,0,0,0,
+           1,0,1,1,1, 0,0,0,0,0, 1,1,1,1,0, 0,1,0,1,1, 1,0,1,0,0)

> nSubj <- 10                                # Anzahl Blöcke / VPn
> year <- factor(rep(1981:1985, nSubj))    # Faktor Messzeitpunkt
> P     <- nlevels(year)                      # Anzahl Messzeitpunkte
> id   <- factor(rep(1:nSubj, each=P))       # Faktor Blockzugehörigkeit
> library(coin)                             # Auswertung mit Paket coin
> symmetry_test(pref ~ year | id, teststat="quad")
Asymptotic General Independence Test
data: pref by year (1981, 1982, 1983, 1984, 1985)
stratified by id
chi-squared = 1.3333, df = 4, p-value = 0.8557
```

Die Ausgabe nennt den Wert der asymptotisch χ^2 -verteilten Teststatistik gefolgt von den Freiheitsgraden (df) und dem empirischen p -Wert (p-value). Das Ergebnis lässt sich auch manuell nachvollziehen, wobei die Messwerte zunächst als Datenmatrix im Wide-Format zusammenfassen sind. Das zentrale Element der Teststatistik sind die Abweichungen der Zeilen- und Spaltensummen in dieser Matrix von ihrem jeweiligen Mittel.

```
# Datenmatrix im Wide-Format
> prefMat <- matrix(pref, nrow=nSubj, ncol=P, byrow=TRUE)
> rSum      <- apply(prefMat, 1, sum)          # Zeilensummen
> cSum      <- apply(prefMat, 2, sum)          # Spaltensummen

# Teststatistik Q
> (Q <- (P*(P-1)*sum((cSum-mean(cSum))^2)) / (P*sum(rSum) - sum(rSum^2)))
[1] 1.333333

> (pVal <- 1-pchisq(Q, P-1))                 # p-Wert
[1] 0.8556952
```

6.4.10 Bowker-Test für zwei abhängige Stichproben

Ein weiterer Spezialfall des Friedman-Tests ist der Bowker-Test für Situationen, in denen eine in nur zwei abhängigen Stichproben erhobene kategoriale Variable mehrere Ausprägungen besitzt. Wieder besteht die Nullhypothese darin, dass die Verteilung der Variable in beiden Bedingungen identisch ist und damit hier eine bzgl. der Hauptdiagonale symmetrische Kontingenztafel der Übereinstimmungen beider Stichproben vorliegt. Die ungerichtete Alternativhypothese besagt, dass es eine systematische Abweichung von Übereinstimmung in eine Richtung gibt. Der Basisumfang von R stellt für den Test keine eigene Funktion bereit, so dass entweder die Funktion `summary(age.comp(), what="symmetry")` aus dem FSA Paket

(Ogle und Spangler, 2009) zu verwenden ist, oder aber eine manuelle Rechnung durchgeführt werden muss.

```
> summary(age.comp((Vektor1), (Vektor2)), what="symmetry")
```

Für `(Vektor1)` sind die Daten aus der ersten Bedingung zu nennen, wobei es sich um einen Vektor mit numerisch codierten Kategorien handeln muss. Unter `(Vektor2)` folgt ein Vektor derselben Länge wie `(Vektor1)`, der die in der zweiten Bedingung erhobenen Daten derselben Variable speichert. Schließlich ist das Argument `what="symmetry"` von `summary()` zu setzen.

Als Beispiel diene jenes aus Bortz et al. (2008, p. 165 ff.): An denselben Personen soll die empfundene Leistungssteigerung eines Medikaments und eines Placebos untersucht werden. Erhoben wird die Einschätzung, ob keine (1), eine geringe (2) oder starke (3) Wirkung vorliegt.

```
> categ <- c(1:3)                                     # AV-Kategorien
> drug  <- rep(categ, c(30, 50, 20))               # Daten Medikament
# Daten Placebo
> plac  <- rep(rep(categ, length(categ)), c(14,7,9, 5,26,19, 1,7,12))    # Auswertung mit FSA Paket
> library(FSA)                                         # Auswertung mit FSA Paket
> summary(age.comp(drug, plac), what="symmetry")
Raw agreement table (square)
  First
Second 1   2   3
  1 14   5   1
  2  7  26   7
  3  9  19  12

Bowker's (Hoenig's) Test of Symmetry
  df   chi.sq      p
1  3 12.27179 0.006507799
```

Die Ausgabe umfasst neben der Kontingenztafel der Übereinstimmungen beider Stichproben die Freiheitsgrade (df) der mit wachsender Stichprobengröße asymptotisch χ^2 -verteilten Teststatistik (chi.sq) und den zugehörigen *p*-Wert (p). Das Ergebnis lässt sich manuell prüfen. Dabei ergibt sich die Teststatistik als Summe der quadrierten Differenzen von an der Hauptdiagonale gespiegelten Einträgen der Kontingenztafel, die zuvor an der Summe beider Einträge relativiert wurden.

```
# berechne zunächst alle relativierten quadrierten Differenzen zwischen
# der Kontingenztafel und ihrer Transponierten
> cTabBow <- table(drug, plac)                      # Kontingenztafel
> sqDiffs  <- (cTabBow-t(cTabBow))^2 / (cTabBow+t(cTabBow))
> P        <- ncol(cTabBow)                           # Anzahl Kategorien

# summiere nur über die Differenzen der oberen Dreiecksmatrix
> (chisqVal <- sum(sqDiffs[upper.tri(cTabBow)]))  # Teststatistik
[1] 12.27179

> (BowDf <- choose(P, 2))                            # Freiheitsgrade
[1] 3
```

```
> P*(P-1)/2                                # einfache Formel
[1] 3

> (pVal <- 1-pchisq(chisqVal, BowDf))      # p-Wert
[1] 0.006507799
```

6.4.11 McNemar-Test für zwei abhängige Stichproben

Ein Spezialfall des Bowker-Tests ist der McNemar-Test für Daten einer dichotomen Variable aus zwei abhängigen Stichproben. Seine Nullhypothese, dass die Verteilung der Abhängigen Variable in beiden Bedingungen identisch ist, lässt sich auch so formulieren, dass die Kontingenztafel der Übereinstimmungen der Daten aus den abhängigen Stichproben bzgl. der Hauptdiagonale symmetrisch ist. Die ungerichtete Alternativhypothese besagt, dass es eine systematische Abweichung von Übereinstimmung in eine Richtung gibt.

```
> mcnemar.test(x, y=NULL, correct=TRUE)
```

Unter x kann die Kontingenztafel der beiden Datenvektoren einer dichotomen Variable aus zwei abhängigen Stichproben angegeben werden. Pro Beobachtungseinheit übereinstimmende Ausprägungen der Variable stehen in dieser Matrix in der Diagonale, voneinander abweichende Ausprägungen außerhalb der Diagonale. Wird für x stattdessen ein die Daten aus einer Stichprobe codierendes Objekt der Klasse `factor` genannt, muss auch y ein Faktor mit denselben Stufen und derselben Länge wie x sein, der die Daten der anderen Stichprobe speichert. Das Argument `correct` legt fest, ob eine Stetigkeitskorrektur durchgeführt wird.

Im Beispiel sei an einer Stichprobe jeweils vor und nach einer Informationskampagne die Variable erhoben worden, ob eine Person raucht.

```
> nSubj  <- 20                                # Anzahl Versuchspersonen
> smPre  <- rbinom(nSubj, size=1, p=0.6)       # Prä-Messung
> smPost <- rbinom(nSubj, size=1, p=0.4)       # Post-Messung

# Konvertierung der numerischen Vektoren in Faktoren
> smPreFac <- factor(smPre, levels=0:1, labels=c("no", "yes"))
> smPostFac <- factor(smPost, levels=0:1, labels=c("no", "yes"))
> cTab      <- table(smPreFac, smPostFac)      # Kontingenztafel
> addmargins(cTab)                            # Randsummen
smPostFac no yes Sum
  no    4   6  10
  yes   5   5  10
  Sum   9  11 20

> mcnemar.test(cTab, correct=FALSE)
McNemar's Chi-squared test
data: cTab
McNemar's chi-squared = 0.0909, df = 1, p-value = 0.763
```

Die Ausgabe nennt den Wert der asymptotisch χ^2 -verteilten Teststatistik gefolgt von den Freiheitsgraden (df) und dem empirischen p -Wert (p-value). Da die Symmetrie einer Kontingenztafel zu testen ist, eignet sich hier wie bei Cochran's Q -Test die `symmetry_test()` Funktion aus dem `coin` Paket zur Auswertung. Sie akzeptiert die Kontingenztafel der Übereinstimmungen beider Bedingungen als Argument.

```
> library(coin) # Auswertung mit coin Paket  
> symmetry_test(cTab, teststat="quad") #...
```

Verglichen mit dem Bowker-Test vereinfacht sich bei der manuellen Berechnung die Formel für die Teststatistik, da in der Kontingenztafel nun nur noch die Differenz der beiden Zellen außerhalb der Diagonale zu berücksichtigen ist.

```

> P <- ncol(cTab)                                     # Anzahl Kategorien

# Teststatistik
> (chisqVal <- (cTab[1, 2] - cTab[2, 1])^2 / (cTab[1, 2] + cTab[2, 1]))
[1] 0.0909091

> (pVal <- 1-pchisq(chisqVal, P*(P-1)/2))        # p-Wert
[1] 0.7630246

```

6.4.12 Stuart-Maxwell-Test für zwei abhängige Stichproben

Der Stuart-Maxwell-Test prüft die Kontingenztafel zweier kategorialer Variablen auf Homogenität der Randverteilungen. Der Test kann z. B. zur Beurteilung der Frage eingesetzt werden, ob zwei Rater die verfügbaren Kategorien mit denselben Grundwahrscheinlichkeiten verwenden (vgl. Abschn. 6.3.3). Verglichen mit dem Bowker-Test bezieht er sich auf nur einen Spezialfall, der zu einer asymmetrischen Kontingenztafel der Übereinstimmungen führt. Zur Berechnung des Tests steht aus dem `coin` Paket die `mh_test()` Funktion bereit.

```
> mh_test(<Kontingenztafel>)
```

Die Funktion akzeptiert mit `⟨Kontingenztafel⟩` die Kontingenztafel der Übereinstimmungen der kategorialen AV in beiden Bedingungen. Als Beispiel sei wie beim Bowker-Test jenes aus Bortz et al. (2008, p. 165 ff.) herangezogen.

```
> library(coin)
> mh_test(cTabBow)
Asymptotic Marginal-Homogeneity Test
data: response by groups (drug, plac) stratified by block
chi-squared = 12.1387, df = 2, p-value = 0.002313
```

Die manuelle Prüfung ist für den Fall von 3×3 Kontingenztafeln wie folgt möglich:

```
> addmargins(cTabBow) # Kontingenztafel mit Randhäufigkeiten
          plac
drug   1   2   3  Sum
  1   14   7   9  30
  2    5  26  19  50
```

```

3   1   7  12  20
Sum  20  40  40 100

```

```

# Hälfte der Summe der an der Hauptdiagonale gespiegelten Häufigkeiten
> (Nij <- ((cTabBow+t(cTabBow)) / 2)[upper.tri(cTabBow)])
[1] 6 5 13

```

```

# Abweichungen der Randverteilungen
> (d <- rowSums(cTabBow) - colSums(cTabBow))
 1   2   3
10 10 -20

> num   <- sum(Nij * rev(d^2))                      # Zähler
> denom <- 2 * sum(apply(combn(Nij, 2), 2, prod))    # Nenner
> (chisqVal <- num / denom)                          # Teststatistik
[1] 12.13873
> (smmhDf <- nrow(cTabBow)-1)                      # Freiheitsgrade
[1] 2

> (pVal <- 1-pchisq(chisqVal, smmhDf))            # p-Wert
[1] 0.002312643

```

Im allgemeinen Fall ist es zur Bestimmung der Teststatistik notwendig, die Kovarianzmatrix der Abweichungen beider Randverteilungen unter der Nullhypothese zu bestimmen und zu invertieren (vgl. Abschn. 2.9). Besitzt die Variable P Kategorien, reicht bereits die Information über die Abweichung in den Randverteilungen bzgl. der ersten $P - 1$ Kategorien aus, da sich die Randsummen der Kontingenztafel zur Anzahl der Objekte summieren.

```

# spätere Kovarianzmatrix der Abweichungen der Randverteilungen unter H0
> S <- -(cTabBow + t(cTabBow))

# Diagonale dieser Kovarianzmatrix
> diag(S) <- rowSums(cTabBow) + colSums(cTabBow) - 2*diag(cTabBow)

# berücksichtige nur Informationen aus den ersten P-1 Kategorien
> keep <- 1:(nrow(cTabBow)-1)

# Teststatistik
> (chisqVal <- t(d[keep]) %*% solve(S[keep, keep]) %*% d[keep])
[1] 12.13873

```


Kapitel 7

Korrelation und Regressionsanalyse

Die Korrelation zweier quantitativer Variablen ist ein Maß ihres linearen Zusammenhangs, ohne dass eine Kausalbeziehung zwischen ihnen impliziert würde. Auch die lineare Regression bezieht sich auf den linearen Zusammenhang von Variablen, um mit seiner Hilfe Variablenwerte einer Zielvariable (des Kriteriums) durch die Werte anderer Variablen (der Prädiktoren) vorherzusagen. Für beide Verfahren lassen sich auch inferenzstatistisch testbare Hypothesen über ihre Parameter aufstellen. Für die statistischen Grundlagen dieser Themen vgl. die darauf spezialisierte Literatur (Bortz, 2005; Hartung et al., 2005; Hays, 1994), die auch für eine vertiefte Behandlung von Regressionsanalysen in R verfügbar ist (Faraway, 2004; Fox, 2002).

7.1 Test auf Korrelation

Die empirische Korrelation zweier normalverteilter Variablen lässt sich daraufhin testen, ob sie mit der Nullhypothese verträglich ist, dass deren theoretische Korrelation 0 ist.¹ Bei gemeinsam normalverteilten Variablen ist dies gleichbedeutend mit der Frage, ob Unabhängigkeit vorliegt.

```
> cor.test(x=<Vektor1>, y=<Vektor2>, alternative=c("two.sided", "less",
+           "greater"), use)
```

Die Daten beider Variablen sind als Vektoren derselben Länge über die Argumente `x` und `y` anzugeben. Ob die Alternativhypothese zwei- (`"two.sided"`), links- (negativer Zusammenhang, `"less"`) oder rechtsseitig (positiver Zusammenhang, `"greater"`) ist, legt das Argument `alternative` fest. Über das Argument `use` können verschiedene Strategien zur Behandlung fehlender Werte ausgewählt werden (vgl. Abschn. 2.12.4).²

¹ Für den Test auf Zusammenhang von Rangdaten vgl. Abschn. 6.3.1.

² Die `rcorr()` Funktion aus dem `Hmisc` Paket berechnet für mehrere Variablen die Korrelationsmatrix nach Pearson sowie nach Spearman und testet die resultierenden Korrelationen gleichzeitig auf Signifikanz.

```

> nSubj <- 20
> vec1 <- sample(0:100, nSubj, replace=TRUE)
> vec2 <- vec1 + rnorm(nSubj, 0, 50)
> cor.test(vec1, vec2)
Pearson's product-moment correlation
data: vec1 and vec2
t = 4.2111, df = 18, p-value = 0.0005251
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
0.3805926 0.8744009
sample estimates:
cor
0.4108645

```

Die Ausgabe beinhaltet den empirischen t -Wert der Teststatistik (t) samt Freiheitsgraden (df) und zugehörigem p -Wert (p -value) sowie das Vertrauensintervall für die Korrelation, deren empirischer Wert ebenfalls aufgeführt ist. Das Ergebnis lässt sich manuell prüfen³:

```

> r      <- cor(vec1, vec2)                      # empirische Korrelation
> (tVal <- sqrt(nSubj-2) * r / sqrt(1-r^2))      # Teststatistik t-Wert
[1] 4.211139

# p-Wert: das Doppelte der Fläche (zweiseitiger Test) rechts von tVal
# unter Dichtefunktion unter H0 - Berechnung über Verteilungsfunktion
> (pVal <- 2 * (1-pt(tVal, nSubj-2)))
[1] 0.0005250652

# 95%-Vertrauensintervall für die wahre Korrelation rho
> fishZ <- 0.5 * log((1+r)/(1-r))            # Fisher z-Transformation
> fishV <- 1 / (nSubj-3)                      # Varianz der Transformierten
> zCrit <- qnorm(1-(0.05/2), 0, 1)           # kritischer z-Wert
> zLo    <- fishZ - zCrit*sqrt(fishV)         # VI für z untere Grenze
> zUp    <- fishZ + zCrit*sqrt(fishV)         # VI für z obere Grenze

# Rücktransformation der Intervallgrenzen
> (ciLo <- (exp(1)^(2*zLo)-1) / (exp(1)^(2*zLo)+1))  # oder: tanh(zLo)
[1] 0.3805926

> (ciUp <- (exp(1)^(2*zUp)-1) / (exp(1)^(2*zUp)+1))  # oder: tanh(zUp)
[1] 0.8744009

```

7.2 Einfache lineare Regression

Bei der einfachen linearen Regression werden anhand der paarweise vorhandenen Daten zweier quantitativer Variablen X und Y die Parameter a und b der Gleichung

³ Für Fishers z -Transformation vgl. die `fisherz()`, für die Rücktransformation die `fisherz2r()` Funktion des `psych` Pakets.

$Y = bX + a$ so bestimmt, dass die Werte von Y (dem Kriterium) bestmöglich anhand der Werte von X (dem Prädiktor) vorhergesagt werden. Als Maß für die Güte der Vorhersage wird dafür die Summe der quadrierten Residuen, also der Abweichungen von vorhergesagten und Kriteriumswerten herangezogen, weswegen a und b auch die Bezeichnung *Ordinary Least Squares* (OLS) Schätzer tragen.⁴

```
> lm(formula=Formel, data=Datensatz, subset=Indexvektor),
+   na.action=Behandlung fehlender Werte)
```

Unter `formula` ist eine Formel als symbolische Spezifikation des linearen Modells anzugeben, das der Regression zugrunde liegen soll (vgl. Abschn. 5.1). Soll das Modell in der Regression ohne den Parameter a gebildet werden, lautet die Formel $\langle\text{Kriterium}\rangle \sim \langle\text{Prädiktor}\rangle - 1$. Sind die angegebenen Variablen nur innerhalb eines Datensatzes bekannt, muss dieser unter `data` übergeben werden. Das Argument `subset` erlaubt es, nur eine Teilmenge der Beobachtungen in die Berechnung einfließen zu lassen, es erwartet einen entsprechenden Indexvektor, der sich auf die Zeilen des Datensatzes bezieht. Mit dem Argument `na.action` kann bestimmt werden, wie mit fehlenden Werten umzugehen ist (vgl. Abschn. 5.3).⁵

Als Beispiel soll das Körpergewicht `weight` als Kriterium mit der Körpergröße `height` als Prädiktor vorhergesagt werden. Die `weight` Daten werden hier entsprechend einem (sicher unrealistischen) linearen Modell aus der Körpergröße und einem zufälligen Fehler simuliert.

```
> nSubj <- 100
> height <- rnorm(nSubj, 175, 7)                      # Prädiktor
> weight <- 0.4*height + 10 + rnorm(nSubj, 0, 3)       # Kriterium
> (model <- lm(weight ~ height))                      # Regression
Call:
lm(formula = weight ~ height)
```

```
Coefficients:
(Intercept)  height
 27.3351     0.3022
```

Im Ergebnis von `lm()` zur Berechnung der Regressionsparameter wird unter der Überschrift `Coefficients` in der Spalte `(Intercept)` der Schnittpunkt a mit der y -Achse und unter dem Namen des Prädiktors (hier: `height`) die Steigung b

⁴ Für Verfahren zur sog. robusten Regression vgl. die Funktionen `lqs()` und `r1m()` aus dem MASS Paket sowie Jurečková und Picek (2006). Für die Bestimmung von Parametern in nichtlinearen Vorhersagemodellen anhand der Methode der kleinsten quadrierten Abweichungen vgl. die `nls()` Funktion und Ritz und Streibig (2009). Für Maximum-Likelihood-Schätzungen der Parameter vgl. die `glm()` Funktion, deren Anwendung im Rahmen der logistischen Regression Abschn. 7.4 demonstriert. Die Ridge-Regression wird durch die Funktion `lm.ridge()` aus dem MASS Paket bereitgestellt.

⁵ Mit `na.action=na.omit` zum Ausschluss aller Fälle mit mindestens einem fehlenden Wert ist zu beachten, dass das Ergebnis entsprechend weniger vorhergesagte Werte und Residuen umfasst. Dies kann etwa dann relevant sein, wenn diese Werte mit den ursprünglichen Datenvektoren in einer Rechnung auftauchen.

ausgegeben, die auch als b -Gewicht bezeichnet wird. Das Ergebnis kann manuell verifiziert werden:

```
> (b <- cov(weight, height) / var(height))      # b-Gewicht
[1] 0.3022421

> (a <- mean(weight) - b*mean(height))        # y-Achsenabschnitt
[1] 27.33506

> fitVals <- b * height + a                   # vorhergesagte Werte
```

Soll statt des b -Gewichts das standardisierte β -Gewicht berechnet werden, so sind die Variablen in der Formel zu z -standardisieren. Der y -Achsenabschnitt sollte in diesem Fall 0 ergeben, was gerundet der Fall ist, das β -Gewicht ist gleich der Korrelation von Prädiktor und Kriterium.

```
> lm(scale(weight) ~ scale(height))
Call:
lm(formula = scale(weight) ~ scale(height))
Coefficients:
(Intercept) scale(height)
 2.027e-15  5.599e-01

> b * sd(height)/sd(weight)                  # manuelle Kontrolle
[1] 0.5599206

> cor(height, weight)
[1] 0.5599206
```

Ein von `lm()` zurückgegebenes Objekt stellt ein deskriptives Modell der Daten dar, in dem in einer Liste die zur Modellanpassung berechneten Größen als Komponenten gespeichert sind und kann in anderen Funktionen weiter verwendet werden. Um auf die gespeicherten Kennwerte zuzugreifen, können ihre Namen mit `names()` erfragt werden.

```
> names(model)
[1] "coefficients"    "residuals"     "effects"      "rank"
[5] "fitted.values"   "assign"       "qr"          "df.residual"
[9] "xlevels"         "call"        "terms"       "model"
```

Zum Extrahieren von Information aus einem von `lm()` erzeugten Modell dienen spezialisierte Funktionen wie `residuals(Model)` zum Anzeigen der Residuen, `coefficients(Model)` zur Ausgabe der Modellparameter, `fitted(Model)` für die vorhergesagten Werte, weiterhin `vcov(Model)` für die Kovarianzmatrix der geschätzten Parameter und für die Extraktion der Designmatrix im Sinne des Allgemeinen Linearen Modells `model.matrix(Model)`.

Für eine graphische Veranschaulichung der Regression können die Daten zunächst als Punktwolke angezeigt werden (Abb. 7.1, vgl. Abschn. 10.2). Die aus der Modellanpassung hervorgehende Schätzung der Regressionsparameter wird dieser Graphik durch die `abline(Model)` Funktion in Form eines Geradenabschnitts hinzugefügt (vgl. Abschn. 10.5). Ebenfalls abgebildet wird hier die Ge-

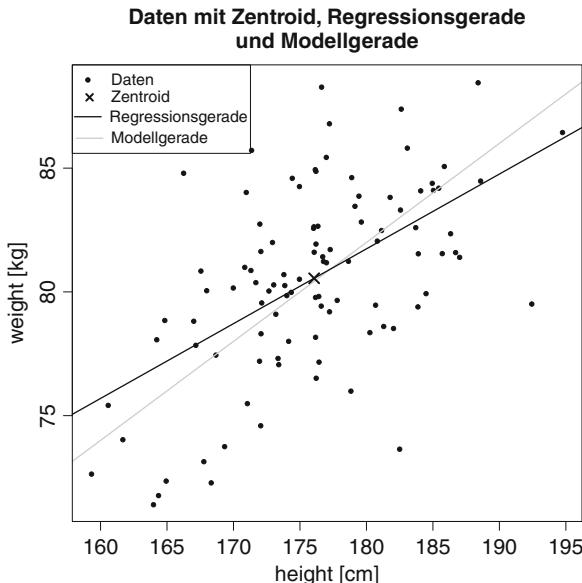


Abb. 7.1 Lineare Regression: Darstellung der Daten mit Modell- und Regressionsgerade

rade der zur Simulation verwendeten fehlerbereinigten Modellgleichung `weight = 0.4*height + 10` und das Zentroid der Daten.

```
# Daten als Streudiagramm darstellen
> plot(height, weight, xlab="height [cm]", ylab="weight [kg]", pch=20,
+      main="Daten mit Zentroid, Regressionsgerade und Modellgerade")

> abline(model, col="blue", lwd=2)           # Regressionsgerade
> abline(a=10, b=0.4, col="gray", lwd=2)       # Modellgerade
> points(mean(height), mean(weight), col="red", pch=4, cex=1.5, lwd=3)
> legend(x="topleft", legend=c("Daten", "Zentroid", "Regressionsgerade",
+      "Modellgerade"), col=c("black", "red", "blue", "gray"),
+      lwd=c(1, 3, 2, 2), pch=c(20, 4, NA, NA), lty=c(NA, NA, 1, 1))
```

7.2.1 Regressionsanalyse

Um weitere Informationen und insbesondere inferenzstatistische Kennwerte eines Modells im Sinne einer Regressionsanalyse zu erhalten, wird die `summary(<Modell>)` Funktion verwendet.⁶ Auch auf von `summary()` erzeugte Ergebnisobjekte sind die Funktionen `residuals()` und `coefficients()` zur Extraktion von Informationen anwendbar, die die entsprechenden Elemente isoliert als

⁶ Für eine Mediationsanalyse mit dem Sobel-Test vgl. die `sobel()` Funktion aus dem `multilevel` Paket, weitergehende Mediationsanalysen sind mit dem `mediation` Paket möglich (Keele et al., 2009).

Vektor bzw. Matrix zurückgeben. Zur Auskunft über weitere Elemente des Ergebnisobjekts eignen sich wieder `names()` und `str()`.

```
> (sumRes <- summary(model))
Call:
lm(formula = weight ~ height)

Residuals:
    Min      1Q  Median      3Q     Max 
-8.8433 -2.1565  0.3454  1.8255  7.5915 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) 27.33506   7.96062   3.434 0.000874 ***
height       0.30224   0.04518   6.690 1.39e-09 ***  
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 3.143 on 98 degrees of freedom
Multiple R-squared: 0.3135, Adjusted R-squared: 0.3065 
F-statistic: 44.76 on 1 and 98 DF, p-value: 1.390e-09
```

Die Ausgabe enthält unter der Überschrift `Call` die Formel, mit der das Modell angepasst wurde, `Residuals` liefert eine Zusammenfassung der beobachtungsweisen Abweichungen vom vorhergesagten zum Kriteriumswert. Unter `Coefficients` werden die Koeffizienten (Spalte `Estimate`), ihr Standardfehler (`Std. Error`), *t*-Wert (`t value`) und der dazugehörige *p*-Wert für den zweiseitigen *t*-Test (`Pr(>|t|)`) ausgegeben. Dieser Test wird mit der Nullhypothese durchgeführt, dass das jeweilige theoretische *b*-Gewicht gleich 0 ist. Die Größe des *p*-Wertes wird mit Sternchen hinter den Werten codiert. Ihre Bedeutung wird unter `Signif. codes` beschrieben.⁷ Für die geschätzten Parameter errechnet `confint.lm` das Konfidenzintervall mit der für das Argument `level` angegebenen Breite.

`Residual standard error` gibt den Standardschätzfehler aus – die Wurzel aus der Mittleren Quadratsumme der Fehler als Schätzung der Fehlervarianz, also aus dem Quotienten der Quadratsumme der Residuen und ihrer Freiheitsgrade. Der Standardschätzfehler liefert damit ein Maß für die Diskrepanz zwischen empirischen und vorhergesagten Werten.

```
# Freiheitsgrade der Residual-Quadratsumme, vgl. model$df.residual
> P          <- 1                                # Anzahl Prädiktoren
> (dfsSE   <- nSubj-P-1)                         # Freiheitsgrade
[1] 98

> MSE <- sum(residuals(model)^2) / dfsSE           # Mittlere QS Residuen
> sqrt(MSE)                                         # Standardschätzfehler
[1] 3.143246
```

⁷ Im folgenden wird dieser Teil des Outputs mit `options(show.signif.stars=FALSE)` unterdrückt.

Ein weiteres Maß für die Güte der Schätzung ist der Determinationskoeffizient R^2 , die quadrierte Korrelation zwischen Vorhersage und Kriterium (Multiple R-squared, auch multiple Korrelation zwischen Prädiktoren und Kriterium genannt). Das nach Wherry korrigierte R^2 (Adjusted R-squared) ist eine lineare Transformation von R^2 und stimmt mit ihm überein, wenn $R^2 = 1$ ist. Während das empirische R^2 im Gegensatz zum korrigierten kein erwartungstreuer Schätzer für das Quadrat der theoretischen Korrelation zwischen Vorhersage und Kriterium ist,⁸ kann das korrigierte auch negativ werden, was seine inhaltliche Interpretation erschwert.

```
> (rSq <- sumRes$r.squared)                                # unkorrigiertes R^2
[1] 0.3135111

# Kontrolle: quadrierte Korrelation von Vorhersage und Kriterium
> fitVals <- fitted(model)                                 # Vorhersage
> cor(fitVals, weight)^2
[1] 0.3135111

> 1 - ((nSubj-1)/(nSubj-P-1)) * (1-rSq)                 # korrigiertes R^2
[1] 0.3065061
```

Alternativ ergibt sich das korrigierte R^2 aus der Differenz von 1 und dem Quotienten aus der mittleren Quadratsumme der Fehler und der mittleren Quadratsumme des Kriteriums.

```
> SScrit    <- sum((weight-mean(weight))^2)                # QS Krit.: unkorrig. Var.
> dfSScrit <- nSubj-1                                      # df QS Kriterium
> 1 - (MSE / (SScrit / dfSScrit))                          # korrigiertes R^2
[1] 0.3065061
```

Schließlich wird mit einem F -Test für das gesamte Modell die Nullhypothese geprüft, dass (bei einer multiplen Regression, vgl. Abschn. 7.3) alle theoretischen b_j -Gewichte gleich 0 sind. Die Einzelheiten des Tests sind der empirische Wert des F -Bruchs (F-statistic) gefolgt von seinen Zähler- und Nenner-Freiheitsgraden (DF) sowie dem p -Wert (p-value).

```
> MSpred   <- sum((fitVals-mean(fitVals))^2) / P      # Mittl. QS Vorhersage
> (Fval    <- MSpred / MSE)                            # Teststatistik F-Wert
[1] 44.75539

> (critF <- qf(1-0.05, P, dfSSE))                   # kritischer F-Wert für alpha=0.05
[1] 3.938111

> (pVal <- 1-pf(Fval, P, dfSSE))                  # p-Wert
[1] 1.390238e-09
```

⁸ Eine theoretische Korrelation von 0 wird systematisch überschätzt, Erwartungswert des empirischen R^2 ist dann $(P - 1)/(N - 1)$.

7.2.2 Regressionsdiagnostik

Soll zur Überprüfung der Voraussetzungen einer Regressionsanalyse eingeschätzt werden, ob die Residuen mit der Annahme einer Normalverteiltheit der Fehler verträglich sind, kann dies visuell-exploratorisch mit einem Quantil-Quantil-Diagramm (Abb. 7.2, vgl. Abschn. 10.6.5) geschehen. Ob die Gültigkeit der Annahme plausibel ist, dass die Residuen unabhängig von der Vorhersage sind, lässt sich ebenfalls graphisch abschätzen.

```
> qqnorm(residuals(model)) # Q-Q-Plot der Residuen
> qqline(residuals(model), col="red", lwd=2) # Referenzgerade für NV

# Residuen gegen Vorhersage darstellen
> plot(fitted(model), residuals(model), pch=20, ylab="Residuen",
+       xlab="Vorhersage", main="Residuen gegen Vorhersage")

> abline(h=0, col="blue", lwd=2) # Referenz für Modellgültigkeit

termplot() erstellt Graphiken, in denen Regressionsterme in Abhängigkeit vom Prädiktor dargestellt werden. Dabei ist es möglich, die Residuen (Argument partial.resid=TRUE) sowie die Verteilung des Prädiktors und der Regressionsterme ebenfalls anzeigen zu lassen (Argument rug=TRUE). termplot() eignet sich gleichermaßen zur graphischen Veranschaulichung von Varianzanalysen (vgl. Abschn. 8.3).

> termplot(model=<Modell>, partial.resid=FALSE, rug=FALSE)

> termplot(model, partial.resid=TRUE, rug=TRUE, xlab="Prädiktor",
+           ylab="Residuen", main="Residuen, Prädiktoren, Regr. gerade")

# zusätzliche Beschriftungen
> mtext(text="Einzelwerte Prädiktor", side=1)
> mtext(text="Einzelwerte Kriterium", side=2)
```

Weitere Diagnosemöglichkeiten werden in ?influence aufgezeigt. Graphisch aufbereitete Informationen aus der Modellanpassung liefert zudem eine Serie von Graphiken, die insbesondere der Veranschaulichung der Verteilung der Residuen dienen

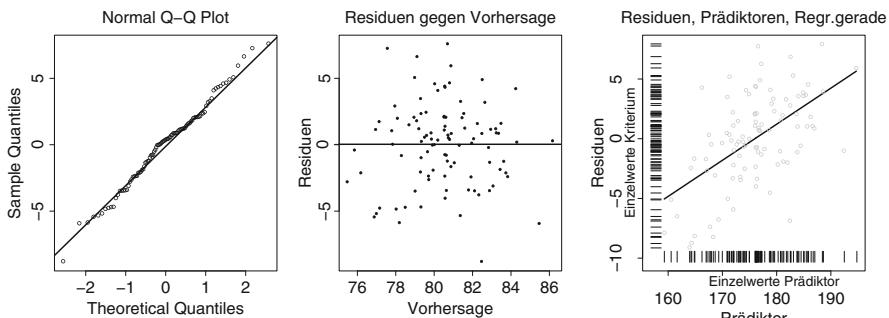


Abb. 7.2 Graphische Prüfung der Voraussetzungen für eine Regressionsanalyse

und so zur Überprüfung auch anderer Voraussetzungen für eine Regressionsanalyse verwendet werden können. Über das Argument `which` können dabei auch einzelne Graphiken der Serie selektiv gezeigt werden:

```
> plot(<Modell>, which=1:6)
```

7.2.3 Vorhersage bei Anwendung auf andere Daten

Soll das von `lm()` bestimmte Regressionsmodell auf weitere Daten angewendet werden, kann dies manuell durch Extrahieren der Parameter der Regression (Steigung sowie y-Achsenabschnitt) und anschließendes Einsetzen der neuen Daten in die Vorhersagegleichung erreicht werden.

```
> myData    <- round(rnorm(100, 75, 10))           # Prädiktor
> crit      <- (0.5*myData + 133) + rnorm(100, 0, 10) # Kriterium
> model     <- lm(crit ~ myData)                  # Regression anpassen
> (coeffs   <- coefficients(model))            # y-Achsenabschnitt und Steigung
(Intercept)  myData
135.4817507 0.4528806

> newData   <- sample(50:150, 5, replace=TRUE)       # neue Daten
> (newPred  <- coeffs[2]*newData + coeffs[1])      # Vorhersage
[1] 187.5630 159.4844 190.2803 202.9610 184.3929
```

Eine weitergehende Möglichkeit zur Anwendung eines Regressionsmodells auf neue Daten stellt die `predict()` Funktion zur Verfügung. Für Hilfe zu dieser Funktion vgl. `?predict.lm`.⁹

```
> predict(object=<Modell>, newdata=<Datensatz>, se.fit=FALSE,
+          interval=NULL, level=(Breite Konfidenzintervall))
```

Als erstes Argument erwartet die Funktion ein von `lm()` erzeugtes Objekt mit den angepassten Regressionsparametern. Werden alle weiteren Argumente weggelassen, liefert die Funktion die vorhergesagten Werte für die ursprünglichen Prädiktorwerte zurück, also `fitted(<Modell>)`. Wird unter `newdata` ein Datensatz übergeben, der eine Variable mit demselben Namen wie dem des ursprünglichen Prädiktors enthält, so wird die Vorhersage für die Werte dieser neuen Variable berechnet.¹⁰ Sollen die Standardabweichungen für die Vorhersagen berechnet werden, ist `se.fit=TRUE` zu setzen. Wird das Argument `interval="confidence"` verwendet, berechnet `predict()` zusätzlich für jeden vorhergesagten Wert die Grenzen des zugehörigen Konfidenzintervalls, dessen Breite mit `level` kontrolliert wird. Die Ausgabe erfolgt

⁹ Dies ist die hier intern automatisch aufgerufene Funktion, da `predict()` generisch ist (vgl. Abschn. 11.1.5).

¹⁰ Handelt es sich etwa im Rahmen einer Kovarianzanalyse (vgl. Abschn. 8.8) um einen kategorialen Prädiktor, mithin ein Objekt der Klasse `factor`, so muss die zugehörige Variable in `newdata` dieselben Stufen in derselben Reihenfolge beinhalten wie die des ursprünglichen Modells – selbst wenn nicht alle Faktorstufen tatsächlich als Ausprägung vorkommen.

in Form einer Matrix, deren erste Spalte (`fit`) die Vorhersage ist, während die beiden weiteren Spalten untere (`lwr`) und obere Grenzen (`upr`) des Vertrauensbereichs nennen.

Im Beispiel wird für den Aufruf von `predict()` ein Datensatz mit einer passend benannten Variable erzeugt. Die erste Spalte der Ausgabe entspricht den manuell berechneten Werten für die Vorhersage auf Basis der neuen Daten, die folgenden beiden Spalten führen jeweils untere und obere Grenzen des Konfidenzintervalls auf.

```
> newDf <- data.frame(myData=newData)
> predict(model, newDf, interval="confidence", level=0.95)
  fit      lwr      upr
1 187.5630 179.6296 195.4964
2 159.4844 154.4441 164.5247
3 190.2803 181.1936 199.3670
4 202.9610 188.4179 217.5040
5 184.3929 177.7894 190.9963
```

Der von `predict()` berechnete Vertrauensbereich um die Vorhersage für die ursprünglichen Daten lässt sich auch graphisch darstellen (Abb. 7.3).

```
> orgPred <- predict(model, interval="prediction", level=0.95)
> plot(myData, crit, pch=20, xlab="Prädiktor", ylab="Kriterium und
+       Vorhersage", main="Daten und Vorhersage durch Regression")
>
> myOrder <- order(myData)                                # für Vertrauensbereich
> polygon(c(myData[myOrder], rev(myData[myOrder])), 
+           c(orgPred[myOrder, "lwr"], orgPred[rev(myOrder), "upr"])),
```

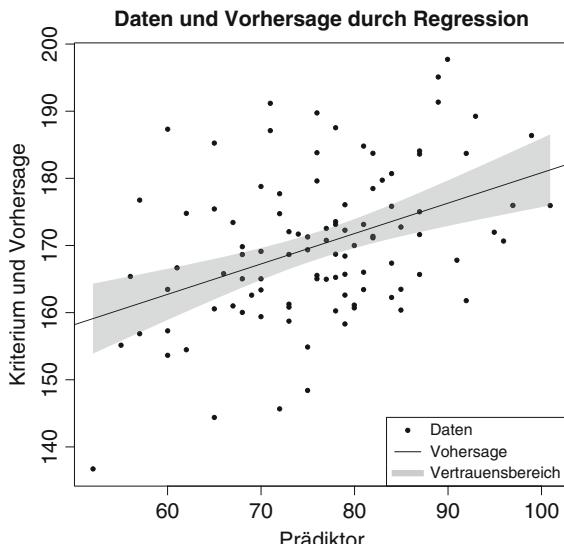


Abb. 7.3 Lineare Regression: Prädiktor, Kriterium und Vorhersage mit Vertrauensbereich

```

+     border=NA, col=rgb(0.7, 0.7, 0.7, 0.6))

> abline(model, col="blue")                      # Regressionsgerade
> legend(x="bottomright", legend=c("Daten", "Vohersage",
+     "Vertrauensbereich"), pch=c(20, NA, NA), lty=c(NA, 1, 1),
+     lwd=c(NA, 1, 8), col=c("black", "blue", "gray"))

```

7.2.4 Partialkorrelation

Ein von `lm()` erzeugtes Objekt kann auch zur Ermittlung der Partialkorrelation zweier Variablen ohne eine dritte verwendet werden (vgl. Abschn. 2.6.6), da sich die Partialkorrelation als Korrelation der Residuen der Regressionen jeweils einer der beiden Variablen auf die dritte ergibt.

```

> z <- runif(100)
> x <- z + rnorm(100, 0, 0.5)
> y <- z + rnorm(100, 0, 0.5)
> cor(cbind(z, x, y))                         # Korrelationsmatrix
      z         x         y
z  1.0000000  0.4820418  0.5533335
x  0.4820418  1.0000000  0.3217158
y  0.5533335  0.3217158  1.0000000

> model1 <- lm(x ~ z)                         # Regression 1
> model2 <- lm(y ~ z)                         # Regression 2
> cor(residuals(model1), residuals(model2))    # Partialkorrelation
[1] 0.0753442

# Kontrolle über herkömmliche Formel
> (cor(x,y)-(cor(x,z)*cor(y,z))) / (sqrt((1-cor(x,z)^2) * (1-cor(y,z)^2)))
[1] 0.0753442

```

7.2.5 Kreuzvalidierung

Da empirische Daten fehlerbehaftet sind, bezieht die Anpassung einer linearen Regression auf Basis eines konkreten Datensatzes immer auch die Messfehler mit ein und orientiert sich damit zu stark an den zufälligen Besonderheiten einer einzelnen Stichprobe. Mit einer Kreuzvalidierung lässt sich die Stabilität der Parameterschätzungen heuristisch beurteilen, wie gut sich also ein berechnetes Regressionsmodell auf andere Stichproben generalisieren lässt. Im Fall einer sog. doppelten Kreuzvalidierung wird dafür die Gesamtmenge der Daten zufällig in zwei disjunkte Mengen partitioniert und für beide separat eine Regression entsprechend desselben theoretischen Modells angepasst. In die sich aus der ersten Teilstichprobe ergebende Regressionsgleichung werden dann die Prädiktor-Werte der zweiten Stichprobe eingesetzt. Die hieraus entstandene Vorhersage wird mit den tatsächlichen Werten des Kriteriums der zweiten Stichprobe korreliert. Für die sich aus der zweiten

Stichprobe ergebende Gleichung wird anhand der Daten der ersten Stichprobe ebenso verfahren.¹¹

```

> nSubj      <- 100                                # Gesamtstichprobe
> predAll   <- sample(50:150, nSubj, replace=TRUE)    # Prädiktor
> critAll   <- predAll + rnorm(nSubj, mean=0, sd=30)  # Kriterium

# Gesamtstichprobe zufällig in zwei Teilstichproben partitionieren
> select     <- sample(rep(c(TRUE, FALSE), nSubj/2), nSubj, replace=FALSE)
> pred1      <- predAll[ select]                  # Prädiktor 1. Teilstichprobe
> crit1      <- critAll[ select]                  # Kriterium 1. Teilstichprobe
> pred2      <- predAll[!select]                 # Prädiktor 2. Teilstichprobe
> crit2      <- critAll[!select]                 # Kriterium 2. Teilstichprobe

# Regressionsmodell in den Stichproben anpassen
> model1    <- lm(crit1 ~ pred1)                # Modell in 1. Teilstichprobe
> model2    <- lm(crit2 ~ pred2)                # Modell in 2. Teilstichprobe
> (coeffs1 <- coefficients(model1))           # Koeff. aus 1. Teilstichprobe
(Intercept) pred1
-26.459314  1.211349

> (coeffs2 <- coefficients(model2))           # Koeff. aus 2. Teilstichprobe
(Intercept) pred2
19.6660302  0.7884972

# Anpassung aus 1. Teilstichprobe auf 2. Teilstichprobe anwenden
> pred1in2 <- coeffs1[2]*pred2 + coeffs1[1]

# Anpassung aus 2. Teilstichprobe auf 1. Teilstichprobe anwenden
> pred2in1 <- coeffs2[2]*pred1 + coeffs2[1]
> (cor(pred1in2, crit2))^2          # Determinationskoeffizienten vergleichen
[1] 0.3742471

> (cor(pred2in1, crit1))^2
[1] 0.6319137

```

Unterschiedlichkeit und mittlere Größe der beiden Determinationskoeffizienten liefern einen Hinweis auf die Stabilität der Parameterschätzungen.

7.3 Multiple lineare Regression

Bei der multiplen linearen Regression dienen mehrere quantitative Variablen X_j als Prädiktoren zur Vorhersage des quantitativen Kriteriums Y .¹² Die Gleichung für das theoretische Regressionsmodell hat hier die Form $Y = b_1X_1 + \dots + b_jX_j +$

¹¹ Kompliziertere Vorgehensweisen sind denkbar, etwa die mehrfache neu-Partitionierung der Gesamtstichprobe in zwei oder mehr Teilstichproben mit anschließender Berechnung der Anpassungsgüte. Das Paket DAAG stellt mit `cv.lm()` eine Funktion für die Kreuzvalidierung linearer Modelle bereit.

¹² Für die multivariate multiple Regression mit mehreren Kriteriumsvariablen Y_k vgl. Abschn. 9.1.

$\dots + b_p X_p + b_0$, wobei die Parameter b_j und b_0 auf Basis der empirischen Daten zu schätzen sind. Dies geschieht auch hier mit `lm(<Formel>)` nach dem Kriterium der kleinsten Summe der quadrierten Abweichungen von Vorhersage und Kriterium. `<Formel>` hat nun die Form `<Kriterium> ~ <Prädiktor 1> + ... + <Prädiktor j> + ... + <Prädiktor p>`, d.h. alle j Prädiktoren werden mit `+` verbunden hinter die `~` geschrieben.

Als Beispiel soll nun `weight` wie bisher aus der Körpergröße `height` vorhergesagt werden, aber auch mit Hilfe des Alters `age` und später ebenfalls mit der Anzahl der Minuten, die pro Woche Sport getrieben wird (`sport`). Der Simulation der Daten wird die Gültigkeit eines entsprechenden linearen Modells mit zufälligen Fehlern zugrundegelegt.

```

> nSubj    <- 100
> height   <- rnorm(nSubj, 175, 7)                      # Prädiktor 1
> age       <- rnorm(nSubj, 30, 8)                      # Prädiktor 2
> sport     <- abs(rnorm(nSubj, 60, 30))                # Prädiktor 3

# Simulation des Kriteriums im Modell der multiplen linearen Regression
> weight   <- 0.5*height - 0.3*age - 0.4*sport + 10 + rnorm(nSubj, 0, 3)
> (model1  <- lm(weight ~ height + age))
Call:
lm(formula = weight ~ height + age)

```

Coefficients:
(Intercept) height age
-72.5404 0.8602 -0.4429

Die Schätzungen für die Parameter b_j und b_0 werden in der Ausgabe unter der Überschrift **Coefficients** genannt, wobei b_0 in der Spalte **(Intercept)** und die b_j -Gewichte unter dem Namen des zugehörigen Prädiktors stehen. Die Ergebnisse lassen sich manuell verifizieren:¹³

```

> X    <- cbind(height, age)          # Matrix der Prädiktoren
> K    <- cov(X)                   # Kovarianzmatrix der Prädiktoren
> k    <- cov(X, weight)           # Kovarianzen Prädiktoren mit Kriterium
> (b   <- solve(K, k))            # berechne K^(-1)*k -> Regr.gewichte
[,1]
height  0.8602345
age     -0.4428933

# absoluter Term - y-Achsenabschnitt
> (a <- as.vector(mean(weight) - t(b) %*% colMeans(X)))
[1] -72.54045

> fitVals <- X %*% b + a          # vorhergesagte Werte

```

Auch hier müssen die Variablen in der Formel z -standardisiert werden, um die standardisierten β_j -Gewichte zu berechnen:

¹³ Hier sei vorausgesetzt, dass die Kovarianzmatrix der Prädiktoren invertierbar ist (vgl. Abschn. 2.9.2), also keine lineare Abhängigkeit zwischen den Prädiktoren vorliegt.

```
> lm(scale(weight) ~ scale(height) + scale(age)) # ...
> (1/sd(weight)) * diag(sqrt(diag(K))) %*% b      # manuelle Kontrolle ...
```

Die graphische Veranschaulichung mit der `scatterplot3d()` Funktion aus dem gleichlautenden Paket (Ligges und Mächler, 2003) zeigt die simulierten Daten zusammen mit der Vorhersageebene sowie die Residuen als vertikale Abstände zwischen ihr und den Daten (Abb. 7.4).

```
> b0 <- model1$coefficients[1] # extrahiere Parameter aus Regression
> b1 <- model1$coefficients[2]
> b2 <- model1$coefficients[3]

> library(scatterplot3d)      # Streudiagramm für dreidimensionale Daten
> sp3 <- scatterplot3d(height, age, weight, pch=16, color="red",
+   zlab="Kriterium: weight", mar=c(3, 3, 4, 2),
+   main="Multiple Regression: Daten und Vorhersageebene")
> sp3$plane3d(model1)        # Vorhersageebene

# Residuen einzeichnen, dafür 3D-Punkte auf 2D Koordinaten abbilden
> xyEmp <- sp3$xyz.convert(height, age, weight)
> xyPred <- sp3$xyz.convert(height, age, b1*height + b2*age + b0)
> segments(xyPred$x, xyPred$y, xyEmp$x, xyEmp$y, col="blue")
```

7.3.1 Modell verändern

Möchte man ein bereits berechnetes Modell nachträglich verändern, ihm etwa einen weiteren Prädiktor hinzufügen, kann dies mit der `update()` Funktion geschehen.

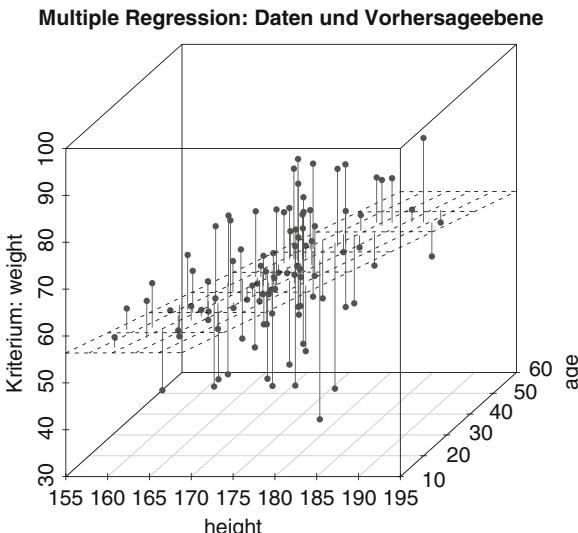


Abb. 7.4 Daten, Vorhersageebene und Residuen einer multiplen linearen Regression

```
> update(model.object=<Modell>, . ~ . + <weiterer Prädiktor>)
```

Unter `model.object` wird das Modell eingetragen, welches erweitert werden soll. Der Term `. ~ .` signalisiert, dass alle bisherigen Prädiktoren beizubehalten sind. Jeder weitere Prädiktor wird dann mit einem `+` angefügt.

```
> (model2 <- update(model1, . ~ . + sport)) # zusätzlicher Prädiktor sport
Call:
lm(formula = weight ~ height + age + sport)

Coefficients:
(Intercept) height      age      sport
     8.1012    0.5149   -0.2866   -0.4160
```

Auf die gleiche Weise wie Prädiktoren hinzugefügt werden können, werden sie auch entfernt, wobei statt des `+` ein `-` zu verwenden ist.

```
> (model3 <- update(model2, . ~ . - age)) # entferne Prädiktor age
Call:
lm(formula = weight ~ height + sport)

Coefficients:
(Intercept) height      sport
     4.9841    0.4855   -0.4255
```

7.3.2 Modell auswählen

Beim Vergleich verschiedener Modelle mit gleichem Kriterium aber unterschiedlichen Sätzen von Prädiktoren kann sich die Frage stellen, inwieweit die Hinzunahme von Prädiktoren zu einer bedeutsamen Verbesserung des Gesamtmodells führt. Dieser Frage kann zum einen mit Hilfe von Modelltests in Form von Varianzanalysen nachgegangen werden, die die Veränderung von R^2 in Abhängigkeit vom verwendeten Prädiktorensatz testen (vgl. Abschn. 8.3.5). Zum anderen existieren weitere Kriterien, mit denen sich eine Modellverbesserung durch Hinzufügen von Prädiktoren bewerten lässt. Um verschiedene Regressionsmodelle miteinander hinsichtlich der Quadratsumme der Residuen sowie des sog. Informationskriteriums AIC¹⁴ vergleichen zu können, lassen sich folgende Funktionen verwenden:

```
> step(object=<Modell>, scope=~ . + <Prädiktoren>)
> add1(object=<Modell>, scope=~ . + <Prädiktoren>, test="none")
> drop1(object=<Modell>, scope=~ . - <Prädiktoren>, test="none")
```

¹⁴ Beim Akaike Information Criterion AIC stehen kleinere Werte für eine höhere Informativität. Es berücksichtigt einerseits die Güte der Modellanpassung, andererseits die Komplexität des Modells, gemessen an der Anzahl zu schätzender Parameter. Besitzen zwei Modelle dieselbe Anpassungsgröße, erhält das Modell mit einer geringeren Anzahl von Parametern den kleineren AIC-Wert, vgl. `?AIC`.

`step()` prüft im sog. Stepwise-Verfahren sequentiell Teilmengen von denjenigen Prädiktoren der unter `object` angegebenen Regression, die in `scope` als Formel definiert werden. Diese Formel beginnt mit einer \sim und enthält danach alle Prädiktoren aus `object` (Voreinstellung) sowie ggf. weitere. `step()` berechnet in jedem Schritt die Auswirkung einer Modellveränderung durch Hinzunahme oder Auslassen eines Prädiktors auf Vergleichskriterien und wählt schließlich ein Modell, das durch diese Schritte nicht substantiell verbessert werden kann.¹⁵ `add1()` berechnet den Effekt der Hinzunahme von Prädiktoren zu einer unter `object` angegebenen Regression, `drop1()` den Effekt des Weglassens von Prädiktoren. Die Änderung in der Quadratsumme der Residuen kann inferenzstatistisch geprüft werden, indem das Argument `test="F"` gesetzt wird. Das einfachste Modell, in dem kein Prädiktor berücksichtigt wird, lautet `<Kriterium> ~ 1` und sorgt dafür, dass für jeden Wert der konstante Mittelwert des Kriteriums vorhergesagt wird.

```
> add1(model1, .~. + sport)
Single term additions
Model:
weight ~ height
  Df Sum of Sq    RSS   AIC
<none>        13211.6 494.4
sport     1    12232.0   979.6 236.2
```

In der Ausgabe werden in den letzten beiden Zeilen die Kennwerte des unter `object` genannten Modells und jenes Modells verglichen, bei dem zusätzlich der Prädiktor `sport` berücksichtigt wird. Die Spalte `Sum of Sq` nennt den Zugewinn an aufgeklärter Varianz beim Wechsel hin zum unter `scope` genannten Modell, also die sog. sequentielle Quadratsumme vom Typ I des zusätzlichen Prädiktors. Sie entspricht der Differenz der in der Spalte `RSS` genannten Quadratsummen der Residuen (Residual Sum of Squares), ist also gleich der Reduktion an Fehlervarianz beim Modellwechsel. Auf ähnliche Weise kann mit `drop1()` die partielle Quadratsumme vom Typ III eines Vorhersageterms berechnet werden (vgl. Abschn. 8.5.2). Die Spalte `AIC` listet den Wert von Akaike's Informationskriterium für beide Modelle auf.

7.3.3 Auf Multikollinearität prüfen

Multikollinearität liegt dann vor, wenn sich die Werte eines Prädiktors aus einer Linearkombination der Werte der übrigen Prädiktoren vorhersagen lassen. Dies ist insbesondere dann der Fall, wenn Prädiktoren paarweise miteinander korrelieren. Für die multiple Regression hat dies als unerwünschte Konsequenz einerseits weniger stabile Schätzungen der Koeffizienten zur Folge, die mit hohen Schätzfehlern versehen sind. Ebenso kann sich die Parameterschätzung bzgl. desselben Prädiktors stark in Abhängigkeit davon ändern, welche anderen Prädiktoren noch berücksich-

¹⁵ Dieses Verfahren ist mit vielen inhaltlichen Problemen verbunden. Für eine Diskussion und verschiedene Strategien zur Auswahl von Prädiktoren vgl. Miller (2002).

tigt werden. Andererseits ergeben sich Schwierigkeiten bei der Interpretation der b_j - bzw. der standardisierten β_j -Gewichte: verglichen mit der Korrelation der zugehörigen Variable mit dem Kriterium können letztere unerwartet große oder kleine Werte annehmen und auch im Vorzeichen von der Korrelation abweichen.

Ob Kollinearität vorliegt, lässt sich anhand der Korrelationsmatrix der Prädiktoren prüfen. Gegebenenfalls sollte auf einige Prädiktoren verzichtet werden, bei denen eine zu hohe Korrelation vorhanden ist.

```
> X3 <- cbind(height, age, sport)           # Datenmatrix der Prädiktoren
> cor(X3)                                    # Korrelationsmatrix
      height        age        sport
height  1.00000000  0.06782798 -0.20937559
age     0.06782798  1.00000000  0.09496699
sport   -0.20937559  0.09496699  1.00000000
```

Weitere Möglichkeiten zur Kollinearitätsdiagnostik enthält das Paket `car`, dessen Funktion `vif(mod=Model1)` den sog. Varianzinflationsfaktor VIF_j eines jeden Prädiktors j berechnet. Als Argument `mod` erwarten `vif()` ein durch `lm()` erstelltes lineares Modell. VIF_j berechnet sich als $1/(1 - R_j^2)$, also als Kehrwert der sog. Toleranz $1 - R_j^2$, wobei R_j^2 der Determinationskoeffizient bei der Regression des Prädiktors j auf alle übrigen Prädiktoren ist.

```
> library(car)
> vif(model2)
      height        age        sport
1.054409  1.017361  1.059110
```

In der Ausgabe findet sich unter dem Namen jedes Prädiktors der jeweils zugehörige VIF_j -Wert. Da geringe Werte für die Toleranz auf lineare Abhängigkeit zwischen den Prädiktoren hindeuten, gilt dasselbe für große VIF-Werte. Konventionell werden VIF-Werte von bis zu ca. 4 als unkritisch, jene über 10 als starke Indikatoren für Multikollinearität gewertet.

Ein weiterer Kennwert zur Beurteilung von Multikollinearität ist die Kondition κ der Designmatrix des meist mit standardisierten Variablen gebildeten linearen Modells (vgl. Abschn. 2.9.5). Werte von $\kappa > 20$ sprechen einer Faustregel folgend für Multikollinearität. Zur Berechnung dient die `kappa(lm-Modell)` Funktion.¹⁶

```
> kappa(lm(scale(weight) ~ scale(height) + scale(age) + scale(support)),
+       exact=TRUE)                         # Kondition kappa
[1] 1.279833
```

7.4 Logistische Regression

Die Vorhersage von Werten einer dichotomen Variable Y (codiert als 0 und 1) kann ebenfalls durch die Anpassung eines Regressionsmodells erfolgen (für Details vgl.

¹⁶ Fortgeschrittene Methoden zur Diagnostik von Multikollinearität enthält das Paket `perturb` (Hendrickx, 2008).

Agresti, 2007). Im Vergleich zur Vorhersage quantitativer Variablen treten aber u. a. zwei Schwierigkeiten auf:

- Im allgemeinen kann eine lineare Funktion von p Prädiktoren X_j der Form $Y = b_1X_1 + \dots + b_jX_j + \dots + b_pX_p + b_0$ bei einem unbeschränkten Definitionsbereich Werte im Intervall von $-\infty$ bis $+\infty$ annehmen. Es können sich also Vorhersagen ergeben, die keine mögliche Ausprägung einer dichotomen Variable sind. Dieses Problem besteht auch dann, wenn statt der Ausprägungen der Variable selbst die Wahrscheinlichkeit eines Treffers (je nach Definition der Wert 0 oder 1) vorhergesagt werden soll, die zwischen 0 und 1 liegt.
- Für eine Regressionsanalyse im Sinne des inferenzstatistischen Tests eines linearen Regressionsmodells ist vorauszusetzen, dass die Abweichungen der Messwerte zur zugehörigen Vorhersage unabhängige Realisierungen einer normalverteilten Variable mit Erwartungswert 0 sind. Bei einer dichotomen Variable kann dies nicht der Fall sein.

Die Modellierung von Y ist möglich, wenn statt der Ausprägungen selbst der natürliche Logarithmus des Wettquotienten (sog. Odds) vorhergesagt werden soll: bezeichnet P die Wahrscheinlichkeit eines Treffers, ist dies $\ln(P/(1 - P))$, die sog. Logit-Funktion. Da Logits einen Wertebereich von $-\infty$ bis $+\infty$ besitzen, können sie besser durch eine lineare Funktion von Prädiktoren vorhergesagt werden.¹⁷ Nimmt man $\ln(P/(1 - P)) = b_1X_1 + \dots + b_jX_j + \dots + b_pX_p + b_0$ als gültiges statistisches Modell an und fasst die rechte Seite der Gleichung unter X zusammen ($\ln(P/(1 - P)) = X$), so ergibt sich mit der logistischen Funktion als Umkehrfunktion der Logit-Funktion $P = e^X / (1 + e^X) = 1 / (1 + e^{-X})$.

Die Anpassung einer logistischen Regression durch eine Maximum-Likelihood-Schätzung der Parameter b_j geschieht mit der `glm()` Funktion, mit der Modelle aus der Familie des Verallgemeinerten Linearen Modells spezifiziert werden können.¹⁸ Die AV muss ein Objekt der Klasse `factor` mit zwei Stufen sein, die Auftretenswahrscheinlichkeit P bezieht sich dann auf jene Ausprägung der AV, die als erste Faktorstufe definiert ist.

```
> glm(formula=Formel, family=Verteilungsfamilie, data=Datensatz)
```

Unter `formula` wird ein Vorhersagemodell angegeben, das zunächst als lineares Modell wie bei der Verwendung von `lm()` formuliert wird, wobei sowohl quantitative wie kategoriale Variablen als Prädiktoren möglich sind. Zusätzlich ist unter `family` ein Objekt anzugeben, das die für die AV angenommene Verteilungsfamilie sowie die Link-Funktion benennt (vgl. `?family`) – im Fall der logistischen Regression ist dieses Argument auf `binomial(link="logit")` zu setzen.¹⁹

¹⁷ Solche Transformationen des eigentlich vorherzusagenden Parameters, die eine lineare Modellierung ermöglichen, heißen auch Link-Funktion.

¹⁸ Für weitere Funktionen zum Erstellen und Analysieren von logistischen Regressionsmodellen vgl. das Paket `rms` (Harrell Jr, 2009b).

¹⁹ Für eine Probit-Regression mit der Umkehrfunktion der Verteilungsfunktion der Standardnormalverteilung als Link-Funktion entsprechend `binomial(link="probit")`. Soll eine gewöhnliche lineare Regression angepasst werden, wäre `gaussian(link="identity")` einzutragen, da

Als Beispiel sei jenes aus der einfachen linearen Regression herangezogen, wobei hier mit der Körpergröße vorhergesagt werden soll, ob das Gewicht einer Person über dem Median liegt.

```
> nSubj <- 100
> height <- rnorm(nSubj, 175, 7)                      # Prädiktor
> weight <- 0.5*height + 10 + rnorm(nSubj, 0, 3)        # Kriterium

# Umwandlung der quantitativen AV in dichotomen Faktor
> wFac <- cut(weight, breaks=c(-Inf, median(weight), Inf))
> levels(wFac) <- c("lo", "hi")
> regDf <- data.frame(wFac, height)                   # Datensatz

# Anpassung des logistischen Regressionsmodells
> (glmRes <- glm(wFac ~ height, family=binomial(link="logit"), regDf))
Call: glm(formula=wFac ~ height, family=binomial(link="logit"), data=regDf)

Coefficients:
(Intercept) height
-33.707    0.193

Degrees of Freedom: 99 Total (i.e. Null); 98 Residual
Null Deviance:      138.6
Residual Deviance: 104.5      AIC: 108.5
```

Die Ausgabe nennt unter der Überschrift **Coefficients** zunächst die Schätzung der Parameter b_j des Modells der logistischen Regression, wobei der in der Spalte **(Intercept)** aufgeführte Wert die Schätzung für b_0 ist. Der Parameter eines Prädiktors ist als Ausmaß der Änderung der Vorhersage von $\ln(P/(1 - P))$ zu interpretieren, wenn der Prädiktor X_j um eine Einheit wächst. Etwas einfacher ist die Bedeutung des exponenzierten Parameters e^{b_j} zu erfassen: dieser Koeffizient gibt an, um welchen Faktor sich die Vorhersage des Wettquotienten ändert, wenn sich X_j um eine Einheit vergrößert. Dies ist das Verhältnis des vorhergesagten Wettquotienten nach der Änderung um eine Einheit zum Wettquotienten vor der Änderung um eine Einheit, also ihre Odds Ratio. e^{b_0} besitzt dagegen keine einfache Bedeutung. Als Güte der Modellpassung wird die Deviance, also die Summe der quadrierten Residuen²⁰ und der Wert des Informationskriteriums AIC ausgegeben, bei dem kleinere Werte gewünscht sind (vgl. Abschn. 7.3.2, Fußnote 14).

```
> exp(glmRes$coefficients)                      # exponenzierte Koeffizienten
(Intercept) height
2.297060e-15 1.212883e+00
```

hier von einer normalverteilten AV ausgegangen wird, die nicht durch eine Link-Funktion transformiert werden muss, um linear modellierbar zu sein.

²⁰ Im Sinne von `sum(residuals((GLM-Modell))^2)`, wobei es sich um sog. Deviance Residuen handelt, vgl. `?residuals.glm`. Dagegen speichert `(GLM-Modell)$residuals` die sog. Working Residuen.

Die logarithmierte Likelihood des Modells berechnet `logLik((GLM-Modell))`. Die vorhergesagte Wahrscheinlichkeit $P = 1/(1+e^{-X})$ ergibt sich durch Einsetzen der Parameterschätzungen in $X = b_1 X_1 + \dots + b_j X_j + \dots + b_p X_p + b_0$ bzw. durch `fitted((GLM-Modell))` oder `predict((GLM-Modell), type="response")`.

```
> logLik(glmRes)                      # log-Likelihood
'log Lik.' -52.27401 (df=2)

> fitted(glmRes)                     # vorhergesagte Wahrscheinlichkeit ...
```

Das Ergebnis lässt sich graphisch darstellen, wobei die auf Basis der Vorhersage für die Trefferwahrscheinlichkeit vorgenommene Klassifikation bzgl. des dichotomen Kriteriums mit der tatsächlichen Ausprägung verglichen werden kann (Abb. 7.5).

```
> b0      <- glmRes$coefficients[1]      # extrahierte Parameter
> b1      <- glmRes$coefficients[2]
> cutoff  <- -b0/b1                      # Trennwert bei p=0.5

# durch den Trennwert der Vorhersage falsch klassifizierte Daten
> wrong   <- (height < cutoff & wFac=="hi") | (height >= cutoff & wFac=="lo")

# zeichne Vorhersage für die Wahrscheinlichkeit
> curve(1/(1+exp(-(b1*x + b0))), from=150, to=200, lwd=2,
+        xlab="height", ylab="p(> Median)" bzw. Gruppe",
+        main="Vorhersage der logistischen Regression")
> abline(h=0.5, v=cutoff, col="gray")    # Trennlinie für Klassifikation

# tatsächliche Gruppenzugehörigkeit, Punkte dabei leicht vertikal versetzt
```

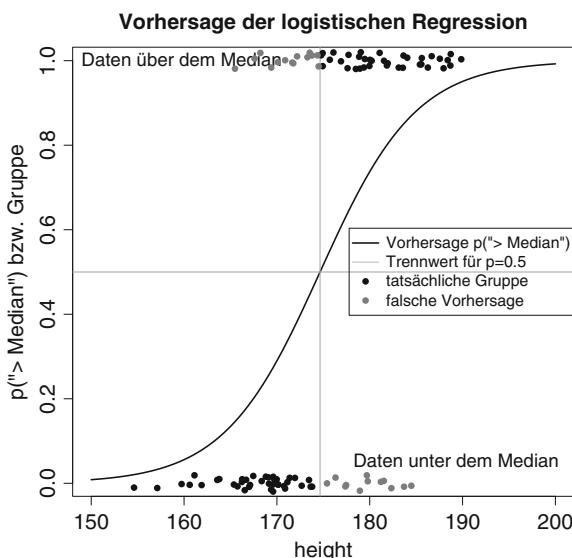


Abb. 7.5 Vorhersage und Daten einer logistischen Regression. Daten, die auf Basis der Vorhersage falsch klassifiziert würden, sind grau hervorgehoben

```
> points(height[!wrong], jitter(as.numeric(wFac[!wrong]=="hi"), 0.1),
+         col="black", pch=16)

> points(height[ wrong], jitter(as.numeric(wFac[ wrong]=="hi"), 0.1),
+         col="red", pch=16)

> text(150, 1, adj=0, label="Daten über dem Median")
> text(200, 0, adj=1, label="Daten unter dem Median")
> legend(x="right", c("Vorhersage p(> Median)", "Trennwert für p=0.5",
+         "tatsächliche Gruppe", "falsche Vorhersage"), col=c("black", "gray",
+         "black", "red"), pch=c(NA, NA, 1, 16), lwd=c(2, 2, NA, NA),
+         lty=c(1, 1, NA, NA))
```

Die Parameter b_j des Modells lassen sich mit `summary(GLM-Modell)` einem Signifikanztest unterziehen. Teststatistik dieses Wald-Tests sind unter der Nullhypothese standardnormalverteilte z -Werte.

```
> summary(glmRes)
Call:
glm(formula = wFac ~ height, family = binomial, data = regDf)

Deviance Residuals:
    Min      1Q  Median      3Q     Max 
-2.00853 -0.81160  0.05788  0.84680  1.96560 

Coefficients:
            Estimate Std. Error z value Pr(>|z|)    
(Intercept) -33.70715   7.13133  -4.727 2.28e-06 ***
height       0.19300   0.04083   4.726 2.29e-06 ***

(Dispersion parameter for binomial family taken to be 1)

Null deviance: 138.63 on 99 degrees of freedom
Residual deviance: 104.55 on 98 degrees of freedom
AIC: 119.06

Number of Fisher Scoring iterations: 4
```

Um das Gesamtmodell mit einem χ^2 -Test auf Signifikanz zu prüfen, muss `anova(0-Modell, GLM-Modell, test="Chisq")` aufgerufen werden. Der Test beruht auf dem Vergleich des angepassten Modells mit einem Modell, das nur eine Konstante als Prädiktor beinhaltet. Teststatistik ist die Deviance-Differenz beider Modelle mit der Differenz ihrer Freiheitsgrade als Anzahl der Freiheitsgrade der asymptotisch gültigen χ^2 -Verteilung.

```
> glm0 <- glm(wFac~1, family=binomial(link="logit"), data=regDf)
> anova(glm0, glmRes, test="Chisq")
Analysis of Deviance Table

Model 1: wFac ~ 1
Model 2: wFac ~ height
Resid. Df Resid. Dev Df Deviance P(>|Chi|)
```

```
1 99 138.63
2 98 104.55 1 34.081 5.285e-09 ***

# manuelle Kontrolle durch Vergleich zum NULL-Modell
# Teststatistik, Freiheitsgrade und p-Wert
> statChisq <- glmRes$null.deviance - glmRes$deviance
> dfChisq   <- glmRes$df.null      - glmRes$df.residual
> (pVal     <- 1-pchisq(statChisq, dfChisq))
[1] 5.285363e-09
```

Kapitel 8

Parametrische Tests für Dispersions- und Lageparameter von Verteilungen

Häufig bestehen in empirischen Untersuchungen Hypothesen über Erwartungswerte von Variablen. Die für solche Hypothesen geeigneten Tests gehen davon aus, dass bestimmte Annahmen über die Verteilungen der Variablen erfüllt sind, dass etwa in allen Bedingungen Normalverteilungen derselben Varianz vorliegen. Bevor auf Tests zum Vergleich von Erwartungswerten selbst eingegangen wird, sollen deshalb zunächst jene Verfahren vorgestellt werden, die sich mit der Prüfung statistischer Voraussetzungen befassen (vgl. auch Abschn. 6.1). Für die statistischen Grundlagen dieser Themen vgl. die hierauf spezialisierte Literatur (Bortz, 2005; Hartung et al., 2005; Kirk, 1995; Maxwell und Delaney, 2004).

8.1 Tests auf Varianzhomogenität

Die ab Abschn. 8.2 vorgestellten Tests auf Erwartungswertunterschiede setzen voraus, dass die Variable in allen Bedingungen normalverteilt mit derselben Varianz ist. Um zu prüfen, ob die erhobenen Werte mit der Annahme von Varianzhomogenität konsistent sind, existieren verschiedene Testverfahren, bei deren Anwendung der Umstand zu berücksichtigen ist, dass i. d. R. die Nullhypothese den gewünschten Zustand darstellt. Um den β -Fehler zu reduzieren, wird häufig mit einem höher als üblichen α -Niveau in der Größenordnung von 0.2 oder 0.25 getestet.

8.1.1 F-Test auf Varianzhomogenität bei zwei Stichproben

Um festzustellen, ob die empirischen Varianzen einer Variable in zwei unabhängigen Stichproben mit der Nullhypothese verträglich sind, dass die Variable in beiden Bedingungen dieselbe theoretische Varianz besitzt, kann die Funktion `var.test()` benutzt werden. Diese vergleicht beide empirischen Varianzen mittels eines

F-Tests, der aber sensibel auf Verletzungen der Voraussetzung reagiert, dass die Variable in den Bedingungen normalverteilt ist.¹

```
> var.test(x=(Vektor), y=(Vektor), conf.level=0.95,
+           alternative=c("two.sided", "less", "greater"))
```

Unter x und y sind die Daten aus beiden Stichproben einzutragen. Alternativ kann auch eine Formel $\langle AV \rangle \sim \langle UV \rangle$ eingegeben werden. Stammen die in der Formel verwendeten Variablennamen aus einem Datensatz, ist dieser unter data zu nennen. Die Argumente alternative und conf.level beziehen sich auf die Größe des *F*-Bruchs unter der Alternativhypothese im Vergleich zu seiner Größe unter der Nullhypothese (1) bzw. auf die Breite seines Vertrauensintervalls.

```
> n1 <- 110                                # Gruppengröße 1
> n2 <- 90                                  # Gruppengröße 2
> one <- rnorm(n1, mean=100, sd=15)          # simulierte AV Gruppe 1
> two <- rnorm(n2, mean=100, sd=13)          # simulierte AV Gruppe 2
> var.test(one, two)
F test to compare two variances
data: one and two
F = 1.6821, num df = 109, denom df = 89, p-value = 0.01157
alternative hypothesis: true ratio of variances is not equal to 1
95 percent confidence interval:
1.124894 2.495117
sample estimates:
ratio of variances
1.682140
```

Die Ausgabe des Tests umfasst den empirischen *F*-Wert (*F*), der sich auch als Quotient der zu testenden Varianzen (ratio of variances) wiederfindet. Zusammen mit den zugehörigen Freiheitsgraden (UV-Effekt: num df, Fehler: denom df) wird der *p*-Wert (p-value) ausgegeben sowie das Konfidenzintervall für das Verhältnis der Varianzen in der gewünschten Breite. Das Ergebnis lässt sich manuell bestätigen:

```
# p-Wert: das Doppelte der Fläche (zweiseitiger Test) rechts der
# Teststatistik unter Dichtefunktion unter H0 -> Verteilungsfunktion
> (pValF <- 2 * (1-pf(var(one)/var(two), n1-1, n2-1)))
[1] 0.0115746
```

```
# zweiseitiges 95%-Vertrauensintervall für den Quotient der Varianzen
> critF1 <- qf(1-(0.05/2), n1-1, n2-1)      # kritischer Wert links
> critF2 <- qf(0.05/2, n1-1, n2-1)          # kritischer Wert rechts
> (ciLo <- var(one) / (critF1*var(two)))    # VI untere Grenze
[1] 1.124894

> (ciUp <- var(one) / (critF2*var(two)))    # VI obere Grenze
[1] 2.495117
```

¹ Für die Erweiterung zu Hartleys F_{\max} -Test für mehrere Stichproben steht in R keine eigene Funktion zur Verfügung. Als Alternativen für den Fall zweier Stichproben existieren mit mood.test() und ansari.test() auch die Tests nach Mood bzw. Ansari-Bradley, die keine Normalverteiltheit voraussetzen.

8.1.2 Fligner-Killeen-Test und Bartlett-Test

Der Fligner-Killeen-Test auf Varianzhomogenität benutzt eine asymptotisch χ^2 -verteilte Teststatistik und gilt als robust gegenüber Verletzungen der Voraussetzung von Normalverteiltheit. Es können auch mehr als zwei Gruppen miteinander verglichen werden.

```
> fligner.test(x=<Vektor>, g=<Faktor>)
```

Unter x wird der Datenvektor eingetragen und unter g die zugehörige Gruppierungsvariable als Objekt der Klasse `factor` derselben Länge wie x, die für jeden Wert in x angibt, aus welcher Bedingung er stammt. Alternativ zu x und g kann eine Formel $\langle AV \rangle \sim \langle UV \rangle$ verwendet und ggf. unter `data` ein Datensatz angegeben werden, aus dem die in der Formel verwendeten Variablen stammen.

Der auf mehr als zwei Gruppen anwendbare Bartlett-Test auf Varianzhomogenität basiert auf einer Teststatistik, die asymptotisch χ^2 -verteilt ist und reagiert sensibel auf Verletzungen der Normalverteiltheit. Die Argumente von `bartlett.test(x=<Vektor>, g=<Faktor>)` besitzen dieselbe Bedeutung wie in `fligner.test()`. Als weitere Alternative lässt sich ein lineares Modell verwenden, wie es `lm()` als Objekt zurückgibt.

8.1.3 Levene-Test

Ein weiterer Test auf Varianzhomogenität ist jener nach Levene aus dem Paket `car`. Der Levene-Test kann ebenfalls die Daten einer in mehr als zwei Stichproben erhobenen Variable auf die Nullhypothese testen, dass die Variable in den zugehörigen Bedingungen gleiche Varianz besitzt.

```
> levene.test(y=<Daten>, group=<Faktor>)
```

Die Daten y können in Form eines Vektors zusammen mit einer zugehörigen Gruppierungsvariable group als Objekt der Klasse `factor` derselben Länge wie y angegeben werden. Alternativ ist dies als Formel $\langle AV \rangle \sim \langle UV \rangle$ oder als lineares Modell möglich, wie es `lm()` als Objekt zurückgibt.

```
> library(car)
> nSubj    <- 30                                # Zellbesetzung
> nGroups  <- 3                                  # Anzahl Gruppen
> myData   <- sample(0:100, nGroups*nSubj, replace=TRUE)  # AV Daten

# VPn zufällig auf Gruppen aufteilen
> group <- factor(sample(rep(1:nGroups, each=nSubj), nGroups*nSubj,
+                      replace=FALSE))

> levene.test(myData, group)
Levene's Test for Homogeneity of Variance
  Df F value Pr(>F)
group  2  0.4797 0.6206
```

Da der Levene-Test letztlich eine Varianzanalyse ist, beinhaltet die Ausgabe einen empirischen F -Wert (`F value`) mit den Freiheitsgraden von Effekt- und Fehler-Quadratsumme (`Df`) sowie den zugehörigen p -Wert (`Pr(>F)`). In der beim Levene-Test durchgeführten Varianzanalyse gehen statt der ursprünglichen Werte der AV die jeweiligen Beträge ihrer Differenz zum zugehörigen Gruppenmedian ein. Der Test lässt sich so auch manuell durchführen (vgl. Abschn. 8.3).

```
# Berechnung mit den absoluten Abweichungen zum Mittelwert jeder Gruppe
> model <- lm(myData ~ group) # für Residuen
> anova(lm(abs(residuals(model)) ~ group)) # Varianzanalyse ...

# Berechnung mit den absoluten Abweichungen zum Median jeder Gruppe
> groupMeds <- tapply(myData, group, median) # Gruppen-Mediane
> absDiff   <- abs(myData - groupMeds[group]) # absolute Abweichungen
> anova(lm(absDiff ~ group)) # Varianzanalyse ...
```

8.2 t -Tests

Hypothesen über den Erwartungswert bzw. die Erwartungswerte einer Variable in einer oder zwei Bedingungen lassen sich mit t -Tests prüfen.

8.2.1 t -Test für eine Stichprobe

Der einfache t -Test prüft, ob die in einer Stichprobe ermittelten Werte einer normalverteilten Variable mit der Nullhypothese verträglich sind, dass diese Variable einen bestimmten Erwartungswert besitzt.

```
> t.test(x=Vektor, alternative=c("two.sided", "less", "greater"),
+         mu=0, conf.level=0.95)
```

Unter `x` ist der Datenvektor einzutragen. Mit `alternative` wird festgelegt, ob die Alternativhypothese gerichtet oder ungerichtet ist. `"less"` und `"greater"` beziehen sich dabei auf die Reihenfolge Erwartungswert unter der Alternativhypothese `"less"` bzw. `"greater"` Erwartungswert unter der Nullhypothese. Das Argument `mu` bestimmt den Erwartungswert unter der Nullhypothese. `conf.level` legt die Breite des je nach Alternativhypothese ein- oder zweiseitigen Konfidenzintervalls für den Erwartungswert fest.

```
> nSubj  <- 100
> myData <- rnorm(nSubj, 5, 20) # simulierte AV
> muH0   <- 0 # Erwartungswert unter H0
> t.test(myData, alternative="two.sided", mu=muH0)
One Sample t-test
data: myData
t = 2.4125, df = 99, p-value = 0.01769
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
```

```
0.8989476 9.2292614
sample estimates:
mean of x
5.064104
```

Die Ausgabe umfasst den empirischen *t*-Wert (*t*) mit den Freiheitsgraden (*df*) und dem zugehörigen *p*-Wert (*p-value*). Weiterhin wird das Konfidenzintervall für den Erwartungswert in der gewünschten Breite sowie der empirische Mittelwert genannt. Das Ergebnis lässt sich manuell verifizieren:

```
# empirischer t-Wert
> (tVal <- (mean(myData)-muH0) / (sd(myData)/sqrt(nSubj)))
[1] 2.412462

# p-Wert: das Doppelte der Fläche (zweiseitiger Test) rechts von tVal
# unter Dichtefunktion unter H0 - Berechnung über Verteilungsfunktion
> (pVal <- 2 * (1-pt(tVal, nSubj-1)))
[1] 0.01768618

# zweiseitiges 95%-Vertrauensintervall für den Erwartungswert
> tCrit <- qt(1-(0.05/2), nSubj-1)           # kritischer t-Wert
> (ciLo <- mean(myData) - tCrit*sd(myData)/sqrt(nSubj))
[1] 0.8989476

> (ciUp <- mean(myData) + tCrit*sd(myData)/sqrt(nSubj))
[1] 9.229261
```

8.2.2 *t*-Test für zwei unabhängige Stichproben

Im *t*-Test für unabhängige Stichproben werden die in zwei unabhängigen Stichproben ermittelten Werte einer normalverteilten Variable daraufhin miteinander verglichen, ob sie mit der Nullhypothese verträglich sind, dass die Variable in den zugehörigen Bedingungen denselben Erwartungswert besitzt.

```
> t.test(x=<Vektor>, y=<Vektor>, paired=FALSE, conf.level=0.95,
+         alternative=c("two.sided", "less", "greater"), var.equal=FALSE)
```

Unter *x* sind die Daten der ersten Stichprobe einzutragen, unter *y* entsprechend die der zweiten. Alternativ kann statt *x* und *y* auch das *formula* Argument verwendet und eine Formel $\langle AV \rangle \sim \langle UV \rangle$ angegeben werden. Dabei ist $\langle UV \rangle$ ein Faktor mit zwei Ausprägungen und derselben Länge wie $\langle AV \rangle$, der für jede Beobachtung die Gruppenzugehörigkeit codiert. Mit *alternative* wird festgelegt, ob die Alternativhypothese gerichtet oder ungerichtet ist. "less" und "greater" beziehen sich dabei auf die Reihenfolge *x* "less" bzw. "greater" *y*. Mit dem Argument *paired* wird bestimmt, ob es sich um unabhängige (FALSE) oder abhängige (TRUE) Stichproben handelt. *var.equal* gibt an, ob von Varianzhomogenität in den beiden Bedingungen ausgegangen werden soll (Voreinstellung ist FALSE). Das von *conf.level* in seiner Breite festgelegte Vertrauensintervall bezieht sich auf die Differenz der Erwartungswerte und ist je nach Alternativhypothese ein- oder zweiseitig.

Als Beispiel soll die Körpergröße von Männern und Frauen betrachtet werden. Die Fragestellung ist gerichtet – getestet werden soll, ob der Erwartungswert bei den Männern größer als jener bei den Frauen ist. Zunächst sei Varianzhomogenität vorausgesetzt.

```
> n1      <- 18                      # Stichprobenumfang 1
> n2      <- 21                      # Stichprobenumfang 2
> dataM  <- rnorm(n1, 180, 10)        # simulierte AV Männer
> dataW  <- rnorm(n2, 175, 6)         # simulierte AV Frauen
> t.test(dataM, dataW, alternative="greater", var.equal=TRUE)

Two Sample t-test
data: dataM and dataW
t = 1.6346, df = 37, p-value = 0.05531
alternative hypothesis: true difference in means is greater than 0
95 percent confidence interval:
-0.1201456      Inf
sample estimates:
mean of x  mean of y
180.1867   176.4451
```

Das Ergebnis lässt sich manuell verifizieren:

```
# Schätzung der Streuung der Erwartungswertdifferenz
> estSigDiff <- sqrt((n1+n2)/(n1*n2)) * sqrt(((n1-1)*var(dataM)
+                               + (n2-1)*var(dataW)) / (n1+n2-2))

> (tVal      <- (mean(dataM)-mean(dataW)) / estSigDiff)      # t-Wert
[1] 1.634606

# p-Wert einseitiger Test: Fläche rechts von tVal unter Dichtefunktion
# unter H0 - Berechnung über Verteilungsfunktion
> (pVal <- 1-pt(tVal, n1+n2-2))
[1] 0.05530639

# untere Grenze des einseitigen 95%-Vertrauensintervalls für mu
> tCrit <- qt(1-0.05, n1+n2-2)      # zweiseitig: qt(1-(0.05/2), n1+n2-2)
> (ciLo <- mean(dataM)-mean(dataW) - tCrit*estSigDiff)
[1] -0.1201456
```

Ohne Voraussetzung von Varianzhomogenität verwendet R die als Welch-Test bezeichnete Variante des *t*-Tests, deren andere Berechnung der Teststatistik sowie der Freiheitsgrade zu einem i. a. etwas konservativeren Test führt.

```
> t.test(dataM, dataW, alternative="greater", var.equal=FALSE)

Welch Two Sample t-test
data: dataM and dataW
t = 1.5681, df = 25.727, p-value = 0.06454
alternative hypothesis: true difference in means is greater than 0
95 percent confidence interval:
-0.3298099      Inf
sample estimates:
mean of x  mean of y
180.1867   176.4451
```

Das Ergebnis lässt sich auch manuell nachvollziehen.

```
> ss1      <- var(dataM)                      # korrigierte Varianz Männer
> ss2      <- var(dataW)                      # korrigierte Varianz Frauen
> num      <- (ss1/n1 + ss2/n2)^2            # Zähler
> denom    <- ss1^2/((n1-1)*n1^2) + ss2^2/((n2-1)*n2^2)  # Nenner
> (dfWelch <- num/denom)                      # Freiheitsgrade im Welch-Test
[1] 25.72683

# Teststatistik
> (tValW  <- (mean(dataM)-mean(dataW)) / sqrt(ss1/n1 + ss2/n2))
[1] 1.568068

> (pVal <- 1-pt(tValW, dfWelch))             # p-Wert einseitiger Test
[1] 0.06454212
```

8.2.3 t-Test für zwei abhängige Stichproben

Der *t*-Test für abhängige Stichproben prüft, ob die Erwartungswerte einer in zwei Bedingungen paarweise erhobenen Variable identisch sind. Er wird wie jener für unabhängige Stichproben durchgeführt, jedoch ist hier das Argument `paired=TRUE` für `t.test()` zu verwenden. Der Test setzt voraus, dass sich die in `x` und `y` angegebenen Daten einander paarweise zuordnen lassen, weshalb `x` und `y` dieselbe Länge besitzen müssen. Abhängige Daten liegen etwa in Situationen mit mehrfacher Messung einer Variable an denselben VPn oder an miteinander in einer engen Beziehung stehenden Personen (z. B. Geschwister oder Ehepartner) vor. Aber auch wenn verschiedene, hinsichtlich einer Störvariable ähnliche VPn einander zugeordnet (gematcht) werden, handelt es sich um abhängige Beobachtungen. Der Test prüft die Nullhypothese, dass die sich aus paarweiser Subtraktion ergebende Differenzvariable von `x` und `y` den Erwartungswert 0 besitzt. Entsprechend bezieht sich auch das Konfidenzintervall auf den Erwartungswert dieser Differenzvariable.

```
> pre  <- rnorm(20, mean=90,  sd=15)
> post <- rnorm(20, mean=100, sd=15)
> t.test(pre, post, alternative="less", paired=TRUE)
Paired t-test
data: pre and post
t = -2.0818, df = 19, p-value = 0.02556
alternative hypothesis: true difference in means is less than 0
95 percent confidence interval:
-Inf -1.790952
sample estimates:
mean of the differences
-10.57236

# äquivalent: 1-Stichproben t-Test der Differenzvariable
> diffVar <- pre-post                                # Differenzvariable
> t.test(diffVar, alternative="less")                 # ...
```

8.3 Einfaktorielle Varianzanalyse (CR-*p*)

Bestehen Hypothesen über die Erwartungswerte einer Variable in mehr als zwei Bedingungen, wird häufig eine Varianzanalyse (ANOVA, Analysis of Variance) zur Prüfung herangezogen. Die einfaktorielle Varianzanalyse ohne Messwiederholung mit p Gruppen (CR-*p*, Completely Randomized Design²) verallgemeinert dabei die Fragestellung eines *t*-Tests für unabhängige Stichproben auf Situationen, in denen Werte einer normalverteilten Variable in mehr als zwei Gruppen ermittelt werden.

Durch die enge Verwandtschaft von linearer Regression und Varianzanalyse ähneln sich die Befehle für beide Analysen in R häufig stark. Zur Unterscheidung ist die Art der modellierenden Variablen bedeutsam: im Fall der Regression sind dies quantitative Prädiktoren (numerische Vektoren), im Fall der Varianzanalyse dagegen kategoriale Gruppierungsvariablen, also Objekte der Klasse `factor`. Damit R auch bei Gruppierungsvariablen mit numerisch codierten Stufen die richtige Interpretation als Faktor und nicht als quantitativer Prädiktor vornehmen kann, ist darauf zu achten, dass die UVn tatsächlich die Klasse `factor` besitzen.

8.3.1 Regression und Varianzanalyse als lineare Modelle

Wiewohl sich Regression und Varianzanalyse insofern unterscheiden, als sich im erstgenannten Modell eine Variable Y aus quantitativen, und im letztgenannten aus kategorialen Variablen ergibt, lassen sich beide auf dieselbe Weise darstellen. Eine vertiefte Darstellung der folgenden Sachverhalte würde den Formalismus des Allgemeinen Linearen Modells erfordern, der hier nicht erarbeitet werden kann (Andres, 1996; Mardia et al., 1980). Da es aber implizit den Berechnungen von R zugrunde liegt und dabei auch bisweilen für den Anwender sichtbar wird, soll kurz skizziert werden, wie sich eine einfaktorielle Varianzanalyse mit p unabhängigen Gruppen über verschiedene Arten der Codierung der UV im Modell der multiplen Regression $Y = b_1X_1 + \dots + b_jX_j + \dots + b_pX_p + b_0$ formulieren lässt (vgl. Abschn. 7.3). Dabei sind verschiedene Codierschemata möglich, von denen die inhaltliche Bedeutung der zu schätzenden Parameter b_0 bis b_p abhängt.

Für die sog. Dummy-Codierung wird zunächst jede der p Faktorstufen zu einem eigenen dichotomen Prädiktor X_j , der auch als Indikatorvariable bezeichnet wird. Beobachtungsobjekte erhalten für ein X_j den Wert 1, wenn sie sich in der zugehörigen Bedingung j befinden, sonst 0. Ein Beobachtungsobjekt erhält also einmal die 1 und $p - 1$ mal die 0. Diese Art der Codierung führt jedoch zu vollständig redundanten Prädiktoren und macht damit die Parameter b_j unbestimmbare. Bereits die Werte derselben $p - 1$ Prädiktoren reichen nämlich aus, um für alle Beobachtungsobjekte zu wissen, welchen Wert sie bzgl. des p -ten Prädiktors haben: ist unter den ersten $p - 1$ Werten eine 1, ist der p -te Wert 0. Sind alle $p - 1$ Werte 0, ist der

² Hier und im folgenden wird für Versuchspläne die Notation von Kirk (1995) übernommen.

p-te Wert 1.³ Um diese Redundanz zu beseitigen, sind verschiedene Nebenbedingungen denkbar. In R wird per Voreinstellung der zur ersten Faktorstufe gehörende Prädiktor nicht mit in die Regression aufgenommen.⁴ Als Konsequenz ergibt sich Y für alle Beobachtungsobjekte aus der ersten Gruppe als absoluter Term b_0 , da für diese Gruppe alle $X_j = 0$ sind (mit $2 \leq j \leq p$). Der Parameter b_0 hat damit die Bedeutung des Erwartungswertes Y der ersten Gruppe. Für Mitglieder der Gruppe j (mit $2 \leq j \leq p$) erhält man Y als $b_0 + b_j$, da dann $X_j = 1$ ist. Mit anderen Worten gibt b_j die Differenz $Y - b_0$, also die Differenz des Erwartungswertes der Gruppe j zum Erwartungswert der Referenzgruppe an.

Offensichtlich lassen sich auch *t*-Tests für zwei Stichproben durch eine Dummy-Codierung als Regression betrachten, wobei das Modell der einfachen Regression $Y = bX + a$ resultiert. Hier ist a der Erwartungswert in der Referenzgruppe und b die Differenz zum anderen Erwartungswert.

Das beschriebene Vorgehen wird in R als Treatment-Kontrast bezeichnet. Der Name leitet sich aus der Situation ab, dass die erste Faktorstufe eine Kontrollgruppe darstellt, während die übrigen zu Treatment-Gruppen gehören. Die Parameter b_j können dann als Wirkung der Stufe j im Sinne der Differenz zur Kontrollgruppe verstanden werden. Die `contr.treatment(n=(Anzahl))` Funktion gibt die Matrix der Dummy-codierten Prädiktoren für n Faktorstufen aus, wobei die Gruppenzugehörigkeiten in den Zeilen, die Werte auf den Prädiktoren in den Spalten stehen.

```
> contr.treatment(4)          # Dummy-Codierung für 4-stufigen Faktor
   2 3 4
1 0 0 0
2 1 0 0
3 0 1 0
4 0 0 1
```

In der mit `options("contrasts")` einsehbaren Voreinstellung werden Treatment-Kontraste verwendet, wenn ungeordnete kategoriale Prädiktoren in einer Regression vorhanden sind und polynomiale Kontraste für ordinale Prädiktoren. Deren Stufen werden dabei als gleichabständig vorausgesetzt, vgl. `?contrasts`.

```
> options("contrasts")
$contrasts
  unordered      ordered
"contr.treatment" "contr.poly"
```

In der Varianzanalyse ist die Konzeption des Effekts einer Bedingung jedoch meist eine andere und als Differenz eines Gruppenerwartungswertes zum mittleren Erwartungswert aller Gruppen definiert. Auch dieser Vergleich lässt sich über die sog. Effektcodierung als Regressionsmodell ausdrücken: hierfür wird zunächst jede Faktorstufe zu einem separaten Prädiktor X_j , der die Werte $-1, 0$ und 1 annehmen kann.

³ Aus der Perspektive des Allgemeinen Linearen Modells bedeutet dies, dass die Designmatrix nicht vollen Rang hat. Dies verhindert die Identifizierbarkeit der Parameter b_j .

⁴ Die weggelassene Gruppe ist die erste Stufe von `levels(Faktor)`. Sie kann durch explizites Setzen der Reihenfolge der Stufen kontrolliert werden (vgl. Abschn. 2.7.4).

Zur Beseitigung der Redundanz wird dann der zur letzten Stufe gehörende Prädiktor X_p gestrichen. Beobachtungsobjekte aus der Gruppe j (mit $1 \leq j \leq p-1$) erhalten für X_j den Wert 1, sonst 0. Beobachtungsobjekte aus der Gruppe p erhalten auf allen Prädiktoren den Wert -1 . Mit dieser Codierung erhält der Parameter b_0 die Bedeutung des mittleren Erwartungswertes und die b_j der jeweiligen Differenz zu ihm. Für Mitglieder der ersten $p-1$ Gruppen ergibt sich Y nämlich als $b_0 + b_j$, für Mitglieder der Gruppe p als $b_0 - (b_1 + \dots + b_j + \dots + b_{p-1})$. Dabei stellt $(b_1 + \dots + b_j + \dots + b_{p-1})$ die Abweichung des Erwartungswertes der p -ten Gruppen vom mittleren Erwartungswert dar, weil sich die Abweichungen der Gruppenerwartungswerte vom mittleren Erwartungswert zu 0 summieren müssen. Die `contr.sum(n=<Anzahl>)` Funktion gibt die Matrix der so codierten Prädiktoren für n Faktorstufen aus, wobei die Gruppenzugehörigkeiten in den Zeilen, die Werte auf den Prädiktoren in den Spalten stehen.

```
> contr.sum(4)                                # Effektcodierung für 4-stufigen Faktor
   [,1]  [,2]  [,3]
1     1     0     0
2     0     1     0
3     0     0     1
4    -1    -1    -1
```

Soll bei nominalskalierten Prädiktoren jede Gruppe nicht mit der Referenzstufe, sondern mit dem Gesamtmittel verglichen werden, ist dies mit `options(contrasts=c("contr.sum", "contr.poly"))` einzustellen.

8.3.2 Auswertung mit `oneway.test()`

In der einfaktoriellen Varianzanalyse wird geprüft, ob die in verschiedenen Bedingungen erhobenen Werte einer Variable mit der Nullhypothese verträglich sind, dass diese Variable in allen Gruppen denselben Erwartungswert besitzt. Die Alternativhypothese ist unspezifisch und lautet, dass sich mindestens zwei Erwartungswerte unterscheiden.

```
> oneway.test(formula=<Formel>, data=<Datensatz>, subset=<Indexvektor>,
+              var.equal=FALSE)
```

Unter `formula` sind Daten und Gruppierungsvariable als Formel $\langle AV \rangle \sim \langle UV \rangle$ einzugeben, wobei $\langle UV \rangle$ ein Faktor derselben Länge wie $\langle AV \rangle$ ist und für jede Beobachtung in $\langle AV \rangle$ die zugehörige UV-Stufe angibt. Geschieht dies mit Variablennamen, die nur innerhalb eines Datensatzes bekannt sind, muss dieser unter `data` eingetragen werden. Das Argument `subset` erlaubt es, nur eine Teilmenge der Beobachtungen einfließen zu lassen, es erwartet einen entsprechenden Indexvektor, der sich auf die Zeilen des Datensatzes bezieht. Mit `var.equal` wird vorgegeben, ob von Varianzhomogenität ausgegangen werden kann (Voreinstellung ist `FALSE`).

```
# Ernteertrag in den Monaten Januar, Februar und März
> yield <- c(2349, 2554, 2278, 2476, 2242, 2124, 2649, 2106, 2107,
+           2185, 2095, 2070)
```

```
# Bedingungen: Erntemonat
> month <- factor(rep(1:3, each=4), labels=c("Jan", "Feb", "Mar"))
> dfCrop <- data.frame(month, yield)
> oneway.test(yield ~ month, data=dfCrop, var.equal=TRUE)
One-way analysis of means
data: yield and month
F = 3.3083, num df = 2, denom df = 9, p-value = 0.08374
```

Die Ausgabe von `oneway.test()` beschränkt sich auf die wesentlichen Ergebnisse des Hypothesentests: dies sind der empirische *F*-Wert (*F*) mit den Freiheitsgraden der Quadratsumme zwischen den Gruppen (`num df`) aus dem Zähler (Numerator) des *F*-Bruchs und jener der Quadratsumme der Fehler (`denom df`) aus dem Nenner (Denominator) des *F*-Bruchs sowie der zugehörige *p*-Wert (`p-value`).

Für ausführlichere Informationen zum Modell, etwa zu den Quadratsummen von Effekt und Fehler, sollte auf die `aov()` oder `anova()` Funktion zurückgegriffen werden (vgl. Abschn. 8.3.3 und 8.3.5). Die Ergebnisse können auch manuell überprüft werden.

```
> P      <- nlevels(month)                      # Anzahl UV-Stufen
> N      <- length(yield)                      # Gesamt-N
> groupNs <- tapply(yield, month, length)       # Gruppengrößen
> groupVs <- tapply(yield, month, var)          # korrig. Gruppenvarianzen
> groupMs <- tapply(yield, month, mean)         # Gruppenmittel
> grandM <- sum((groupNs/N) * groupMs)        # gewichtetes Gesamtmittel
> SSw    <- sum((groupNs-1) * groupVs)         # Quadratsumme within
> SSb    <- sum(groupNs * (groupMs-grandM)^2)   # Quadratsumme between
> MSw    <- SSw / (N-P)                         # MS within
> MSb    <- SSb / (P-1)                          # MS between
> (Fval  <- MSb / MSw)                         # Teststatistik F-Wert
[1] 3.308287

> (pVal <- 1-pf(Fval, P-1, N-P))              # p-Wert
[1] 0.0837443
```

Ist von Varianzhomogenität nicht auszugehen und deshalb das Argument `var.equal=FALSE` gesetzt, führt `oneway.test()` einen auf mehr als zwei Stichproben verallgemeinerten Welch-Test durch. Dieser Test berechnet *F*-Wert und Freiheitsgrade der Fehler anders als die konventionelle Varianzanalyse, wodurch er i. a. etwas konservativer ist.

```
> oneway.test(yield ~ month, data=dfCrop, var.equal=FALSE)
One-way analysis of means (not assuming equal variances)
data: yield and month
F = 9.2333, num df = 2.000, denom df = 4.664, p-value = 0.02389
```

8.3.3 Auswertung mit `aov()`

Wenn Varianzhomogenität anzunehmen ist, kann eine Varianzanalyse auch mit der Funktion `aov()` berechnet werden.

```
> aov(formula=<Formel>, data=<Datensatz>)
```

Unter `formula` werden Daten und Gruppierungsvariable als Formel $\langle AV \rangle \sim \langle UV \rangle$ eingetragen, wobei $\langle UV \rangle$ ein Faktor derselben Länge wie $\langle AV \rangle$ ist und für jede Beobachtung in $\langle AV \rangle$ die zugehörige UV-Stufe angibt. Unter `data` ist ggf. der Datensatz als Quelle der Variablen anzugeben.

```
> (aovCRp <- aov(yield ~ month, data=dfCrop))
Call:
aov(formula = yield ~ month, data = dfCrop)
```

Terms:

	month	Residuals
Sum of Squares	180682.7	245768.2
Deg. of Freedom	2	9
Residual standard error:	165.2501	
Estimated effects may be unbalanced		

Der Output beinhaltet unter der Überschrift `Terms` in der Zeile `Sum of Squares` die Zerlegung der gesamten Quadratsumme in jene des UV-Effekts (hier: `month`) und jene des Fehlers (`Residuals`) mit zugehörigen Freiheitsgraden (Zeile `Deg. of Freedom`). Der Standardfehler der Residuen (`Residual standard error`) ist die Wurzel aus der Mittleren Quadratsumme der Fehler, also aus dem Quotienten der Quadratsumme der Fehler und den zugehörigen Freiheitsgraden. Das von `aov()` zurückgegebene Objekt besteht ähnlich wie das Ergebnis von `lm()` aus einer Liste, die u. a. die berechneten Quadratsummen, Freiheitsgrade, Fehler und vorhergesagten Werte enthält, wobei letztere hier den Gruppenmittelwerten entsprechen (vgl. `names(aov-Objekt))`).

Um von UV-Effekt (hier Zeile `month`) und Fehler (Zeile `Residuals`) auch die Quadratsummen (Spalte `Sum Sq`), Mittleren Quadratsummen (Spalte `Mean Sq`) sowie den aus letzteren durch Division gebildeten F -Wert (`F value`) und zugehörigen p -Wert (`Pr(>F)`) zu erhalten, muss `summary()` auf den Output angewendet werden:

```
> summary(aovCRp)
   Df Sum Sq Mean Sq F value    Pr(>F)
month     2 180683  90341  3.3083 0.08374 .
Residuals 9 245768  27308
```

Eine tabellarische Übersicht über den Gesamtmittelwert und die Mittelwerte in den einzelnen Bedingungen erzeugt die Funktion `model.tables()`.

```
> model.tables(x=(aov-Objekt), type=c("means", "effects"), se=FALSE)
```

Im ersten Argument `x` muss das Ergebnis einer durch `aov()` vorgenommenen Modellanpassung stehen. Über das Argument `type` wird bestimmt, ob die Mittelwerte oder Effekte (Voreinstellung) ausgegeben werden sollen. Sollen Standardfehler genannt werden, ist `se=TRUE` zu setzen.

```
> model.tables(aovCRp, type="means")
Tables of means
Grand mean
2269.583
```

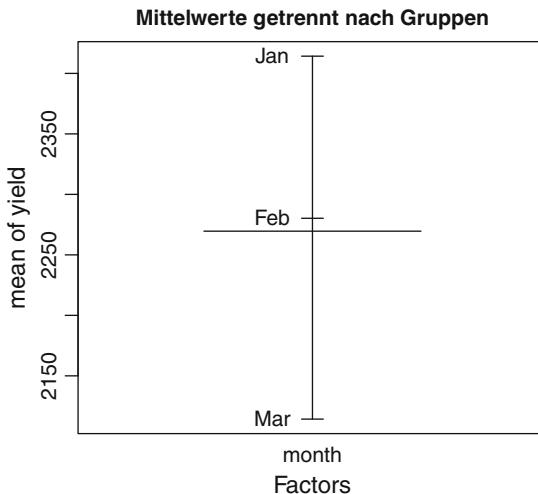


Abb. 8.1 `plot.design()` zur Darstellung der Mittelwerte pro Gruppe

```
month
Feb     Jan     Mar
2280.3 2414.3 2114.3
```

Graphisch aufbereitet werden deskriptive Kennwerte der AV in den einzelnen Gruppen mit der Funktion `plot.design()` (Abb. 8.1). In der Voreinstellung sind dies die Mittelwerte, über das Argument `fun` lassen sich jedoch durch Übergabe einer geeigneten Funktion auch beliebige andere Kennwerte berechnen. Die Anwendung von `plot.design()` ist insbesondere sinnvoll, wenn mehr als ein Faktor als UV variiert wird (vgl. Abschn. 8.5.1, Abb. 8.4).

```
> plot.design(x=<Formel>, fun=mean, data=<Datensatz>)

> plot.design(yield ~ month, fun=mean, data=dfCrop, main="Mittelwerte
+                   getrennt nach Gruppen")
```

8.3.4 Graphische Prüfung der Voraussetzungen

Voraussetzung einer varianzanalytischen Auswertung ist u. a. die Annahme, dass die Fehler unabhängige, normalverteilte Variablen mit Erwartungswert 0 und fester Streuung sind. Ob empirische Daten einer Stichprobe mit diesen Annahmen konsistent sind, kann heuristisch durch eine Reihe von Diagrammen abgeschätzt werden. Dazu können die Fehler gegen die Gruppenmittelwerte abgetragen werden – bei Modellgültigkeit sollten sie unabhängig vom Gruppenmittelwert zufällig um 0 streuen. Ebenso kann die empirische Verteilung der Fehler in jeder Gruppe mit Hilfe von Boxplots veranschaulicht werden (vgl. Abschn. 10.6.3) – die Fehlerverteilungen sollten ähnlich sein. Schließlich können die Fehler mittels eines Quantil-Quantil-

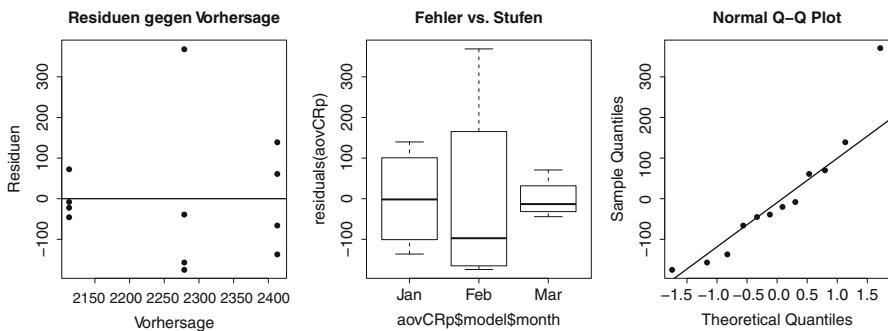


Abb. 8.2 Graphische Prüfung der Voraussetzungen für die einfaktorielle Varianzanalyse

Plots hinsichtlich ihrer Verträglichkeit mit der Annahme von Normalverteiltheit eingeschätzt werden (Abb. 8.2, vgl. Abschn. 10.6.5).

```
# Residuen gegen Vorhersage darstellen
> plot(fitted(aovCRp), residuals(aovCRp), pch=20, ylab="Residuen",
+       xlab="Vorhersage", main="Residuen gegen Vorhersage")

> abline(h=0, col="blue", lwd=2) # Referenz für Modellgültigkeit

# Verteilung der Residuen in den Gruppen veranschaulichen
> plot(residuals(aovCRp) ~ aovCRp$model$month, main="Fehler vs. Stufen")

# Normalverteiltheit der Fehler prüfen
> qqnorm(residuals(aovCRp), pch=20)      # Q-Q-Plot der Residuen
> qqline(residuals(aovCRp), col="red")     # Referenzgerade
```

8.3.5 Auswertung mit `anova()`

Äquivalent zum Aufruf von `summary(aov())` ist die Verwendung der `anova()` Funktion, wenn sie auf ein lineares Modell als Output von `lm()` angewendet wird (vgl. Kap. 7).

```
> anova(object=<Modell>)
```

Unter `object` ist das Ergebnis eines Aufrufs von `lm()` zu übergeben. Hier soll zunächst der Test mit dem Datensatz `dfCrop` aus Abschn. 8.3.2 wiederholt werden.

```
> anova(lm(yield ~ month, data=dfCrop))
Analysis of Variance Table
Response: yield
  Df Sum Sq Mean Sq F value    Pr(>F)
month     2 180683   90341   3.3083 0.08374 .
Residuals 9 245768   27308
```

Werden mehrere `lm()` Objekte übergeben, testet `anova()`, ob sich die Modelle signifikant in ihrer Varianzaufklärung unterscheiden. So lassen sich lineare Regressions mit demselben Kriterium aber unterschiedlichen Sätzen von Prädiktoren

hinsichtlich der Varianzaufklärung miteinander vergleichen (vgl. Abschn. 7.3.2). Insbesondere kann getestet werden, ob durch die Hinzunahme von Prädiktoren die Varianzaufklärung statistisch signifikant steigt.

Im Beispiel soll ein Kriterium entweder durch einen oder durch zwei Prädiktoren vorhergesagt werden. Dazu wird zunächst eine Regressionsanalyse des vollständigen Modells mit `summary(lm(<Modell>))` durchgeführt (vgl. Abschn. 7.2.1).

```
> nSubj <- 100
> pred1 <- rnorm(nSubj, 175, 7)                      # Prädiktor 1
> pred2 <- rnorm(nSubj, 30, 8)                         # Prädiktor 2

# Simulation des Kriteriums im Modell der multiplen linearen Regression
> crit      <- 0.5*pred1 - 0.3*pred2 + 10 + rnorm(nSubj, 0, 6)
> (sumRes <- summary(lm(crit ~ pred1 + pred2)))       # Regressionsanalyse
Call:
lm(formula = crit ~ pred1 + pred2)

Residuals:
    Min      1Q  Median      3Q     Max 
-20.1886 -4.5096  0.2253  4.6388 13.8862 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) 8.37578   16.58640   0.505  0.614719    
pred1        0.51550    0.09286   5.551  2.47e-07 *** 
pred2       -0.32877    0.08239  -3.990  0.000128 *** 
                                               
Residual standard error: 6.346 on 97 degrees of freedom
Multiple R-squared: 0.3361, Adjusted R-squared: 0.3224 
F-statistic: 24.55 on 2 and 97 DF, p-value: 2.358e-09
```

Der Test beider Regressionsgewichte fällt hier signifikant aus, damit natürlich auch jener des Gesamtmodells. Die Varianzanalyse zum Vergleich der Modelle mit und ohne den Prädiktor `pred2` soll Auskunft darüber geben, ob seine Hinzunahme zu einer bedeutsam höheren Varianzaufklärung führt.

```
# F-Test für die Hinzunahme von pred2
> (anovaRes <- anova(lm(crit ~ pred1), lm(crit ~ pred1 + pred2)))
Analysis of Variance Table
Model 1: crit ~ pred1
Model 2: crit ~ pred1 + pred2

  Res.Df   RSS Df Sum of Sq    F    Pr(>F)  
1     98 4546.9
2     97 3905.8  1   641.1 15.921 0.0001282 ***
```

In der Ausgabe von `anova()` zum Vergleich der Modelle wird in der Spalte `RSS` die Quadratsumme der Fehler (Residual Sum of Squares) für jedes Modell aufgeführt, wobei diejenige des umfassenderen Modells notwendigerweise geringer ist. Die Reduktion an Fehlervarianz, also die partielle Quadratsumme als Differenz der

beiden RSS-Werte findet sich in der Spalte `Sum of Sq`, daneben der partielle F -Wert (Spalte `F`) samt des zugehörigen p -Wertes (Spalte `Pr(>F)`).

Der Wert des F -Bruchs ergibt sich als Quotient der partiellen Quadratsumme geteilt durch ihre Freiheitsgrade (Spalte `Df`) und der Fehler-Quadratsumme des umfassenderen Modells geteilt durch ihre Freiheitsgrade (Spalte `Res.Df`). Die Freiheitsgrade eines Modells berechnen sich als Differenz zwischen der Stichprobengröße und der Anzahl zu schätzender Parameter. Hier sind im umfassenderen Modell mit zwei Regressionsgewichten und dem absoluten Term insgesamt drei Parameter zu schätzen, im eingeschränkten Modell entsprechend zwei. Die Freiheitsgrade der partiellen Quadratsumme berechnen sich aus der Differenz zwischen der Anzahl der Freiheitsgrade des eingeschränkten und des umfassenderen Modells, wofür sich hier 1 ergibt. Da der F -Bruch einen Zähler-Freiheitsgrad hat, ist er auch das Quadrat des t -Wertes aus dem Test des zugehörigen Regressionsgewichts auf Unterschiedlichkeit von 0.

```
> dfMod1 <- nSubj - (1+1)           # Freiheitsgrade eingeschränktes Modell
> dfMod2 <- nSubj - (2+1)           # Freiheitsgrade umfassenderes Modell

# Mittlere Quadratsumme des Effekts
> MSMod2  <- anovaRes[["Sum of Sq"]][2] / (dfMod1 - dfMod2)
> MSE      <- sum(residuals(sumRes)^2 / dfMod2)    # mittlere Fehler-QS
> MSMod2 / MSE                      # F-Bruch
[1] 15.92144

> (sumRes$coefficients[3, 3])^2      # Quadrat des t-Wertes von pred2
[1] 15.92144
```

8.3.6 Einzelvergleiche (Kontraste)

Ein Kontrast ψ meint im folgenden eine Linearkombination $\sum c_j \mu_j$ der Gruppenerwartungswerte μ_j mit den Koeffizienten c_j , wobei $1 \leq j \leq p$ sowie $\sum c_j = 0$ gilt. Solche Kontraste dienen einem spezifischen Vergleich zwischen den p experimentellen Bedingungen. Über die Größe von ψ lassen sich Hypothesen aufstellen, deren Test im folgenden beschrieben wird.

8.3.6.1 Beliebige a-priori Kontraste

Zum Testen beliebiger Kontraste stellt das Basispaket von R keine spezialisierten Funktionen bereit, jedoch lässt sich der Test zum einen manuell durchführen, zum anderen komfortabler mit Hilfe der `glht()` Funktion des `multcomp` Pakets (Hothorn et al., 2008). In beiden Fällen wird dafür zunächst der sog. Kontrastvektor c aus den Koeffizienten der Linearkombination der Erwartungswerte in den einzelnen Gruppen gebildet.

```
> glht((aov-Modell), linfct=mcp((UV)=(Kontrastkoeffizienten)),
+       alternative=c("two.sided", "less", "greater"))
```

Als erstes Argument ist ein mit `aov()` erstelltes Modell zu übergeben, dessen Gruppen einem spezifischen Vergleich unterzogen werden sollen. Dieser Vergleich kann mit dem `linfct` Argument definiert werden, was auf verschiedenen Wegen möglich ist. Einer davon verwendet seinerseits die `mcp()` Funktion, die eine Zuweisung der Form $\langle\text{UV}\rangle=\langle\text{Kontrastkoeffizienten}\rangle$ als Argument erwartet. Dabei ist $\langle\text{UV}\rangle$ der Name der UV aus dem mit `aov()` erstellten Modell (ohne Anführungszeichen), $\langle\text{Kontrastkoeffizienten}\rangle$ eine zeilenweise aus (ggf. benannten) Kontrastvektoren zusammengestellte Matrix. Mit `alternative` wird festgelegt, ob die Alternativhypothese gerichtet oder ungerichtet (`two.sided`) ist. "less" und "greater" beziehen sich dabei auf die Reihenfolge $\psi < 0$ bzw. $\psi > 0$ – der Größe ψ_0 des Kontrasts unter der Nullhypothese.

Das Ergebnis kann mit `summary(glht-Modell, test=adjusted("alpha-Adjustierung"))` auf Signifikanz getestet werden. Das Argument `test` erlaubt für das Testen von mehreren Kontrasten gleichzeitig, eine Methode zur α -Adjustierung auszuwählen (vgl. `?summary.glht`) – soll diese unterbleiben, ist `test=adjusted("none")` zu setzen.

Im Beispiel soll ein CR-*p* Design mit drei Gruppen vorliegen, wofür der Datensatz `dfCrop` herangezogen wird (vgl. Abschn. 8.3.2). Dabei ist zu beachten, dass die Reihenfolge der Gruppen durch die Reihenfolge der Faktorstufen bestimmt wird (vgl. Abschn. 2.7.4). Getestet werden soll zunächst nur der Vergleich der dritten Gruppe mit dem Mittel der ersten beiden Gruppen ($c = (1/2, 1/2, -1)$). Unter der Nullhypothese soll der Kontrast gleich 0 sein, unter der Alternativhypothese größer als 0.

```
> library(multcomp)                                     # für glht()
> aovCRp <- aov(yield ~ month, data=dfCrop)          # aov-Modell

# Matrix der Kontrastkoeffizienten - hier nur eine Zeile
> cntrMat <- rbind("0.5*(Jan+Feb)-Mar"=c(1/2, 1/2, -1))
> summary(glht(aovCRp, linfct=mcp(month=cntrMat), alternative="greater"))
Simultaneous Tests for General Linear Hypotheses
Multiple Comparisons of Means: User-defined Contrasts
Fit: aov(formula = yield ~ month, data = dfCrop)
Linear Hypotheses:
Estimate Std. Error t value Pr(>t)
0.5*(Jan+Feb)-Mar <= 0     233.0      101.2    2.302 0.0234 *
---
(Adjusted p values reported -- single-step method)
```

Die Ausgabe führt den Namen des getesteten Kontrasts aus der Matrix der Kontrastkoeffizienten auf, gefolgt von der Schätzung $\hat{\psi}$, für die die Erwartungswerte in der Linearkombination durch die Gruppenmittelwerte ersetzt werden ($\hat{\psi} = \sum c_j M_j$). Es folgt der Standardfehler dieser Schätzung (Std. Error), der Wert der *t*-Teststatistik (t value) und der zugehörige empirische *p*-Wert (Pr(>*t*)). Die zuletzt genannte gewählte Methode der α -Adjustierung (single-step) ist hier irrelevant, da nur ein Kontrast getestet wurde.

Die Differenz $\hat{\psi} - \psi_0$ zum unter der Nullhypothese erwarteten Wert für ψ (meist 0) bildet den Zähler der Teststatistik *t*. Für den Nenner wird die quadrierte Länge

$\|c\|^2$ des Kontrastvektors benötigt, für dessen Berechnung eine Gewichtung mit den Zellbesetzungen vorzunehmen ist. Das Produkt $\|c\|^2 \cdot MS_w$ der quadrierten Länge mit der mittleren Quadratsumme der Fehler aus der zugehörigen Varianzanalyse bildet den quadrierten Nenner der Teststatistik und stellt die Schätzung der Varianz des Kontrasts dar. Die Teststatistik ist im Fall von a-priori Kontrasten unter der Nullhypothese zentral t -verteilt mit den Freiheitsgraden der Quadratsumme der Fehler in der zugehörigen Varianzanalyse. Das α -Niveau betrage 0.05.

```
> anovaCRp <- anova(lm(yield ~ month, dfCrop))           # zugehörige ANOVA
> MSw       <- anovaCRp[["Mean Sq"]][2]                  # MS within
> P         <- nlevels(dfCrop$month)                      # Anzahl der Gruppen
> groupMs   <- tapply(dfCrop$yield, dfCrop$month, mean)    # Gruppen-Ms
> groupNs   <- tapply(dfCrop$yield, dfCrop$month, length)   # Gruppen-Ns
> dfSSw     <- sum(groupNs)-P                           # df von SS within
> psiHat    <- sum(cntrMat[1, ] * groupMs)            # Kontrastschätzung
> lenSq     <- sum((cntrMat[1, ]^2) * (1/groupNs))      # quadrierte Länge
> statTc    <- psiHat / sqrt(lenSq*MSw)                 # Teststatistik t
> abs(statTc)                                         # deren Betrag
[1] 2.302495

> (critT    <- qt(1-0.05, dfSSw))                     # kritischer t-Wert einseitig
[1] 1.833113

> (pValT   <- 1-pt(abs(statTc), dfSSw))             # p-Wert einseitig
[1] 0.02340396

# untere Grenze des einseitigen 95%-Vertrauensintervalls für psi
> (ciLo    <- psiHat - critT*sqrt(lenSq*MSw))
[1] 47.4989
```

Der a-priori Test fällt signifikant aus, wie einerseits am p -Wert, andererseits daran zu sehen ist, dass der Betrag der Teststatistik größer als der kritische t -Wert ist. Zudem enthält das einseitige Vertrauensintervall für ψ nicht ψ_0 , da die untere Grenze größer als 0 ist.

Sollen mehrere Kontraste gleichzeitig getestet werden, ist dies durch Zusammenstellung der zugehörigen Kontrastvektoren als (ggf. benannte) Zeilen einer Matrix möglich. Hier werden vier Kontraste mit `glht()` aus dem `multcomp` Paket zunächst ohne Adjustierung des α -Niveaus gerichtet getestet, dann manuell mit α -Adjustierung nach Bonferroni.

```
> cntrMat <- rbind( "Jan-0.5*(Feb+Mar)" = c( 1, -1/2, -1/2),
+                      "Feb-Jan"        = c( -1, 1, 0),
+                      "0.5*(Jan+Feb)-Mar" = c(1/2, 1/2, -1),
+                      "Feb-Mar"        = c( 0, 1, -1))

> summary(glht(aovCRp, linfct=mcp(month=cntrMat),
+               alternative="greater"), test=adjusted("none"))
Simultaneous Tests for General Linear Hypotheses
Multiple Comparisons of Means: User-defined Contrasts
Fit: aov(formula = yield ~ month, data = dfCrop)
```

```

Linear Hypotheses:
Estimate Std. Error t value Pr(>t)
Jan-0.5*(Feb+Mar) <= 0    217.0     101.2   2.144 0.0303 *
Feb-Jan <= 0           -134.0     116.8  -1.147 0.8595
0.5*(Jan+Feb)-Mar <= 0   233.0     101.2   2.302 0.0234 *
Feb-Mar <= 0            166.0     116.8   1.421 0.0946 .
---
(Adjusted p values reported -- none method)

# manuelle Berechnung mit alpha-Adjustierung nach Bonferroni
> (psiHats <- cntrMat %*% groupMs)                      # Kontrastschätzungen
[,1]
Jan-0.5*(Feb+Mar)  217
Feb-Jan          -134
0.5*(Jan+Feb)-Mar 233
Feb-Mar           166

> alphaAdj <- 0.05/nrow(cntrMat)                         # Bonferroni Adjustierung
> lenSqs   <- cntrMat^2 %*% (1/groupNs)                  # quadrierte Längen
> statTcs  <- psiHats / sqrt(lenSqs*MSw)                 # empirische Teststat.

# kritische adjustierte t-Werte
> critTs   <- rep(qt(1-alphaAdj, dfSSw), nrow(cntrMat))
> pValTs   <- 1-pt(abs(statTcs), dfSSw)                  # p-Werte einseitige Tests
> (resDf   <- data.frame(statTcs, critTs, pVals=pValTs,
+      alphaAdj=rep(alphaAdj, nrow(cntrMat)), sig=abs(statTcs)>critTs))
  statTcs critTs pVals alphaAdj sig
Jan-0.5*(Feb+Mar)  2.144384 2.685011 0.03029387 0.0125 FALSE
Feb-Jan           -1.146775 2.685011 0.14052348 0.0125 FALSE
0.5*(Jan+Feb)-Mar  2.302495 2.685011 0.02340396 0.0125 FALSE
Feb-Mar            1.420632 2.685011 0.09456422 0.0125 FALSE

```

Im abschließend ausgegebenen Datensatz wird für jeden Kontrast der empirische Wert der Teststatistik, der (für alle identische) kritische *t*-Wert, der jeweils zugehörige *p*-Wert, das adjustierte α sowie schließlich das Ergebnis der Signifikanzprüfung im Sinne des Vergleichs von *p*-Wert und adjustiertem α aufgeführt.

8.3.6.2 Beliebige post-hoc Kontraste nach Scheffé

Beliebige Kontraste im Sinne von Linearkombinationen von Gruppenerwartungswerten können auch im Anschluss an eine signifikante Varianzanalyse getestet werden. Die Varianzanalyse prüft implizit simultan alle möglichen Kontraste – spezifische Hypothesen liegen also bei ihrer Anwendung nicht vor. Aus diesem Grund muss im Anschluss an eine Varianzanalyse bei Einzeltests eine geeignete α -Adjustierung vorgenommen werden, hier vorgestellt nach der Methode von Scheffé.

Zunächst gilt für das Aufstellen eines Kontrastes alles bereits für beliebige a-priori Kontraste Ausgeführte. Lediglich die Wahl des kritischen Wertes weicht ab und ergibt sich zur α -Adjustierung aus einer *F*-Verteilung. Dieser kritische Wert ist mit dem Quadrat der a-priori Teststatistik zu vergleichen – die etwa in der Ausgabe

von `summary(glht())` abgelesen werden kann. Hier soll wieder die dritte Gruppe gegen das Mittel der ersten beiden gerichtet getestet werden.

```
> dfSSb    <- P-1                                # df von SS between
> (statF  <- psiHat^2 / (lenSq*MSw))           # quadrierte Teststatistik
[1] 5.301482

# kritischer F-Wert für quadrierte Teststatistik
> (critF <- dfSSb * qf(1-0.05, df1=dfSSb, df2=dfSSw))
[1] 8.51299

> (pValF <- 1-pf(statF/dfSSb, dfSSb, dfSSw))      # p-Wert einseitig
[1] 0.1244161
```

Der a-priori signifikant getestete Kontrast ist nun aufgrund der konservativeren Testung nicht mehr signifikant. Während der empirische Wert der Teststatistik derselbe wie im a-priori Fall ist, erhöht sich der kritische Wert.

Die Wahl des kritischen Wertes erfolgte hier so, dass alle möglichen Kontraste zugelassen sind. Sollen sich die Kontraste dagegen nur auf einen Teil der Erwartungswerte beziehen (und damit aus einem Unterraum des $(p - 1)$ -dimensionalen Kontrastraumes stammen), kann der kritische Wert entsprechend anders gewählt werden. In diesem Fall erfolgt keine gleichzeitige α -Adjustierung für alle möglichen Kontraste, sondern nur für Kontraste aus dem gewählten Unterraum, woraus ein etwas geringerer kritischer Wert resultiert. Ist q mit $q < p$ die Anzahl der relevanten Gruppen, wäre der kritische F -Wert $(q-1) * qf(1-0.05, df1=q-1, df2=dfSSw)$.

8.3.6.3 Paarvergleiche mit t -Tests und α -Adjustierung

Taucht nach einer signifikanten Varianzanalyse die spezifischere Frage auf, welche paarweisen Gruppenunterschiede vorliegen, besteht eine andere Herangehensweise in der Anwendung mehrerer t -Tests, um die Unterschiede jeweils zweier Gruppen auf Signifikanz zu prüfen. Das α -Niveau ist zu adjustieren, da sonst mehrfach die Möglichkeit bestünde, denselben Fehler erster Art zu begehen und das tatsächliche α -Niveau über dem nominellen läge. Anstatt mehrere solcher t -Tests manuell durchzuführen steht die `pairwise.t.test()` Funktion zur Verfügung, die alle möglichen Paarvergleiche testet und verschiedene Korrekturverfahren zur α -Adjustierung anbietet.

```
> pairwise.t.test(x=<Vektor>, g=<Faktor>, p.adjust.method="holm",
+                   paired=FALSE, pool.sd=!paired, alternative="two.sided")
```

Der Vektor `x` muss die Daten enthalten, `g` ist der zugehörige Faktor derselben Länge wie `x`, der für jedes Element von `x` codiert, in welcher Bedingung der Wert erhoben wurde. Über das Argument `p.adjust.method` wird die α -Adjustierung gesteuert, als Methoden stehen u. a. jene nach Holm (Voreinstellung) und Bonferroni zur Auswahl, für weitere vgl. `?p.adjust`. Wird bei als gegeben angesehener Varianzhomogenität `pool.sd=TRUE` gesetzt, wird die Fehlerstreuung auf Basis aller, also nicht nur anhand der beiden beim jeweiligen t -Test beteiligten Gruppen bestimmt. Dies

ist jedoch nur möglich, wenn keine abhängigen Stichproben vorliegen, was mit dem Argument paired anzugeben ist. Gerichtete Tests sind möglich, wenn das Argument alternative auf "less" oder "greater" gesetzt wird – Voreinstellung ist "two.sided" für ungerichtete Tests.

```
# simulierte Daten von je 50 VPn in vier UV-Stufen
> nSubj <- 50                                     # Gruppengröße
> DVa     <- rnorm(nSubj, 0, 1)
> DVb     <- rnorm(nSubj, 0.3, 1)
> DVC     <- rnorm(nSubj, 0.6, 1)
> DVd     <- rnorm(nSubj, 1.0, 1)
> DV      <- c(DVa, DVb, DVC, DVd)             # kombinierte AV-Daten
> IV      <- factor(rep(LETTERS[1:4], each=nSubj)) # Gruppenzugehörigkeit
> anova(lm(DV ~ IV))                            # Varianzanalyse

Analysis of Variance Table

Response: DV
Df   Sum Sq Mean Sq F value    Pr(>F)
IV       3   20.808  6.9359  5.9311  0.0006822 ***
Residuals 196 229.203  1.1694

> pairwise.t.test(DV, IV, p.adjust.method="bonferroni") # Einzelvergleiche
Pairwise comparisons using t tests with pooled SD
data: DV and IV
  A      B      C
B 0.1531 -     -
C 0.0029 0.1505 -
D 9.3e-05 0.201  0.3565
P value adjustment method: bonferroni
```

Die in Form einer Matrix ausgegebenen *p*-Werte für den Vergleich der jeweils in Zeile und Spalte angegebenen Gruppen berücksichtigen bereits die α -Adjustierung, sie können also direkt mit dem gewählten Signifikanzniveau verglichen werden. Setzt man p.adjust.method="none", unterbleibt eine α -Adjustierung, was zusammen mit pool.sd=FALSE zu jenen Ergebnissen führt, die man mit a-priori durchgeführten zweiseitigen *t*-Tests ohne α -Adjustierung erhalten hätte:

```
> pairwise.t.test(DV, IV, p.adjust.method="none", pool.sd=FALSE)      # ...
> t.test(DV[IV %in% c("A", "B")] ~ IV[IV %in% c("A", "B")])      # ...
> t.test(DV[IV %in% c("A", "C")] ~ IV[IV %in% c("A", "C")])      # ...
> t.test(DV[IV %in% c("A", "D")] ~ IV[IV %in% c("A", "D")])      # ...
> t.test(DV[IV %in% c("B", "C")] ~ IV[IV %in% c("B", "C")])      # ...
> t.test(DV[IV %in% c("B", "D")] ~ IV[IV %in% c("B", "D")])      # ...
> t.test(DV[IV %in% c("C", "D")] ~ IV[IV %in% c("C", "D")])      # ...
```

8.3.6.4 Simultane Konfidenzintervalle nach Tukey

Die Konstruktion simultaner Vertrauensintervalle für die Erwartungswertdifferenz bei paarweisen Gruppenvergleichen nach Tukey (Tukey Honestly Significant

Differences) stellt eine weitere Möglichkeit dar, im Anschluss an eine signifikante Varianzanalyse nach Unterschieden zwischen jeweils zwei Gruppen zu suchen. Diese Methode ist bei einer größeren Zahl von Vergleichen weniger konservativ als die Herangehensweise mit mehrfachen *t*-Tests und α -Adjustierung.

```
> TukeyHSD(x=(aov-Objekt), which, ordered=FALSE, conf.level=0.95)
```

Als Argument *x* erwartet *TukeyHSD()* ein von *aov()* erstelltes Objekt. Wurde in *x* mehr als ein Faktor berücksichtigt, kann mit *which* in Form eines Vektors aus Zeichenketten angegeben werden, welche dieser Faktoren für die Bildung von Gruppen herangezogen werden sollen. Ist gewünscht, die Gruppen entsprechend der Größe der zugehörigen empirischen Mittelwerte in eine Reihenfolge zu bringen, ehe sie miteinander verglichen werden, ist das Argument *ordered=TRUE* zu setzen. Über *conf.level* wird die Breite des Vertrauensintervalls für die jeweilige Differenz zweier Erwartungswerte festgelegt.

Im Beispiel mit denselben Daten wie bei *pairwise.t.test()* ist zu erkennen, dass die *p*-Werte teilweise niedriger als dort sind – bei den gegebenen Daten wäre es also bereits günstiger, Paarvergleiche nach Tukey statt *t*-Tests mit einer α -Adjustierung nach Bonferroni verwenden.

```
> (tHSD <- TukeyHSD(aov(DV ~ IV)))
```

```
Tukey multiple comparisons of means
95% family-wise confidence level
Fit: aov(formula = DV ~ IV)
```

\$IV					
	diff	lwr	upr	p	adj
B-A	0.3353342	-0.22508656	0.8957549	0.4095158	
C-A	0.6615085	0.10108783	1.2219293	0.0134128	
D-A	0.8452367	0.28481602	1.4056574	0.0007343	
C-B	0.3261744	-0.23424632	0.8865951	0.4345788	
D-B	0.5099026	-0.05051813	1.0703233	0.0888762	
D-C	0.1837282	-0.37669253	0.7441489	0.8306335	

Die Ausgabe umfasst für jeden Gruppenvergleich die empirische Mittelwertsdifferenz, ihr Vertrauensintervall sowie den zugehörigen *p*-Wert. Die Ergebnisse lassen sich auch manuell nachvollziehen, wobei hier zu beachten ist, dass gleiche Gruppengrößen vorliegen. Kritischer Wert und Verteilungsfunktion der Teststatistik sind mit *qtukey()* bzw. *ptukey()* zu berechnen.

```
> anovaRes <- anova(lm(DV ~ IV))          # Varianzanalyse
> MSE <- anovaRes[["Mean Sq"]][2]           # MS innerhalb
> Ms  <- tapply(DV, IV, mean)                # Gruppenmittelwerte
> P   <- nlevels(IV)                         # Anzahl Gruppen
> dfE <- (nSubj*P)-P                         # Freiheitsgrade SS innerhalb

# alle paarweisen Mittelwertsdifferenzen
> diffMat <- outer(Ms, Ms, "-")
> (diffs <- diffMat[lower.tri(diffMat)])
[1] 0.3353342 0.6615085 0.8452367 0.3261744 0.5099026 0.1837282
```

```

> qTs      <- abs(diffs) / sqrt(MSE / nSubj)    # Teststatistiken
> (pVals  <- 1-ptukey(qTs, P, dfE))           # p-Werte
[1] 0.4095158213 0.0134128212 0.0007343386 0.4345787687
[5] 0.0888762477 0.8306334843

# halbe Breite Vertrauensintervall für paarweise Mittelwertsdifferenz
> tWidth <- qtukey(1-0.05, P, dfE) * sqrt(MSE / nSubj)
> diffs - tWidth      # Vertrauensintervalle untere Grenzen
[1] -0.22508656 0.10108783 0.28481602 -0.23424632 -0.05051813 -0.37669253

> diffs + tWidth      # Vertrauensintervalle obere Grenzen
[1] 0.8957549 1.2219293 1.4056574 0.8865951 1.0703233 0.7441489

```

Die Konfidenzintervalle können graphisch veranschaulicht werden, indem das Ergebnis von TukeyHSD() einem Objekt zugewiesen und dies an die plot() Funktion übergeben wird (Abb. 8.3). Die gestrichelt gezeichnete senkrechte Linie markiert die 0 – Intervalle, die die 0 enthalten, entsprechen einem nicht signifikanten Paarvergleich.

```
> plot(tHSD)
```

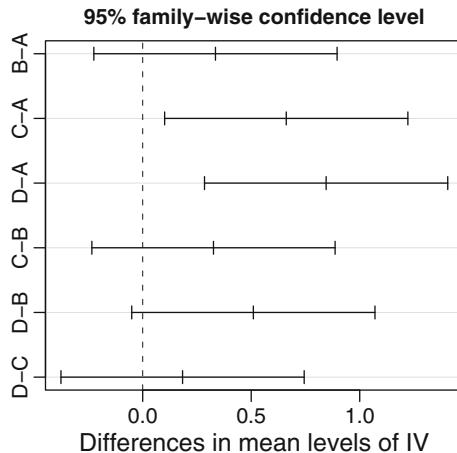


Abb. 8.3 Graphische Darstellung simultaner Vertrauensintervalle nach Tukey

8.4 Einfaktorielle Varianzanalyse mit abhängigen Gruppen (RB-*p*)

In einem RB-*p* Design werden in *p* Bedingungen einer UV abhängige Beobachtungen gemacht. Diese Situation verallgemeinert jene eines *t*-Tests für zwei abhängige Stichproben auf mehr als zwei Gruppen, d. h. auf jeweils mehr als zwei voneinander abhängige Beobachtungen.

8.4.1 Univariat formulierte Auswertung mit `aov()`

Jede Menge aus p abhängigen Beobachtungen (eine aus jeder Bedingung) wird als Block bezeichnet. Versuchsplanerisch ist zu beachten, dass im Fall der Messwiederholung die Reihenfolge der Beobachtungen für jede VP randomisiert werden muss, weswegen das Design unter Randomized Block Design firmiert. Bei gematchten VPn ist zunächst die Blockbildung so vorzunehmen, dass jeder Block p VPn umfasst, die bzgl. relevanter Störvariablen homogen sind. Innerhalb jedes Blocks müssen daraufhin die VPn randomisiert den Bedingungen zugeordnet werden.

Im Vergleich zum CR- p Design wirkt im Modell zum RB- p Design ein systematischer Effekt mehr am Zustandekommen einer Beobachtung mit: zusätzlich zum Effekt der Zugehörigkeit zu einer Bedingung ist dies der Blockeffekt aus der Zugehörigkeit einer Beobachtung zu einem Block. Im Gegensatz zum sog. festen (Fixed) Gruppenfaktor stellt die Blockzugehörigkeit einen sog. zufälligen (Random) Faktor dar. Die Bezeichnungen erscheinen vor dem Hintergrund plausibel, dass die Faktorstufen inhaltlich bedeutsam sind, durch den Versuchsleiter als UV im versuchsplanerischen Sinn willentlich festgelegt werden und eine Generalisierung der Ergebnisse über diese Stufen hinaus nicht erfolgt. Die vorliegenden Ausprägungen der Blockzugehörigkeit (im Fall der Messwiederholung die VPn selbst) sind dagegen selbst nicht bedeutsam. Stattdessen sind sie vom Versuchsleiter nicht kontrollierte Realisierungen einer Zufallsvariable im statistischen Sinn. Dies erlaubt es den Ergebnissen, über die tatsächlich realisierten Werte hinaus Gültigkeit beanspruchen können.

Für die Durchführung der Varianzanalyse ergeben sich wichtige Änderungen im Vergleich zum CR- p Design. Zunächst sind die statistischen Voraussetzungen andere: die jeweils pro Block zu erhebenden Daten müssen, als Zufallsvektoren aufgefasst, gemeinsam normalverteilt sein. Darüber hinaus muss Zirkularität gelten (vgl. Abschn. 8.4.2). Für die Durchführung des Tests mit `aov()` muss der Datensatz so strukturiert sein, dass jede Zeile nicht die Werte eines einzelnen Blocks zu allen Messzeitpunkten beinhaltet (Wide-Format), sondern auch die Messwerte eines Blocks aus verschiedenen Messzeitpunkten in separaten Zeilen stehen (Long-Format, vgl. Abschn. 3.2.8). Der Messzeitpunkt muss mit einem Objekt der Klasse `factor` codiert werden. Weiterhin muss es eine Variable geben, die codiert, von welcher VP, bzw. allgemeiner aus welchem Block ein Wert der AV stammt. Diese Variable sei im folgenden als `(BlockNr)` bezeichnet und muss ein Objekt der Klasse `factor` sein.⁵

Weil ein Random-Faktor vorhanden ist, ändert sich die Formel in den Aufrufen von `lm()` oder `aov()` dahingehend, dass explizit anzugeben ist, aus welchen additiven Komponenten sich die Quadratsumme innerhalb der Gruppen zusammensetzt. Im konkreten Fall drückt `Error((BlockNr)/(UV))` aus, dass `(BlockNr)` in `(UV)`

⁵ Bei Verwendung von `reshape()` werden sowohl die Blockzugehörigkeit als auch der Messzeitpunkt als numerischer Vektor codiert. Beide Variablen sind dementsprechend manuell in einen Faktor umzuwandeln.

verschachtelt (nested) ist, weil nicht alle möglichen Blöcke in jeder Stufe der UV auch experimentell realisiert sind, sondern nur eine Zufallsauswahl.⁶ Der durch die Variation der UV entstehende Effekt in der AV ist innerhalb der durch `<BlockNr>` definierten Blöcke zu analysieren.

```
> aov(<AV> ~ <UV> + Error(<BlockNr>/<UV>), data=<Datensatz>)

> nSubj    <- 10                                # Zellbesetzung
> P        <- 3                                  # Anzahl UV-Stufen
> id       <- factor(rep(1:nSubj, P))           # Blockzugehörigkeit
> timeFac <- factor(rep(1:P, each=nSubj))       # Intra-Gruppen Faktor
> DV_t1   <- round(rnorm(nSubj, -0.3, 1), 2)    # Simulation AV zu t1
> DV_t2   <- round(rnorm(nSubj, -0.2, 1), 2)    # Simulation AV zu t2
> DV_t3   <- round(rnorm(nSubj, 0.5, 1), 2)    # Simulation AV zu t3

# Datensatz im Long-Format und Varianzanalyse mit aov()
> dfRBpL <- data.frame(id, timeFac, DV=c(DV_t1, DV_t2, DV_t3))
> aovRBp  <- aov(DV ~ timeFac + Error(id/timeFac), data=dfRBpL)
> summary(aovRBp)

Error: id
      Df  Sum Sq Mean Sq F value Pr(>F)
Residuals  9  9.2086  1.0232

Error: id:timeFac
      Df  Sum Sq Mean Sq F value Pr(>F)
timeFac     2  6.3022  3.15108  5.1448  0.01709 *
Residuals 18 11.0246  0.61248
```

Die beim Test eines Effekts jeweils verwendete Quelle der Fehlervarianz wird durch die `Error: <Effekt>` Überschriften kenntlich gemacht, ihre Quadratsumme findet sich in der Zeile `Residuals`. Es ist zu erkennen, dass im RB-*p* Design die Quadratsumme des festen Effekts (`timeFac`) gegen die Quadratsumme der Interaktion von festem und Random-Faktor getestet wird (`Error: id:timeFac`). Dies wird auch deutlich, wenn die Daten eines RB-*p* Designs mit einer zweifaktoriellen Varianzanalyse im CRF-*pq* Design analysiert werden (vgl. Abschn. 8.5), wobei der Random und der feste Faktor jeweils die Rolle einer UV einnehmen:

```
> anova(lm(DV ~ id + timeFac + id:timeFac, data=dfRBpL))
Analysis of Variance Table
      Df  Sum Sq Mean Sq F value Pr(>F)
id       9  9.2086  1.02318
timeFac  2  6.3022  3.15108
id:timeFac 18 11.0246  0.61248
Residuals  0  0.0000
```

⁶ Ausgeschrieben lautet der Term `Error(<BlockNr>+<BlockNr>:<UV>)`. Dies sind die beiden Effekte, deren Quadratsummen sich zur Quadratsumme innerhalb der Gruppen addieren. Dabei stellt letztlich die Quadratsumme der Interaktion `<BlockNr>:<UV>` die Quadratsumme der Fehler dar, gegen die der feste UV-Effekt getestet wird.

Die sich in dieser Varianzanalyse ergebende Quadratsumme der Interaktion beider UVn (`id:timeFac`) ist identisch zu jener der Residuen beim Test des festen Effekts im RB-*p* Design. Analoges gilt für den Effekt der im CRF-*pq* Design als UV behandelten Variable `id`, dessen Quadratsumme der ersten Fehler-Quadratsumme im RB-*p* Design entspricht. Die Effekt-Quadratsumme von `timeFac` ist in beiden Varianzanalysen notwendigerweise identisch. Da aus jedem Block nur eine Beobachtung pro Stufe von `timeFac` vorliegt, beträgt im konstruierten CRF-*pq* Design die Zellbesetzung 1. Deshalb kann es keine Abweichungen zum Zellmittelwert geben (Residuals beträgt 0).

Die Ergebnisse lassen sich auch manuell prüfen, wobei nach dem Bilden der blockweise zentrierten Daten wie im CR-*p* Design vorgegangen werden kann, lediglich die Freiheitsgrade der Fehler unterscheiden sich.

```
# blockweise zentrierte Werte
> DVctr <- dfRBpL$DV - ave(dfRBpL$DV, dfRBpL$id, FUN=mean)
> tfMs <- tapply(DVctr, dfRBpL$timeFac, mean) # Messzeitpunkt-Mittel
> grandM <- mean(tfMs) # Gesamtmittel
> SSb <- sum(nSubj * (tfMs-grandM)^2) # Quadratsumme UV

# Interaktion (BlockNr):(UV) -> für Quadratsumme der Fehler
> IDxTF <- DVctr - ave(DVctr, dfRBpL$timeFac, FUN=mean) + grandM
> SSE <- sum(IDxTF^2) # Quadratsumme Fehler
> dfSSb <- P-1 # Freiheitsgrade UV
> dfSSE <- (nSubj-1) * (P-1) # Freiheitsgrade Fehler
> (MSb <- SSb / dfSSb) # Mittlere QS UV
[1] 3.151083

> (MSE <- SSE / dfSSE) # Mittlere QS Fehler
[1] 0.6124796

> (Fval <- MSb / MSE) # Teststatistik F-Wert
[1] 5.144797

> (pVal <- 1-pf(Fval, P-1, (nSubj-1)*(P-1))) # p-Wert
[1] 0.01709276
```

8.4.2 Zirkularität der Kovarianzmatrix prüfen

Die Gültigkeit der dargestellten Varianzanalyse für Daten aus einem RB-*p* Design hängt davon ab, ob neben den anderen Voraussetzungen auch die Annahme von Zirkularität der theoretischen Kovarianzmatrix der AV in den einzelnen Gruppen gilt. Sie ist gegeben, wenn alle aus je zwei unterschiedlichen Variablen gebildeten Differenzvariablen dieselbe theoretische Varianz besitzen.⁷

⁷ Dies ist bei nur zwei Gruppen immer der Fall. Als Test auf Zirkularität steht mit `mauchly.test()` jener nach Mauchly zur Verfügung, von dessen Verwendung aber aufgrund seiner geringen Power oft abgeraten wird. Ein Spezialfall zirkulärer Matrizen sind homogene Matrizen, bei denen alle

Für Situationen ohne gegebene Zirkularität existieren Korrekturformeln, die verhindern sollen, dass bei Durchführung einer Varianzanalyse die tatsächliche Wahrscheinlichkeit eines Fehlers erster Art höher als das nominelle α -Niveau ist. Ist p die Anzahl der Gruppen, besteht die Korrektur in der Multiplikation der Zähler- und Nenner-Freiheitsgrade des in der Varianzanalyse getesteten F -Bruchs mit einem Skalar ε , für den $1/(p - 1) \leq \varepsilon \leq 1$ gilt. ε ist genau dann 1, wenn die theoretische Kovarianzmatrix zirkulär ist, andernfalls schwankt ε in Abhängigkeit vom „Ausmaß der Abweichung von Zirkularität“. Zur Schätzung von ε sind u. a. die Methoden von Box (auch als Greenhouse-Geisser-Korrektur bezeichnet) sowie von Huynh und Feldt anwendbar. Es folgt eine manuelle Berechnung dieser Schätzungen, vgl. Abschn. 8.4.3 für die automatische Berechnung durch Anova().

```
# Schätzung von epsilon nach Greenhouse + Geisser
> dataMat <- cbind(DV_t1, DV_t2, DV_t3) # Daten im Wide-Format
> nSubj <- nrow(dataMat) # Anzahl der Beobachtungen
> S <- var(dataMat) # empirische Kovarianzmatrix
> P <- nrow(S) # Anzahl Variablen (= der UV-Stufen)
> mdS <- mean(diag(S)) # Mittelwert der Diagonalelemente von S
> mS <- mean(S) # Mittelwert aller Elemente von S
> mSr <- rowMeans(S) # Zeilenmittelwerte von S
> N <- P^2 * (mdS-mS)^2 # Zähler
> D <- (P-1)*(sum(S^2) - 2*P*sum(mSr^2) + P^2*mS^2) # Nenner
> (epsGG <- N/D)
[1] 0.9411227

# Schätzung von epsilon nach Huynh + Feldt
> (epsHF <- (nSubj*(P-1)*epsGG-2) / ((P-1)*((nSubj-1)-((P-1)*epsGG))))
[1] 1.181725

# setze epsilon nach Hyhn + Feldt auf 1, wenn es größer ist
> epsHF <- ifelse(epsHF > 1, 1, epsHF)

# Multiplikation der Freiheitsgrade mit epsilon und Vergleich der p-Werte
> dfEpsGG <- c(dfSSb, dfSSE) * epsGG # Greenhouse + Geisser
> (pEpsGG <- 1-pf(Fval, df1=dfEpsGG[1], df2=dfEpsGG[2]))
[1] 0.01934478

> dfEpsHF <- c(dfSSb, dfSSE) * epsHF # Huynh + Feldt
> (pEpsHF <- 1-pf(Fval, df1=dfEpsHF[1], df2=dfEpsHF[2]))
[1] 0.01709276
```

Im Beispiel bringt die Korrektur der Freiheitsgrade einen größeren p -Wert mit sich, wobei dies insbesondere für kleinere empirische F -Werte nicht in jeder Situation

Varianzen (die Diagonale der Matrix) übereinstimmen und auch alle Kovarianzen identisch sind. Bei nicht bestehender Zirkularität ist eine multivariate Herangehensweise (vgl. Abschn. 8.4.3 und 9.6) oder ein univariater Zugang nach dem Linear Mixed Effects Modell (Pinheiro und Bates, 2000) mit den Paketen lme4 (Bates et al., 2009) und nlme (Pinheiro et al., 2009) u. U. geeigneter. Im letztgenannten Fall wird die Korrelationsstruktur zwischen den abhängigen Messungen explizit mit modelliert.

der Fall sein muss. Die Korrektur nach Greenhouse und Geisser gilt in Bereichen oberhalb von 0.9 als etwas zu konservativ. Dies ist nicht der Fall für die Korrektur nach Huynh und Feldt, deren Schätzung aber auch wie hier zu Werten größer als 1 führen kann. In diesen Fällen sollte die Schätzung auf 1 gesetzt werden. Soll dagegen konservativ getestet werden, ist die Schätzung für ε pauschal auf das theoretische Minimum $1/(p - 1)$ zu setzen.

Im vorangehenden Abschnitt wurde bereits die Quadratsumme der Interaktion $\langle \text{BlockNr} \rangle : \langle \text{UV} \rangle$ manuell berechnet, die als Quadratsumme der Fehler beim Test des festen Effekts dient. Hier kann zur Schätzung von ε nach Greenhouse und Geisser deshalb auch die etwas einfachere Formel verwendet werden, die auf der Kovarianzmatrix der Fehler basiert.

```
# dem Datensatz Interaktion als neue Variable hinzufügen
> dfRBpL$IDxTF <- IDxTF

# Fehler als Matrix im Wide-Format extrahieren
> errMat <- as.matrix(unstack(dfRBpL, IDxTF ~ timeFac))
> Serr <- cov(errMat) # Kovarianzmatrix der Fehler
> (epsGG <- (1/(P-1)) * sum(diag(Serr))^2 / sum(Serr^2))
[1] 0.9411227
```

8.4.3 Multivariat formulierte Auswertung mit Anova()

Die Anova() Funktion aus dem car Paket erlaubt die Durchführung von Varianzanalysen mit Messwiederholung, wobei sowohl die bereits beschriebene univariate wie auch eine multivariate Auswertung möglich ist. Bei der multivariaten Varianzanalyse (vgl. Abschn. 9.6) werden dabei die Messwerte jeweils eines Blocks (z. B. die Werte einer VP zu unterschiedlichen Messzeitpunkten) in einem Datenvektor zusammengefasst. Die Analyse zielt auf den Effekt der manipulierten UV auf das Muster der Datenvektoren ab.

Ein Vorteil der Verwendung von Anova() ist die Möglichkeit, mit einer leicht nachvollziehbaren Syntax direkt anzugeben, bzgl. welcher Faktoren eines Versuchsplans abhängige Messungen vorliegen. Zudem kann ein Datensatz im Wide-Format benutzt werden, so dass eine Konvertierung ins Long-Format entfällt. Weiterhin berechnet Anova() den Mauchly-Test auf Zirkularität und die Schätzungen des Parameters ε als Ausdruck für die Abweichung der Kovarianzmatrix von Zirkularität mit den Methoden von Greenhouse und Geisser sowie von Huynh und Feldt samt der korrigierten p -Werte.

```
> Anova(mod=lm(<Formel>), type=c("II", "III"), idata=<Datensatz>,
+       idesign=<Formel>)
```

Unter mod ist ein mit lm() erstelltes lineares Modell zu übergeben, das bei abhängigen Designs multivariat formuliert werden muss – statt einer einzelnen gibt es also mehrere AVn. In der Formel $\langle \text{AV} \rangle \sim \langle \text{UV} \rangle$ werden dazu auf der linken Seite der \sim die Daten der AV als Matrix zusammengefügt, wobei jede Spalte die Werte

der AV in einer Gruppe (z. B. zu einem Messzeitpunkt) beinhaltet. Auf der rechten Seite der \sim werden die Zwischen-Gruppen Faktoren aufgeführt, wenn es sich um ein gemischtes Design handelt (vgl. Abschn. 8.7). Liegt ein reines Intra-Gruppen Design vor, ist als UV nur die Konstante 1 anzugeben. Sind etwa die Werte einer an denselben VPn gemessenen AV zu drei Messzeitpunkten in den Variablen t1, t2 und t3 des Datensatzes myDf gespeichert, lautet das Modell `lm(cbind(t1, t2, t3) ~ 1, data=myDf)`.

Mit `type` kann der im Fall ungleicher Zellbesetzungen relevante Quadratsummen-Typ beim Test festgelegt werden (Typ II oder Typ III, vgl. Abschn. 8.5.2). Die Argumente `idata` und `idesign` dienen der Spezifizierung der Intra-Gruppen Faktoren. Hierfür erwartet `idata` einen Datensatz, der die Struktur der unter `mod` in einer Matrix zusammengefassten Daten der AV beschreibt. Im RB-*p* Design beinhaltet `idata` eine Variable der Klasse `factor`, die die Stufen der Intra-Gruppen UV codiert. In jeder Zeile von `idata` gibt dieser Faktor an, aus welcher Bedingung die zugehörige Spalte der Datenmatrix stammt. Im oben skizzierten Design mit drei Messzeitpunkten lautet das Argument damit `idata=data.frame(timeFac=factor(1:3))` – die erste Spalte der Datenmatrix entspricht der ersten Zeile von `idata`, also der Bedingung 1, usw. Unter `idesign` wird eine Formel mit den Intra-Gruppen Vorhersagetermen auf der rechten Seite der \sim angegeben, die linke Seite bleibt leer. Gibt es nur eine Intra-Gruppen UV `timeFac`, lautet das Argument also `idesign= ~ timeFac`.

```
> dfRBpW <- data.frame(DV_t1, DV_t2, DV_t3)           # Datensatz Wide-Format

# Zwischen-Gruppen Design (hier kein Zwischen-Gruppen Faktor)
> modelRBp <- lm(cbind(DV_t1, DV_t2, DV_t3) ~ 1, data=dfRBpW)
> intraRBp <- data.frame(timeFac=factor(1:P))        # Intra-Gruppen Design
> library(car)                                         # für Anova()
> (AnovaRBp <- Anova(modelRBp, idata=intraRBp, idesign=~ timeFac)) # ...
```

Der ausführliche Test des Modells inklusive der Schätzungen für ε erfolgt mit `summary()`. Als Besonderheit bei der Anwendung auf ein von `Anova()` erzeugtes Modell kann mit den Argumenten `univariate` und `multivariate` angegeben werden, ob die univariaten und multivariaten Kennwerte berechnet werden sollen.

```
> summary(AnovaRBp, multivariate=FALSE, univariate=TRUE)      # ...
```

Die unkorrigierten Ergebnisse der univariaten Auswertung stimmen mit den von `aov()` ermittelten überein, sofern alle Zellen dieselbe Anzahl an Beobachtungen aufweisen. Die Auswertung des RB-*p* Designs in multivariater Formulierung ist auch mit der bereits bekannten `anova()` Funktion möglich und identisch zur univariaten, sofern die Zirkularität der Kovarianzmatrix angenommen wird (Argument `test="Spherical"`).⁸ Wie bei `Anova()` muss das Modell multivariat formuliert und die AV-Struktur in Form des Datensatzes `idata` angegeben werden.

⁸ Bei der multivariaten Formulierung des Modells wird intern aufgrund der generischen `anova()` Funktion automatisch die `anova.mlm()` Funktion verwendet, ohne dass dies explizit angegeben

Die Angabe des Versuchsdesigns in `anova()` mit den Argumenten `M` und `X` benötigt Kenntnisse des Allgemeinen Linearen Modells. Auch bei `anova()` wird ε mit den Methoden von Greenhouse und Geisser sowie von Huynh und Feldt geschätzt und samt der korrigierten p -Werte ausgegeben. Für Einzelheiten vgl. `?anova.mlm`.

```
> anova(modelRBp, idata=intraRBp, X=~ 1, test="Spherical") # ...
```

8.4.4 Einzelvergleiche (Kontraste)

Einzelvergleiche nach dem Muster jener in Abschn. 8.3.6 für das CR- p Design beschriebenen sind im Rahmen einer Varianzanalyse von Daten eines RB- p Designs im Prinzip ebenfalls möglich. Dabei ist die Mittlere Quadratsumme der Interaktion von festem und Random-Effekt (`Error: (BlockNr):(UV)`) als Residualvarianz in der Rolle der Mittleren Quadratsumme innerhalb der Gruppen beim CR- p Design zu verwenden. So gebildete Einzelvergleiche gelten aber als anfällig gegenüber Verletzungen der Zirkularitäts-Voraussetzung. Im Gegensatz zum Test des Haupteffekts ist hier die Korrektur der Freiheitsgrade mit einer Schätzung von ε kein probates Mittel, um die Gefahr eines erhöhten tatsächlichen α -Fehlers zu verringern. Aus diesen Gründen wird auf ihre Darstellung hier verzichtet. Für eine weitergehende Diskussion mit alternativen Strategien zur Auswertung vgl. Kirk (1995) und Maxwell und Delaney (2004).

8.5 Zweifaktorielle Varianzanalyse (CRF- pq)

Bei zwei UVn bestehen verschiedene Möglichkeiten, diese versuchsplanerisch zu Kombinationen von Experimentalbedingungen zu verbinden. Hier sei der Fall betrachtet, dass alle Kombinationen von Bedingungen realisiert werden, es sich also um vollständig gekreuzte Faktoren handelt. Zudem sollen beide UVn Zwischen-Gruppen Faktoren darstellen, jede VP also in nur einer Bedingung beobachtet werden. Die Zuteilung von VPn auf Bedingungen erfolge randomisiert. In diesem Fall liegt ein sog. CRF- pq Design (Completely Randomized Factorial) mit p Stufen der ersten und q Stufen der zweiten UV vor.

Bei allen Varianzanalysen mit mehr als einem Faktor ist es wichtig, ob in jeder experimentellen Bedingung dieselbe Anzahl an Beobachtungen vorliegt. Ist dies nicht der Fall, handelt es sich um ein sog. unbalanciertes oder nicht-orthogonales Design.⁹ Dann lässt sich die Gesamt-Quadratsumme nicht mehr eindeutig in die

werden muss (vgl. Abschn. 11.1.5). Ohne das Argument `test="Spherical"` wird – mit entsprechend anderen Ergebnissen – multivariat getestet.

⁹ Die Nicht-Orthogonalität bezieht sich auf die Kontrastvektoren der einzelnen Effekte im durch die Versuchspersonen aufgespannten Raum: bei ungleichen Zellbesetzungen stehen sie dort nicht mehr senkrecht aufeinander.

Quadratsummen der einzelnen Effekte als additive Komponenten zerlegen (vgl. Abschn. 8.5.2). Im folgenden sei ein balanciertes Design vorausgesetzt.

8.5.1 Auswertung mit aov()

Für das CRF-*pq* Design erlaubt es das Modell der Varianzanalyse, drei Effekte zu testen: den der ersten sowie der zweiten UV und den Interaktionseffekt. Jeder dieser Effekte kann in die Formel im Aufruf von lm() oder aov() als modellierender Term eingehen. Hierbei sei daran erinnert, dass der Interaktionseffekt zweier Variablen in einer Formel durch den Doppelpunkt : symbolisiert wird.

```
> aov(<AV> ~ <UV1> + <UV2> + <UV1>:<UV2>), data=<Datensatz>)      # bzw.  
> aov(<AV> ~ <UV1> * <UV2>), data=<Datensatz>)
```

Im Beispiel soll neben den beiden eigentlichen UVn auch eine auf diese Bezug nehmende Variable in den Datensatz aufgenommen werden: sie ignoriert die Faktorstruktur und codiert die aus der Kombination beider UVn resultierenden Bedingungen als Ausprägungen einer einzelnen UV im Sinne der assoziierten einfaktoriellen Varianzanalyse (vgl. auch ? interaction).

```
> nSubj  <- 8                                # Zellbesetzung  
> P      <- 2                                # Anzahl Stufen UV1  
> Q      <- 3                                # Anzahl Stufen UV2  
> IV1   <- factor(rep(1:P, Q*nSubj))        # Ausprägungen UV1  
> IV2   <- factor(rep(1:Q, each=P*nSubj))    # Ausprägungen UV2  
  
# Kombination zu einem Faktor -> assoziierte einfaktorielle ANOVA  
> IVcomb <- factor(c(rep(c("1-1", "2-1"), nSubj),  
+                      rep(c("1-2", "2-2"), nSubj), rep(c("1-3", "2-3"), nSubj)))  
  
# Simulation der AV mit Effekt beider UVn, ohne Interaktion  
> DV          <- rnorm(P*Q*nSubj, 0, 1)        # Start-Werte  
> DV[IV1 == 1] <- DV[IV1 == 1] + 0.5          # Effekt UV1-1  
> DV[IV1 == 2] <- DV[IV1 == 2] - 0.5          # Effekt UV1-2  
> DV[IV2 == 1] <- DV[IV2 == 1] - 0.5          # Effekt UV2-1  
> DV[IV2 == 3] <- DV[IV2 == 3] + 0.5          # Effekt UV2-3  
  
# Datensatz und Varianzanalyse mit aov()  
> dfCRFpq <- data.frame(id=1:(P*Q*nSubj), IV1, IV2, IVcomb, DV)  
> summary(aov(DV ~ IV1*IV2, data=dfCRFpq))  
Df Sum Sq Mean Sq F value    Pr(>F)  
IV1       1 17.650  17.650 14.5870 0.0004349 ***  
IV2       2  1.899  0.949  0.7846 0.4628961  
IV1:IV2   2  3.742  1.871  1.5462 0.2249378  
Residuals 42 50.818  1.210
```

Die einer mehrfaktoriellen Varianzanalyse zugrundeliegende Struktur der Daten lässt sich deskriptiv zum einen in Form von Mittelwerts- bzw. Effekttabellen mit model.tables() darstellen, wobei die Zellbesetzungen mit aufgeführt werden.

Zum anderen kann die Datenlage aber auch graphisch über Mittelwertsdiagramme veranschaulicht werden. Neben der Möglichkeit, dies mit `plot.design()` zu tun, bietet sich auch `interaction.plot()` an. Das Muster der Mittelwertsverläufe wird häufig als heuristische Datenquelle für die Beurteilung herangezogen, auf welche Art einer ggf. vorliegenden statistischen Interaktion die Daten hindeuten.

```
> interaction.plot(x.factor=(UV1), trace.factor=(UV2), response=(AV),
+                   fun=mean, col="Linienstärke)
```

Unter `x.factor` wird jener Gruppierungsfaktor angegeben, dessen Ausprägungen auf der Abszisse abgetragen werden. `trace.factor` kontrolliert, welcher zusätzliche Faktor im Diagramm berücksichtigt wird, seine Stufen werden durch unterschiedliche Linien repräsentiert. Sind die Variablen Teil eines Datensatzes, muss dieser den Variablennamen in der Form `(Datensatz)$<Variable>` vorangestellt werden. Das Argument `response` erwartet die auf der Ordinate abzutragende AV. Soll nicht der Mittelwert, sondern ein anderer Kennwert pro Gruppe berechnet werden, akzeptiert das Argument `fun` auch andere Funktionsnamen als die Voreinstellung `mean`. Über `col` und `lwd` können Farbe und Stärke der Linien kontrolliert werden (Abb. 8.4).

```
> plot.design(DV ~ IV1*IV2, data=dfCRFpq, main="Gruppen-Mittelwerte")
> interaction.plot(IV1, IV2, DV, main="Mittelwertsverläufe",
+                   col=c("red", "blue", "green"), lwd=2)
```

Die varianzanalytische Auswertung eines Designs mit drei vollständig gekreuzten Zwischen-Gruppen Faktoren (CRF-*pqr*) unterscheidet sich nur wenig von der beschriebenen zweifaktoriellen Situation. Lediglich die Formel mit den zu berücksichtigenden Effekten ist entsprechend anzupassen. Sollen alle Haupteffekte, Interaktionseffekte zweiter Ordnung und der Interaktionseffekt dritter Ordnung eingehen, könnte die Formel etwa $\langle AV \rangle \sim \langle UV1 \rangle * \langle UV2 \rangle * \langle UV3 \rangle$ lauten. Um den Interaktionseffekt dritter Ordnung auszuschließen, wäre die Formulierung $\langle AV \rangle \sim \rightarrow \langle UV1 \rangle * \langle UV2 \rangle * \langle UV3 \rangle - \langle UV1 \rangle : \langle UV2 \rangle : \langle UV3 \rangle$ möglich.

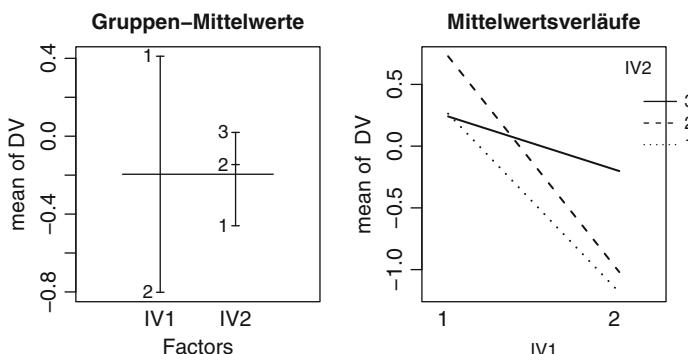


Abb. 8.4 Darstellung der Gruppen-Mittelwerte durch die Funktionen `plot.design()` und `interaction.plot()`

8.5.2 Quadratsummen vom Typ I und III

In wegen unbalanciert ungleicher Zellbesetzungen nicht-orthogonalen mehrfaktoriellen Designs offenbart sich, dass R eine andere Methode zur Berechnung der Quadratsummen heranzieht, als es der Voreinstellung etwa von SPSS und SAS entspricht.

R berechnet sog. sequentielle Quadratsummen vom Typ I, für die bei unbalancierten Designs die Reihenfolge der im Modell berücksichtigten Terme bedeutsam ist. Die Quadratsumme eines Effekts wird hier als Dekrement der Quadratsumme der Fehler (Residual Sum of Squares, RSS), also dem Zugewinn an aufgeklärter Varianz, beim Wechsel zwischen den folgenden beiden Modellen berechnet: dem Modell mit allen Vorhersagetermen, die in der Formel vor dem zu testenden Effekt auftauchen und dem Modell, in dem zusätzlich dieser Effekt selbst berücksichtigt wird (vgl. Abschn. 7.3.2). Der auf der so berechneten Effekt-Quadratsumme (RSS-Differenz) fußende Test soll mithin die Hypothese testen, ob ein Effekt eine signifikante Steigerung an aufgeklärter Varianz im Vergleich zum Modell mit allen vor ihm berücksichtigten Vorhersagetermen mit sich bringt, wobei alle übrigen Effekte vernachlässigt werden. Dies ist bei Haupteffekten äquivalent zu der Frage, ob die zugehörigen gewichteten Randerwartungswerte identisch sind, wobei die Gewichtung der Zellerwartungswerte bei ihrer Mittelung mit den Zellbesetzungen erfolgt. Die Summe aller Effekt-Quadratsummen ist hier gleich der RSS-Differenz vom vollständigen Modell und jenem, in dem kein Effekt, also nur der Gesamterwartungswert eingeht ($\langle AV \rangle \sim 1$).

SPSS dagegen verwendet in der Voreinstellung sog. partielle Quadratsummen vom Typ III. Die Quadratsumme eines Effekts wird hier als RSS-Reduktion beim Wechsel zwischen zwei anderen Modellen berechnet: dem Modell mit allen Vorhersagetermen außer dem interessierenden Effekt (egal ob vor oder nach diesem in der Formel stehend) und dem vollständigen Modell mit allen Termen. Die Summe aller Effekt-Quadratsummen hat hier keine Bedeutung. Der auf der so berechneten Effekt-Quadratsumme beruhende Test soll die Hypothese testen, ob ein Effekt einen signifikanten Zugewinn an aufgeklärter Varianz im Vergleich zum Modell mit allen anderen Vorhersagetermen mit sich bringt. Dies ist bei Haupteffekten äquivalent zu der Frage, ob die ungewichteten Randerwartungswerte übereinstimmen.

Es handelt sich also bei Quadratsummen vom Typ I und vom Typ III letztlich nicht einfach um unterschiedliche Berechnungsmethoden desselben Kennwertes, sondern um Tests inhaltlich unterschiedlicher Fragestellungen, die sich einmal auf gewichtete (Typ I) und einmal auf ungewichtete (Typ III) Randerwartungswerte beziehen. Dies äußert sich in unterschiedlichen Koeffizienten der zu einem Effekt gehörenden Kontraste (für eine vertiefte Darstellung vgl. Maxwell und Delaney, 2004). Im Spezialfall balancierter Designs fällt bei der empirischen Berechnung der Quadratsummen die Gewichtung von Zellmittelwerten mit den Zellbesetzungen mit einer Gleichgewichtung zusammen, weshalb hier Quadratsummen vom Typ I und III gleiche Ergebnisse liefern.

Das folgende Beispiel eines nicht-orthogonalen CRF-33 Designs mit ungleichen Zellbesetzungen ist jenes aus Maxwell und Delaney (2004, p. 338 ff.). Zunächst folgen die Modellvergleiche, die zu Quadratsummen vom Typ I führen, daraufhin die Berechnung der Quadratsummen vom Typ III.

```

> mdP <- 3                                     # Anzahl Stufen UV1
> mdQ <- 3                                     # Anzahl Stufen UV2

# AV Daten aus Bedingungen 1-1, 1-2, 1-3, 2-1, 2-2, 2-3, 3-1, 3-2, 3-3
> g11 <- c(41, 43, 50)
> g12 <- c(51, 43, 53, 54, 46)
> g13 <- c(45, 55, 56, 60, 58, 62, 62)
> g21 <- c(56, 47, 45, 46, 49)
> g22 <- c(58, 54, 49, 61, 52, 62)
> g23 <- c(59, 55, 68, 63)
> g31 <- c(43, 56, 48, 46, 47)
> g32 <- c(59, 46, 58, 54)
> g33 <- c(55, 69, 63, 56, 62, 67)
> mdDV <- c(g11, g12, g13, g21, g22, g23, g31, g32, g33) # alle Daten

# Zugehörige Faktoren UV1, UV2, Datensatz
> mdIV1 <- factor(rep(1:mdP, c(3+5+7, 5+6+4, 5+4+6)))
> mdIV2 <- factor(rep(rep(1:mdQ, mdP), c(3,5,7, 5,6,4, 5,4,6)))
> dfMD <- data.frame(IV1=mdIV1, IV2=mdIV2, DV=mdDV)

# Quadratsummen vom Typ I aus anova()
> (anovaFull <- anova(lm(DV ~ IV1 + IV2 + IV1:IV2, data=dfMD)))
Analysis of Variance Table
Response: DV
Df Sum Sq Mean Sq F value    Pr(>F)
IV1     2   101.11   50.56   1.8102   0.1782
IV2     2  1253.19   626.59  22.4357 4.711e-07 ***
IV1:IV2  4   14.19     3.55   0.1270   0.9717
Residuals 36  1005.42   27.93

# manuelle Berechnung der Quadratsummen vom Typ I aus
# Vergleich der sequentiell aufeinander aufbauenden Modelle
> SSI1 <- anova(lm(DV ~ 1,           dfMD), lm(DV ~ IV1,           dfMD))
> SSI2 <- anova(lm(DV ~ IV1,         dfMD), lm(DV ~ IV1+IV2,       dfMD))
> SSIi <- anova(lm(DV ~ IV1+IV2,   dfMD), lm(DV ~ IV1+IV2+IV1:IV2, dfMD))

> SSI1[["Sum of Sq"]][2]                      # QS Typ I von UV1
[1] 101.1111

> SSI2[["Sum of Sq"]][2]                      # QS Typ I von UV2
[1] 1253.189

> SSIi[["Sum of Sq"]][2]                       # QS der Interaktion
[1] 14.18714

# Vergleich des Gesamtmodells mit jenem ohne Effekte

```

```
> SSIt <- anova(lm(DV ~ 1, dfMD), lm(DV ~ IV1+IV2+IV1:IV2, dfMD))
> SSIt[["Sum of Sq"]][2] # QS des Gesamtmodells
[1] 1368.487

# Summe aller einzelnen Quadratsummen vom Typ I
> SSI1[["Sum of Sq"]][2] + SSI2[["Sum of Sq"]][2] + SSIi[["Sum of Sq"]][2]
[1] 1368.487
```

In R gibt es im wesentlichen zwei Möglichkeiten, Quadratsummen vom Typ III zu erhalten. Zum einen kann mit `drop1()` (vgl. Abschn. 7.3.2) die RSS-Änderung berechnet werden, die sich jeweils ergibt, wenn ein Vorhersageterm aus der Formel gestrichen wird und alle anderen beibehalten werden. Wie oben ausgeführt liegen diese Vergleiche gerade Quadratsummen vom Typ III zugrunde. Die Quadratsumme der Fehler steht in der Ausgabe in der Zeile `<none>`. Zum anderen testet die `Anova()` Funktion aus dem car Paket mit Quadratsummen vom Typ III, wenn das Argument `type="III"` gesetzt ist (vgl. Abschn. 8.4.3). In beiden Fällen ist das `contrasts` Argument für `lm()` notwendig, um von der Dummy- zur Effektcodierung der UVn zu wechseln (vgl. Abschn. 8.3.1).

```
# lineares Modell mit Effektcodierung
> modelIII <- lm(DV ~ IV1 + IV2 + IV1:IV2, data=dfMD,
+                   contrasts=list(IV1=contr.sum, IV2=contr.sum))

> drop1(modelIII, ~ ., test="F") # QS Typ III aus drop1()
Single term deletions
Model:
DV ~ IV1 + IV2 + IV1:IV2

      Df  Sum of Sq    RSS     AIC F value    Pr(F)
<none>          1005.42 157.79
IV1     2      204.76 1210.19 162.13  3.6658  0.03556 *
IV2     2     1181.11 2186.53 188.75 21.1452 8.447e-07 ***
IV1:IV2 4      14.19 1019.61 150.42  0.1270  0.97170

> library(car) # QS Typ III aus Anova()
> Anova(modelIII, type="III") # ...
```

Die Quadratsumme der Interaktion unterscheidet sich nicht zwischen Typ I und Typ III, da sich die Interaktion nur auf Zellerwartungswerte bezieht und Fragen der Gewichtung somit entfallen. Dies ist auch an den Modellvergleichen zu erkennen, die hier dieselben für die Quadratsummen beider Typen sind – das eingeschränkte Modell ist in beiden Fällen $\langle DV \rangle \sim \langle UV1 \rangle + \langle UV2 \rangle$, das vollständige bzw. sequentiell folgende Modell $\langle DV \rangle \sim \langle UV1 \rangle + \langle UV2 \rangle + \langle UV1 \rangle : \langle UV2 \rangle$.

Es folgt die manuelle Berechnung der Quadratsummen vom Typ III der Haupteffekte.

```
# Zellmittelwerte
> cellMs <- tapply(dfMD$DV, list(dfMD$IV1, dfMD$IV2), mean)
1   2   3
1 44.66667 49.40 56.85714
```

```
2 48.60000 56.00 61.25000
3 48.00000 54.25 62.00000
```

```
# Zellbesetzungen
> (cellNs <- tapply(dfMD$DV, list(dfMD$IV1, dfMD$IV2), length))
  1 2 3
1 3 5 7
2 5 6 4
3 5 4 6

> rowMs <- rowMeans(cellMs)                      # ungewichtete Zeilenmittel
> colMs <- colMeans(cellMs)                      # ungewichtete Spaltenmittel

# effektive Rand-Ns aus harmonischem Mittel der Zellen-Ns
> effNj <- mdP / apply(1/cellNs, 1, sum)        # harmonisches Mittel Zeilen-N
> effNk <- mdQ / apply(1/cellNs, 2, sum)        # harmonisches Mittel Spalten-N

# Gesamtmittel aus gewichteten Zeilen-Ms, bzw. gewichteten Spalten-Ms
> gM1   <- sum(effNj*rowMs) / sum(effNj)       # aus Zeilen-Ms
> gM2   <- sum(effNk*colMs) / sum(effNk)       # aus Spalten-Ms

# Quadratsummen - mdP bzw. mdQ * quadrierte Differenzen der Randmittel
# zum zugehörigen Gesamtmittel, gewichtet mit dem effektiven Rand-N
> (SSIIII1 <- mdP * sum(effNj * (rowMs-gM1)^2))          # QS Typ III UV1
[1] 204.7617

> (SSIIII2 <- mdQ * sum(effNk * (colMs-gM2)^2))          # QS Typ III UV2
[1] 1181.105

# QS Fehler - Summe der quadrierten Differenzen zum Zellenmittel
> (SSE     <- sum((dfMD$DV - ave(dfMD$DV,dfMD$IV1,dfMD$IV2,FUN=mean))^2))
[1] 1005.424
```

Der *F*-Bruch eines Effekts ergibt sich als Quotient der Effekt-Quadratsumme dividiert durch ihre Freiheitsgrade und der Quadratsumme der Fehler des vollständigen Modells dividiert durch ihre Freiheitsgrade.

```
> dfSS1 <- mdP-1                                # Freiheitsgrade QS UV1
> dfSS2 <- mdQ-1                                # Freiheitsgrade QS UV2
> dfSSE <- sum(cellNs-1)                         # Freiheitsgrade QS Fehler vollst. Modell
> (Fval1 <- (SSIIII1/dfSS1) / (SSE/dfSSE))      # Teststatistik F-Wert UV1
[1] 3.665829

> (Fval2 <- (SSIIII2/dfSS2) / (SSE/dfSSE))      # Teststatistik F-Wert UV2
[1] 21.14520

> (pVal1 <- 1-pf(Fval1, dfSS1, dfSSE))         # p-Wert UV1
[1] 0.03555901

> (pVal2 <- 1-pf(Fval2, dfSS2, dfSSE))         # p-Wert UV2
[1] 8.44678e-07
```

8.5.3 Beliebige *a-priori* Kontraste

Im folgenden sei wieder das balancierte Design aus Abschn. 8.5.1 betrachtet. Im Vergleich zum einfaktoriellen Fall im CRF-*p* Design ergeben sich für beliebige *a-priori* Kontraste im CRF-*pq* Design einige Änderungen, da es nun verschiedene Typen von Kontrasten gibt. Zunächst sind dies jene Kontraste, die sich auf die sog. assoziierte einfaktorielle Varianzanalyse beziehen. Dies bedeutet, dass die faktorielle Struktur der Bedingungskombinationen ignoriert wird, stattdessen werden alle Bedingungskombinationen als Stufen einer einzigen (künstlichen) UV betrachtet. Aus einem Design mit zwei Stufen der ersten und drei Stufen der zweiten UV würde so eines mit $2 \cdot 3 = 6$ Stufen einer einzigen UV. Innerhalb dieser assoziierten einfaktoriellen Situation lassen sich nun Kontraste wie in Abschn. 8.3.6.1 formulieren und testen.

Durch die Gleichheit der Gruppengrößen vereinfacht sich die Formel für die Teststatistik, da nicht mehr gruppenweise mit der Anzahl der Beobachtungen gewichtet werden muss. Die Mittlere Quadratsumme der Fehler aus der zweifaktoriellen Varianzanalyse ist dieselbe wie die Mittlere Quadratsumme innerhalb der Gruppen aus der assoziierten einfaktoriellen Varianzanalyse. Zu beachten ist die Reihenfolge der Stufen in der assoziierten einfaktoriellen Situation. Sie wird hier durch die Stufen des künstlichen Faktors bestimmt, dessen Ausprägung von der Kombination der Faktorstufen der beiden UVn abhängt.

Im Beispiel soll das bereits verwendete CRF-23 Design mit dem in Tabelle 8.1 dargestellten Schema vorliegen.

Als assoziiertes einfaktorielles Schema ergibt sich das in Tabelle 8.2 aufgeführte.

```
# assoziierte einfaktorielle sowie die zweifaktorielle ANOVA
> CRFPQ1 <- anova(lm(DV ~ IVcomb, data=dfCRFPQ))
> CRFPQ2 <- anova(lm(DV ~ IV1*IV2, data=dfCRFPQ))

# Vergleich von MS within und MS Error -> sollten identisch sein
> isTRUE(all.equal(CRFPQ1[["Mean Sq"]][2], CRFPQ2[["Mean Sq"]][4]))
[1] TRUE
```

Tabelle 8.1 CRF-23 Designschema mit Zell- und Randerwartungswerten

	UV2 – 1	UV2 – 2	UV2 – 3	Mittel
UV1 – 1	μ_{11}	μ_{12}	μ_{13}	$\mu_{1.}$
UV1 – 2	μ_{21}	μ_{22}	μ_{23}	$\mu_{2.}$
Mittel	$\mu_{.1}$	$\mu_{.2}$	$\mu_{.3}$	μ

Tabelle 8.2 Assoziiertes einfaktorielles Schema zum CRF-23 design

UVcomb – 1	UVcomb – 2	UVcomb – 3	UVcomb – 4	UVcomb – 5	UVcomb – 6	M
μ_{11}	μ_{12}	μ_{13}	μ_{21}	μ_{22}	μ_{23}	μ

Mit der `glht()` Funktion des `multcomp` Pakets soll im Beispiel das Mittel der Erwartungswerte μ_{11} und μ_{12} gegen das Mittel der verbleibenden vier Gruppenerwartungswerte getestet werden. Unter der Nullhypothese soll der Kontrast gleich 0 sein, unter der Alternativhypothese größer als 0.

```
> library(multcomp) # für glht()
> aovCRFpq <- aov(DV ~ IVcomb, data=dfCRFpq) # aov-Modell

# Matrix der Kontrastkoeffizienten - hier nur eine Zeile
> cntrMat <- rbind("contrast 01"=c(1/2, 1/2, -1/4, -1/4, -1/4, -1/4))
> summary(glht(aovCRFpq, linfct=mcp(IVcomb=cntrMat), alternative="greater"))
Simultaneous Tests for General Linear Hypotheses
Multiple Comparisons of Means: User-defined Contrasts
Fit: aov(formula = DV ~ IVcomb, data = dfCRFpq)
Linear Hypotheses:
Estimate Std. Error t value Pr(>t)
contrast 01 <= 0 1.0372 0.3368 3.08 0.00182 **
---
(Adjusted p values reported -- single-step method)

# manuelle Berechnung
> cellMs <- tapply(dfCRFpq$DV, dfCRFpq$IVcomb, mean) # Zellen-Ms
> MSE <- CRFpq2[["Mean Sq"]][4] # MS Error
> dfSSE <- (nSubj-1)*P*Q # df von SS Error
> psiHat <- sum(cntrMat[1, ] * cellMs) # Kontrastschätzung
> lenSq <- (1/nSubj) * sum(cntrMat[1, ]^2) # quadrierte Länge
> statTc <- psiHat / sqrt(lenSq*MSE) # Teststatistik t
> abs(statTc) # deren Betrag
[1] 3.079714

> (critT <- qt(1-0.05, dfSSE)) # kritischer t-Wert einseitig
[1] 1.681952

> (pValT <- 1-pt(abs(statTc), dfSSE)) # p-Wert einseitig
[1] 0.001822898

# untere Grenze des einseitigen 95%-Vertrauensintervalls für psi
> (ciLo <- psiHat - critT*sqrt(lenSq*MSE))
[1] 0.4707625
```

Der Test fällt signifikant aus, wie einerseits am p -Wert, andererseits daran zu sehen ist, dass der Betrag der Teststatistik größer als der kritische t -Wert ist. Zudem ist die untere Grenze des einseitigen Vertrauensintervalls für ψ größer als 0, das Intervall enthält ψ_0 also nicht.

Sollen mehrere Kontraste gleichzeitig getestet werden, ist dies durch Zusammenstellung der zugehörigen Kontrastvektoren als (ggf. benannte) Zeilen einer Matrix möglich. Hier werden zunächst mit `glht()` aus dem `multcomp` Paket drei Kontraste ohne Adjustierung des α -Niveaus getestet, dann manuell mit α -Adjustierung nach Bonferroni.

```

# Matrix der Kontrastkoeffizienten
> cntrMat <- rbind( "contrast 01"=c(1/2, 1/2, -1/4, -1/4, -1/4, -1/4),
+                      "contrast 02"=c( 0,   1,    -1,     0,     0,     0),
+                      "contrast 03"=c( 0,   0,     0,   1/2,   -1,   1/2))

> summary(glht(aovCRFpq, linfct=mcp(IVcomb=cntrMat),
+                alternative="greater"), test=adjusted("none"))
Simultaneous Tests for General Linear Hypotheses
Multiple Comparisons of Means: User-defined Contrasts
Fit: aov(formula = DV ~ IVcomb, data = dfCRFpq)
Linear Hypotheses:
Estimate Std. Error t value Pr(>t)
contrast 01 <= 0  1.0372    0.3368   3.080 0.00182 **
contrast 02 <= 0  0.4878    0.5500   0.887 0.19009
contrast 03 <= 0  0.3274    0.4763   0.687 0.24783
---
(Adjusted p values reported -- none method)

# manuelle Berechnung mit alpha-Adjustierung nach Bonferroni
> alphaAdj <- 0.05/nrow(cntrMat)                      # Bonferroni Adjustierung
> psiHats <- cntrMat %*% cellMs                         # Kontrastschätzungen

# quadrierte Längen
> lenSqs <- (1/nSubj) * cntrMat^2 %*% (rep(1, ncol(cntrMat)))
> statTs <- psiHats / sqrt(lenSqs*MSE)                  # Teststatistiken

# kritische adjustierte t-Werte
> critTs <- rep(qt(1-alphaAdj, dfSSE), nrow(cntrMat))
> pValTs <- 1-pt(abs(statTs), dfSSE)                   # p-Werte einseitig
> (resDf <- data.frame(statTs, critTs, pVals=pValTs,
+                        alphaAdj=rep(alphaAdj, nrow(cntrMat)),
+                        sig=abs(statTs)>critTs))
      statTs    critTs      pVals   alphaAdj    sig
contrast 01  3.0797137  2.200368  0.001822898 0.016666667 TRUE
contrast 02  0.8869062  2.200368  0.190090003 0.016666667 FALSE
contrast 03  0.6873081  2.200368  0.247833311 0.016666667 FALSE

```

Im abschließend ausgegebenen Datensatz wird für jeden Kontrast der empirische Wert der Teststatistik, der (für alle identische) kritische *t*-Wert, der jeweils zugehörige *p*-Wert, das adjustierte α sowie schließlich das Ergebnis der Signifikanzprüfung im Sinne des Vergleichs von *p*-Wert und adjustiertem α aufgeführt.

Neben allgemeinen Kontrasten, die sich auf das assoziierte einfaktorielle Design beziehen, gibt es in der zweifaktoriellen Situation drei weitere Kontrasttypen, auch *Familien* von Kontrasten genannt: Linearkombinationen der mittleren Erwartungswerte in den Stufen der ersten UV (*A-Kontraste*), solche der mittleren Erwartungswerte in den Stufen der zweiten UV (*B-Kontraste*) und Interaktions-Kontraste (*I-Kontraste*). Alle drei zeichnen sich dadurch aus, dass sich die Koeffizienten der Linearkombination nicht nur insgesamt zu 0 summieren, sondern zudem weitere Nebenbedingungen erfüllen, wenn sie in das Designschema eingetragen werden:

- A-Kontraste werden mit Koeffizienten gebildet, die im Designschema in jeder Zeile konstant sind und sich pro Spalte zu 0 summieren. Im obigen Beispiel mit der entsprechenden Reihenfolge der Zellen würde etwa der Kontrast $(1, 1, 1, -1, -1, -1)$ den mittleren Erwartungswert μ_1 . gegen den mittleren Erwartungswert μ_2 . testen.
- B-Kontraste werden mit Koeffizienten gebildet, die im Designschema in jeder Spalte konstant sind und sich pro Zeile zu 0 summieren. Im obigen Beispiel mit der entsprechenden Reihenfolge der Zellen würde etwa der Kontrast $(1, -1/2, -1/2, 1, -1/2, -1/2)$ den mittleren Erwartungswert $\mu_{1,1}$ gegen das Mittel der mittleren Erwartungswerte $\mu_{1,2}$ und $\mu_{1,3}$ testen.
- I-Kontraste werden mit Koeffizienten gebildet, die sich im Designschema sowohl pro Zeile als auch pro Spalte zu 0 summieren. Im obigen Beispiel mit der entsprechenden Reihenfolge der Zellen würde etwa der Kontrast $(1, -1/2, -1/2, -1, 1/2, 1/2)$ die Differenz der Gruppenerwartungswerte $\mu_{11} - \mu_{21}$ gegen das Mittel der beiden Differenzen der Gruppenerwartungswerte $\mu_{12} - \mu_{22}$ und $\mu_{13} - \mu_{23}$ testen.

A-, B- und I-Kontraste können genauso innerhalb der assoziierten einfaktoriellen Varianzanalyse getestet werden wie allgemeine Kontraste. Statt mit den Gruppenerwartungswerten können A- und B-Kontraste zudem auch mit den mittleren Erwartungswerten formuliert werden. In diesem Fall ist die Teststatistik entsprechend anders zu bilden, der kritische Wert ändert sich dagegen nicht.

Im Beispiel sollen die oben genannten A- und B-Kontraste mit Hilfe der mittleren Erwartungswerte formuliert werden.

```
# A-Kontrast aus Randmittelwerten der UV1
> meansA  <- tapply(dfCRFpq$DV, dfCRFpq$IV1, mean)      # Zeilenmittel
> cntrVecA <- c(1, -1)                                     # Kontrastvektor
> psiHatA  <- sum(cntrVecA * meansA)                      # Kontrastschätzung
> lenSqA   <- (1/(Q*nSubj)) * sum(cntrVecA^2)            # quadrierte Länge
> statA    <- psiHatA / sqrt(lenSqA*MSE)                  # Teststatistik

# B-Kontrast aus Randmittelwerten der UV2
> meansB  <- tapply(dfCRFpq$DV, dfCRFpq$IV2, mean)      # Spaltenmittel
> cntrVecB <- c(1, -1/2, -1/2)                            # Kontrastvektor
> psiHatB  <- sum(cntrVecB * meansB)                      # Kontrastschätzung
> lenSqB   <- (1/(P*nSubj)) * sum(cntrVecB^2)            # quadrierte Länge
> statB    <- psiHatB / sqrt(lenSqB*MSE)                  # Teststatistik
```

8.5.4 Beliebige post-hoc Kontraste nach Scheffé

Beliebige Kontraste im Sinne von Linearkombinationen von Gruppenerwartungswerten können auch im Anschluss an eine signifikante Varianzanalyse getestet werden. Die zweifaktorielle Varianzanalyse prüft beim Test der Effekte (zwei Haupteffekte, ein Interaktionseffekt) implizit simultan alle möglichen zugehörigen Kontraste (A , B oder I) – spezifische Hypothesen liegen also bei ihrer Anwendung nicht

vor. Aus diesem Grund muss im Anschluss an eine Varianzanalyse bei Einzeltests eine geeignete α -Adjustierung vorgenommen werden, hier vorgestellt nach der Methode von Scheffé.

Zunächst gilt für das Aufstellen eines Kontrasts alles bereits für beliebige a-priori Kontraste Ausgeführt. Lediglich die Wahl des kritischen Wertes weicht ab und ergibt sich zur α -Adjustierung aus einer F -Verteilung. Dieser kritische Wert ist mit dem Quadrat der a-priori Teststatistik zu vergleichen – die etwa in der Ausgabe von `summary(glht())` abgelesen werden kann. Hier soll wieder das Mittel der Erwartungswerte μ_{11} und μ_{12} gegen das Mittel der verbleibenden vier Gruppenerwartungswerte gerichtet getestet werden. Da es sich hierbei nicht um einen A , B oder I -Kontrast handelt, soll in der assoziierten einfaktoriellen Situation getestet werden, was zu einer sehr konservativen α -Adjustierung führt.

```
> (statF <- psiHat^2 / (lenSq*MSE)) # quadrierte Teststatistik
[1] 9.484637

# df von SS between aus der assoziierten einfaktoriellen Varianzanalyse
> dfSSba <- P*Q - 1

# kritischer F-Wert für quadrierte Teststatistik
> (critF <- dfSSba*qf(1-0.05, df1=dfSSba, df2=dfSSE))
[1] 12.18846

> (pValF <- 1-pf(statF/dfSSba, dfSSba, dfSSE)) # p-Wert einseitig
[1] 0.1152858
```

Während der empirische Wert der Teststatistik derselbe wie im a-priori Fall ist, erhöht sich der kritische Wert aufgrund der simultanen α -Adjustierung deutlich.

Die Wahl des kritischen Wertes erfolgte hier so, dass alle möglichen Kontraste aus der assoziierten einfaktoriellen Varianzanalyse zugelassen sind. Sollen die Kontraste dagegen nur aus einem Unterraum des Kontrastraumes stammen, etwa weil jeweils nur Kontraste aus einer Familie (A , B oder I) von Interesse sind, kann der kritische Wert entsprechend anders gewählt werden. In diesem Fall erfolgt keine gleichzeitige α -Adjustierung für alle möglichen Kontraste, sondern nur für Kontraste aus der gewählten Familie, woraus ein geringerer kritischer Wert resultiert. Für A -Kontraste (p Stufen) wäre der kritische F -Wert $(P-1) * qf(1-0.05, df1=P-1, df2=dfSSE)$, für B -Kontraste (q Stufen) $(Q-1) * qf(1-0.05, df1=Q-1, df2=dfSSE)$, für I -Kontraste $(P-1) * (Q-1) * qf(1-0.05, df1=(P-1)*(Q-1), df2=dfSSE)$.

8.6 Zweifaktorielle Varianzanalyse mit zwei Intra-Gruppen Faktoren (RBF- pq)

Wird jede VP in jeder der von zwei Faktoren gebildeten Bedingungskombinationen beobachtet, spricht man von einem RBF- pq Design (Randomized Block Factorial) mit p Stufen der ersten und q Stufen der zweiten UV.

8.6.1 Univariat formulierte Auswertung mit `aov()`

Die Reihenfolge, in der die $p \cdot q$ Bedingungen durchlaufen werden, ist für jede VP zu randomisieren. Ein Block aus $p \cdot q$ voneinander abhängigen Beobachtungen (eine aus jeder Bedingung) kann dabei auch von unterschiedlichen, z. B. bzgl. relevanter Störvariablen gematchten VPn stammen. Hierbei ist innerhalb eines Blocks die Zuteilung von VPn zu Bedingungen zu randomisieren.

Im Vergleich zum CRF- pq Design wirkt im Modell zum RBF- pq Design ein systematischer Effekt mehr am Zustandekommen einer Beobachtung mit: zusätzlich zu den drei festen Effekten, die auf die Gruppenzugehörigkeit bzgl. der beiden UVn zurückgehen (beide Haupteffekte und der Interaktionseffekt), ist dies der zufällige Blockeffekt.

Wie beim RB- p Design müssen die Daten im Long-Format (vgl. Abschn. 3.2.8) inklusive einer Variable vorliegen, die die Blockzugehörigkeit codiert. Diese Variable sei hier `<BlockNr>` und muss ein Objekt der Klasse `factor` sein. Zudem muss in der Formel explizit angegeben werden, aus welchen additiven Komponenten sich die Quadratsumme innerhalb der Gruppen zusammensetzt. Im konkreten Fall drückt `Error(<BlockNr>/(<UV1>*<UV2>))` aus, dass `<BlockNr>` in den durch die Kombination von `<UV1>` und `<UV2>` entstehenden Bedingungen verschachtelt ist, nicht alle möglichen Blöcke also in jeder Kombination von Bedingungen experimentell realisiert sind, sondern nur eine Zufallsauswahl (`<BlockNr>` ist ein Random-Faktor).¹⁰ Die durch die kombinierte Variation von UV1 und UV2 entstehenden Effekte in der AV (beide Einzeleffekte und der Interaktionseffekt) sind jeweils innerhalb der durch `<BlockNr>` definierten Blöcke zu analysieren.

```
> aov(<AV> ~ <UV1>*<UV2> + Error(<BlockNr>/(<UV1>*<UV2>)),
+       data=<Datensatz>)

> nSubj      <- 10                                # Zellbesetzung
> P          <- 2                                 # Messzeitpunkte UV1
> Q          <- 3                                 # Messzeitpunkte UV2
> id         <- factor(rep(1:nSubj, P*Q))        # Blockzugehörigkeit
> timeFac1  <- factor(rep(rep(1:P, each=nSubj), Q))  # Intra-Gruppen UV1
> timeFac2  <- factor(rep(rep(1:Q, each=P*nSubj)))  # Intra-Gruppen UV2

# Simulation der AV mit Effekt beider UVn, ohne Interaktion
> DV_t11    <- round(rnorm(nSubj, -0.8, 1), 2)    # AV zu t1-1
> DV_t12    <- round(rnorm(nSubj, -0.7, 1), 2)    # AV zu t1-2
> DV_t13    <- round(rnorm(nSubj, 0.0, 1), 2)    # AV zu t1-3
> DV_t21    <- round(rnorm(nSubj, 0.2, 1), 2)    # AV zu t2-1
> DV_t22    <- round(rnorm(nSubj, 0.3, 1), 2)    # AV zu t2-2
> DV_t23    <- round(rnorm(nSubj, 1.0, 1), 2)    # AV zu t2-3
```

¹⁰ Der Term lautet `Error(<BlockNr>+<BlockNr>:<UV1>+<BlockNr>:<UV2>+<BlockNr>:<UV1>:<UV2>)`, wenn er ausgeschrieben wird. Dies sind die vier Effekte, deren Quadratsummen sich zur Quadratsumme innerhalb der Gruppen addieren.

```
# Datensatz im Long-Format und Auswertung mit aov()
> dfRBFpqL <- data.frame(id, timeFac1, timeFac2,
+                           DV=c(DV_t11, DV_t21, DV_t12, DV_t22, DV_t13, DV_t23))

> aovRBFpq <- aov(DV~timeFac1*timeFac2 + Error(id/(timeFac1*timeFac2)),
+                     data=dfRBFpqL)

> summary(aovRBFpq)
Error: id
      Df  Sum Sq Mean Sq F value Pr(>F)
Residuals  9  3.6306  0.4034

Error: id:timeFac1
      Df  Sum Sq Mean Sq F value Pr(>F)
timeFac1   1 16.865 16.8646 11.209 0.00855 **
Residuals  9 13.541  1.5045

Error: id:timeFac2
      Df  Sum Sq Mean Sq F value Pr(>F)
timeFac2   2 13.797  6.8987  4.5641 0.02493 *
Residuals 18 27.207  1.5115

Error: id:timeFac1:timeFac2
      Df  Sum Sq Mean Sq F value Pr(>F)
timeFac1:timeFac2 2  2.2468  1.1234  0.9608 0.4014
Residuals        18 21.0473  1.1693
```

Der Output unterscheidet sich von jenem im CRF-*pq* Design, da hier nicht mehr dieselbe Fehler-Quadratsumme für den Test jeder der drei Effekte herangezogen wird. Die beim Test eines Effekts jeweils verwendete Quelle der Fehlervarianz wird durch die Error: <Effekt> Überschriften kenntlich gemacht, ihre Quadratsumme findet sich in der Zeile Residuals. Die Quadratsumme des festen Faktors timeFac1 wird mit der Quadratsumme aus der Interaktion von timeFac1 und dem Random-Faktor id in der Rolle als Fehler-Quadratsumme verglichen. Analoges gilt für den festen Faktor timeFac2. Die Quadratsumme der Interaktion der beiden festen Faktoren wird entsprechend mit der Quadratsumme der Interaktion dritter Ordnung vom Random-Faktor und beiden festen Faktoren getestet.

Der Test eines Haupteffekts ist äquivalent zum Test im einfaktoriellen RB-*p* Design der zuvor blockweise über die Stufen der jeweils anderen UV gemittelten Daten. Für den Test der UV1 ergibt sich dabei als neuer Wert jedes Blocks für eine Stufe der UV1 der zugehörige Mittelwert des Blocks über die Stufen der UV2 – der Test der UV2 erfolgt analog. Der Test der Interaktion von UV1 und UV2 ist u. a. äquivalent zum Test, ob die bedingten Haupteffekte der UV1 für jede Stufe der UV2 identisch sind. Da die UV1 hier nur zwei Stufen hat, können ihre bedingten Haupteffekte durch die blockweisen Differenzen zwischen beiden Stufen der UV1 geschätzt werden: als neuer Wert jedes Blocks für eine Stufe der UV2 ergibt sich die Differenz der zugehörigen Messwerte zwischen den Stufen der UV1. Mit diesen Daten lässt sich hier ein zum Test der Interaktion äquivalenter Test im RB-*p* Design durchführen.

```

> dfG <- dfRBFpqL                                # kürzerer Name für Datensatz

# Datensatz aus blockweise über Stufen der (UV2) gemittelten Daten
> mDf1 <- aggregate(dfG$DV, list(dfG$id, dfG$timeFac1), mean)

# Datensatz aus blockweise über Stufen der (UV1) gemittelten Daten
> mDf2 <- aggregate(dfG$DV, list(dfG$id, dfG$timeFac2), mean)

# hier: Datensatz aus blockweisen Differenzen zwischen Stufen der (UV1)
> dDfI <- aggregate(dfG$DV, list(dfG$id, dfG$timeFac2), diff)

# äquivalent zum Test des Haupteffekts von (UV1) im RBF-pq Design
> summary(aov(x ~ Group.2 + Error(Group.1/Group.2), data=mDf1))  # ...

# äquivalent zum Test des Haupteffekts von (UV2) im RBF-pq Design
> summary(aov(x ~ Group.2 + Error(Group.1/Group.2), data=mDf2))  # ...

# hier: äquivalent zum Test der Interaktion (UV1):(UV2) im RBF-pq Design
> summary(aov(x ~ Group.2 + Error(Group.1/Group.2), data=dDfI))  # ...

Im allgemeinen Fall wäre der Test der Interaktion wie folgt manuell durchzuführen.

# für Quadratsumme Interaktion: Mittel der Zellen, (UV1), (UV2), gesamt
> cellMs  <- tapply(dfG$DV, list(dfG$timeFac1, dfG$timeFac2), mean)
> tf1Ms   <- tapply(dfG$DV, dfG$timeFac1, mean)           # timeFac1-Mittel
> tf2Ms   <- tapply(dfG$DV, dfG$timeFac2, mean)           # timeFac2-Mittel
> grandM  <- mean(cellMs)                                 # Gesamtmittel

# Interaktion (UV1):(UV2)
> TF1xTF2 <- c(sweep(sweep(cellMs, 1, tf1Ms), 2, tf2Ms)) + grandM
> (SSI    <- nSubj * sum(TF1xTF2^2))                      # Quadratsumme Interaktion
[1] 2.246813

# Quadratsumme der Fehler: jeden Wert durch zugehöriges Mittel ersetzen
> cellMsL <- ave(dfG$DV, dfG$timeFac1, dfG$timeFac2, FUN=mean)
> blTf1MsL <- ave(dfG$DV, dfG$id,      dfG$timeFac1, FUN=mean)
> blTf2MsL <- ave(dfG$DV, dfG$id,      dfG$timeFac2, FUN=mean)
> blockMsL <- ave(dfG$DV, dfG$id,      FUN=mean)
> tf1MsL   <- ave(dfG$DV, dfG$timeFac1, FUN=mean)
> tf2MsL   <- ave(dfG$DV, dfG$timeFac2, FUN=mean)

# Interaktion (BlockNr):(UV1):(UV2) -> Fehler bei Test von (UV1):(UV2)
> IDxTF1xTF2 <- dfG$DV - blTf1MsL - blTf2MsL - cellMsL
+                  + blockMsL + tf1MsL + tf2MsL - grandM

> (SSE <- sum(IDxTF1xTF2^2))                         # Kontrolle: QS Fehler
[1] 21.04732

> dfSSI  <- (P-1) * (Q-1)                            # Freiheitsgrade Interaktion
> dfSSE  <- (P-1) * (Q-1) * (nSubj-1)                # Freiheitsgrade Fehler
> (FvalI <- (SSI/dfSSI) / (SSE/dfSSE))             # F-Wert Interaktion
[1] 0.9607551

```

```
> (pValI <- 1-pf(FvalI, dfSSI, dfSSE))      # p-Wert Interaktion
[1] 0.4013768
```

Die varianzanalytische Auswertung eines dreifaktoriellen Designs mit drei Intra-Gruppen Faktoren (RBF-*pqr*) unterscheidet sich nur wenig von der beschriebenen zweifaktoriellen Situation. Lediglich die Formel mit den zu berücksichtigenden Effekten ist entsprechend anzupassen. Sollen alle Haupteffekte, Interaktionseffekte zweiter Ordnung und der Interaktionseffekt dritter Ordnung eingehen, könnte die Formel etwa $\langle AV \rangle \sim \langle UV1 \rangle * \langle UV2 \rangle * \langle UV3 \rangle + \rightarrow Error(\langle BlockNr \rangle / (\langle UV1 \rangle * \langle UV2 \rangle * \langle UV3 \rangle))$ lauten.

8.6.2 Zirkularität der Kovarianzmatrizen prüfen

Auch bei einer Varianzanalyse für Daten aus einem RBF-*pq* Design muss für jeden Test der drei möglichen Effekte die Voraussetzung der Zirkularität der zugehörigen Kovarianzmatrix erfüllt sein (vgl. Abschn. 8.4.2). Eine hinreichende Bedingung hierfür ist, dass die $p \cdot q \times p \cdot q$ Kovarianzmatrix aus der assoziierten einfaktoriellen Varianzanalyse zirkulär ist.¹¹ Dies ist jedoch keine notwendige Bedingung, d. h. es können einzelne Kovarianzmatrizen zirkulär sein, ohne dass es andere sind und ohne dass es jene der assoziierten einfaktoriellen Varianzanalyse ist. Aus diesem Grund wird meist so vorgegangen, dass separat für jeden Test eine Korrektur der Freiheitsgrade auf Basis der Schätzungen für ε nach Greenhouse und Geisser bzw. nach Huynh und Feldt erfolgt.

Bei der Schätzung von ε für den Test der Haupteffekte ist zunächst die oben beschriebene blockweise Mittelung vorzunehmen. Die jeweils resultierende Datenmatrix der gemittelten Werte umfasst im Wide-Format so viele Spalten, wie die zu testende UV Stufen besitzt und repräsentiert nunmehr Daten aus einem einfaktoriellen RB-*p* Design. Die Voraussetzung der Zirkularität bezieht sich auf ihre theoretische Kovarianzmatrix, mit dem empirischen Pendant deshalb der Korrekturfaktor ε geschätzt wird. Mit zwei Stufen der UV1 erübrigts sich hier die Notwendigkeit einer Korrektur für den Test der UV2, da 2×2 Kovarianzmatrizen immer zirkulär sind. Auch der Test der Interaktion lässt sich hier wie beschrieben auf Daten eines RB-*p* Designs zurückführen und ε auf Basis der Kovarianzmatrix der Fehler schätzen.

¹¹ Unter dieser Voraussetzung könnte jeder Effekt mit derselben Mittleren Quadratsumme der Fehler getestet werden. Diese wäre das gewichtete Mittel der drei einzelnen Fehler-Quadratsummen, also $(SS_{1:S} + SS_{2:S} + SS_{1:2:S}) / (df_{1:S} + df_{2:S} + df_{1:2:S})$, wenn $1 : S$ für die Interaktion $\langle UV1 \rangle : \langle BlockNr \rangle$ (andere analog), SS für Quadratsumme und df für ihre Freiheitsgrade steht. Die F-Brüche hätten dann entsprechend mehr $(df_{1:S} + df_{2:S} + df_{1:2:S})$ Nenner-Freiheitsgrade und die Tests damit mehr Power als bei Verwendung separater Fehler-Quadratsummen. Da die Gültigkeit der dafür notwendigen Voraussetzung in den meisten Situationen schwer zu rechtfertigen sein dürfte, wird dieser Test nicht von R verwendet, sondern müsste manuell berechnet werden.

```
# Schätzung von epsilon für Test von <UV2>: Daten als Matrix im Wide-Format
> dataMat <- tapply(dfG$DV, list(dfG$id, dfG$timeFac2), mean)
```

Die weiteren Berechnungen würde mit jenen in Abschn. 8.4.2 übereinstimmen.

```
# Schätzung von epsilon für Test der Interaktion <UV1>:<UV2> aus Fehlern
> dfG <- cbind(dfG, IDxTF1xTF2) # dem Datensatz Fehler hinzufügen

# extrahiere Fehler als Matrix im Wide-Format
> errMat <- as.matrix(unstack(dfG, IDxTF1xTF2 ~ timeFac2))
> Serr <- cov(errMat) # Kovarianzmatrix der Fehler
> epsGGI <- (1 / (Q-1)) * sum(diag(Serr))^2 / sum(Serr^2)
> epsHFI <- (nSubj*(Q-1)*epsGGI-2) / ((Q-1)*((nSubj-1)-((Q-1)*epsGGI)))
```

8.6.3 Multivariat formulierte Auswertung mit Anova()

Für eine Beschreibung der Anova() Funktion aus dem car Paket sowie für die multivariate Anwendung von anova() vgl. Abschn. 8.4.3 und 9.6. Im Vergleich zur RB-p Situation ändert sich hier im wesentlichen das Intra-Gruppen Design unter idata und idesign. Da nun zwei Intra-Gruppen Faktoren vorliegen, muss der unter idata anzugebende Datensatz zwei Variablen der Klasse factor beinhalten. In jeder Zeile von idata geben diese Faktoren an, aus welcher Bedingungskombination der beiden UVn die zugehörige Spalte der Datenmatrix im Wide-Format stammt. Unter idesign ist es nun möglich, als Vorhersageterme in der Formel einerseits die beiden Haupteffekte und andererseits die Interaktion der beiden Intra-Gruppen Faktoren einzutragen.

```
# Datensatz im Wide-Format
> dfRBFpqW <- data.frame(DV_t11, DV_t21, DV_t12, DV_t22, DV_t13, DV_t23)

# Zwischen-Gruppen Design - hier keine Zwischen-Gruppen UV
> modelRBFpq <- lm(cbind(DV_t11, DV_t21, DV_t12, DV_t22, DV_t13,
+ DV_t23) ~ 1, data=dfRBFpqW)

# Intra-Gruppen Design
> (intraRBFpq <- data.frame(timeFac1=factor(rep(1:P, Q)),
+ timeFac2=factor(rep(1:Q, each=P))))
```

	timeFac1	timeFac2
1	1	1
2	2	1
3	1	2
4	2	2
5	1	3
6	2	3

```
> library(car) # für Anova()
> AnovaRBFpq <- Anova(modelRBFpq, idata=intraRBFpq,
```

```
+           idesign=~timeFac1*timeFac2)

> summary(AnovaRBpq, multivariate=FALSE, univariate=TRUE)      # ...
```

Die unkorrigierten Ergebnisse der univariaten Auswertung stimmen mit den von `aov()` ermittelten überein, sofern alle Zellen dieselbe Anzahl an Beobachtungen aufweisen. Die Ergebnisse der univariaten Auswertung sind auch jene, die man mit der Auswertung bei multivariater Formulierung des Modells mit Hilfe von `anova()` erhält, sofern die Zirkularität der Kovarianzmatrix angenommen wird (Argument `test="Spherical"`, vgl. Fußnote 8). Die Schätzungen von ε sowie die entsprechend korrigierten *p*-Werte werden ebenfalls ausgegeben. Die Spezifizierung des Designs mit den Argumenten `M` und `X` setzt Kenntnisse des Allgemeinen Linearen Modells voraus.

```
# Test des ersten Intra-Gruppen Faktors timeFac1
> anova(modelRBpq, M=~timeFac1 + timeFac2, X=~timeFac2,
+        idata=intraRBpq, test="Spherical")                      # ...

# Test des zweiten Intra-Gruppen Faktors timeFac2
> anova(modelRBpq, M=~timeFac1 + timeFac2, X=~timeFac1,
+        idata=intraRBpq, test="Spherical")                      # ...

# Test der Interaktion von timeFac1 und timeFac2
> anova(modelRBpq, X=~timeFac1 + timeFac2, idata=intraRBpq,
+        test="Spherical")                                     # ...
```

8.6.4 Einzelvergleiche (Kontraste)

Prinzipiell ist es möglich, auch für ein RBF-*pq* Design Einzelvergleiche als Tests von Linearkombinationen von Erwartungswerten durchzuführen. Dabei ist die Situation analog zu jener im CRF-*pq* Design, wie sie in Abschn. 8.5.3 beschrieben wurde. Auch hier wären zunächst Vergleiche der mittleren Erwartungswerte des ersten Faktors (*A*-Kontraste) bzw. des zweiten Faktors (*B*-Kontraste) von Interaktionskontrasten (*I*-Kontraste) und allgemeinen Zellvergleichen zu unterscheiden. Im Vergleich zum CRF-*pq* Design ergeben sich jedoch Unterschiede bei der jeweils zu verwendenden Quadratsumme der Fehler: während diese im CRF-*pq* Design für jeden Test dieselbe ist, wird im RBF-*pq* Design jeder Test mit einer anderen Fehler-Quadratsumme durchgeführt. Die Effekt-Quadratsumme eines *A*-Kontrasts wäre im RBF-*pq* Design gegen die Quadratsumme Error: $\langle \text{BlockNr} \rangle : \langle \text{UV1} \rangle$ der Interaktion des Blocks mit der UV1 zu testen, ein *B*-Kontrast entsprechend gegen Error: $\langle \text{BlockNr} \rangle : \langle \text{UV2} \rangle$, ein *I*-Kontrast gegen Error: $\langle \text{BlockNr} \rangle : \langle \text{UV1} \rangle : \langle \text{UV2} \rangle$. Für Vergleiche einzelner Zellen müsste das assoziierte einfaktorielle Design herangezogen werden. Auch hier stellt sich jedoch die Frage, ob Einzelvergleiche in abhängigen Designs sinnvollerweise durchgeführt werden sollten. (vgl. Abschn. 8.4.4).

8.7 Zweifaktorielle Varianzanalyse mit Split-Plot-Design (SPF- $p \cdot q$)

Ein Split-Plot-Design liegt im zweifaktoriellen Fall vor, wenn die Bedingungen eines Zwischen-Gruppen Faktors mit jenen eines Faktors kombiniert werden, bzgl. dessen Stufen abhängige Beobachtungen resultieren – etwa weil es sich um einen Messwiederholungsfaktor handelt. Hat der Zwischen-Gruppen Faktor UV1 p und der Intra-Gruppen Faktor UV2 q Stufen, wobei jede mögliche Stufenkombination auch realisiert wird, spricht man von einem SPF- $p \cdot q$ Design (Split Plot Factorial).

8.7.1 Univariat formulierte Auswertung mit `aov()`

Versuchsplanerisch müssen zunächst Blöcke aus jeweils q Beobachtungen gebildet werden. Die Anzahl der Blöcke muss dabei ein ganzzahliges Vielfaches von p sein, um gleiche Zellbesetzungen erhalten zu können. Im zweiten Schritt werden die einzelnen Blöcke randomisiert auf die Stufen der UV1 verteilt, wobei sich letztlich in jeder der p Stufen dieselbe Anzahl von Blöcken befinden muss. Als weiterer Schritt der Randomisierung wird im Fall der Messwiederholung innerhalb jedes Blocks die Reihenfolge der Beobachtungen bzgl. der q Stufen von UV2 randomisiert. Ein Block kann sich auch aus Beobachtungen unterschiedlicher, z. B. bzgl. relevanter Störvariablen homogener VPn zusammensetzen. Dann ist die Zuordnung der q VPn pro Block zu den q Stufen der UV2 zu randomisieren.

UV1 und die Blockzugehörigkeit sind im SPF- $p \cdot q$ Design konfundiert, da jeder Block nur Beobachtungen aus einer Stufe der UV1 (aber aus allen Stufen der UV2) enthält. Im Vergleich zum RBF- pq Design können deshalb Blockeffekt und Interaktion des Blocks mit der UV1 nicht voneinander isoliert werden. Vom festen Effekt der UV2 lässt sich der Blockeffekt dagegen trennen, weil jeder Block in allen Stufen der UV2 beobachtet wird. Dies hat zur Folge, dass beim Test des Effekts von UV1 eine andere Quadratsumme der Fehler verwendet wird als beim Test des Effekts von UV2.

Wie beim RBF- pq Design müssen die Daten im Long-Format (vgl. Abschn. 3.2.8) inklusive einer Variable vorliegen, die die Blockzugehörigkeit codiert. Diese Variable sei `(BlockNr)` und muss ein Objekt der Klasse `factor` sein. Zudem muss in der Formel explizit angegeben werden, aus welchen additiven Komponenten sich die Quadratsumme innerhalb der Gruppen zusammensetzt. Im konkreten Fall drückt `Error((BlockNr)/(UV2))` aus, dass `(BlockNr)` in `(UV2)` verschachtelt ist, nicht alle möglichen Blöcke also in jeder Bedingung realisiert sind, sondern nur eine Zufallsauswahl (`(BlockNr)` ist ein Random-Faktor).¹² Der Effekt von UV2 ist jeweils innerhalb der durch `(BlockNr)` definierten Blöcke zu analysieren.

¹² Ausgeschrieben lautet der Term `Error((BlockNr)+ (BlockNr):(UV2))`. Dies sind die beiden Effekte, deren Quadratsummen sich zur Quadratsumme innerhalb der Gruppen addieren.

```

> aov(<AV> ~ (<UV1>*<UV2>) + Error(<BlockNr>/<UV2>), data=<Datensatz>)

> nSubj    <- 10                                # Zellbesetzung
> P        <- 3                                 # Anzahl Stufen UV1
> Q        <- 3                                 # Messzeitpunkte UV2
> id       <- factor(rep(1:(P*nSubj), Q))      # Blockzugehörigkeit
> group    <- factor(rep(LETTERS[1:P], Q*nSubj)) # Zwischen-Gruppen UV1
> timeFac  <- factor(rep(1:Q, each=P*nSubj))   # Intra-Gruppen UV2

# Simulation ohne Effekt von UV1, mit Effekt von UV2, ohne Interaktion
> DV_t1 <- round(rnorm(P*nSubj, -0.5, 1), 2)    # AV zu t1
> DV_t2 <- round(rnorm(P*nSubj, 0, 1), 2)        # AV zu t2
> DV_t3 <- round(rnorm(P*nSubj, 0.5, 1), 2)       # AV zu t3

# Datensatz im Long-Format und Auswertung mit aov()
> dfSPFpqL <- data.frame(id, group, timeFac, DV=c(DV_t1,DV_t2,DV_t3))
> aovSPFpq <- aov(DV ~ group*timeFac + Error(id/timeFac), dfSPFpqL)
> summary(aovSPFpq)

Error: id
      Df  Sum Sq Mean Sq F value Pr(>F)
group      2  1.2424  0.6212  0.5533  0.5814
Residuals 27 30.3135  1.1227

Error: id:timeFac
      Df  Sum Sq Mean Sq F value Pr(>F)
timeFac     2  9.494  4.7470  6.2443 0.003635 ***
group:timeFac 4  4.830  1.2074  1.5882 0.190704
Residuals    54 41.052  0.7602

```

Die beim Test eines Effekts jeweils verwendete Quelle der Fehlervarianz wird durch die `Error: <Effekt>` Überschriften kenntlich gemacht, ihre Quadratsumme findet sich in der Zeile `Residuals`. Im Output ist zu erkennen, dass die Quadratsumme des Effekts des Zwischen-Gruppen Faktors `group` mit einer anderen Fehler-Quadratsumme verglichen wird als jene des Effekts vom Intra-Gruppen Faktor `timeFac` und von der Interaktion beider UVn.

8.7.2 Voraussetzungen und Prüfen der Zirkularität

Die statistischen Voraussetzungen im SPF- $p \cdot q$ Design unterscheiden sich z.T. von jenen in der RBF- pq Situation. Für den Test des Haupteffekts des Zwischen-Gruppen Faktors UV1 besteht u.a. die Bedingung der Varianzhomogenität. Sie bezieht sich auf die Daten, die innerhalb jeder Stufe der UV1 durch blockweises Mitteln der Werte über die Stufen der UV2 entstehen. Die so gebildeten Mittelwerte müssen in jeder Stufe der UV1 dieselbe theoretische Varianz besitzen. Es gelten also alle Voraussetzungen wie für eine Varianzanalyse im zugehörigen CR- p Design, die zum Test des Haupteffekts der UV1 im SPF- $p \cdot q$ Design äquivalent ist.

```
# Datensatz aus blockweise gemittelten Werten
> mDf <- aggregate(dfSPFpqL$DV, list(dfSPFpqL$id, dfSPFpqL$group), mean)

# einfaktorielle ANOVA -> äquivalent zum Test der UV1 im SPF-p.q Design
> summary(aov(x ~ Group.2, data=mDf)) # ...
```

Für den Test des Haupteffekts des Intra-Gruppen Faktors UV2 und der Interaktion von UV1 und UV2 gelten folgende Voraussetzungen: Die für jede Stufe von UV1 gebildeten theoretischen Kovarianzmatrizen der UV2 müssen identisch sein. Die Voraussetzung der Zirkularität (vgl. Abschn. 8.4.2) bedeutet, dass diese Matrizen zudem zirkulär sein müssen. Da im Gegensatz zum RBF- pq Design beide Tests auf derselben Fehler-Quadratsumme basieren, gibt es nur eine für beide Tests gültige Korrektur der Freiheitsgrade auf Basis der Schätzungen für ε nach Greenhouse und Geisser bzw. nach Huynh und Feldt.

Für die manuelle Berechnung der Schätzung von ε ist es hier notwendig, zunächst explizit die Interaktion von $\langle \text{BlockNr} \rangle$ und Zwischen-Gruppen Faktor $\langle \text{UV1} \rangle$ zu berechnen, da sie die Fehler liefert, deren Quadratsumme in den Tests von $\langle \text{UV2} \rangle$ und der Interaktion $\langle \text{UV1} \rangle : \langle \text{UV2} \rangle$ im Nenner des F -Bruchs steht.

```
# ersetze AV durch zugehörige Mittelwerte der Zellen, Blöcke, UV1-Stufen
> cellMs <- ave(dfSPFpqL$DV, dfSPFpqL$group, dfSPFpqL$timeFac, FUN=mean)
> blockMs <- ave(dfSPFpqL$DV, dfSPFpqL$id, FUN=mean)
> groupMs <- ave(dfSPFpqL$DV, dfSPFpqL$group, FUN=mean)

# Interaktion  $\langle \text{BlockNr} \rangle : \langle \text{UV2} \rangle$  - Fehler bei Test von  $\langle \text{UV2} \rangle$ ,  $\langle \text{UV1} \rangle : \langle \text{UV2} \rangle$ 
> IDxTF <- dfSPFpqL$DV - blockMs - cellMs + groupMs
> sum(IDxTF^2) # Kontrolle: QS Fehler
[1] 41.05164

# Fehler zu dfSPFpqL hinzufügen, als Matrix im Wide-Format extrahieren
> dfSPFpqL <- cbind(dfSPFpqL, IDxTF)
> errMat <- as.matrix(unstack(dfSPFpqL, IDxTF ~ timeFac))

# Schätzung von epsilon nach Greenhouse + Geisser auf Basis der Fehler
> Serr <- cov(errMat) # Kovarianzmatrix der Fehler
> (epsGG <- (1 / (Q-1)) * sum(diag(Serr))^2 / sum(Serr^2))
[1] 0.9125422

# Schätzung von epsilon nach Huynh + Feldt
> nSubjEps <- nrow(errMat) - (P-1)
> (epsHF <- (nSubjEps * (Q-1) * epsGG - 2) /
+ ((Q-1) * ((nSubjEps-1) - ((Q-1) * epsGG))))
[1] 0.975224
```

8.7.3 Multivariat formulierte Auswertung mit Anova()

Für eine Beschreibung der Anova() Funktion aus dem car Paket sowie für die multivariate Anwendung von anova() vgl. Abschn. 8.4.3 und 9.6. Als wesentlicher Unterschied zur RB- p Situation ergibt sich im SPF- $p \cdot q$ Design die Änderung

des mit `lm(<Formel>)` multivariat spezifizierten Zwischen-Gruppen Designs. In der Formel ist nun auf der rechten Seite der \sim der Zwischen-Gruppen Faktor zu nennen.

```
# Datensatz im Wide-Format
> groupW <- factor(rep(LETTERS[1:P], nSubj))
> dfSPFpqW <- data.frame(groupW, DV_t1, DV_t2, DV_t3)

# Zwischen-Gruppen Design
> modelSPFpq <- lm(cbind(DV_t1, DV_t2, DV_t3) ~ groupW, data=dfSPFpqW)
> intraSPFpq <- data.frame(timeFac=factor(1:Q))      # Intra-Gruppen Design
> library(car)                                         # für Anova()
> AnovaSPFpq <- Anova(modelSPFpq, idata=intraSPFpq, idesign=~timeFac)
> summary(AnovaSPFpq, multivariate=FALSE, univariate=TRUE) # ...
```

Die unkorrigierten Ergebnisse der univariaten Auswertung stimmen mit den von `aov()` ermittelten überein, sofern alle Zellen dieselbe Anzahl an Beobachtungen aufweisen. Das Ergebnis des univariaten Tests für den Intra-Gruppen Faktor ist auch jenes, das man mit der Auswertung des multivariat formulierten Modells mit Hilfe von `anova()` erhält, sofern die Zirkularität der Kovarianzmatrix angenommen wird (Argument `test="Spherical"`, vgl. Fußnote 8). Die Schätzungen von ε sowie die entsprechend korrigierten p -Werte werden ebenfalls ausgegeben. Die Spezifizierung des Designs mit den Argumenten `M` und `X` setzt Kenntnisse des Allgemeinen Linearen Modells voraus.

```
# Test Intra-Gruppen Faktor timeFac sowie Interaktion group*timeFac
> anova(modelSPFpq, X=~ 1, idata=intraSPFpq, test="Spherical")      # ...

# Test des Zwischen-Gruppen Faktors group
> anova(modelSPFpq, M=~ 1, idata=intraSPFpq, test="Spherical")      # ...
```

8.7.4 Einzelvergleiche (Kontraste)

Prinzipiell ist es möglich, auch für ein SPF- $p \cdot q$ Design Einzelvergleiche als Tests von Linearkombinationen von Erwartungswerten durchzuführen. Dabei ist die Situation analog zu jener im CRF- pq Design, wie sie in Abschn. 8.5.3 beschrieben wurde. Auch hier wären zunächst Vergleiche der mittleren Erwartungswerte des Zwischen-Gruppen Faktors (A -Kontraste) bzw. des Intra-Gruppen Faktors (B -Kontraste) von Interaktionskontrasten (I -Kontraste) zu unterscheiden. Im Vergleich zum CRF- pq Design ergeben sich jedoch Unterschiede bei den jeweils zu verwendenden Quadratsummen der Fehler, die dort immer dieselbe ist. Die Effekt-Quadratsumme eines A -Kontrasts wäre dagegen im SPF- $p \cdot q$ Design gegen die Quadratsumme `Error: <BlockNr>` zu testen, ein B -Kontrast sowie ein Interaktionskontrast entsprechend gegen die Quadratsumme `Error: <BlockNr>:<UV2>`. Auch hier stellt sich jedoch die Frage, ob Einzelvergleiche sinnvollerweise durchgeführt werden sollten (vgl. Abschn. 8.4.4).

8.7.5 Erweiterung auf dreifaktorielles SPF- $p \cdot qr$ Design

Die varianzanalytische Auswertung eines dreifaktoriellen Designs mit einer Zwischen-Gruppen UV und zwei Intra-Gruppen Faktoren (SPF- $p \cdot qr$) unterscheidet sich nur wenig von jener im SPF- $p \cdot q$ Design. Abgesehen von der etwas komplizierteren Datenstruktur ist bei Verwendung von aov() nur die Formel mit den zu berücksichtigenden Effekten anzupassen.

```

> nSubj      <- 10                                # Zellbesetzung
> P          <- 2                                # Anzahl Stufen UV1
> Q          <- 3                                # Messzeitpunkte UV2
> R          <- 2                                # Messzeitpunkte UV3
> id         <- factor(rep(1:(P*nSubj), Q*R))    # Blockzugehörigkeit
> group      <- factor(rep(LETTERS[1:P], Q*R*nSubj))  # Zwischen-Gr. UV1
> timeFac1   <- factor(rep(1:Q, each=P*R*nSubj))    # Intra-Gruppen UV2
> timeFac2   <- factor(rep(rep(1:R, each=P*nSubj), Q)) # Intra-Gruppen UV3

# Simulation ohne Effekt UV1, mit Effekt UV2 und UV3, ohne Interaktion
> DV_t11 <- round(rnorm(P*nSubj, 8, 2), 2)        # AV zu t1-1
> DV_t12 <- round(rnorm(P*nSubj, 10, 2), 2)        # AV zu t1-2
> DV_t21 <- round(rnorm(P*nSubj, 13, 2), 2)        # AV zu t2-1
> DV_t22 <- round(rnorm(P*nSubj, 15, 2), 2)        # AV zu t2-2
> DV_t31 <- round(rnorm(P*nSubj, 13, 2), 2)        # AV zu t3-1
> DV_t32 <- round(rnorm(P*nSubj, 15, 2), 2)        # AV zu t3-2

# Datensatz im Long-Format und Auswertung mit aov()
> dfSPFp.qrL <- data.frame(id, group, timeFac1, timeFac2,
+                             DV=c(DV_t11, DV_t12, DV_t21, DV_t22, DV_t31, DV_t32))

> aovSPFp.qr <- aov(DV ~ group*timeFac1*timeFac2
+                     + Error(id/(timeFac1*timeFac2)), data=dfSPFp.qrL)

> summary(aovSPFp.qr)                               # ...

```

Wird die Anova() Funktion aus dem car Paket eingesetzt, sind das Zwischen-Gruppen Design sowie die Intra-Gruppen Struktur zu ändern.

```

# Datensatz im Wide-Format
> groupW    <- factor(rep(LETTERS[1:P], nSubj))
> dfSPFp.qrW <- data.frame(groupW, DV_t11, DV_t12, DV_t21, DV_t22,
+                            DV_t31, DV_t32)

# Zwischen-Gruppen Design
> modelSPFp.qr <- lm(cbind(DV_t11, DV_t12, DV_t21, DV_t22, DV_t31,
+                            DV_t32) ~ groupW, data=dfSPFp.qrW)

# Intra-Gruppen Design
> (intraSPFp.qr <- data.frame(timeFac1=factor(rep(1:Q, each=R)),
+                                 timeFac2=factor(rep(1:R, Q))))
  timeFac1  timeFac2

```

```

1      1      1
2      1      2
3      2      1
4      2      2
5      3      1
6      3      2

> library(car) # für Anova()
> AnovaSPFp.qr <- Anova(modelSPFp.qr, idata=intraSPFp.qr,
+                         idesign=~timeFac1*timeFac2)

> summary(AnovaSPFp.qr, multivariate=FALSE, univariate=TRUE)          # ...

```

8.7.6 Erweiterung auf dreifaktorielles SPF- $pq \cdot r$ Design

Die varianzanalytische Auswertung eines dreifaktoriellen Designs mit zwei Zwischen-Gruppen UVn und einem Intra-Gruppen Faktor (SPF- $pq \cdot r$) mit `aov()` bringt ebenfalls eine etwas komplexere Datenstruktur mit sich und macht eine Anpassung der Formel zur Spezifizierung des Modells notwendig.

```

> nSubj   <- 10                                # Zellbesetzung
> P       <- 2                                  # Anzahl Stufen UV1
> Q       <- 2                                  # Anzahl Stufen UV2
> R       <- 3                                  # Messzeitpunkte UV3
> id      <- factor(rep(1:(P*Q*nSubj), R))    # Blockzugehörigkeit
> IV1     <- factor(rep(1:P, Q*R*nSubj))      # Zwischen-Gr. UV1
> IV2     <- factor(rep(rep(1:Q, each=P*nSubj), R)) # Zwischen-Gr. UV2
> timeFac <- factor(rep(1:R, each=P*Q*nSubj))  # Intra-Gruppen UV3

# Simulation ohne Effekt von UV1, UV2, mit Effekt UV3, ohne Interaktion
> DV_t1 <- round(rnorm(P*Q*nSubj, -3, 2), 2)    # AV zu t1
> DV_t2 <- round(rnorm(P*Q*nSubj, 1, 2), 2)      # AV zu t2
> DV_t3 <- round(rnorm(P*Q*nSubj, 2, 2), 2)      # AV zu t3

# Datensatz im Long-Format und Auswertung mit aov()
> dfSPFpq.rL <- data.frame(id, IV1, IV2, timeFac,
+                             DV=c(DV_t1, DV_t2, DV_t3))

> aovSPFpq.r <- aov(DV ~ IV1*IV2*timeFac + Error(id/timeFac), dfSPFpq.rL)
> summary(aovSPFpq.r)                           # ...

```

Wird die `Anova()` Funktion aus dem `car` Paket eingesetzt, sind das Zwischen-Gruppen Design sowie die Intra-Gruppen Struktur zu ändern.

```

# Datensatz im Wide-Format
> IV1W      <- factor(rep(LETTERS[1:P], Q*nSubj))
> IV2W      <- factor(rep(c("+", "-"), each=P*nSubj))
> dfSPFpq.rW <- data.frame(IV1W, IV2W, DV_t1, DV_t2, DV_t3)

# Zwischen-Gruppen Design
> modelSPFpq.r <- lm(cbind(DV_t1,DV_t2,DV_t3) ~ IV1W*IV2W, dfSPFpq.rW)

```

```
> intraSPFpq.r <- data.frame(timeFac=factor(1:R)) # Intra-Gruppen Design
> library(car) # für Anova()
> AnovaSPFpq.r <- Anova(modelSPFpq.r, idata=intraSPFpq.r,
+                         idesign=~ timeFac)

> summary(AnovaSPFpq.r, multivariate=FALSE, univariate=TRUE) # ...
```

8.8 Kovarianzanalyse

Bei der Kovarianzanalyse (ANCOVA) werden die Daten einer Abhängigen Variable in einem linearen Modell aus quantitativen und kategorialen Variablen vorhergesagt. Sie stellt damit eine Kombination von linearer Regression und Varianzanalyse dar. Die Modellierung erfolgt mit den aus Regressions- und Varianzanalyse bekannten Funktionen, wobei in der $\langle AV \rangle \sim \langle UV \rangle$ Formel in der Rolle von $\langle UV \rangle$ sowohl quantitative Variablen wie Faktoren als Vorhersageterme auftauchen.

8.8.1 Test der Effekte von Gruppenzugehörigkeit und Kovariate

Im folgenden sei die Situation mit einer quantitativen und einer kategorialen UV vorausgesetzt. Die kategoriale UV wird dabei als Treatment-Variable, die quantitative Variable als Kovariate bezeichnet. Aus der Perspektive der Regression kann die Kovarianzanalyse als Prüfung betrachtet werden, ob die theoretischen Parameter des in jeder Treatment-Gruppe gültigen Regressionsmodells mit der Kovariate als Prädiktor und der AV als Kriterium identisch sind.

Aus der Perspektive der Varianzanalyse ist man an der Wirkung der Treatment-Variable interessiert, hält jedoch den Einfluss einer quantitativen Störvariable für möglich, der in jeder Treatment-Gruppe als linear angenommen wird. Da der Einfluss der Störvariable die Heterogenität der AV-Werte innerhalb jeder Gruppe erhöht, scheint es erstrebenswert, die Einflüsse der Treatment-UV und der Kovariate voneinander trennen zu können. Dies ist stärker noch der Fall, wenn davon ausgegangen werden muss, dass die Verteilung der Störvariable in den Treatment-Gruppen nicht identisch, eine systematische Konfundierung beider Einflüsse also möglich ist.

Meist wird die Fragestellung dahingehend eingeschränkt, dass in den Treatment-Gruppen nur unterschiedliche y -Achsenabschnitte der Regressionsgleichung zugelassen, die Steigungen dagegen als identisch vorausgesetzt werden. Die Treatment-Wirkung wäre dann in den Unterschieden der y -Achsenabschnitte erkennbar.

Als Beispiel diene jenes aus Maxwell und Delaney (2004, p. 429 ff.): An Depressionskranken sei ein Maß der Schwere ihrer Krankheit vor und nach einer therapeutischen Intervention erhoben worden, bei der es sich entweder um ein Medikament mit sog. SSRI Wirkstoff (Selective Serotonin Re-uptake Inhibitor), um ein Placebo oder um den Verbleib auf einer Warteliste handelt. Die Vorher-Messung soll als Kovariate für die entscheidende Nachher-Messung dienen.

```
# Schweregrad in den Bedingungen bei Vorher- und Nachher-Messung
> SSRIpre <- c(18, 16, 16, 15, 14, 20, 14, 21, 25, 11) # SSRI vor
> SSRIpost <- c(12, 0, 10, 9, 0, 11, 2, 4, 15, 10) # SSRI nach
> PlacPre <- c(18, 16, 15, 14, 20, 25, 11, 25, 11, 22) # Placebo vor
> PlacPost <- c(11, 4, 19, 15, 3, 14, 10, 16, 10, 20) # Placebo nach
> WLpre <- c(15, 19, 10, 29, 24, 15, 9, 18, 22, 13) # Warteliste vor
> WLpost <- c(17, 25, 10, 22, 23, 10, 2, 10, 14, 7) # Warteliste nach
> nSubj <- length(SSRIpre) # VPn pro Gruppe

# Faktor der Gruppenzugehörigkeiten
> IV <- factor(rep(1:3, each=nSubj), labels=c("SSRI", "Placebo", "WL"))
> P <- nlevels(IV) # Anzahl Bedingungen
> DVpre <- c(SSRIpre, PlacPre, WLpre) # alle prä-Messungen
> DVpost <- c(SSRIpost, PlacPost, WLpost) # alle post-Messungen
> dfAncova <- data.frame(id=1:(P*nSubj), IV, DVpre, DVpost) # Datensatz
```

Als Veranschaulichung wird die Verteilung der Nachher-Werte in den Gruppen durch Boxplots dargestellt, um daraufhin mit `anova()` die Varianzanalyse zunächst ohne, dann die Kovarianzanalyse mit Kovariate zu berechnen (Abb. 8.5, vgl. Abschn. 10.6.3). Dabei sei vorausgesetzt, dass der Steigungsparameter in allen Gruppen identisch ist, im Modell findet also kein Interaktionsterm von Treatment-Variable und Kovariate Berücksichtigung. Während der Gruppeneffekt ohne Kovariate nicht signifikant getestet wird, fällt sowohl der Test des Gruppeneffekts als auch jener der Kovariate in der Kovarianzanalyse signifikant aus.

```
> plot(DVpost ~ IV, main="Boxplots der Scores für jede Gruppe")
# vollständiges Modell, ohne Kovariate, ohne Gruppierungsfaktor
> modelFull <- lm(DVpost ~ IV + DVpre, data=dfAncova)
> modelGrp <- lm(DVpost ~ IV, data=dfAncova)
```

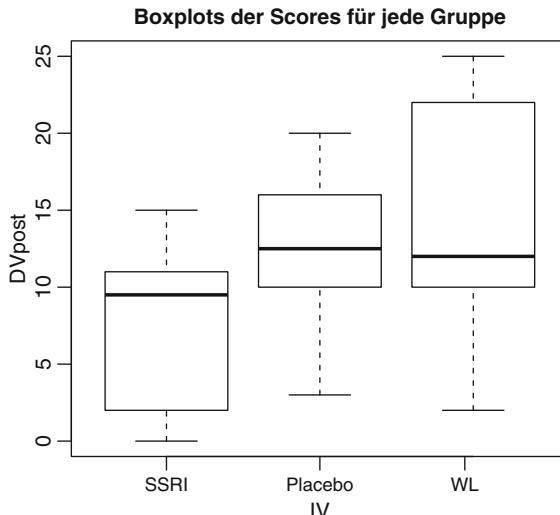


Abb. 8.5 Kovarianzanalyse: Boxplots zum Vergleich der AV-Verteilungen in den Gruppen

```
> modelRegr <- lm(DVpost ~ DVpre, data=dfAncova)
> anova(modelGrp) # Test ohne Kovariate
Analysis of Variance Table
Response: DVpost
Df Sum Sq Mean Sq F value Pr(>F)
IV 2 240.47 120.233 3.0348 0.06473 .
Residuals 27 1069.70 39.619

> anova(modelFull) # Test mit Kovariate, QS Typ I
Analysis of Variance Table
Response: DVpost
Df Sum Sq Mean Sq F value Pr(>F)
IV 2 240.47 120.23 4.1332 0.027629 *
DVpre 1 313.37 313.37 10.7723 0.002937 **
Residuals 26 756.33 29.09
```

Zur Berechnung von Quadratsummen vom Typ III kann zum einen auf die Anova() Funktion aus dem car Paket zurückgegriffen werden. Zum anderen lassen sich diese Quadratsummen mit anova() durch den Test zweier geeigneter Modelle gegeneinander ermitteln (vgl. Abschn. 8.5.2): für den Effekt der Kovariaten sind dies auf der einen Seite das Modell ohne Kovariate als Vorhersageterm, auf der anderen Seite das vollständige Modell. Für den Effekt der Treatment-Variable entsprechend auf der einen Seite das Modell ohne Treatment-Variable als Vorhersageterm, auf der anderen Seite das vollständige Modell.

```
> library(car) # QS Typ III mit Anova()
> Anova(modelFull, type="III") # Test Treatment und Kovariate
Anova Table (Type III tests)
Response: DVpost
Sum Sq Df F value Pr(>F)
(Intercept) 27.83 1 0.9567 0.337029
IV 217.15 2 3.7324 0.037584 *
DVpre 313.37 1 10.7723 0.002937 **
Residuals 756.33 26

# Quadratsummen vom Typ III über Modellvergleiche
> anova(modelRegr, modelFull) # Test Treatment ...
> anova(modelGrp, modelFull) # Test Kovariate ...
```

Die Ergebnisse lassen sich auch manuell nachvollziehen, indem die Fehler-Quadratsummen für das vollständige Modell, das Regressionsmodell ohne Treatment-Variable und das ANOVA-Modell ohne Kovariate berechnet werden. Die Effekt-Quadratsummen vom Typ III ergeben sich dann jeweils als Differenz der Fehler-Quadratsummen des Modells, in dem der Effekt nicht berücksichtigt wird und der Fehler-Quadratsumme des vollständigen Modells.

```
> X <- DVpre # kürzerer Name Kovariate
> Y <- DVpost # kürzerer Name AV
> XMs <- tapply(X, IV, mean) # Gruppenmittel Kovariate
> YMs <- tapply(Y, IV, mean) # Gruppenmittel AV
> N <- length(Y) # Gesamtanzahl VPn
```

```

# Fehler-Quadratsumme vollständiges Modell
# zunächst gruppenweise zentrierte Daten der Kovariate und AV
> Xctr      <- X - ave(X, IV, FUN=mean)    # zentrierte Kovariate
> Yctr      <- Y - ave(Y, IV, FUN=mean)    # zentrierte AV
> bFull     <- cov(Xctr, Yctr) / var(Xctr)  # b-Gewicht (alle Gruppen)
> aFull     <- YM - bFull*XMs              # y-Achsenabschnitte
> YhatFull  <- bFull*X + aFull[IV]         # Vorhersage
> SSEfull   <- sum((Y-YhatFull)^2)          # Fehler-QS
> dfSSEfull <- N-P-1                        # Freiheitsgrade Fehler-QS
> (MSEfull  <- SSEfull / dfSSEfull)        # mittlere Fehler-QS
[1] 29.0898

# Fehler-Quadratsumme Regressionsmodell ohne Treatment-Variable
> bRegr     <- cov(X, Y) / var(X)           # b-Gewicht
> aRegr     <- mean(Y) - bRegr*mean(X)       # y-Achsenabschnitt
> YhatRegr  <- bRegr*X + aRegr             # Vorhersage
> SSEregr   <- sum((Y-YhatRegr)^2)          # Fehler-QS
> (MSEregr  <- SSEregr / (N-2))            # mittlere Fehler-QS
[1] 34.76729

# Fehler-Quadratsumme ANOVA-Modell ohne Kovariate
> grpNs    <- tapply(Y, IV, length)         # Gruppengrößen
> grpVs    <- tapply(Y, IV, var)             # Gruppenvarianzen
> grandM   <- sum((grpNs/N) * YM)           # Gesamt-Mittel
> SSEgrp   <- sum((grpNs-1) * grpVs)        # Fehler-QS
> (MSEgrp  <- SSEgrp / (N-P))              # mittlere Fehler-QS
[1] 39.61852

# Effekt-Quadratsummen und F-Werte der zugehörigen Tests
> SSregr   <- SSEgrp-SSEfull                # Effekt-QS Kovariate
> (MSregr <- SSregr / 1)                     # df = 1
[1] 313.3653

> (Fegr <- MSregr / MSEfull)                 # F-Wert Kovariate
[1] 10.77234

> SSgrp   <- SSEregr-SSEfull                # Effekt-QS Treatment
> (MSgrp <- SSgrp / (P-1))                  # df = P-1
[1] 108.5747

> (Fgrp  <- MSgrp / MSEfull)                 # F-Wert Treatment
[1] 3.732399

```

Mit `summary(lm())` lassen sich die in den verschiedenen Gruppen angepassten Regressionsparameter ausgeben und einzeln auf Signifikanz testen.

```

> (sumRes <- summary(modelFull))
Call:
lm(formula = DVpost ~ IV + DVpre, data = dfAncova)

```

Residuals:

	Min	1Q	Median	3Q	Max
	-10.6842	-3.9615	0.6448	3.8773	9.9675

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	-3.6704	3.7525	-0.978	0.33703
IVPlacebo	4.4483	2.4160	1.841	0.07703 .
IVWL	6.4419	2.4133	2.669	0.01292 *
DVpre	0.6453	0.1966	3.282	0.00294 **

Residual standard error: 5.393 on 26 degrees of freedom
 Multiple R-squared: 0.4227, Adjusted R-squared: 0.3561
 F-statistic: 6.346 on 3 and 26 DF, p-value: 0.002252

Die unter Coefficients aufgeführten Testergebnisse sind so zu interpretieren, dass die SSRI-Gruppe als Referenzgruppe verwendet wurde, da sie die erste Faktorstufe in IV darstellt. Ihre Koeffizienten finden sich in der Zeile (Intercept). Für diese Gruppe ist der unter Estimate genannte Wert der y-Achsenabschnitt der Regressionsgerade. Die Estimate Werte für die Gruppen Placebo und WL geben jeweils die Differenz des y-Achsenabschnitts in dieser Gruppe im Vergleich zur Referenzgruppe an. Der in der letzten Spalte genannte *p*-Wert gibt Auskunft auf die Frage, ob dieser Unterschied signifikant von 0 verschieden ist. Die für alle Gruppen identische Steigung ist als Estimate für die Kovariate DVpre abzulesen. Ob sie signifikant von 0 verschieden ist, ergibt sich aus dem in der letzten Spalte genannten *p*-Wert. Verantwortlich für die Art des Gruppenvergleichs zu einer Referenzstufe ist die Dummy-Codierung der Gruppierungsvariable, um aus ihr einen Prädiktor zu machen (vgl. Abschn. 8.3.1).

Eine graphische Veranschaulichung des linearen Zusammenhangs zwischen Vorher- und Nachher-Messwert in den einzelnen Gruppen erfolgt in Abb. 8.6.

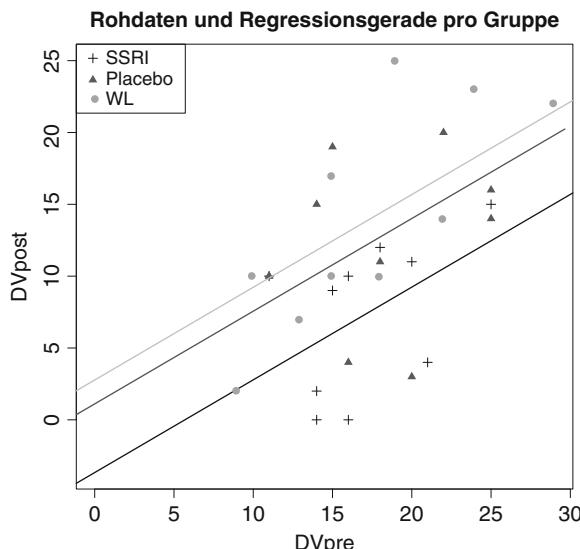


Abb. 8.6 Kovarianzanalyse: nach Gruppen getrennte Regressionsgeraden

```

# Steigung und y-Achsenabschnitte der Regressionsgeraden extrahieren
> iCeptSSRI <- sumRes$coefficients[1, 1] # Referenzgruppe
> iCeptPlac <- sumRes$coefficients[2, 1] + iCeptSSRI
> iCeptWL <- sumRes$coefficients[3, 1] + iCeptSSRI
> slopeAll <- sumRes$coefficients[4, 1]

# Darstellungsbereich für x- und y-Achse wählen und Punktwolke darstellen
> xLims <- c(0, max(dfAncova$DVpre))
> yLims <- c(min(iCeptSSRI, iCeptPlac, iCeptWL), max(dfAncova$DVpost))
> plot(DVpost ~ DVpre, data=dfAncova, xlim=xLims, ylim=yLims,
+       pch=rep(c(3, 17, 19), each=nSubj), col=rep(c("red", "green", "blue"),
+       each=nSubj), main="Rohdaten und Regressionsgerade pro Gruppe")

> legend(x="topleft", legend=levels(group), pch=c(3, 17, 19),
+         col=c("red", "green", "blue"))

# Regressionsgeraden einfügen
> abline(iCeptSSRI, slopeAll, col="red")
> abline(iCeptPlac, slopeAll, col="green")
> abline(iCeptWL, slopeAll, col="blue")

```

Sollen in der Kovarianzanalyse die Steigungen der Regressionsgeraden in den Gruppen nicht als identisch festgelegt, sondern auch bzgl. dieses Parameters Gruppenunterschiede zugelassen werden, lautet das Modell:

```
> summary(lm(DVpost ~ IV + DVpre + IV:DVpre, data=dfAncova)) # ...
```

8.8.2 Beliebige *a-priori* Kontraste

Ähnlich wie bei Varianzanalysen lassen sich bei Kovarianzanalysen spezifische Vergleiche zwischen experimentellen Bedingungen in der Form von Kontrasten, also Linearkombinationen von Gruppenerwartungswerten testen (vgl. Abschn. 8.3.6). Die Kovariate findet dabei Berücksichtigung, indem hier der Vergleich zwischen sog. korrigierten Erwartungswerten der Abhängigen Variable stattfindet: auf der empirischen Ebene müssen für deren Schätzung zunächst die Regressionsparameter pro Gruppe bestimmt werden, wobei wie oben das *b*-Gewichte konstant sein und nur die Variation des *y*-Achsenabschnitts zwischen den Gruppen zugelassen werden soll. Der Gesamtmittelwert der Kovariate über alle Gruppen hinweg wird nun pro Gruppe in die ermittelte Regressionsgleichung eingesetzt. Das Ergebnis ist der korrigierte Gruppenmittelwert, der ausdrücken soll, welcher Wert in der AV zu erwarten wäre, wenn alle Personen denselben Wert auf der Kovariate (nämlich deren Mittelwert) hätten und sich nur in der Gruppenzugehörigkeit unterscheiden würden.

Im Beispiel werden vier Kontraste zunächst mit `glht()` aus dem `multcomp` Paket ohne α -Adjustierung getestet, dann manuell mit α -Adjustierung nach Bonferroni. Dafür ist es notwendig, die zugehörigen Kontrastvektoren als (ggf. benannte) Zeilen einer Matrix zusammenzustellen.

```
# Matrix der Kontrastkoeffizienten
> cntrMat <- rbind( "SSRI-Placebo" = c(-1, 1, 0),
```

```

+
  "SSRI-WL"      = c(-1, 0, 1),
+
  "Placebo-WL"   = c( 0, -1, 1),
+
  "SSRI-0.5(P+WL)" = c(-2, 1, 1))

> library(multcomp)                      # Auswertung mit glht()
> aovAncova <- aov(DVpost ~ IV + DVpre, data=dfAncova)
> summary(glht(aovAncova, linfct=mcp(IV=cntrMat),
+               alternative="greater"), test=adjusted("none"))
Simultaneous Tests for General Linear Hypotheses
Multiple Comparisons of Means: User-defined Contrasts
Fit: aov(formula = DVpost ~ IV + DVpre, data = dfAncova)
Linear Hypotheses:
Estimate Std. Error t value Pr(>|t|)
SSRI-Placebo <= 0     4.448     2.416   1.841  0.03852 *
SSRI-WL <= 0       6.442     2.413   2.669  0.00646 **
Placebo-WL <= 0      1.994     2.413   0.826  0.20808
SSRI-0.5(P+WL) <= 0 10.890     4.183   2.603  0.00753 **

---
(Adjusted p values reported -- none method)

# manuelle Berechnung mit alpha-Adjustierung nach Bonferroni
# quadrierte Längen der Kontrastvektoren
> lenSqs    <- (1/nSubj) * cntrMat^2 %*% (rep(1, ncol(cntrMat)))
> alphaAdj  <- 0.05/nrow(cntrMat)           # Bonferroni-Adjustierung
> (YMsAdj   <- bFull*mean(X) + aFull)        # korrigierte Gruppenmittel
      SSRI   Placebo     WL
7.536616 11.984895 13.978489

> psiHats   <- cntrMat %*% YMsAdj          # Kontrast-Schätzungen
> fracs     <- (cntrMat %*% XMs)^2 / sum((X-ave(X, IV, FUN=mean))^2)
> varPsiHats <- MSEfull * (lenSqs + fracs)    # Varianz der Schätzungen
> sqrt(varPsiHats)
 [,1]
SSRI-Placebo  2.415968
SSRI-WL        2.413326
Placebo-WL     2.412766
SSRI-0.5(P+WL) 4.183378

> statTs <- psiHats / sqrt(varPsiHats)        # Teststatistiken
> critTs <- rep(qt(1-alphaAdj, dfSSEfull), nrow(cntrMat)) # krit. Wert
> pValTs <- 1-pt(abs(statTs), dfSSEfull)      # p-Werte einseitig
> (resDf <- data.frame(statTs, critTs, pVals=pValTs,
+                         alphaAdj=rep(alphaAdj, nrow(cntrMat)), sig=abs(statTs)>critTs))
      statTs    critTs    pVals alphaAdj    sig
SSRI-Placebo  1.8411994 2.378786 0.038515271  0.0125 FALSE
SSRI-WL        2.6692923 2.378786 0.006461541  0.0125 TRUE
Placebo-WL     0.8262695 2.378786 0.208084776  0.0125 FALSE
SSRI-0.5(P+WL) 2.6031958 2.378786 0.007529061  0.0125 TRUE

```

Im abschließend ausgegebenen Datensatz wird für jeden Kontrast der empirische Wert der Teststatistik, der (für alle identische) kritische t -Wert, der jeweils zugehörige

rige p -Wert, das adjustierte α sowie schließlich das Ergebnis der Signifikanzprüfung im Sinne des Vergleichs von p -Wert und adjustiertem α aufgeführt.

8.8.3 Beliebige post-hoc Kontraste nach Scheffé

Beliebige Kontraste können auch im Anschluss an eine signifikante Kovarianzanalyse getestet werden, die implizit simultan alle möglichen Kontraste prüft – spezifische Hypothesen liegen also bei ihrer Anwendung nicht vor. Aus diesem Grund muss im Anschluss an eine Kovarianzanalyse bei Einzeltests eine geeignete α -Adjustierung vorgenommen werden, hier vorgestellt nach der Methode von Scheffé.

Zunächst gilt für das Aufstellen eines Kontrasts alles bereits für beliebige a-priori Kontraste Ausgeführte. Lediglich die Wahl des kritischen Wertes weicht ab und ergibt sich zur α -Adjustierung aus einer F -Verteilung. Dieser kritische Wert ist mit dem Quadrat der a-priori Teststatistik zu vergleichen – die etwa in der Ausgabe von `summary(glht())` abgelesen werden kann. Hier sollen dieselben Kontraste wie im a-priori Fall gerichtet getestet werden.

```
> dfSSgrp <- P-1                                     # Freiheitsgrade Treatment-QS
> statFs   <- psiHats^2 / varPsiHats                # quadrierte Teststatistiken

# kritischer F-Wert für quadrierte Teststatistiken
> critFs <- rep(dfSSgrp*qf(1-0.05, dfSSgrp, dfSSEfull), nrow(cntrMat))
> pValFs <- 1-pf(statFs/dfSSgrp, dfSSgrp, dfSSEfull) # p-Werte einseitig
> (resDf <- data.frame(statFs, critFs, pVals=pValFs,
+                         alpha=rep(0.05, nrow(cntrMat)), sig=statFs>critFs))
      statFs     critFs    pVals alpha    sig
SSRI-Placebo 3.3900154 6.738033 0.20326188 0.05 FALSE
SSRI-WL       7.1251213 6.738033 0.04291472 0.05 TRUE
Placebo-WL    0.6827212 6.738033 0.71394043 0.05 FALSE
SSRI-0.5(P+WL) 6.7766283 6.738033 0.04923999 0.05 TRUE
```

Im abschließend ausgegebenen Datensatz wird für jeden Kontrast der empirische Wert der Teststatistik, der (für alle identische) kritische F -Wert, der jeweils zugehörige p -Wert, das α -Niveau sowie schließlich das Ergebnis der Signifikanzprüfung im Sinne des Vergleichs von p -Wert und α aufgeführt.

8.9 Power und notwendige Stichprobengrößen berechnen

Mit Hilfe der Funktionen von Zufallsvariablen (vgl. Abschn. 5.2) lässt sich die Power der vorgestellten inferenzstatistischen Tests berechnen, sofern eine exakte Alternativhypothese (H_1) vorliegt. Hierfür ist es notwendig, zunächst auf Basis der Verteilung der Teststatistik unter der Nullhypothese (H_0) mit der Quantilfunktion `q(Funktionsfamilie)()` den kritischen Wert zu bestimmen. Mit Hilfe der zugehörigen Verteilungsfunktion `p(Funktionsfamilie)()` kann dann unter Gültigkeit der Alternativhypothese die Wahrscheinlichkeit dafür berechnet werden, dass die Teststatistik Werte größer als dieser kritische Wert annimmt.

8.9.1 Binomialtest

Im Beispiel soll zunächst der Fall eines rechtsseitigen Binomialtests betrachtet werden. Die Punktwahrscheinlichkeiten der einzelnen Ereignisse bei Gültigkeit von H_0 und H_1 sind zusammen mit dem kritischen Wert in Abb. 8.7 dargestellt.

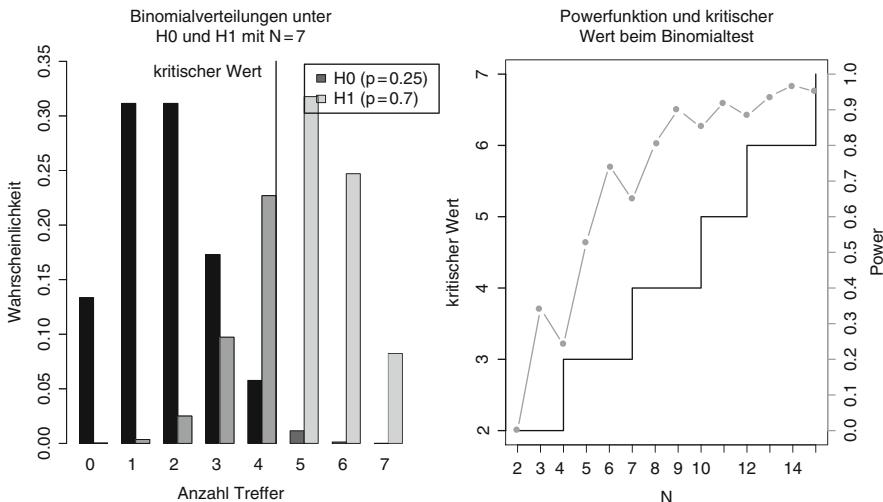


Abb. 8.7 Binomialverteilung: Wahrscheinlichkeiten von Treffern unter H_0 und H_1 sowie kritischer Wert. Powerfunktion und kritischer Wert in Abhängigkeit von der Stichprobengröße

```
> N <- 7 # Stichprobengröße
> pH0 <- 0.25 # Wahrscheinlichkeit Treffer unter H0
> pH1 <- 0.7 # Wahrscheinlichkeit Treffer unter H1
> alpha <- 0.05 # Signifikanzniveau
> (critB <- qbinom(1-alpha, N, pH0)) # kritischer Wert
[1] 4

> (powB <- 1-pbinom(critB, N, pH1)) # Power für konkrete H0, H1, N
[1] 0.6470695

> sum(dbinom(5:7, N, pH1)) # Kontrolle: Summe Einzelwahrscheinlichkeiten
[1] 0.6470695

# Säulendiagramm zur Veranschaulichung der Wahrscheinlichkeitsfunktionen
> dH0 <- dbinom(0:N, N, pH0) # Verteilung unter H0
> dH1 <- dbinom(0:N, N, pH1) # Verteilung unter H1
> mat <- rbind(dH0, dH1)
> rownames(mat) <- c("H0 (p=0.25)", "H1 (p=0.7)")
> colnames(mat) <- 0:N
> barsX <- barplot(mat, beside=TRUE, ylim=c(0, 0.35),
+ xlab="Anzahl Treffer", ylab="Wahrscheinlichkeit",
+ main="Binomialverteilungen unter H0 und H1 mit N=7",
+ col=c(rgb(1, 0.2, 0.2, 0.7), rgb(0.3, 0.3, 1, 0.6))),
```

```

+     names.arg=colnames(mat), legend.text=rownames(mat))

> barplot(mat[, 1:(critB+1)], beside=TRUE, ylim=c(0, 0.35),
+           col=c("red", "blue"), add=TRUE)

# Hilfslinie für kritischen Wert
> xx <- barsX[2, critB+1] + (barsX[1, critB+2] - barsX[2, critB+1])/4
> abline(v=xx, col="green", lwd=2)
> text(xx-0.4, 0.34, adj=1, label="kritischer Wert")

```

Beim diskreten Binomialtest ist die Power keine monotone Funktion der Stichprobengröße, sondern kann auch sinken, wenn der Test bei fester H_0 und H_1 mit Daten von mehr Beobachtungsobjekten durchgeführt wird. Dies liegt an der Veränderung des kritischen Wertes, die zusammen mit der Powerfunktion in Abb. 8.7 abgebildet ist.

```

> Nvec      <- 2:15                      # betrachteter Bereich für N
> critBvec <- qbinom(1-alpha, size=Nvec, prob=pH0)    # kritische Werte
> powBvec   <- 1-pbinom(critBvec, size=Nvec, prob=pH1)  # zugehörige Power

# Veranschaulichung des Verlaufs der kritischen Werte
> op <- par(mar=c(5, 4, 4, 4))          # breiterer rechter Rand
> plot(Nvec, critBvec, xlab="N", xaxt="n", yaxt="n", lwd=2, type="s",
+       pch=16, col="red", main="Powerfunktion und kritischer Wert
+ beim Binomialtest", ylab="kritischer Wert")

# zeichne Achsen separat ein
> axis(side=1, at=seq(Nvec[1], Nvec[length(Nvec)], by=1))
> axis(side=2, at=seq(min(critBvec), max(critBvec), by=1), col="red")
> par(new=TRUE)                         # füge Powerfunktion hinzu
> plot(Nvec, powBvec, ylim=c(0, 1), type="b", lwd=2, pch=16,
+       col="blue", xlab=NA, ylab=NA, axes=FALSE)

# zeichne rechte Achse mit Achsenbeschriftung separat ein
> axis(side=4, at=seq(0, 1, by=0.1), col="blue")
> mtext(text="Power", side=4, line=3, cex=1.4)
> par(op)                               # stelle ursprüngliche Ränder wieder her

```

8.9.2 t-Test

Für einen t -Test mit einer Stichprobe ist zur Bestimmung der Verteilung der Teststatistik unter der H_1 die Berechnung des Nonzentralitätsparameters δ erforderlich, für den die wahre Streuung sowie der Erwartungswert unter H_0 und H_1 bekannt sein muss.¹³ Abbildung 8.8 zeigt die Verteilungen von t für das gegebene Hypo-

¹³ Für zwei unabhängige Stichproben mit Gruppengrößen n_1 und n_2 und Erwartungswerten μ_1 und μ_2 unter H_1 würde sich δ als $\sqrt{(n_1*n_2)/(n_1+n_2)}*((\mu_2-\mu_1)/\sigma)$ berechnen. Für zwei abhängige Stichproben des jeweiligen Umfangs n mit wahren Streuungen σ_1 und σ_2 sowie der wahren Korrelation ρ wäre δ gleich $\sqrt{n} * ((\mu_2-\mu_1)/\sqrt(\sigma_1^2 + \sigma_2^2 + 2*\rho*\sigma_1*\sigma_2))$.

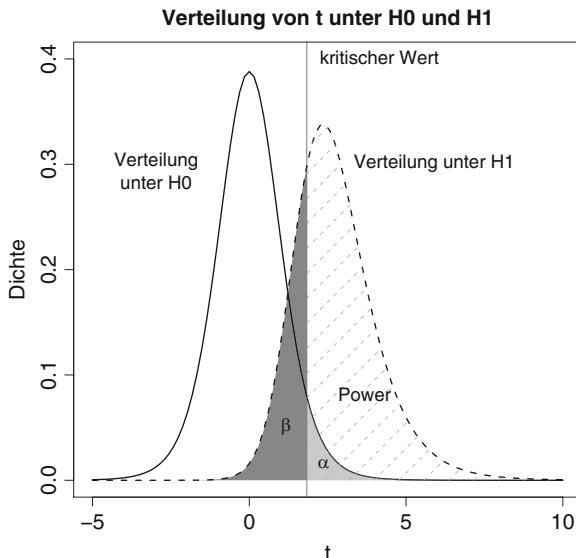


Abb. 8.8 Rechtsseitiger t -Test für eine Stichprobe: Verteilung von t unter H_0 und H_1 , kritischer Wert, α , β und Power

thesenpaar und kennzeichnet die Flächen, deren Größe α , β und Power bei einem rechtsseitigen Test entsprechen.

```

> nSubj <- 10                                # Stichprobengröße
> muH0  <- 0                                 # Erwartungswert unter H0
> muH1  <- 1.6                               # Erwartungswert unter H1
> alpha <- 0.05                               # Signifikanzniveau
> sigma <- 2                                  # wahre Streuung
> (d    <- (muH1-muH0) / sigma)             # Effektstärke d
[1] 0.8

> (delta <- (muH1-muH0) / (sigma/sqrt(nSubj))) # NZP, oder sqrt(nSubj)*d
[1] 2.529822

> (critT <- qt(1-alpha, nSubj-1))           # kritischer t-Wert
[1] 1.833113

> (powT <- 1-pt(critT, nSubj-1, delta))      # Power
[1] 0.7544248

# graphische Veranschaulichung der t-Verteilungen
> tVals <- seq(-5, 10, length.out=100)        # t-Werte
> yH0   <- dt(tVals, nSubj-1, 0)              # Dichte unter H0
> yH1   <- dt(tVals, nSubj-1, delta)           # Dichte unter H1
> plot(tVals, yH0, type="l", lwd=2, col="red", xlab="t", ylab="Dichte",
+      main="Verteilung von t unter H0 und H1", ylim=c(0, 0.4))
> points(tVals, yH1, type="l", lwd=2, lty=2, col="blue")

```

```
# Markierung der Flächen für alpha, beta und Power
> tLeft <- seq(-5, critT, length.out=100) # links vom krit. Wert
> tRight <- seq(critT, 10, length.out=100) # rechts vom krit. Wert
> yH0r <- dt(tRight, nSubj-1, 0) # Dichte unter H0 rechts
> yH1l <- dt(tLeft, nSubj-1, delta) # Dichte unter H1 links
> yH1r <- dt(tRight, nSubj-1, delta) # Dichte unter H1 rechts
> polygon(c(tRight, rev(tRight)), c(yH0r, numeric(length(tRight))),
+           border=NA, col=rgb(1, 0.3, 0.3, 0.6))

> polygon(c(tLeft, rev(tLeft)), c(yH1l, numeric(length(tLeft))),
+           border=NA, col=rgb(0.3, 0.3, 1, 0.6))

> polygon(c(tRight, rev(tRight)), c(yH1r, numeric(length(tRight))),
+           border=NA, density=5, lty=2, angle=45, col="darkgray")

# zusätzliche Beschriftungen
> abline(v=critT, lty=1, lwd=2, col=rgb(0, 1, 0, 0.5))
> text(critT+0.2, 0.4, adj=0, label="kritischer Wert")
> text(critT-2.8, 0.3, adj=1, label="Verteilung unter H0")
> text(critT+1.5, 0.3, adj=0, label="Verteilung unter H1")
> text(critT+1.0, 0.08, adj=0, label="Power")
> text(critT-0.7, 0.05, expression(beta))
> text(critT+0.5, 0.015, expression(alpha))
```

Wie für Binomial- und *t*-Tests demonstriert kann die Power analog für viele andere Tests manuell berechnet werden. Für die Berechnung der Power von *t*-Tests, bestimmten χ^2 -Tests und einfaktoriellen Varianzanalysen ohne Messwiederholung stehen in R auch eigene Funktionen bereit, deren Name nach dem Muster `power.<Test>.test()` aufgebaut ist. Diese Funktionen dienen gleichzeitig der Ermittlung der Stichprobengröße, die notwendig ist, damit ein Test bei fester Effektstärke eine vorgegebene Mindest-Power erzielt.¹⁴

```
> power.t.test(n, delta, sd, sig.level, power, strict=FALSE,
+               type=c("two.sample", "one.sample", "paired"),
+               alternative=c("two.sided", "one.sided"))
```

Von den Argumenten `n` für die Gruppengröße, `delta` für die Differenz der Erwartungswerte unter H_0 und H_1 , `sd` für die theoretische Streuung, `sig.level` für den α -Fehler und `power` für die Power sind genau vier mit konkreten Werten zu nennen und eines auf `NULL` zu setzen. Das auf `NULL` gesetzte Argument wird dann auf Basis der übrigen berechnet. `n` bezieht sich im Fall zweier Stichproben auf die Größe jeder

¹⁴ Diese Funktionalität ist jedoch recht eingeschränkt, so ist die Berechnung der Mindeststichprobengröße nur für die genannten drei Tests möglich. Mehr Möglichkeiten bietet das `pwr` Paket (Champely, 2009). Insbesondere besitzt jedoch das kostenlose verfügbare Programm G*Power einen deutlich breiteren Einsatzbereich hinsichtlich der unterstützten Tests, zudem bietet es vielfältige Möglichkeiten zur Visualisierung der Zusammenhänge von Effektstärke, Power und Stichprobengröße (Faul et al., 2007).

Gruppe – es werden also auch bei unabhängigen Stichproben gleiche Gruppengrößen vorausgesetzt.¹⁵

Welche Art von *t*-Test vorliegt, kann über `type` angegeben werden, `alternative` legt fest, ob die H_1 gerichtet oder ungerichtet ist. Das Argument `strict` bestimmt, ob im zweiseitigen Test für die Power die Wahrscheinlichkeit berücksichtigt werden soll, auch auf der falschen Seite (relativ zur Lage der tatsächlichen Verteilung unter H_1) die H_0 zu verwerfen.

Eine Fragestellung für den Einsatz von `power.t.test()` ist die Aufgabe, eine Stichprobengröße zu ermitteln, für die der Test bei einem als gegeben vorausgesetzten Effekt eine gewisse Power erreicht. In diesem Fall ist also das Argument `n=NULL` zu übergeben, alle anderen sind zu spezifizieren. Die ausgegebene Gruppengröße ist i. d. R. nicht ganzzahlig, muss also in der konkreten Anwendung aufgerundet werden, wodurch sich die tatsächliche Power des Tests leicht erhöht.

Im Beispiel soll für die oben gegebene Situation herausgefunden werden, wie viele VPn notwendig sind, damit der Test eine Power von 0.9 besitzt.

```
> power.t.test(n=NULL, delta=muH1-muH0, sd=sigma, sig.level=0.05,
+                 power=0.9, type="one.sample", alternative="one.sided")
One-sample t test power calculation

      n    = 14.84346
      delta = 1.6
      sd    = 2
      sig.level = 0.05
      power   = 0.9
      alternative = one.sided
```

Eine andere Frage ist, wie groß bei einer gegebenen Stichprobengröße der tatsächliche Effekt sein muss, damit der Test eine bestimmte Power erreicht. Hier ist `delta=NULL` zu übergeben und alle anderen Argumente zu spezifizieren.

8.9.3 Einfaktorielle Varianzanalyse

Die analog zu `power.t.test()` arbeitende Funktion für eine einfaktorielle Varianzanalyse ohne Messwiederholung lautet `power.anova.test()`.

```
> power.anova.test(groups, n, between.var, within.var, sig.level, power)
```

¹⁵ Für unterschiedliche Gruppengrößen n_1 und n_2 lassen sich annähernd richtige Ergebnisse erzielen, wenn $2*((n_1*n_2)/(n_1+n_2))$ für das Argument `n` übergeben wird, wodurch die Berechnung des Nonzentralitätsparameters δ als $\sqrt{n/2} * (\text{delta}/\text{sd})$ korrekt ist. Statt mit der richtigen Zahl der Freiheitsgrade $n_1 + n_2 - 2$ rechnet `power.t.test()` dann aber mit $4 \cdot ((n_1 \cdot n_2)/(n_1 + n_2)) - 2$. Der so entstehende Fehler wächst zwar mit der Differenz von n_1 und n_2 , bleibt jedoch absolut gesehen gering.

Von den Argumenten `groups` für die Anzahl der Gruppen, `n` für die Gruppengröße, `between.var` für die Varianz der Erwartungswerte unter H_1 ,¹⁶ `within.var` für die theoretische Varianz innerhalb der Gruppen, `sig.level` für den α -Fehler und `power` für die Power sind genau fünf mit konkreten Werten zu nennen und eines auf `NULL` zu setzen. Das auf `NULL` gesetzte Argument wird dann auf Basis der übrigen berechnet. `n` bezieht sich auf die Größe jeder Gruppe – es werden also gleiche Gruppengrößen vorausgesetzt.

Im folgenden Beispiel einer einfaktoriellen Varianzanalyse ohne Messwiederholung soll die notwendige Stichprobengröße berechnet werden, damit der Test bei gegebenen Erwartungswerten unter H_1 eine bestimmte Power erzielt.

```
> mus    <- c(100, 110, 115)                      # Erwartungswerte unter H1
> sigma <- 10                                     # theoretische Streuung
> power.anova.test(groups=3, n=NULL, between.var=var(mus),
+                     within.var=sigma^2, sig.level=0.05, power=0.9)
Balanced one-way analysis of variance power calculation

groups = 3
n     = 11.90949
between.var = 58.33333
within.var  = 100
sig.level   = 0.05
power       = 0.9
```

NOTE: `n` is number in each group

Im folgenden Beispiel soll die manuelle Berechnung der Power und der Maße für die Effektstärke einer Varianzanalyse im CR- p Design mit ungleichen Zellbesetzungen demonstriert werden.

```
> P      <- 3                                     # Anzahl der Gruppen
> nSubj <- c(21, 17, 19)                         # Gruppengrößen
> mus    <- c(100, 110, 115)                     # Gruppenerwartungswerte
> sigma  <- 15                                    # wahre Streuung
> grandMu <- sum(nSubj * mus) / sum(nSubj)        # mittlerer EW, gewichtet
> alphas <- mus - grandMu                        # Gruppeneffekte
> varMus <- sum(nSubj * alphas^2) / sum(nSubj)    # Varianz der Erw.werte

# Maße für die Effektstärke - Bezeichnungen uneinheitlich
> (fSq <- varMus / sigma^2)                       # f^2
[1] 0.1826887

> (etaSq <- varMus / (sigma^2 + varMus))          # eta^2 bzw. omega^2
[1] 0.1544690
```

¹⁶ Enthält der Vektor `mus` die Erwartungswerte der Gruppen unter H_1 , muss wegen der in `power.anova.test()` verwendeten Formel für den Nonzentralitätsparameter λ die korrigierte Varianz der Erwartungswerte, also `var(mus)`, für das Argument `between.var` übergeben werden.

```
# Nonzentralitätsparameter lambda bzw. delta^2 - Bezeichnungen uneinheitlich
> (lambda <- sum(nSubj * alphas^2) / sigma^2)      # oder: sum(nSubj) * fSq
[1] 10.41326

> (critF <- qf(1-0.05, P-1, sum(nSubj)-P))    # kritischer Wert, alpha=0.05
[1] 3.168246

> (powF <- 1-pf(critF, P-1, sum(nSubj)-P, lambda))  # Power
[1] 0.8090387
```

Liegt keine exakte Alternativhypothese vor, ist die Power als Funktion der Effektstärke f darstellbar (Abb. 8.9).

```
> fVals <- seq(0, 1.2, length.out=100)      # Effektstärken f auf x-Achse
> nn     <- seq(10, 25, by=5)                 # Stichprobengrößen: separate Kurven
# Funktion: berechne Power für verschiedene Stichprobengrößen und f-Werte
> getFpow <- function(n) {
+   critF <- qf(1-0.05, P-1, P*n - P)
+   1-pf(critF, P-1, P*n - P, P*n * fVals^2)
+ }

> powsF <- sapply(nn, getFpow)      # Power-Werte für jede Stichprobengröße

# Beschriftungen vorbereiten
> yStr <- "Wahrscheinlichkeit der Annahme von H1"
> mStr <- substitute(paste("Power als Funktion der Effektstärke f (",
+                           alpha, "=0.05")))
```

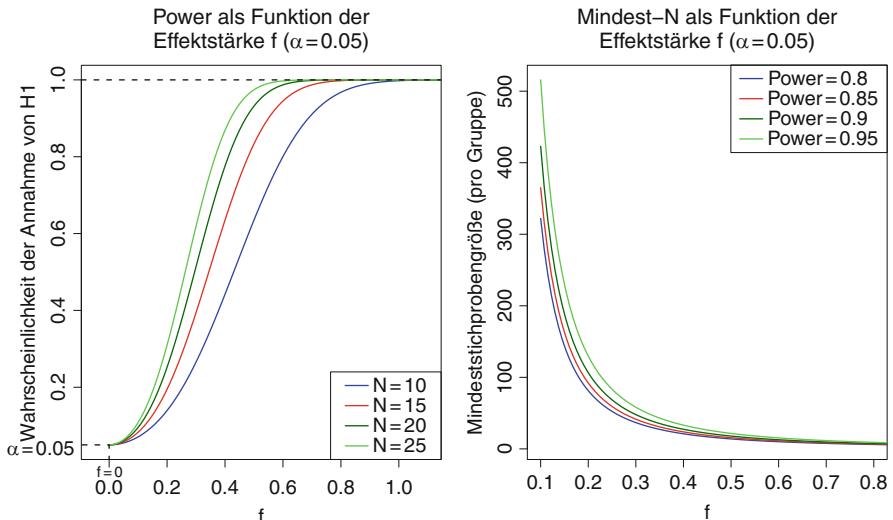


Abb. 8.9 Einfaktorielle Varianzanalyse: Power als Funktion der Effektstärke f für verschiedene Gruppengrößen sowie Mindeststichprobengröße als Funktion von f für verschiedene Power-Werte

```
# Power-Werte darstellen
> matplot(fVals, powsF, type="l", lty=1, lwd=2, xlab="f", ylab=yStr,
+           xlim=c(-0.05, 1.1), main=mStr,
+           col=c("blue", "red", "darkgreen", "green"))

# zusätzliche Beschriftungen
> lines(c(-2, 0, 0), c(0.05, 0.05, -2), lty=2, lwd=2)
> abline(h=1, lty=2, lwd=2)
> mtext("f=0", side=1, at=0)
> mtext(substitute(paste(alpha, "=0.05", sep="")), side=2, at=0.05, las=1)
> legend(x="bottomright", legend=paste("N =", c(10, 15, 20, 25)), lwd=2,
+         col=c("blue", "red", "darkgreen", "green"))
```

Liegt keine exakte Alternativhypothese vor, lässt sich auch die Mindeststichprobengröße als Funktion der Effektstärke f darstellen (Abb. 8.9).

```
# Funktion, um für gegebene Power und gegebene var.between
# die Mindeststichprobengröße zu berechnen
> getOneFn <- function(pp, varB) {
+   res <- power.anova.test(groups=P, n=NULL, between.var=varB,
+                           within.var=sigma^2, sig.level=0.05, power=pp)
+   res$n
+ }

# Funktion, um für mehrere Effektstärken f und mehrere Power-Werte
# gleichzeitig die Mindeststichprobengröße zu berechnen
# berechnet aus f zunächst var.between für power.anova.test()
> getManyFn <- function(ff, powF) {
+   varB <- ff^2 * (P/(P-1)) * sigma^2
+   sapply(powF, getOneFn, varB)
+ }

> fVals <- seq(0.1, 0.85, length.out=100) # Effektstärken f auf x-Achse
> pows <- seq(0.8, 0.95, by=0.05)          # Power-Werte: separate Kurven
> minN <- sapply(fVals, getManyFn, pows) # Mindeststichprobengrößen

# Beschriftungen vorbereiten
> yStr <- "Mindeststichprobengröße (pro Gruppe)"
> mStr <- substitute(paste("Mindest-N als Funktion der Effektstärke f
+ (" , alpha, " =0.05")"))

# Mindeststichprobengrößen als Funktion von f darstellen
> matplot(fVals, t(minN), type="l", lty=1, lwd=2, xlab="f", ylab=yStr,
+           xlim=c(0.1, 0.8), main=mStr,
+           col=c("blue", "red", "darkgreen", "green"))

> legend(x="topright", legend=paste("Power = ", c(0.80, 0.85, 0.90,
+                                         0.95)), lwd=2, col=c("blue", "red", "darkgreen", "green"))
```


Kapitel 9

Multivariate Verfahren

Liegen von Beobachtungsobjekten Daten mehrerer Variablen vor, können sich Fragestellungen nicht nur auf jede Variable einzeln, sondern auch auf die gemeinsame Verteilung der Variablen beziehen. Solche Fragestellungen sind mit multivariaten Verfahren zu bearbeiten (Backhaus et al., 2008; Härdle und Simar, 2007; Mardia et al., 1980). Aus diesem umfangreichen Themengebiet kann hier nur die Umsetzung ausgewählter, als bekannt vorausgesetzter Methoden vorgestellt werden (für eine ausführlichere Darstellung vgl. Everitt, 2005). Insbesondere Klassifikationsverfahren wie die Diskriminanz- und Clusteranalyse (vgl. die Abschnitte Cluster und Multivariate der Task Views, R Development Core Team, 2009a) sowie eine separate Behandlung des Allgemeinen Linearen Modells bleiben ausgespart.

Die Verfahren basieren in ihren Grundlagen auf Konzepten und Rechentechniken der Linearen Algebra, deren Anwendung in R Abschn. 2.9 beschreibt. Abschnitt 10.6.8 und insbesondere 10.8 thematisieren Möglichkeiten, Diagramme zur Visualisierung multivariater Daten zu erzeugen.

9.1 Multivariate Multiple Regression

Die univariate multiple Regression (vgl. Abschn. 7.3) lässt sich zur multivariaten multiplen Regression verallgemeinern, bei der durch p Prädiktoren X_j (mit $1 \leq j \leq p$) nicht nur ein Kriterium Y vorhergesagt werden soll, sondern q Kriteriumsvariablen Y_k (mit $1 \leq k \leq q$) gleichzeitig. Der multivariate Fall ist jedoch auf den univariaten zurückführbar: das im Sinne der geringsten Quadratsumme der Abweichungen optimale Ergebnis ist die Zusammenstellung der q unabhängig voneinander durchgeführten Regressionen von jeweils einem Kriterium Y_k auf alle Prädiktoren X_j . Dafür werden zum einen die Regressionsgewichte $b_{1k}, \dots, b_{jk}, \dots, b_{pk}$ aus der Regression mit den Prädiktoren X_j und dem Kriterium Y_k spaltenweise zu einer $(p \times q)$ -Matrix B zusammengestellt. Zum anderen bilden die q absoluten Terme a_k einen q -Vektor a . Stellt man die jeweils ermittelten Vorhersagen \hat{Y}_k spaltenweise zu einer Matrix \hat{Y} zusammen, gilt damit insgesamt $\hat{Y} = X \cdot B + a$.

In der Berechnung der multivariaten multiplen Regression mittels `lm(<Formel>)` ist `<Formel>` wie im univariaten Fall als `<Kriterium> ~ <Prädiktor 1> + ... + <Prädiktor j> + ... + <Prädiktor p>` aufzubauen. Im Gegensatz zum

univariaten Fall ist (Kriterium) hier jedoch kein Vektor sondern muss eine spaltenweise aus den einzelnen Kriteriumsvariablen zusammengestellte Matrix sein.

Im Beispiel sollen anhand der Prädiktoren Alter, Körpergröße und wöchentliche Dauer sportlicher Aktivitäten die Kriterien Körpergewicht und Gesundheit (im Sinne eines geeigneten quantitativen Maßes) vorhergesagt werden.

```

> nSubj <- 100 # Anzahl Versuchspersonen
> height <- rnorm(nSubj, 175, 7) # Prädiktor 1
> age <- rnorm(nSubj, 30, 8) # Prädiktor 2
> sport <- abs(rnorm(nSubj, 60, 30)) # Prädiktor 3

# modellgerechte Simulation beider Kriterien
> weight <- 0.5*height - 0.3*age - 0.4*sport + 10 + rnorm(nSubj, 0, 3)
> health <- -0.3*age + 0.6*sport + rnorm(nSubj, 4)
> Y <- cbind(weight, health) # Matrix der Kriterien

# multivariat formulierte Formel für lm()
> (model <- lm(Y ~ height + age + sport))
Call:
lm(formula = Y ~ height + age + sport)

Coefficients:
            weight      health
(Intercept) 10.41235  2.44980
height       0.50721  0.01031
age          -0.34730 -0.29678
sport        -0.40894  0.59826

# Kontrolle: beide univariaten Regressionen separat
# Kriterium: Körpergewicht
> lm(weight ~ height + age + sport)
Call:
lm(formula = weight ~ height + age + sport)

Coefficients:
(Intercept) height      age      sport
           10.4124  0.5072 -0.3473 -0.4089

# Kriterium: Gesundheitsmaß
> lm(health ~ height + age + sport)
Call:
lm(formula = health ~ height + age + sport)

Coefficients:
(Intercept) height      age      sport

```

Für die manuelle Kontrolle sei vorausgesetzt, dass die Kovarianzmatrix der Prädiktoren invertierbar ist, also keine lineare Abhängigkeit zwischen den Prädiktoren vorliegt.

```
> X <- cbind(height, age, sport)      # Matrix der Prädiktoren
> Sx <- cov(X)                    # Kovarianzmatrix der Prädiktoren
```

```

# Kovarianzmatrix der Prädiktoren mit den Kriterien
> Sxy <- cov(X, Y)
> (B <- solve(Sx, Sxy))           # Matrix der Regressionsgewichte
      weight      health
height  0.5072084  0.01030549
age     -0.3473046 -0.29677895
sport    -0.4089361  0.59826298

# Vektor der absoluten Terme
> (a <- as.vector(colMeans(Y) - t(B) %*% colMeans(X)))
[1] 10.412355 2.449798

# vorhergesagte Werte: X*B + a
> fitMat <- sweep(X %*% B, 2, a, "+")

# Kontrolle: Vergleich der manuell berechneten Vorhersage mit
# jener aus lm()
> all.equal(c(fitMat), c(fitted(model)))
[1] TRUE

```

9.2 Hauptkomponentenanalyse

Die Hauptkomponentenanalyse multivariater Daten dient dazu, die Hauptstreuungsrichtungen der Daten im durch die Variablen aufgespannten Raum zu identifizieren. Hauptkomponenten sind neue Variablen, die als Linearkombinationen der ursprünglichen Variablen gebildet werden und folgende Eigenschaften besitzen:

- Die Linearkombinationen sind standardisiert, d. h. die Koeffizientenvektoren haben jeweils die Länge 1. Dies sind die Eigenvektoren der Kovarianzmatrix der Daten.
- Die Hauptkomponenten sind unkorreliert,¹ außerdem sind alle Skalarprodukte ihrer Koeffizientenvektoren 0.
- Die Hauptkomponenten kann man in einem Koordinatensystem ablesen, das seinen Ursprung im Zentroid der Daten hat. Die Achsen weisen in Richtung der Koeffizientenvektoren und haben dieselbe Einheit wie das Standard-Koordinatensystem (Abb. 9.1).
- Projiziert man die Daten auf eine Gerade, die in Richtung eines Eigenvektors verläuft, so ist die Streuung der projizierten Daten entlang der Geraden gleich der Streuung der zugehörigen Hauptkomponente. Die Streuung ist auch gleich der Wurzel des zugehörigen Eigenwertes der Kovarianzmatrix der Daten.
- Die Streuung der Hauptkomponenten ist im folgenden Sinn sukzessive maximal: die erste Hauptkomponente ist diejenige unter allen standardisierten Linearkombinationen mit der größten Streuung. Unter allen standardisierten

¹ Genauer gesagt ist ihre Kovarianz 0 – da ihre Varianz auch 0 sein kann, ist die Korrelation nicht immer definiert.

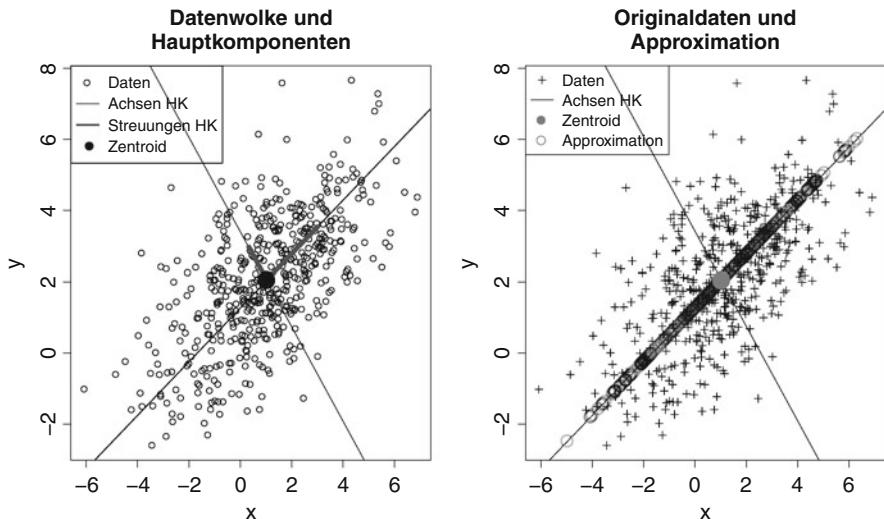


Abb. 9.1 Hauptkomponentenanalyse: Koordinatensystem mit Achsen in Richtung der ins Zentrum verschobenen Eigenvektoren sowie Streuungen der Hauptkomponenten. Originaldaten und Approximation

Linearkombinationen, die mit der ersten Hauptkomponente unkorreliert sind, ist die zweite Hauptkomponente dann wieder diejenige mit der größten Streuung. Für die weiteren Hauptkomponenten gilt dies analog.

Für die Berechnung der Hauptkomponenten samt ihrer jeweiligen Streuung stehen die Funktionen `prcomp(x=Matrix)` und `princomp(x=Matrix)` zur Verfügung, wobei für `x` die spaltenweise aus den Variablen zusammengestellte Datenmatrix anzugeben ist.

```
# 2x2 Kovarianzmatrix für simulierte Zufallsvektoren
> sigma <- matrix(c(4, 2, 2, 3), ncol=2)
> mu      <- c(1, 2)                                # Zentroid für Zufallsvektoren
> nSubj   <- 500                                    # Anzahl VPn
> library(mvtnorm)                                 # Zufallsvektoren mit mvtnorm
> X       <- rmvnorm(n=nSubj, mean=mu, sigma=sigma) # Multinormalverteilung
> (pca <- prcomp(X))                               # Hauptkomponentenanalyse
Standard deviations:
[1] 2.534967 1.205418
```

Rotation:

	PC1	PC2
[1,]	0.7989777	-0.6013607
[2,]	0.6013607	0.7989777

Die Ausgabe von `prcomp()` ist eine Liste mit zwei Komponenten: die erste enthält den Vektor der korrigierten Streuungen der Hauptkomponenten (Überschrift `Standard deviations`). Dies sind gleichzeitig die Wurzeln aus den Eigenwer-

ten der Kovarianzmatrix der Daten (vgl. Abschn. 2.9.5). Die zweite Komponente beinhaltet die als Spalten einer Matrix C zusammengestellten Koeffizienten der Linearkombinationen zur Bildung der Hauptkomponenten (Überschrift Rotation). Dies sind gleichzeitig die normierten Eigenvektoren der Kovarianzmatrix (bis auf potentiell unterschiedliche Vorzeichen im Sinne von Rotationen um 180°). Da die Linearkombinationen standardisiert sind und die Koeffizientenvektoren jeweils Skalarprodukt 0 miteinander haben, ist C eine Orthogonalmatrix ($C^t = C^{-1}$).

```

> coeff <- pca[[2]]           # Koeffizienten der Hauptkomponenten
> t(coeff) %*% coeff         # bilden Orthogonalmatrix
      PC1          PC2
PC1  1.000000e+00 -2.602085e-18
PC2 -2.602085e-18  1.000000e+00

# graphische Darstellung der Daten und Hauptkomponenten
> ctr     <- colMeans(X)      # Zentroid der Daten
> eig     <- eigen(cov(X))    # Eigenwerte, -vektoren Kovarianzmatrix
> eigVal  <- eig$values       # Eigenwerte
> eigVec  <- eig$vectors      # Eigenvektoren

# multiplizierte Eigenvektoren mit Wurzel aus zugehörigem Eigenwert
> eigScl <- sweep(eigVec, 2, sqrt(eigVal), "*")
> xMat   <- rbind(ctr[1] - eigScl[1, ], ctr[1])
> yMat   <- rbind(ctr[2] - eigScl[2, ], ctr[2])

# berechne Punkt-Steigungsform der durch die Eigenvektoren
# definierten Achsen
> p1 <- ctr + eigScl[, 1]        # Punkt Achse 1
> p2 <- ctr + eigScl[, 2]        # Punkt Achse 2
> b1 <- (p1[2]-ctr[2]) / (p1[1]-ctr[1])  # Steigung Achse 1
> a1 <- ctr[2] - b1*ctr[1]        # Nulldurchgang Achse 1
> b2 <- (p2[2]-ctr[2]) / (p2[1]-ctr[1])  # Steigung Achse 2
> a2 <- ctr[2] - b2*ctr[1]        # Nulldurchgang Achse 2

# Datenwolke darstellen
> plot(X[, 1], X[, 2], xlab="x", ylab="y",
+       main="Datenwolke und Hauptkomponenten")

> abline(a=a1, b=b1)             # Achse 1
> abline(a=a2, b=b2)             # Achse 2
> matlines(xMat, yMat, lty=1, lwd=6, col="blue")  # Streuungen HK
> points(ctr[1], ctr[2], pch=16, col="red", cex=3) # Zentroid

# Legende einfügen
> legend(x="topleft", legend=c("Daten", "Achsen HK",
+                               "Streuungen HK", "Zentroid"), pch=c(1, NA, NA, 16),
+                               lty=c(NA, 1, 1, NA), col=c("black", "black", "blue", "red")))

```

Aus der Matrix C der Koeffizienten der standardisierten Linearkombinationen und der Datenmatrix X berechnen sich die Hauptkomponenten als neue Variablen durch $X \cdot C$. Dies ist gleichzeitig die orthogonale Projektion der Daten auf die Gera-

den in Richtung der Eigenvektoren (vgl. Abschn. 2.9.4). Die Streuungen der projizierten Daten entlang der Geraden sind deshalb gleich jenen der zugehörigen Hauptkomponenten, wie sie in der Komponente Standard deviations der Ausgabe von `prcomp()` genannt werden.

```
# manuelle Berechnung der Koeffizienten zur Bildung der HK
> sqrt(eig$values)           # Wurzel aus den Eigenwerten
[1] 2.534967 1.205418

> eig$vectors               # Eigenvektoren
[,1]      [,2]
[1,] -0.7989777 0.6013607
[2,] -0.6013607 -0.7989777

> pc      <- X %*% coeff      # Berechnung der Hauptkomponenten
> (pcSd <- apply(pc, 2, sd))  # Kontrolle: Streuungen
PC1      PC2
2.534967 1.205418
```

Die Streuungen der Hauptkomponenten, den Anteil ihrer Varianz an der Gesamtvarianz (im Sinne der Spur der Kovarianzmatrix der Daten) sowie den kumulativen Anteil der Varianz der Hauptkomponenten an der Gesamtvarianz gibt `summary(PCA)` aus. Dabei ist `(PCA)` das Ergebnis einer Hauptkomponentenanalyse mit `prcomp()` oder `princomp()`.

```
> summary(pca)
Importance of components:
              PC1     PC2
Standard deviation    2.535 1.205
Proportion of Variance 0.816 0.184
Cumulative Proportion 0.816 1.000

# Kontrolle: Anteil der durch HK aufgeklärten Varianz
> pca[[1]]^2 / sum(diag(cov(X)))
[1] 0.8155839 0.1844161
```

Häufig dient die Hauptkomponentenanalyse der Datenreduktion: wenn von Beobachtungsobjekten Werte von sehr vielen Variablen vorliegen, ist es oft wünschenswert, die Daten durch Werte von weniger Variablen möglichst gut zu approximieren. Als Kriterium dient dabei die Summe der quadrierten Abstände zwischen Originaldaten und der Approximation. Eine perfekte Reproduktion der Originaldaten lässt sich zunächst wie folgt erreichen: sei die Matrix B spaltenweise aus den Koeffizientenvektoren der Hauptkomponenten zusammengestellt. Zusätzlich sollen die Spalten von B als Länge die Streuung der jeweils zugehörigen Hauptkomponente besitzen. Weiter sei H_s die Matrix der normierten Hauptkomponenten, die jeweils Mittelwert 0 und Varianz 1 besitzen. Ferner sei c das Zentroid der Daten. Für die Datenmatrix X gilt dann $X = B \cdot H_s + c$.

```
# multiplizierte Koeffizientenvektoren mit Streuung der HK
> B    <- sweep(coeff, 2, pca[[1]], "*")
> Hs   <- scale(pc)          # standardisierte Hauptkomponenten
```

```
> mat <- t(B %*% t(Hs))      # B * H_s
# Reproduktion der Originaldaten: addiere Zentroid: B * H_s + c
> repr <- sweep(mat, 2, ctr, "+")
> all.equal(X, repr)          # Kontrolle: Übereinstimmung mit X
[1] TRUE

> sum((X-repr)^2)            # Summe der quadrierten Abweichungen: 0
[1] 1.696667e-28
```

Sollen die ursprünglichen Daten nun im obigen Sinne optimal durch weniger Variablen repräsentiert werden, können die Spalten von B von rechts kommend nacheinander gestrichen werden, wodurch sich etwa eine Matrix B' ergibt. Ebenso sind die zugehörigen standardisierten Hauptkomponenten, also die Spalten von H_s , zu streichen, wodurch die Matrix H'_s entsteht. Die approximierten Daten $X' = B' \cdot H'_s + c$ liegen dann in einem affinen Unterraum mit niedrigerer Dimension als die ursprünglichen Daten (Abb. 9.1).

```
# approximiere Originaldaten durch niedrig-dimensionale Daten
> mat1 <- t(B[, 1] %*% t(Hs[, 1]))           # B' * H_s'
> repr1 <- sweep(mat1, 2, ctr, "+")             # B' * H_s' + c
> sum((X-repr1)^2)                            # Summe der quadrierten Abweichungen
[1] 725.0627

# graphische Darstellung der Originaldaten und der Approximation
> plot(X[, 1], X[, 2], xlab="x", ylab="y", pch=3,
+       main="Originaldaten und Approximation")

> abline(a=a1, b=b1)                          # Achse 1
> abline(a=a2, b=b2)                          # Achse 2
> points(repr1, pch=1, col=rgb(0, 1, 0, 0.2)) # Approximation
> points(ctr[1], ctr[2], pch=16, col="red", cex=3) # Zentroid

# Legende einfügen
> legend(x="topleft", legend=c("Daten", "Achsen HK",
+                               "Zentroid", "Approximation"), pch=c(3, NA, 16, 1),
+                               lty=c(NA, 1, NA, NA), col=c("black", "black", "red", "green"))
```

B hat auch die Eigenschaft, dass sich die Kovarianzmatrix S der Daten als $S = B \cdot B^t$ darstellen lässt (vgl. Abschn. 2.9.5). Die Reproduktion von S durch die um rechte Spalten gestrichene Matrix B' mit $B' \cdot B'^t$ wird schlechter, bleibt aber optimal im Vergleich zu allen anderen Matrizen gleicher Dimensionierung.

```
> B %*% t(B)                                # reproduziert Kovarianzmatrix von X
[1,]      [,2]
[1,] 4.627640 2.389409
[2,] 2.389409 3.251450

> cov(X)                                    # Kontrolle ...

# approximiere Kovarianzmatrix von X
> B[, 1] %*% t(B[, 1])
```

```
[,1]      [,2]
[1,] 4.102173 3.087553
[2,] 3.087553 2.323885
```

Die von `princomp()` berechnete Hauptkomponentenanalyse unterscheidet sich von der durch `prcomp()` erzeugten in den ausgegebenen Streuungen: bei `prcomp()` sind dies die korrigierten, bei `princomp()` die unkorrigierten.²

```
> (pcaPrin <- princomp(X))
Call:
princomp(x = X)
Standard deviations:
Comp.1   Comp.2
2.532431 1.204212
2 variables and 500 observations.
```

In der Ausgabe von `princomp()` sind die Koeffizienten der standardisierten Linearkombination zunächst nicht mit aufgeführt, sind jedoch in der Komponente `loadings` der zurückgelieferten Liste enthalten.

```
> pcaPrin$loadings           # Koeffizientenmatrix
Loadings:
          Comp.1  Comp.2
[1,] -0.799  0.601
[2,] -0.601 -0.799

          Comp.1  Comp.2
SS loadings     1.0    1.0
Proportion Var  0.5    0.5
Cumulative Var 0.5    1.0
```

```
# Kontrolle: unkorrigierte Streuungen der Hauptkomponenten
> covMat <- cov.wt(pc, method="ML")$cov
> sqrt(diag(covMat))
PC1      PC2
2.532431 1.204212
```

Die Varianzen der Hauptkomponenten werden häufig in einem Liniendiagramm (vgl. Abschn. 10.2) als sog. Scree-Plot dargestellt, der mit `plot(PCA, type="b")` aufzurufen ist. Dabei ist `(PCA)` das Ergebnis einer Hauptkomponentenanalyse mit `prcomp()` oder `princomp()`. Für eine graphische Veranschaulichung der Hauptkomponenten selbst über die Hauptachsen des die gemeinsame Verteilung der Daten charakterisierenden Streungsellipsoids vgl. Abschn. 10.6.8, Abb. 10.23. `biplot()` stellt die Hauptkomponenten gleichzeitig mit den Objekten in einem Diagramm dar, das eine inhaltliche Interpretation der Hauptkomponenten fördern soll.

² Weiterhin basiert `prcomp()` intern auf der Singulärwertzerlegung mit `svd()`, `princomp()` hingegen auf der Berechnung der Eigenwerte mit `eigen()` (vgl. Abschn. 2.9.5). Die Singulärwertzerlegung gilt als numerisch stabiler bei schlecht konditionierten Matrizen im Sinne der Kondition κ (vgl. Abschn. 2.9.5) – `prcomp()` sollte also vorgezogen werden.

9.3 Faktorenanalyse

Der Faktorenanalyse liegt die Vorstellung zugrunde, dass sich die korrelativen Zusammenhänge zwischen vielen beobachtbaren Merkmalen aus wenigen latenten Variablen (den sog. Faktoren) speisen, die ursächlich auf die Ausprägung der beobachtbaren Merkmale wirken.³ Die Wirkungsweise wird dabei zum einen als linear angenommen, d.h. die Ausprägung eines beobachtbaren Merkmals soll sich als Linearkombination der Ausprägungen der latenten Variablen zzgl. eines zufälligen Fehlers ergeben. Zum anderen beruht die Faktorenanalyse auf der Annahme, dass sich bei unterschiedlichen Beobachtungsobjekten zwar die Ausprägungen der latenten Variablen unterscheiden, die den Einfluss der Faktoren auf die beobachtbaren Merkmale charakterisierenden Koeffizienten der Linearkombinationen dagegen fest, also unabhängig von den Beobachtungsobjekten sind.

Zur Vereinbarung der Terminologie sei x die Variable der p beobachtbaren Merkmalsausprägungen, f die Variable der q Faktorausprägungen, e die Variable der p Fehler und Λ die $(p \times q)$ -Matrix der zeilenweise zusammengestellten Koeffizienten der Linearkombinationen für jede beobachtbare Variable. Λ wird auch als Ladungsmatrix bezeichnet. Das Modell geht von standardisierten beobachtbaren Variablen und Faktoren aus, die also Erwartungswert 0 und Varianz 1 besitzen sollen. Weiterhin sollen die Fehler untereinander und mit den Faktoren unkorreliert sein. Das Modell lässt sich damit insgesamt als $x = \Lambda \cdot f + e$ formulieren. Die beobachtbaren Variablen ohne addierten Fehlerterm seien als reduzierte Variablen $\hat{x} = \Lambda \cdot f$ bezeichnet.

Es sind nun zwei Varianten denkbar. Zum einen kann das Modell unkorrelierter Faktoren angenommen werden, bei denen die Korrelationsmatrix der Faktorwerte K_f gleich der $(q \times q)$ -Einheitsmatrix ist. Dieses Modell wird auch als orthogonal bezeichnet, da die Faktoren in einer geeigneten geometrischen Darstellung senkrecht aufeinander stehen. In diesem Fall ist Λ gleichzeitig die auch als Faktorstruktur bezeichnete Korrelationsmatrix von x und f . Zum anderen ist das Modell potentiell korrelierter Faktoren mit Einträgen von K_f ungleich 0 außerhalb der Hauptdiagonale möglich. Hier bilden die Faktoren in einer geeigneten Darstellung keinen rechten Winkel, weshalb das Modell auch als schiefwinklig bezeichnet wird. Die Faktorstruktur berechnet sich zu $\Lambda \cdot K_f$, zur Unterscheidung wird Λ selbst als Faktormuster bezeichnet.

Für die Korrelationsmatrix K_x der beobachtbaren Variablen ergibt sich im Modell korrelierter Faktoren $K_x = \Lambda \cdot K_f \cdot \Lambda^t + D_e$, wenn die Diagonalmatrix D_e die Kovarianzmatrix der Fehler ist. Im Modell unkorrelierter Faktoren vereinfacht sich die Gleichung zu $K_x = \Lambda \cdot \Lambda^t + D_e$. Analog zu den reduzierten Variablen \hat{x} sei $K_{\hat{x}} = \Lambda \cdot K_f \cdot \Lambda^t$ bzw. $K_{\hat{x}} = \Lambda \cdot \Lambda^t$ als reduzierte Korrelationsmatrix bezeichnet. Dabei ist zu beachten, dass $K_{\hat{x}}$ nicht die Korrelationsmatrix der \hat{x} ist – die

³ Für weitere Verfahren, die die Beziehungen latenter und beobachtbarer Variablen modellieren – etwa lineare Strukturgleichungsmodelle oder die Analyse latenter Klassen – vgl. den Abschnitt *Psychometrics* der Task Views (R Development Core Team, 2009a).

Diagonalelemente sind nicht 1, sondern ergänzen die Diagonalelemente von D_e (die Fehlervarianzen) zu 1. Die Diagonalelemente von $K_{\hat{x}}$ heißen auch Kommunalitäten von x . Sie sind gleichzeitig die Varianzen von $\Lambda \cdot f$ und damit ≥ 0 .

Bei der exploratorischen Faktorenanalyse besteht der Wunsch, bei vorausgesetzter Gültigkeit eines der beiden Modelle auf Basis vieler Messwerte der beobachtbaren Variablen eine Schätzung der Ladungsmatrix $\hat{\Lambda}$ sowie ggf. der Korrelationsmatrix der Faktoren $K_{\hat{f}}$ zu erhalten.⁴ Die Anzahl der latenten Faktoren sei dabei vorgegeben. Praktisch immer lassen sich jedoch durch Rotation (s. u.) viele Ladungs- und ggf. Korrelationsmatrizen finden, die zum selben Resultat führen.

Die Aufgabe kann analog zur Hauptkomponentenanalyse auch so verstanden werden, dass es die empirisch gegebene korrelative Struktur der beobachtbaren Variablen im Sinne der Korrelationsmatrix K_x möglichst gut durch die Matrix $\hat{\Lambda}$ und ggf. $K_{\hat{f}}$ zu reproduzieren gilt: $\hat{\Lambda} \cdot \hat{\Lambda}^t$ bzw. $\hat{\Lambda} \cdot K_{\hat{f}} \cdot \hat{\Lambda}^t$ soll also nur außerhalb der Diagonale nennenswert von K_x abweichen und auf der Diagonale positive Einträge ≤ 1 besitzen. Die mit $\hat{\Lambda}$ geschätzten Einflüsse der latenten Faktoren auf beobachtbare Variablen sollen gleichzeitig dazu dienen, die Faktoren mit Bedeutung zu versehen, also inhaltlich interpretierbar zu machen.

In R wird die Faktorenanalyse mit `factanal()` durchgeführt.

```
> factanal(x=<Datenmatrix>, covmat=<Kovarianzmatrix>,
+           n.obs=<Anzahl Beobachtungen>, factors=<Anzahl Faktoren>,
+           scores=<Schätzung Faktorwerte>, rotation=<Rotationsart>)
```

Für `x` ist die spaltenweise aus den Daten der beobachtbaren Variablen zusammengestellte Matrix zu übergeben. Alternativ lässt sich das Argument `covmat` nutzen, um statt der Daten ihre Kovarianzmatrix zu übergeben. In diesem Fall ist für `n.obs` die Anzahl der Beobachtungen zu nennen, die `covmat` zugrunde liegen. Die gewünschte Anzahl an Faktoren muss für `factors` genannt werden. Sollen die Schätzungen der Faktorwerte \hat{f} ebenfalls berechnet werden, ist `scores` auf "regression" oder "Bartlett" zu setzen. Mit der Voreinstellung "none" für das Argument `rotation` liegt der Rechnung das Modell unkorrelierter Faktoren zugrunde. Weitere Rotationsarten, etwa für das Modell korrelierter Faktoren, stellt das Paket GPArotation (Bernaards und Jennrich, 2005) zur Verfügung. Es enthält Funktionen, deren Namen an das Argument `rotation` übergeben werden können, z. B. "oblimin" für eine schiefwinklige Rotation. Für eine vollständige Liste vgl. `?rotations`, nachdem das Paket installiert und mit `library()` geladen wurde.

Das folgende Beispiel behandelt den Fall, dass unkorrelierte Faktoren angenommen werden.

```
> nSubj <- 200                      # Anzahl Beobachtungsobjekte
> nVar  <- 6                        # Anzahl beobachteter Variablen
> nFac  <- 2                        # simulierte Anzahl von Faktoren
```

⁴ Die konfirmatorische Faktorenanalyse, bei der theoretische Erwägungen ein bestimmtes, auf Konsistenz mit den Daten zu testendes $\hat{\Lambda}$ vorgeben, ist mit Hilfe linearer Strukturgleichungsmodelle durchzuführen – etwa mit dem `sem` Paket (Fox et al., 2009b).

```

# hypothetische Ladungsmatrix für die Simulation der Daten
> (lambda <- matrix(c(0.7,-0.4, 0.8,0, -0.2,0.9, -0.3,0.4,
+                               0.3,0.7, -0.8,0.1), nrow=nVar, ncol=nFac, byrow=TRUE))
   [,1] [ ,2]
[1,] 0.7 -0.4
[2,] 0.8  0.0
[3,] -0.2 0.9
[4,] -0.3 0.4
[5,] 0.3  0.7
[6,] -0.8 0.1

# hypothetische Kovarianzmatrix der Faktoren. Hier: Einheitsmatrix
> Kf <- diag(nFac)          # Modell unkorrelierter Faktoren
> mu <- c(5, 15)            # hypothetisches Zentroid der Faktoren
> library(mvtnorm)          # Zufallsvektoren mit mvtnorm
> f  <- t(rmvnorm(nSubj, mean=mu, sigma=Kf))  # simulierte Faktorwerte
> e  <- rnorm(nSubj, 0, 10)      # simulierter Fehler

# simuliere beobachtete Variablen im Modell der Faktorenanalyse
> X  <- t(lambda %*% f + e)        # Datenmatrix
> (fa <- factanal(X, factors=2, scores="regression"))
Call:
factanal(x = X, factors = 2)

Uniquenesses:
[1] 0.833 0.877 0.753 0.879 0.521 0.957

Loadings:
  Factor1 Factor2
[1,] -0.200  0.357
[2,]         -0.348
[3,]  0.216  0.447
[4,] -0.282 -0.203
[5,]  0.686
[6,] -0.207

  Factor1 Factor2
SS loadings    0.681  0.498
Proportion Var  0.114  0.083
Cumulative Var  0.114  0.197

Test of the hypothesis that 2 factors are sufficient.
The chi square statistic is 1.76 on 4 degrees of freedom.
The p-value is 0.779

```

Die Ausgabe von `factanal()` liefert unter der Überschrift `Uniqueness` die Fehlervarianzen von x , also die Einträge von D_e , die sich mit den Kommunalitäten zu 1 ergänzen. Die geschätzte Ladungsmatrix $\hat{\Lambda}$ ist unter `Loadings` aufgeführt, wobei nur Werte über einer bestimmten absoluten Größe angezeigt werden (Abb. 9.2). Die Faktoren sind dabei entsprechend der durch sie aufgeklärten Varianz geordnet. Die berechnete Ladungsmatrix weicht deutlich von der zur Simulation verwendeten

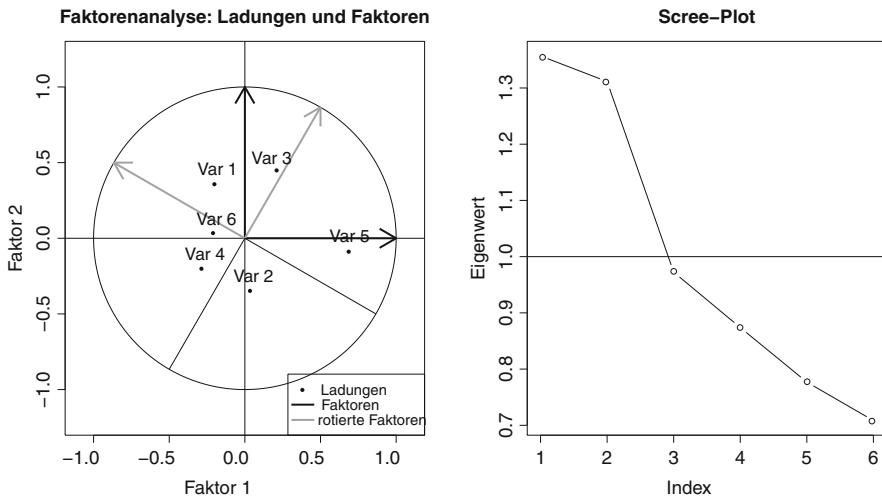


Abb. 9.2 Faktorenanalyse: Faktorladungen der Variablen sowie rotierte Faktoren. Scree-Plot der Eigenwerte der Korrelationsmatrix der Daten

ab. Ursache hierfür ist zum einen die bereits angesprochene Uneindeutigkeit der Lösung, zum anderen der in die simulierten Daten eingeflossene Fehlerterm.

In der abschließenden Tabelle gibt `SS loadings` die jeweilige Spaltensumme der quadrierten Ladungen eines Faktors, also die durch ihn aufgeklärte Varianz aller Variablen an – im Sinne der Spur der Korrelationsmatrix der Variablen, also der Anzahl der Variablen. Dies ist bei der durch `factanal()` verwendeten Methode, um ein $\hat{\Lambda}$ zu erzeugen, gleichzeitig der zugehörige Eigenwert der geschätzten reduzierten Korrelationsmatrix R_r aus den nicht rotierten geschätzten Ladungsmatrizen ($R_r = \hat{\Lambda} \cdot \hat{\Lambda}^T$).⁵ Proportion Var ist der vom Faktor aufgeklärte Anteil an der Gesamtvarianz aller Variablen und Cumulative Var der kumulative Anteil der durch die Faktoren aufgeklärten Varianz.

```
# Spaltensumme der quadrierten Ladungen -> aufgeklärte Varianz
> colSums((fa$loadings)^2)
  Factor1   Factor2
0.6814618 0.4982222

# Spaltensummen der quadrierten Ladungen geteilt durch Spur der
# Korrelationsmatrix -> Anteil der aufgeklärten Varianz
> colSums((fa$loadings)^2) / sum(diag(cor(X)))
  Factor1   Factor2
0.11357696 0.08303703

# Eigenwerte der geschätzten reduzierten Korrelationsmatrix
# hier: von Faktoren aufgeklärte Varianzen
```

⁵ Bei der Methode handelt es sich um die iterative Maximum-Likelihood-Kommunalitätenschätzung. Im allgemeinen gilt die Gleichheit von aufgeklärter Varianz eines Faktors und einem Eigenwert nicht.

```
> Rr      <- fa$loadings %*% t(fa$loadings)
> eigRr  <- eigen(Rr)
> eigRr$values                      # nur die ersten beiden > 0
[1] 6.815111e-01 4.981728e-01 7.850375e-17 1.118884e-17 -6.854921e-18
[6] -6.093836e-17
```

Schließlich folgt der χ^2 -Anpassungstest mit der Nullhypothese, dass das Modell mit der angegebenen Zahl an Faktoren tatsächlich gilt. Ein kleiner p -Wert deutet daraufhin, dass mehr Faktoren berücksichtigt werden müssen. Weitere Hilfestellung zur Frage, wieviele Faktoren zu wählen sind, liefert etwa das sog. Kaiser-Kriterium, dem zufolge nur so viele Faktoren zu berücksichtigen sind, wie es Eigenwerte der Korrelationsmatrix der Daten > 1 gibt (Abb. 9.2). Eine andere Herangehensweise besteht darin, den Abfall der Eigenwerte der Korrelationsmatrix der Daten zu betrachten, um einen besonders markanten Sprung zu identifizieren (Scree-Test). Hierbei hilft die graphische Darstellung dieser Eigenwerte als Liniendiagramm im sog. Scree-Plot (Abb. 9.2). Die Entscheidung für eine bestimmte Anzahl an Faktoren wird durch das Paket nFactors (Raiche und Magis, 2009) unterstützt.

Mit dem von factanal() zurückgegebenen Objekt sollen nun die Kommunalitäten berechnet sowie die geschätzte Korrelationsmatrix der beobachtbaren Variablen ermittelt werden. Schließlich sollen die Faktorladungen der Variablen grafisch dargestellt werden (Abb. 9.2).

```
> 1 - fa$uniqueness          # Kommunalitäten: 1 - Fehlervarianzen
[1] 0.16735135 0.12276446 0.24671733 0.12062093 0.47897514 0.04329568

> rowSums((fa$loadings)^2)      # Zeilensummen der quadrierten Ladungen
[1] 0.16733566 0.12273049 0.24671101 0.12059916 0.47897272 0.04333491

> Vx <- cov(X)                # Kovarianzmatrix der Daten

# geschätzte Korrelationsmatrix der beobachtbaren Variablen
# Lambda * Lambda^t + D_e
> KxEst <- fa$loadings %*% t(fa$loadings) + diag(fa$uniquenesses)

# geschätzte Kovarianzmatrix der beobachtbaren Variablen
> VxEst <- diag(sqrt(diag(Vx))) %*% KxEst %*% diag(sqrt(diag(Vx)))

# graphische Darstellung der Faktorladungen
> plot(fa$loadings, xlab="Faktor 1", ylab="Faktor 2",
+       xlim=c(-1.1, 1.1), ylim=c(-1.1, 1.1), pch=20, asp=1,
+       main="Faktorenanalyse: Ladungen und (rotierte) Faktoren")

> abline(h=0, v=0)              # Achsen mit Ursprung 0

# Faktoren einzeichnen
> arrows(0, 0, c(1, 0), c(0, 1), col="blue", lwd=2)
> angles <- seq(0, 2*pi, length.out=200)      # Winkel Einheitskreis
> circ   <- cbind(cos(angles), sin(angles))    # Koordinaten Einheitskr.
> lines(circ)                                # Einheitskreis
```

Durch orthogonale oder schiefwinklige Rotation der Faktoren lassen sich aus der von factanal() geschätzten Ladungsmatrix und den zugehörigen Faktorwer-

ten weitere Ladungsmatrizen berechnen, die eine ebensogute Reproduktion der Korrelationsmatrix K_x der beobachtbaren Variablen liefern. Ist dabei die Rotationsmatrix G eine Orthogonalmatrix ($G^{-1} = G^t$), ergibt sich die neue Ladungsmatrix $\tilde{\Lambda}$ im Modell unkorrelierter Faktoren aus der alten Ladungsmatrix durch $\tilde{\Lambda} = \hat{\Lambda} \cdot G$, und die Faktoren sind weiterhin unkorreliert (Abb. 9.2). Die neue geschätzte Korrelationsmatrix $\tilde{\Lambda} \cdot \tilde{\Lambda}^t + D_e$ stimmt dann mit $\hat{\Lambda} \cdot \hat{\Lambda}^t + D_e$ überein.

```

> ang <- pi/3                                # Rotationswinkel

# Matrix für orthogonale Rotation der Faktoren
> G <- matrix(c(cos(ang), -sin(ang), sin(ang), cos(ang)), nrow=2, byrow=TRUE)
> (Lrot <- fa$loadings %*% G)    # neue Ladungsmatrix
      [,1]      [,2]
[1,]  0.20872262  0.3518104
[2,] -0.28287244 -0.2066729
[3,]  0.49535060  0.0365896
[4,] -0.31652270  0.1428725
[5,]  0.26556867 -0.6390978
[6,] -0.08347933  0.1906990

# neue geschätzte Korrelationsmatrix der beobachtbaren Variablen
> KxEstRot <- Lrot %*% t(Lrot) + diag(fa$uniquenesses)
> all.equal(KxEst, KxEstRot)      # Kontrolle: stimmt mit alter überein
[1] TRUE

# rotierte Faktoren einzeichnen
> arrows(0, 0, G[1, ], G[2, ], col="green", lwd=2)
> segments(0, 0, -G[1, ], -G[2, ])

# Beschriftung der Variablen
> text(fa$loadings[, 1], fa$loadings[, 2]+0.06,
+       label=paste("Var", 1:nVar))

# Legende einfügen
> legend(x="bottomright", legend=c("Ladungen", "Faktoren",
+        "rotierte Faktoren"), pch=c(20, NA, NA),
+        lty=c(NA, 1, 1), col=c("black", "blue", "green"))

# Scree-Plot der Eigenwerte der Korrelationsmatrix der Daten
> plot(eigen(cor(X))$values, type="b", ylab="Eigenwert",
+       main="Scree-Plot")

> abline(h=1)                                # Referenzlinie für Kaiser-Kriterium

```

Ist G eine schiefwinklige Rotationsmatrix, ergibt sich die neue Ladungsmatrix $\tilde{\Lambda}$ aus der alten Ladungsmatrix durch $\tilde{\Lambda} = \hat{\Lambda} \cdot G^{t-1}$. Die neue Korrelationsmatrix der Faktoren ist $K_{\tilde{f}} = G^t \cdot K_{\hat{f}} \cdot G$. Die Faktorstruktur, also die Matrix der Korrelationen zwischen beobachtbaren Variablen und Faktoren, berechnet sich hier als $\hat{\Lambda} \cdot K_{\tilde{f}} \cdot G$. Die neue geschätzte Korrelationsmatrix $\tilde{\Lambda} \cdot K_{\tilde{f}} \cdot \tilde{\Lambda}^t + D_e$ stimmt dann mit $\hat{\Lambda} \cdot K_{\tilde{f}} \cdot \hat{\Lambda}^t + D_e$ überein.

```
# Matrix für schiefwinklige Rotation der Faktoren
> G <- matrix(c(5/3, -1, -4/3, 8/5), nrow=2, byrow=TRUE)

# neue Korrelationsmatrix der Faktoren (bisher: Einheitsmatrix)
> KfRot <- t(G) %*% diag(nFac) %*% G

# neue Ladungsmatrix
> Lrot <- fa$loadings %*% solve(t(G))

# neue Faktorstruktur
> facStruct <- fa$loadings %*% diag(nFac) %*% G

# neue geschätzte Korrelationsmatrix der beobachtbaren Variablen
> KxEstRot <- Lrot %*% KfRot %*% t(Lrot) + diag(fa$uniquenesses)
> all.equal(KxEst, KxEstRot)      # Kontrolle: stimmt mit alter überein
[1] TRUE
```

9.4 Multidimensionale Skalierung

Die Multidimensionale Skalierung ist ein weiteres Verfahren zur Dimensionsreduktion, das bestimmte Relationen zwischen Objekten durch deren Anordnung auf möglichst wenig Merkmalsdimensionen zu repräsentieren sucht. Die betrachtete Eigenschaft ist hier die globale Unähnlichkeit von je zwei Objekten. Die Ausgangssituation unterscheidet sich von jener in der Hauptkomponentenanalyse und der Faktorenanalyse dahingehend, dass zunächst unbekannt ist, bzgl. welcher Merkmale Objekte beurteilt werden, wenn über ihre generelle Ähnlichkeit zu anderen Objekten eine Aussage getroffen wird. Anders formuliert ist die Anzahl und die Bedeutung der Variablen, für die Objekte Werte besitzen, nicht gegeben. Dementsprechend werden in einer empirischen Erhebung auch nicht separat bestimmte Eigenschaften von einzelnen Objekten gemessen – vielmehr gilt es, in einem Paarvergleich je zwei Objekte im Sinne ihrer Unähnlichkeit zu beurteilen. Die Multidimensionale Skalierung sucht nun nach Merkmalsdimensionen, auf denen sich die Objekte als Punkte so anordnen lassen, dass die paarweisen Unähnlichkeitswerte möglichst gut mit dem euklidischen Abstand zwischen den Punkten übereinstimmen. Die metrische Multidimensionale Skalierung wird in R mit der `cmdscale()` Funktion durchgeführt.

```
> cmdscale(d=(Distanzmatrix), k=(Anzahl an Dimensionen), x.ret=FALSE)
```

Als Argument d ist eine symmetrische Matrix zu übergeben, deren Zeilen und Spalten dieselben Objekte repräsentieren. Die Abstände zwischen verschiedenen Objekten im Sinne von Werten eines Unähnlichkeitsmaßes sind in den Zellen außerhalb der Hauptdiagonale enthalten, die selbst überall 0 ist. Liegen als Daten nicht bereits die Abstände zwischen Objekten vor, sondern die separat für alle Objekte erhobenen Werte bzgl. derselben Variablen, können diese mit `dist()` in eine geeignete Distanzmatrix transformiert werden (vgl. Abschn. 2.9.3). Für k ist anzugeben, durch wieviele Variablen der Raum aufgespannt werden soll, in dem `cmdscale()` die Objekte anordnet und ihre euklidische Distanzen berechnet. Liegen die

Unähnlichkeitswerte zwischen n Objekten vor, kann die gewünschte Dimension höchstens $n - 1$ sein. Voreinstellung ist 2.

Die Ausgabe umfasst eine Matrix mit den n Koordinaten der Objekte im k -dimensionalen Variablenraum. Mit `x.ret=TRUE` enthält das Ergebnis zusätzlich die Matrix der paarweisen Distanzen der Objekte bzgl. der ermittelten Merkmalsdimensionen und ein Maß für die Güte der Anpassung der ursprünglichen Unähnlichkeiten durch die euklidischen Distanzen.

Als Beispiel liegen die Straßen-Entfernungen zwischen deutschen Städten vor, auf deren Basis die Multidimensionale Skalierung eine räumliche Anordnung auf zwei Dimensionen hervorbringen soll.

```
# Städte, deren Entfernungen berücksichtigt werden
> cities <- c("Augsburg", "Berlin", "Dresden", "Hamburg", "Hannover",
+           "Karlsruhe", "Kiel", "München", "Rostock", "Stuttgart")

> nCities <- length(cities)                                # Anzahl der Städte

# erstelle Distanzmatrix aus Vektor der Entfernungen
dstMat <- matrix(numeric(nCities^2), nrow=nCities)
cityDst <- c(596, 467, 743, 599, 226, 838, 65, 782, 160, 194, 288,
+          286, 673, 353, 585, 231, 633, 477, 367, 550, 542, 465,
+          420, 510, 157, 623, 96, 775, 187, 665, 480, 247, 632,
+          330, 512, 723, 298, 805, 80, 872, 206, 752, 777, 220, 824)

> dstMat[upper.tri(dstMat)] <- rev(cityDst)
> dstMat <- t(dstMat[, ncol(dstMat):1])[ , ncol(dstMat):1]
> dstMat[lower.tri(dstMat)] <- t(dstMat)[lower.tri(dstMat)]

# füge Städtenamen in Zeilen und Spalten hinzu
> dimnames(dstMat) <- list(city=cities, city=cities)
> (mds <- cmdscale(dstMat, k=2))                      # Multidimensionale Skalierung
      [,1]      [,2]
Augsburg   399.60802  -70.51443
Berlin    -200.20462  -183.39465
Dresden    -18.47337  -213.01950
Hamburg    -316.39991  130.72245
Hannover   -161.38209  120.21761
Karlsruhe   333.38724  212.12523
Kiel       -409.08703  147.62226
München    408.55752  -144.46446
Rostock    -401.19605  -126.20651
Stuttgart   365.19030  126.91200
```

Eine graphische Darstellung der ermittelten Koordinaten erlaubt es, das Ergebnis mit den tatsächlichen Positionen zu vergleichen (Abb. 9.3). Hier ist zu erkennen, dass das Ergebnis nur für manche Städte gut mit der realen Topographie übereinstimmt, auch wenn die West-Ost-Richtung gespiegelt wird.

```
> xLims <- range(mds[, 1]) + c(0, 250)      # x-Achsenbereich
> plot(mds, xlim=xLims, xlab="West-Ost", ylab="Süd-Nord", pch=16,
+       main="Anordnung der Städte nach MDS")
```

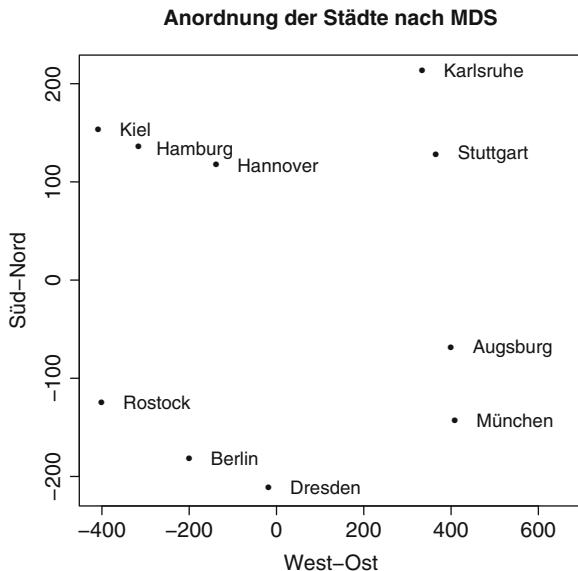


Abb. 9.3 Ergebnis der Multidimensionalen Skalierung auf Basis der Distanzen zwischen deutschen Städten

```
# füge Städtenamen hinzu
> text(mds[, 1]+50, mds[, 2], adj=0, label=cities)
```

Die metrische, nichtlineare Multidimensionale Skalierung versucht nicht, die euklidische Distanzen in exakte Übereinstimmung mit den Unähnlichkeiten zu bringen, sondern nur bis auf lineare Transformationen. Sie wird durch die `isoMDS()` Funktion aus dem MASS Paket bereit gestellt, die nonmetrische nichtlineare Variante nach Sammon und Kruskal durch `sammon()` aus demselben Paket. Beide Funktionen arbeiten ebenfalls auf Basis einer zu übergebenden Distanzmatrix und liefern zusätzlich zu den berechneten Koordinaten einen sog. Stress-Wert als Maß für die Übereinstimmung zwischen Unähnlichkeiten und Distanzen.

9.5 Hotellings T^2

9.5.1 Test für eine Stichprobe

Hotellings T^2 -Test für eine Stichprobe prüft die Datenvektoren mehrerer gemeinsam normalverteilter Variablen daraufhin, ob sie mit der Nullhypothese konsistent sind, dass ihr theoretisches Zentroid mit einem bestimmten Vektor übereinstimmt.⁶ Der Test lässt sich mit der `HotellingsT2()` Funktion aus dem ICSNP Paket

⁶ Für den Shapiro-Wilk-Test auf gemeinsame Normalverteilung vgl. die `mshapiro()` Funktion des `mvnormtest` Pakets.

(Nordhausen et al., 2010) durchführen. Sie arbeitet analog zu `t.test()` für den univariaten t -Test (vgl. Abschn. 8.2). Dieser ist zum Hotellings T^2 -Test äquivalent, wenn eine eindimensionale Variable getestet wird.

```
> HotellingsT2(X=<Datenmatrix>, mu=<Zentroid unter H0>)
```

Unter X ist die Datenmatrix einzutragen, bei der sich die Beobachtungsobjekte in den Zeilen und die Variablen in den Spalten befinden. Das Argument mu legt das theoretische Zentroid unter der Nullhypothese fest.

Im Beispiel sollen die Werte zweier Variablen betrachtet werden, deren Datenvektoren mit dem `mvtnorm` Paket simuliert werden. Zunächst sind nur die Daten aus einer Stichprobe (von später dreien) relevant.

```
# theoretische 2x2 Kovarianzmatrix für Zufallsvektoren
> sigma <- matrix(c(16, -1, -1, 9), byrow=TRUE, ncol=2)
> mu11 <- c(-3, 2) # theoretisches Zentroid Zufallsvektoren
> nSubj <- c(15, 20, 25) # Gruppengrößen - hier: nur 1. Gruppe
> library(mvtnorm) # Zufallsvektoren mit mvtnorm

# Datenmatrix mit Variablen in den Spalten
> Y11 <- round(rmvnorm(n=nSubj[1], mean=mu11, sigma=sigma))
> muH0 <- c(-1, 2) # Zentroid unter H0
> library(ICSNP) # Hotellings T^2 mit ICSNP Paket
> HotellingsT2(Y11, mu=muH0)
Hotelling's one sample T2-test
data: Y11
T.2 = 3.7599, df1 = 2, df2 = 13, p-value = 0.05146
alternative hypothesis: true location is not equal to c(-1,2)
```

Die Ausgabe nennt den empirischen Wert der Teststatistik (T.2), bei dem es sich nicht um Hotellings T^2 selbst, sondern bereits um den transformierten Wert handelt, der dann F -verteilt ist. Weiterhin sind die Freiheitsgrade der zugehörigen F -Verteilung (df1, df2) und der entsprechende p -Wert (p-value) aufgeführt.

Alternativ lässt sich der Test auch mit der aus der univariaten Varianzanalyse bekannten `anova(lm(<Formel>))` Funktion durchführen (vgl. Abschn. 8.3.5). In der für `lm()` anzugebenden Formel $\langle AV \rangle \sim \langle UV \rangle$ sind die $\langle AV \rangle$ Werte dabei multivariat, d. h. in Form einer spaltenweise aus den einzelnen Variablen zusammengestellten Matrix zu übergeben, von deren Zeilenvektoren zuvor das Zentroid unter der Nullhypothese abgezogen wurde. Der rechte $\langle UV \rangle$ Teil der Formel besteht hier nur aus dem absoluten Term 1. Als weitere Änderung lässt sich im multivariaten Fall das Argument `test` von `anova()` verwenden, um eine von mehreren multivariaten Teststatistiken auszuwählen.⁷ Dies kann "Hotelling-Lawley" für die Hotelling-Lawley-Spur sein, wobei die Wahl hier nicht relevant ist: wenn nur, wie hier, eine Bedingung oder zwei Bedingungen vorliegen, sind alle auswählbaren Teststatistiken zu Hotellings T^2 -Statistik äquivalent.

⁷ Bei der multivariaten Formulierung des Modells wird intern aufgrund der generischen `anova()` Funktion automatisch die `anova.mlm()` Funktion verwendet, ohne dass dies explizit angegeben werden muss (vgl. Abschn. 11.1.5).

```
# ziehe Zentroid unter H0 von allen Zeilenvektoren ab
> Y11ctr <- sweep(Y11, 2, muH0, "-")
> (anRes <- anova(lm(Y11ctr ~ 1), test="Hotelling-Lawley"))
   Df Hotelling-Lawley approx F num Df den Df Pr(>F)
(Intercept) 1          0.57845  3.7599      2     13  0.05146 .
Residuals   14
```

Das Ergebnis lässt sich auch manuell prüfen. Dabei kann verifiziert werden, dass der nicht transformierte T^2 -Wert gleich dem n -fachen der quadrierten Mahalanobisdistanz zwischen Zentroid der Daten und dem Zentroid unter der Nullhypothese bzgl. der korrigierten Kovarianzmatrix der Daten ist. Außerdem ist T^2 gleich dem $(n - 1)$ -fachen der Hotelling-Lawley-Spur.

```
> n    <- nSubj[1]                                # Stichprobengröße
> ctr <- colMeans(Y11)                           # Zentroid

# Hotellings T^2 Teststatistik
> (T2 <- n * (t(ctr-muH0) %*% solve(cov(Y11)) %*% (ctr-muH0)))
[1,]
[1,] 8.098296

# Kontrolle: n-faches der quadrierten Mahalanobisdistanz
> n * mahalanobis(ctr, muH0, cov(Y11))
[1] 8.098296

# Kontrolle: (n-1)-faches der Hotelling-Lawley-Spur
> (n-1) * anRes$"Hotelling-Lawley"[1]
[1] 8.098296

> P      <- 2                                    # Anzahl Variablen
> (T2f <- ((n-P) / (P*(n-1))) * T2)           # transformiertes T^2
[1,]
[1,] 3.759923

> (pVal <- 1-pf(T2f, P, n-P))                  # p-Wert
[1,]
[1,] 0.0514636
```

9.5.2 Test für zwei Stichproben

Hotellings T^2 -Test für zwei Stichproben prüft die in zwei Bedingungen erhobenen Datenvektoren mehrerer gemeinsam normalverteilter Variablen mit homogenen Kovarianzmatrizen daraufhin, ob sie mit der Nullhypothese konsistent sind, dass ihre theoretischen Zentroide übereinstimmen. Der Test ist äquivalent zum univariaten t -Test für zwei unabhängige Stichproben, wenn eine eindimensionale Variable getestet wird. Hotellings T^2 -Test lässt sich mit der `HotellingsT2()` Funktion aus dem ICSNP Paket durchführen.

```
> HotellingsT2(X=<Datenmatrix>, Y=<Datenmatrix>)
```

Unter X ist die Datenmatrix aus der ersten Bedingung einzutragen, bei der sich die Beobachtungsobjekte in den Zeilen und die Variablen in den Spalten befinden. Für Y ist die ebenso aufgebaute Datenmatrix aus der zweiten Bedingung zu nennen.

Das im vorangehenden Abschnitt begonnene Beispiel soll nun um die in einer zweiten Bedingung erhobenen Daten der betrachteten Variablen erweitert werden.

```
> mu12 <- c(-2, 0)                      # Zentroid Zufallsvektoren 2. Bedingung

# Datenmatrix aus 2. Bedingung mit Variablen in den Spalten
> Y12 <- round(rmvnorm(n=nSubj[2], mean=mu12, sigma=sigma))
> library(ICSNP)
> HotellingsT2(Y11, Y12)
Hotelling's two sample T2-test
data: Y11 and Y12
T.2 = 1.6223, df1 = 2, df2 = 32, p-value = 0.2133
alternative hypothesis: true location difference is not equal to c(0,0)
```

Die Ausgabe nennt den empirischen Wert der bereits transformierten Teststatistik ($T.2$, s. o.), die Freiheitsgrade der passenden F -Verteilung ($df1$, $df2$) und den zugehörigen p -Wert ($p\text{-value}$).

Alternativ lässt sich der Test auch mit `anova(lm((Formel)))` durchführen. Dabei sind in der Formel $\langle AV \rangle \sim \langle UV \rangle$ die $\langle AV \rangle$ Werte beider Gruppen in Form einer spaltenweise aus den einzelnen Variablen zusammengestellten Matrix zu übergeben, deren Zeilen von den Beobachtungsobjekten gebildet werden. $\langle UV \rangle$ codiert in Form eines Faktors für jede Zeile der $\langle AV \rangle$ Matrix, aus welcher Bedingung der zugehörige Datenvektor stammt. Das Argument `test` von `anova()` kann auf "Hotelling-Lawley" für die Hotelling-Lawley-Spur gesetzt werden.

```
> YH <- rbind(Y11, Y12)                  # Gesamt-Datenmatrix beider Bedingungen

# codiere für jede Zeile der Datenmatrix die zugehörige Bedingung
> facH <- factor(rep(1:2, nSubj[1:2]))
> anova(lm(YH ~ facH), test="Hotelling-Lawley")
Analysis of Variance Table

  Df Hotelling-Lawley approx F num Df den Df Pr(>F)
(Intercept) 1          1.0936   17.4970     2      32 7.342e-06 ***
facT         1          0.1014   1.6223     2      32      0.2133
Residuals   33
```

Die Ausgabe gleicht der aus der univariaten Anwendung von `anova()`, die Kennwerte des Tests des Faktors stehen in der mit seinem Namen bezeichneten Zeile.

Schließlich existiert mit `manova()` eine Funktion, die `aov()` (vgl. Abschn. 8.3.3) auf den multivariaten Fall verallgemeinert und ebenso wie diese aufzurufen ist. Die Formel ist wie in der multivariaten Anwendung von `anova(lm((Formel)))` zu formulieren. In der Anwendung von `summary()` auf das von `manova()` erzeugte Modell ist wieder als Teststatistik die Hotelling-Lawley-Spur zu wählen, indem das Argument `test` auf "Hotelling-Lawley" gesetzt wird.

```
> (sumRes <- summary(manova(YH ~ facT), test="Hotelling-Lawley"))
   Df Hotelling-Lawley approx F num Df den Df Pr(>F)
facT      1          0.10140    1.6223      2     32  0.2133
Residuals 33
```

Das Ergebnis lässt sich auch manuell prüfen. Dabei kann verifiziert werden, dass der nicht transformierte T^2 -Wert bis auf einen Vorfaktor gleich der quadrierten Mahalanobisdistanz beider Zentroide bzgl. einer geeigneten Schätzung der Kovarianzmatrix der Differenzvektoren ist. Außerdem ist T^2 gleich dem $(n_1 + n_2 - 1)$ -fachen der Hotelling-Lawley-Spur.

```
> n1 <- nSubj[1]                                # Gruppengröße 1
> n2 <- nSubj[2]                                # Gruppengröße 2
> ctr1 <- colMeans(Y11)                          # Zentroid 1. Bedingung
> ctr2 <- colMeans(Y12)                          # Zentroid 2. Bedingung

# unkorrigierte Kovarianzmatrizen aus beiden Bedingungen
> S1 <- cov.wt(Y11, method="ML")$cov
> S2 <- cov.wt(Y12, method="ML")$cov

# mit Stichprobengröße gewichtete Summe der Kovarianzmatrizen
> Su <- (1 / (n1+n2-2)) * (n1*S1 + n2*S2)

# Hotellings T^2
> (T2 <- ((n1*n2) / (n1+n2)) * (t(ctr2-ctr1) %*% solve(Su) %*% (ctr2-ctr1)))
[1]
[1,] 3.346066

# Kontrolle: quadrierte Mahalanobisdistanz beider Zentroide bzgl. Su
> ((n1*n2) / (n1+n2)) * mahalanobis(ctr1, ctr2, Su)
[1] 3.346066

# Kontrolle: T^2 = (n1+n2-2) * Hotelling-Lawley-Spur
> (n1+n2-2) * sumRes$stats[1, 2]
[1] 3.346066

> P <- 2                                         # Anzahl Variablen
> (T2f <- ((n1+n2-P-1) / ((n1+n2-2)*P)) * T2)  # transformiertes T^2
[1]
[1,] 1.622335

> (pVal <- 1-pf(T2f, P, n1+n2-P-1))           # p-Wert
[1]
[1,] 0.2132576
```

9.6 Multivariate Varianzanalyse (MANOVA)

9.6.1 Einfaktorielle MANOVA

Die einfaktorielle multivariate Varianzanalyse prüft die in mehreren Bedingungen einer UV erhobenen Datenvektoren mehrerer gemeinsam normalverteilter Variablen

mit homogenen Kovarianzmatrizen daraufhin, ob sie mit der Nullhypothese konsistent sind, dass ihre theoretischen Zentroide übereinstimmen. Der Test ist äquivalent zur univariaten einfaktoriellen Varianzanalyse, wenn eine eindimensionale Variable getestet wird.

Zur Durchführung eignet sich wie bei Hotellings T^2 -Test für zwei Stichproben die `manova()` Funktion als Verallgemeinerung von `aov()`. Dabei ist der `(AV)` Teil der Formel multivariat zu formulieren, also eine spaltenweise aus den Variablenwerten zusammengestellte Matrix mit allen Beobachtungen in den Zeilen zu übergeben. Für `(UV)` ist ein Faktor zu nennen, der für jede Zeile von `(AV)` codiert, aus welcher Bedingung der Datenvektor stammt.

Im Test der von `manova()` durchgeführten Modellanpassung mit `summary()` können verschiedene Teststatistiken über das Argument `test` gewählt werden: Voreinstellung ist "Pillai" für die Pillai-Bartlett-Spur, andere Optionen sind "Wilks" für Wilks' Λ , "Roy" für Roys Maximalwurzel und "Hotelling-Lawley" für die Hotelling-Lawley-Spur. Bei zwei Gruppen sind alle Teststatistiken äquivalent, ebenso bei nur einer Abhängigen Variable.

Das im vorangehenden Abschnitt begonnene Beispiel soll nun um die in einer dritten Bedingung erhobenen Daten der betrachteten beiden Variablen erweitert werden.

```
> mu13 <- c(-4, -1)                                     # Zentroid Zufallsvektoren 3. Bedingung

# Datenmatrix aus 3. Bedingung mit Variablen in den Spalten
> Y13 <- round(rmvnorm(n=nSubj[3], mean=mu13, sigma=sigma))
> YM  <- rbind(Y11, Y12, Y13)                         # Gesamt-Datenmatrix aller Bedingungen

# codiere für jede Zeile der Datenmatrix die zugehörige Bedingung
> facM   <- factor(rep(1:3, nSubj))
> manRes0 <- manova(YM ~ facM)                      # manova() Modell
> summary(manRes0, test="Pillai")                     # Pillai-Bartlett-Spur
      Df    Pillai approx F num Df den Df   Pr(>F)
facM     2  0.23326   3.7628       4      114  0.006511 **
Residuals 57

> summary(manRes0, test="Wilks")                      # Wilks' Lambda
      Df    Wilks approx F num Df den Df   Pr(>F)
facM     2  0.77238   3.8598       4      112  0.00563 ***
Residuals 57

> summary(manRes0, test="Roy")                        # Roys Maximalwurzel
      Df      Roy approx F num Df den Df   Pr(>F)
facM     2  0.25924   7.3882       2      57  0.001403 **
Residuals 57
> summary(manRes0, test="Hotelling-Lawley")          # Hotelling-Lawley-Spur
      Df Hotelling-Lawley approx F num Df den Df   Pr(>F)
facM     2           0.2874   3.9518       4      110  0.004909 **
Residuals 57
```

Das Ergebnis lässt sich auch manuell prüfen, indem zunächst die (noch redundante) Designmatrix ohne absoluten Term erstellt wird (vgl. Abschn. 8.3.1). Sie enthält für

jede Bedingung eine Indikatorvariable als Spalte, die für jede Zeile der Datenmatrix angibt, ob sie aus der zugehörigen Bedingung stammt (1) oder nicht (0). Mit Hilfe dieser Designmatrix lassen sich die relevanten orthogonalen Projektionen definieren (vgl. Abschn. 2.9.4). Diese führen letztlich zu den Matrizen B (SSCP-Matrix der Gruppenzentroide), W (SSCP-Matrix der Residuen) und T (SSCP-Matrix der Daten) als multivariate Verallgemeinerung der Quadratsummen zwischen (Between, B) und innerhalb (Within, W) der Gruppen, sowie der totalen Quadratsumme (T) als Summe von B und W . Die Teststatistiken berechnen sich auf Basis dieser Matrizen wie folgt:

- Pillai-Bartlett-Spur: $\text{Spur}(T^{-1} \cdot B)$.
- Wilks' Λ : $\det(W)/\det(W + B) = \det(W)/\det(T)$.
- Roys Maximalwurzel: entweder der größte Eigenwert θ_1 von $T^{-1} \cdot B$ (so definiert in R) oder der größte Eigenwert λ_1 von $W^{-1} \cdot B$. Für die Umrechnung von θ_1 und λ_1 gilt $\lambda_1 = \theta_1/(1 - \theta_1)$ sowie $\theta_1 = \lambda_1/(1 + \lambda_1)$.
- Hotelling-Lawley-Spur: $\text{Spur}(W^{-1} \cdot B)$.

```
# Designmatrix
> X <- cbind(as.numeric(facM == 1), as.numeric(facM == 2),
+              as.numeric(facM == 3))

# orthogonale Projektion auf durch Designmatrix aufgespannten Raum
> P      <- X %*% solve(t(X) %*% X) %*% t(X)
> Pw     <- diag(nrow(P)) - P           # "within"-Projektion
> ones   <- rep(1, nrow(X))            # Basisvektor aus 1-Einträgen

# Projektion auf entsprechenden Unterraum
> Pm <- ones %*% solve(t(ones) %*% ones) %*% t(ones)
> Pt <- diag(nrow(Pm)) - Pm          # "total"-Projektion
> Pb <- P - Pm                      # "between"-Projektion
> TT <- t(YM) %*% Pt %*% YM        # Matrix T
> BB <- t(YM) %*% Pb %*% YM        # B
> WW <- t(YM) %*% Pw %*% YM        # W
> all.equal(TT, BB + WW)             # Kontrolle: T=B+W
[1] TRUE

> (PB <- sum(diag(solve(TT) %*% BB))) # Pillai-Bartlett-Spur
[1] 0.2332605

> (WL <- det(WW) / det(TT))          # Wilks Lambda
[1] 0.7723788

# Roys Maximalwurzel - theta
> (RLRt <- max(eigen(solve(TT) %*% BB)$values))
[1] 0.2058678

# Roys Maximalwurzel - lambda
> (RLRl <- max(eigen(solve(WW) %*% BB)$values))
[1] 0.2592361
```

```
> RLRt / (1-RLRt) # lambda aus theta
[1] 0.2592361

> (HL <- sum(diag(solve(WW) %*% BB))) # Hotelling-Lawley-Spur
[1] 0.2874003
```

9.6.2 Mehrfaktorielle MANOVA

Die mehrfaktorielle multivariate Varianzanalyse ist wie die einfaktorielle MANOVA mit `manova()` durchzuführen, lediglich die Spezifikation der `(UV)` Seite der Formel erweitert sich um die zusätzlich zu berücksichtigenden Effekte. In der Rolle der `(AV)` steht weiterhin eine spaltenweise aus den Variablen zusammengesetzte Matrix, deren Zeilen aus allen Beobachtungen bestehen. Für `(UV)` können nun die Faktoren genannt werden, deren Stufen die Bedingungskombinationen festlegen, in denen die AVn erhoben wurden. Jeder Faktor gibt für jede Zeile der Datenmatrix an, aus welcher Bedingung bzgl. der durch ihn codierten UV der Datenvektor stammt. Sollen bei zwei UVn nur die Haupteffekte berücksichtigt werden, lautete die Formel wie im univariaten Fall $\langle AV \rangle \sim \langle UV_1 \rangle + \langle UV_2 \rangle$, mit Einbeziehung der Interaktion entsprechend $\langle AV \rangle \sim \langle UV_1 \rangle * \langle UV_2 \rangle$.

```
> mu21 <- c(-2, 1)
> mu22 <- c(-1, -1)
> mu23 <- c(-5, -2)
> Y21 <- round(rmvnorm(nSubj[1], mu21, sigma))
> Y22 <- round(rmvnorm(nSubj[2], mu22, sigma))
> Y23 <- round(rmvnorm(nSubj[3], mu23, sigma))
> YMt <- rbind(YM, Y21, Y22, Y23)
# codiere für Zeilen der Datenmatrix zugehörige Bedingung beider UVn
> facT1 <- rep(facM, 2)
> facT2 <- factor(rep(1:2, each=sum(nSubj)))
> manResT <- manova(YMt ~ facT1 + facT2) # Test der Haupteffekte

> summary(manResT, test="Pillai") # Pillai-Bartlett-Spur
      Df    Pillai approx F num Df den Df   Pr(>F)
facT1     2  0.278003   9.3636      4     232 4.942e-07 ***
facT2     1  0.080618   5.0420      2     115  0.007962 **
Residuals 116

> summary(manResT, test="Wilks") # Wilks' Lambda
      Df    Wilks approx F num Df den Df   Pr(>F)
facT1     2  0.73824   9.4219      4     230 4.537e-07 ***
facT2     1  0.91938   5.0420      2     115  0.007962 **
Residuals 116

> summary(manResT, test="Roy") # Roys Maximalwurzel
      Df      Roy approx F num Df den Df   Pr(>F)
facT1     2  0.241417  14.002      2     116 3.571e-06 ***
facT2     1  0.087687   5.042      2     115  0.007962 **
Residuals 116
```

```
> summary(manResT, test="Hotelling-Lawley")      # Hotelling-Lawley-Spur
      Df  Hotelling-Lawley approx F num Df den Df   Pr(>F)
facT1      2          0.33256   9.4781      4     228 4.181e-07 ***
facT2      1          0.08769   5.0420      2     115  0.007962 **
Residuals 116
```


Kapitel 10

Diagramme erstellen

Daten lassen sich in R mit Hilfe einer Vielzahl von Diagrammtypen graphisch darstellen. Nur auf wenige der verfügbaren Typen kann dabei hier Bezug genommen werden.¹ In R werden zwei Arten von Graphikfunktionen unterschieden: sog. High-Level-Funktionen erstellen eigenständig ein vollständiges Diagramm inklusive Achsen, während sog. Low-Level-Funktionen lediglich ein bestimmtes Element einem bestehenden Diagramm hinzufügen. Einen kurzen Überblick über die Gestaltungsmöglichkeiten vermittelt `demo(graphics)`.

10.1 Graphik-Devices

10.1.1 Aufbau und Verwaltung von Graphik-Devices

Die Ausgabe von Befehlen zum Erstellen einer Graphik kann in verschiedenen Ausgabekanälen, sog. Devices erfolgen. Ein Device lässt sich wie ein leeres Blatt Papier vorstellen, auf dem mit Graphikfunktionen einzelne Inhalte eingezeichnet werden. Die Fläche des Devices ist dabei in drei Regionen eingeteilt: die gesamte Device-Region mit den äußereren Rändern, die innerhalb dieser Ränder liegende sog. Figure-Region und die in diese eingebettete sog. Plot-Region, in die die Datenpunkte und andere Graphikelemente eingezeichnet werden (Abb. 10.1).² In der Voreinstellung ist ein Device ein separates Graphikfenster innerhalb der R-Umgebung, es können

¹ Zudem soll fast ausschließlich auf das graphische Basissystem, nicht aber auf fortgeschrittene Alternativen eingegangen werden, wie sie etwa durch die Pakete `lattice` (Sarkar, 2002, 2008; vgl. Abschn. 10.8.3) oder `ggplot2` (Wickham, 2009, 2010) realisiert werden. Ebenso bleiben Funktionen weitgehend ausgespart, die eine Interaktion in Echtzeit mit Graphiken erlauben, vgl. dazu die Pakete `ipplots` (Urbanek und Wichtrey, 2009) und `rggobi` (Temple Lang et al., 2009) sowie Abschn. 10.8.2. Für eine umfassende Dokumentation der Möglichkeiten inklusive Beispielen vgl. Murrell (2005) und die R Graph Gallery (François 2006).

² Die Regionen besitzen unterschiedliche Koordinatensysteme, über die sich Elemente nicht nur in der Plot-Region, sondern auch direkt in der Figure- oder der gesamten Device-Region plazieren lassen. Die Beschränkung, nur in die Plot-Region zeichnen zu können (sog. Clipping), kann über das Argument `xpd` der `par()` Funktion aufgehoben werden (vgl. Abschn. 10.3.1 sowie die `cnvrt.coords()` Funktion aus dem Paket `Hmisc`).

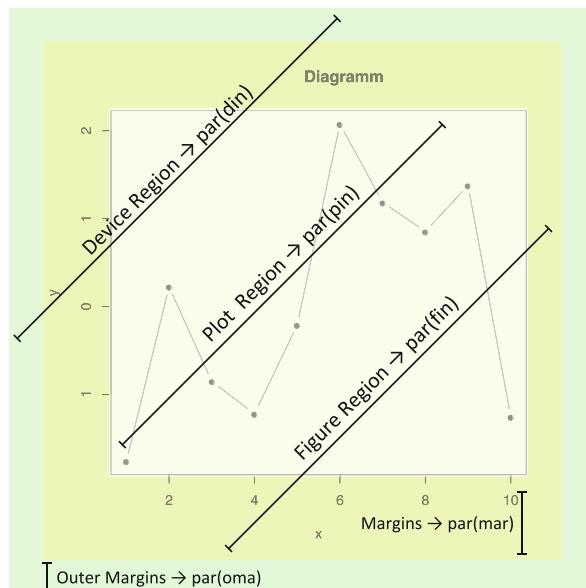


Abb. 10.1 Regionen und Ränder eines Graphik-Devices samt Möglichkeiten, ihre Größe mit `par()` ausgeben und ggf. verändern zu können (vgl. Abschn. 10.3.1)

aber auch Dateien in verschiedenen Graphikformaten oder Drucker als Graphik-Device dienen.

Sofern noch kein Graphikfenster existiert, öffnet es sich mit Eingabe des ersten High-Level-Graphikbefehls automatisch. In dieses Fenster werden dann alle weiteren Ausgaben graphischer Funktionen hinein gezeichnet, wobei im Fall von High-Level-Funktionen ein ggf. bereits vorhandener Inhalt gelöscht wird. Soll für die Ausgabe einer Graphikfunktion zusätzlich zu bereits bestehenden ein neues, zunächst leeres Fenster geöffnet werden, geschieht dies mit

```
> windows(width=<Breite>, height=<Höhe>)
```

Unter MacOS X ist `quartz()` und unter Unix/Linux `x11()` der äquivalente Befehl. Breite und Höhe des Fensters können über die Argumente `width` und `height` in der Einheit inch bestimmt werden (Voreinstellung ist 7 in). Bei mehreren geöffneten Graphik-Devices sind alle bis auf eines inaktiv, das im Fenstertitel die Bezeichnung (ACTIVE) trägt. Die Bezeichnung bedeutet, dass in dieses Device die Ausgabe der folgenden Graphikfunktion gezeichnet wird, während die Inhalte der anderen Devices unverändert bleiben. Um sich einen Überblick über alle aktuell geöffneten Graphik-Devices zu verschaffen, dient der Befehl `dev.list()`.

```
> windows(); windows(); windows()
> dev.list()
windows windows windows
  2      3      4
```

Der Output zeigt, welche Ausgabekanäle offen sind und zu welchem Typ sie gehören. Weiterhin enthält die Ausgabe die laufende Nummer jedes Devices – das erste Device erhält dabei die Nummer 2. Um zu erfahren, welcher der Ausgabekanäle aktiv ist, dient die Funktion `dev.cur()` (Device Current).

```
> dev.cur()
windows
4
```

Der Output besteht in der fortlaufenden Nummer des aktiven Devices und seines Typs. Die Funktionen `dev.prev()` (Device Previous) und `dev.next()` geben bei mehreren geöffneten Devices die Nummer desjenigen Devices zurück, das sich unmittelbar vor bzw. unmittelbar hinter dem aktiven Device befindet. Diese Information kann etwa zum Wechseln des aktiven Devices mit `dev.set(which=<Nummer>)` verwendet werden, damit die Ausgabe der folgenden Graphikfunktionen dort erfolgt.

```
> dev.set(3)
windows
3

> dev.set(dev.next())
windows
4
```

Das aktive Device wird mit `dev.off()` geschlossen. Handelt es sich um ein Graphikfenster, hat dies denselben Effekt, wie das Fenster per Maus zu schließen. Die Ausgabe von `dev.off()` gibt an, welches fortan das aktive Ausgabe-Device ist. In Abwesenheit geöffneter Devices ist dies das sog. NULL Device mit der Nummer 1. Alle offenen Devices lassen sich gleichzeitig mit `graphics.off()` schließen.

```
> dev.off()                                # aktuelles Device schließen
windows
2

> graphics.off()                            # alle Devices schließen
```

10.1.2 Graphiken speichern

Alles, was sich in einem Graphikfenster anzeigen lässt, kann auch als Datei gespeichert werden: ist ein Graphikfenster aktiviert, so ändert sich das Menü der R-Umgebung dahingehend, dass über Datei: Speichern als: die Graphik in vielen Formaten gespeichert werden kann. Als Alternative erlaubt ein sich durch Rechtsklick auf das Graphikfenster öffnendes Kontextmenü, die Graphik in wenigen Formaten zu speichern. Dasselbe Kontextmenü enthält auch Einträge, um die Graphik in einem bestimmten Format in die Zwischenablage zu kopieren und so direkt anderen Programmen verfügbar zu machen.

Graphiken lassen sich auch befehlsgesteuert ohne den Umweg eines Graphikfensters in Dateien speichern. Unabhängig davon, in welchem Format dies geschehen

soll, sind dafür drei Arbeitsschritte notwendig: Zunächst muss die Datei als Ausgabekanal (also als aktives Graphik-Device) festgelegt werden. Dazu dient etwa die `pdf()` Funktion, wenn die Graphik im PDF-Format zu speichern ist. Es folgen Befehle zum Erstellen von Diagrammen oder Einfügen von Graphikelementen, deren Ausgabe nicht auf dem Bildschirm erscheint, sondern direkt in die Datei umgeleitet wird. Schließlich ist der `dev.off()` oder `graphics.off()` Befehl notwendig, um die Ausgabe in die Datei zu beenden und das Device zu schließen.

Als Dateiformate stehen viele der üblichen bereit, vgl. `?device` für eine Aufstellung. Beispielhaft seien hier die `pdf()` und `jpeg()` Funktionen betrachtet.

```
> pdf(file="(Dateiname)", width=(Breite), height=(Höhe))
> jpeg(filename="(Dateiname)", width=(Breite), height=(Höhe),
+      units="px", quality=(Bildqualität))
```

Unter `file` bzw. `filename` ist der Name der Ausgabedatei einzutragen. Sollen mehrere Graphiken mit gleichem Namensschema unter Zuhilfenahme einer fortlaufenden Nummer erzeugt werden, ist ein spezielles Namensformat zu verwenden, das in der Hilfe erläutert wird. Mit `width` und `height` wird die Größe der Graphik kontrolliert. Beide Angaben sind in der `pdf()` Funktion in der Einheit inch zu tätigen, während bei `jpeg()` über das Argument `units` festgelegt werden kann, auf welche Maßeinheit sie sich beziehen. Voreinstellung ist die Anzahl der Pixel, als Alternativen stehen `in` (inch), `cm` und `mm` zur Auswahl. Schließlich kann bei Bildern im JPEG-Format festgelegt werden, wie stark die Daten komprimiert werden sollen, wobei die Kompression mit einem Verlust an Bildinformationen verbunden ist. Das Argument `quality` erwartet einen sich auf die höchstmögliche Bildqualität beziehenden Prozentwert, wobei ein kleinerer Wert eine geringere Bildqualität bedeutet, die dann stärkere Kompression dafür aber auch zu geringeren Dateigrößen führt.

Hier soll demonstriert werden, wie eine einfache Graphik im PDF-Format gespeichert wird.

```
> pdf("pdf_test.pdf") # Device öffnen (mit Dateinamen)
> plot(1:10, rnorm(10)) # Graphik einzeichnen
> dev.off() # Device schließen
```

Die Inhalte eines aktiven Graphik-Devices können durch `dev.copy()` in ein neues Device kopiert werden. Dabei ist entweder die Nummer des bereits geöffneten Ziel-Devices zu nennen oder anzugeben, welche Art von Device mit den bestehenden Inhalten neu geöffnet werden soll.

```
> dev.copy(device=<Device-Typ>, ..., which=<Device-Nummer>)
```

Ist das Ziel-Device noch nicht geöffnet, findet das device Argument Verwendung, das (ohne Anführungszeichen) den Namen einer Funktion erwartet, mit der ein Device eines bestimmten Typs geöffnet werden kann, etwa pdf. Andernfalls gibt which die Nummer des geöffneten Ziel-Devices an. Benötigt das neu zu erstellende Device seinerseits Argumente – etwa Dateinamen oder Angaben zur Größe der Graphik, so können diese anstelle der . . . genannt werden.

```
> plot(1:10, rnorm(10)) # Graphikfenster mit Diagramm öffnen
```

```
# Inhalt des Fensters in JPEG-Bild speichern,
# dabei Dateinamen und Kompressionsgrad angeben
> dev.copy(jpeg, filename="copied.jpg", quality=90)
> graphics.off()                                # beide offenen Devices schließen
```

10.2 Streu- und Liniendiagramme

In zweidimensionalen Streudiagrammen (sog. Scatterplots) werden mit der `plot()` Funktion Wertepaare in Form von Punkten in einem kartesischen Koordinatensystem dargestellt, wobei ein Wert die Position des Punkts entlang der Abszisse und der andere Wert die Position des Punkts entlang der Ordinate bestimmt. Die Punkte können dabei für ein Liniendiagramm durch Linien verbunden oder für ein Streudiagramm als Punktfolge belassen werden.³

10.2.1 Streudiagramme mit `plot()`

```
> plot(x=<Vektor>, y=<Vektor>, type=<Option>, main=<Diagrammtitel>,
       sub=<Untertitel>, asp=<Seitenverhältnis Höhe/Breite>)
```

Unter x und y sind die x- bzw. y-Koordinaten der Punkte jeweils als Vektor einzutragen. Wird nur ein Vektor angegeben, werden seine Elemente als y-Koordinaten interpretiert und die x-Koordinate jedes Datenpunkts gleich dem Index des zugehörigen Vektorelements gesetzt. Das Argument `type` hat mehrere mögliche Ausprägungen, die das Aussehen der Datenmarkierungen im Diagramm bestimmen (Tabelle 10.1, Abb. 10.2). Der Diagrammtitel kann als Zeichenkette für `main` angegeben werden, der Untertitel für `sub`. Das Verhältnis von Höhe zu Breite jeweils einer Skaleneinheit im Diagramm kontrolliert das Argument `asp` (Aspect Ratio), für weitere Argumente vgl. `?plot.default`.

```
> vec <- rnorm(1:10)

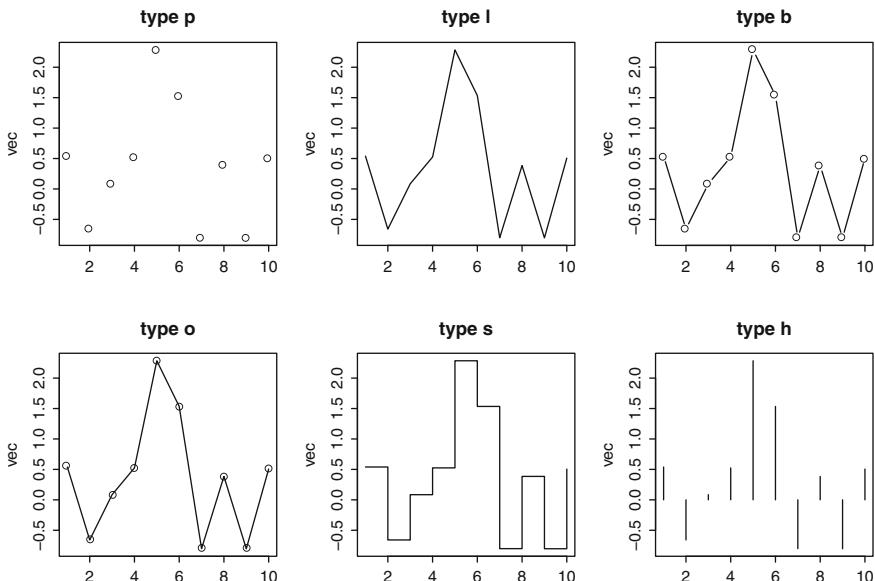
# das Argument xlab=NA entfernt die Bezeichnung der x-Achse
> plot(vec, type="p", xlab=NA, main="type p")
> plot(vec, type="l", xlab=NA, main="type l")
> plot(vec, type="b", xlab=NA, main="type b")
> plot(vec, type="o", xlab=NA, main="type o")
> plot(vec, type="s", xlab=NA, main="type s")
> plot(vec, type="h", xlab=NA, main="type h")
```

Die Koordinaten der Punkte können auch über eine Formel angegeben werden, womit der Aufruf `plot(<y-Koordinaten> ~ <x-Koordinaten>, ...)`

³ Das Paket `car` enthält mit `scatterplot()` eine erweiterte Funktion für Streudiagramme.

Tabelle 10.1 Mögliche Werte für das Argument type der plot() Funktion

Wert für type	Bedeutung
"p"	Punkte
"l"	durchgehende Linien. Durch eng gesetzte Punkte können Funktionskurven beliebiger Form approximiert werden.
"b"	Punkte und Linien
"c"	unterbrochene Linien
"o"	Punkte und Linien, aber überlappend
"h"	senkrechte Linien zu jedem Datenpunkt (Spike Plot)
"s"	Stufendiagramm
"S"	Stufendiagramm mit anderer Reihenfolge von vertikalen und horizontalen Schritten zur nächsten Stufe
"n"	fügt dem Diagramm keine Datenpunkte hinzu (No Plotting)

**Abb. 10.2** Mit `plot(type = "<option>")` erzeugbare Diagrammarten

`→data=(Datensatz)`) lautet.⁴ Wenn die in der Formel verwendeten Variablen Teil eines Datensatzes sind, kann dieser unter `data` genannt werden.

Sind nur wenige Wertepaare möglich und treten deswegen in den Daten mehrfach auf, würden mehrere gleiche Wertepaare durch dasselbe Symbol im Diagramm dargestellt. Sollen dagegen unter Verzicht auf die präzise Positionierung ebenso viele Symbole wie Wertepaare angezeigt werden, kann dies mit `jitter(<Variable>)` erreicht werden. Diese Funktion ändert die Werte der Variable um einen kleinen zufälligen Betrag und bewirkt dadurch beim Zeichnen jedes Datenpunkts einen zufälligen Versatz entlang der zugehörigen Achse (Abb. 10.3).⁵

```
> vec1 <- sample(1:10, 100, replace=TRUE)
> vec2 <- sample(1:10, 100, replace=TRUE)
> plot(vec2 ~ vec1, main="Punktfolke")      # Datenpunkte ohne jitter()

# Datenpunkte mit zufälligem Versatz in Richtung der y-Achse durch jitter()
> plot(jitter(vec2) ~ vec1, main="Punktfolke mit jitter")
```

Bei sehr vielen darzustellenden Punkten taucht mitunter das Problem auf, dass die Datenpunktssymbole einander überdecken und einzelne Werte so nicht mehr identifizierbar sind. Indem als Datenpunktssymbol mit `pch=". "` der Punkt gewählt wird, lässt sich dies bis zu einem gewissen Grad verhindern. Auch der Einsatz von simulierter Transparenz für die Datenpunktssymbole kann einzelne Werte identifizierbar halten: bei einer sehr durchlässig gewählten Farbe erzeugt ein einzelner Datenpunkt nur einen schwachen Abdruck, während viele aufeinander liegende Punkte die Farbe an dieser Stelle des Diagramms immer gesättigter machen (vgl. Abschn. 10.3.2, Fußnote 6 und Abb. 9.1 für ein Beispiel). Alternativ lässt sich mit der `hexbin()` Funktion aus dem gleichnamigen Paket (Carr et al., 2009) ein Diagramm erstellen,

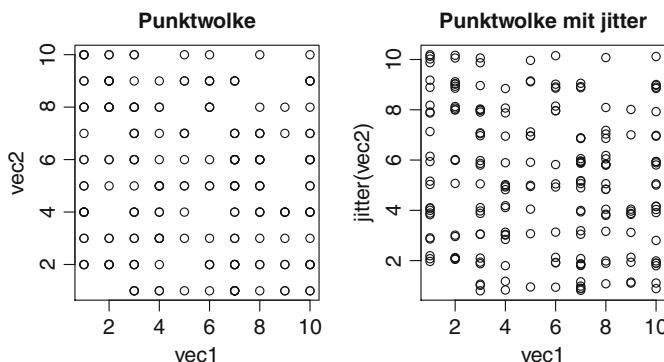


Abb. 10.3 Streudiagramm ohne und mit Anwendung von `jitter()`

⁴ Hat die Formel dagegen die Form `<y-Koordinaten>~<Faktor>`, werden in einem Diagramm getrennt für die von `<Faktor>` definierten Gruppen Boxplots dargestellt (vgl. Abschn. 10.6.3), da `plot()` eine generische Funktion ist (vgl. Abschn. 11.1.5).

⁵ Als Alternative zu diesem Vorgehen kommt die Funktion `sunflowerplot()` in Betracht.

dass die Diagrammfläche in hexagonale Regionen einteilt und die Dichte der Datenpunkte in jeder Region ähnlich einem Höhenlinien-Diagramm (vgl. Abschn. 10.8.1) farblich repräsentiert.

10.2.2 Datenpunkte eines Streudiagramms identifizieren

Werden viele Daten in einem Streudiagramm dargestellt, ist es häufig nicht ersichtlich, zu welchem Wert ein bestimmter Datenpunkt gehört. Diese Information kann jedoch interessant sein, wenn etwa erst die graphische Betrachtung eines Datensatzes Besonderheiten der Verteilung verrät und die für bestimmte Datenpunkte verantwortlichen Untersuchungseinheiten identifiziert werden sollen. Die `identify()` Funktion erlaubt eine solche interaktive Identifikation von Werten in einem Streudiagramm.

```
> identify(x=(x-Koordinaten), y=(y-Koordinaten))
```

Für `x` und `y` sollten dieselben Daten in Form von Vektoren mit `x`- und `y`-Koordinaten übergeben werden, die zuvor in einem noch geöffneten Graphikfenster als Streudiagramm dargestellt wurden. Wird nur ein Vektor angegeben, werden seine Werte als `y`-Koordinaten interpretiert und die `x`-Koordinaten durch die Indizes des Vektors gebildet. Durch Ausführen von `identify()` ändert sich der Mauszeiger zu einem Kreuz, sobald er über der Diagrammfläche plaziert wird. Mit einem Klick der linken Maustaste wird derjenige Datenpunkt identifiziert, der der Mausposition am nächsten liegt und sein Index dem Diagramm neben dem Datenpunkt hinzugefügt. Die Konsole ist in dieser Zeit blockiert. Der Vorgang kann mehrfach wiederholt und schließlich durch Klicken der rechten Maustaste über ein Kontextmenü beendet werden, woraufhin `identify()` die Indizes der ausgemachten Punkte zurückgibt.

```
> vec <- rnorm(10)
> plot(vec)
> identify(vec)
```

10.2.3 Streudiagramme mit `matplot()`

So wie durch `plot()` ein Streudiagramm einer einzelnen Datenreihe erstellt wird, erzeugt `matplot()` ein Streudiagramm für mehrere Datenreihen gleichzeitig (Abb. 10.4).

```
> matplot(x=(Matrix), y=(Matrix), type="(Option)", pch="(Symbol)")
```

Die Argumente sind dieselben wie für `plot()`, lediglich `x`- und `y`-Koordinaten können nun als Matrizen an `x` und `y` übergeben werden, wobei jede ihrer Spalten als eine separate Datenreihe interpretiert wird. Haben dabei alle Datenreihen dieselben `x`-Koordinaten, kann `x` auch ein Vektor sein. Wird nur eine Matrix mit Koordinaten angegeben, werden diese als `y`-Koordinaten gedeutet und die `x`-Koordinaten durch die Zeilenindizes der Werte gebildet. In der Voreinstellung werden die Datenreihen in unterschiedlichen Farben dargestellt. Als Symbol für jeden Datenpunkt dienen

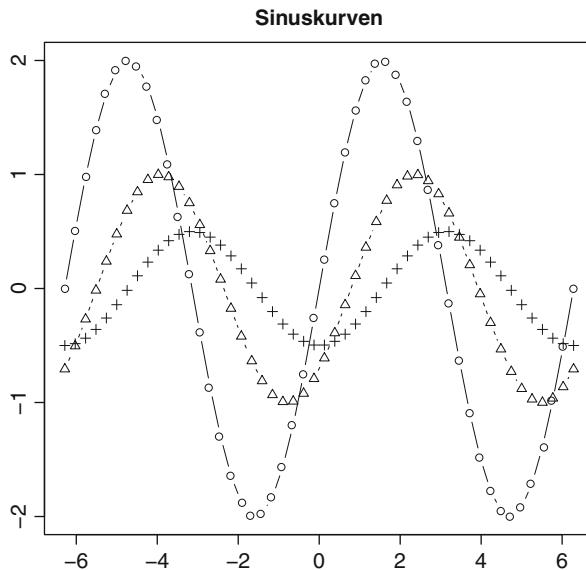


Abb. 10.4 Mit `matplotlib()` erzeugtes Streudiagramm

die Ziffern 1–9, die mit der zur Datenreihe gehörenden Spaltennummer korrespondieren. Mit dem Argument `pch` können auch andere Symbole Verwendung finden (Abb. 10.5, vgl. Abschn. 10.3.1).

```
> vec <- seq(from=-2*pi, to=2*pi, length.out=50)
> mat <- cbind(2*sin(vec), sin(vec-(pi/4)), 0.5*sin(vec-(pi/2)))
> matplot(vec, mat, type="b", xlab=NA, ylab=NA, pch=1:3,
+         main="Sinuskurven")
```

10.3 Diagramme formatieren

Die `plot()` Funktion akzeptiert wie auch andere High-Level-Funktionen zum Erstellen von Diagrammen eine Vielzahl weiterer Argumente, mit denen die Graphik weitestgehend flexibel angepasst werden kann. Einige der wichtigsten Möglichkeiten zur individuellen Diagrammgestaltung werden im folgenden vorgestellt.

10.3.1 Graphikelemente formatieren

Die Formatierung von Graphikelementen ist in vielen Aspekten variabel, etwa hinsichtlich der Art, Größe und Farbe der verwendeten Symbole oder der Orientierung der Achsenbeschriftungen. Zu diesem Zweck akzeptieren die meisten Funktionen zum Erstellen von Diagrammen einen gemeinsamen Satz zusätzlicher Argumente, auch wenn diese nicht immer in den jeweiligen Hilfeseiten mit aufgeführt sind. Darunter befinden sich einige der in Tabelle 10.2 beschriebenen Argumente für die

Tabelle 10.2 Graphikoptionen steuern mit Argumenten der `par()` Funktion

Argument	Wert	Bedeutung
<code>bty</code>	"o", "l", "n"	Bei "o" wird ein Rahmen um die die Datenpunkte enthaltende Plot-Region gezogen, bei "l" und "n" nicht
<code>cex</code>	(Zahl)	Vergrößerungsfaktor für die Datenpunkt-Symbole. Voreinstellung ist der Wert 1
<code>cex.axis</code>	(Zahl)	Vergrößerungsfaktor für die Achsenbeschriftungen. Voreinstellung ist der Wert 1
<code>cex.lab</code>	(Zahl)	Vergrößerungsfaktor für die Schrift der Achsenbezeichnungen. Voreinstellung ist der Wert 1
<code>col</code>	"(Farbe)"	Farbe der Datenpunkt-Symbole sowie bei <code>par()</code> zusätzlich des Rahmens um die Plot-Region (vgl. Abschn. 10.3.2 für mögliche Werte)
<code>las</code>	0, 1, 2, 3	Orientierung der Achsenbeschriftungen. Für senkrecht zur Achse stehende Beschriftungen ist der Wert auf 2 zu setzen
<code>lty</code>	1, 2, 3, 4, 5, 6 bzw. (\ →Schlüsselwort)	Linientyp: Schlüsselwörter sind "solid", "dashed", "dotted", "dotdash", "longdash", "twodash" (Abb. 10.5)
<code>lwd</code>	(Zahl)	Linienstärke (Line Width), auch bei Datenpunktssymbolen. Voreinstellung ist der Wert 1
<code>mar</code>	(Vektor)	Ränder zwischen Plot- und Figure-Region eines Diagramms (Abb. 10.1). Angabe als Vielfaches der Zeilenhöhe in Form eines Vektors mit vier Elementen, die jeweils dem unteren, linken, oberen und rechten Rand entsprechen. Voreinstellung ist <code>c(5, 4, 4, 2)</code>
<code>mai</code>	(Vektor)	Wie <code>mar</code> , jedoch in der Einheit inch
<code>oma</code>	(Vektor)	Ränder zwischen Figure- und Device-Region einer Graphik (Abb. 10.1). Bei aufgeteilten Diagrammen zwischen den zusammengefassten Figure-Regionen und der Device-Region. Angabe wie bei <code>mar</code> . Voreinstellung ist <code>c(0, 0, 0, 0)</code> , d. h. die Figure-Region füllt die Device-Fläche vollständig aus
<code>omi</code>	(Vektor)	Wie <code>oma</code> , jedoch in der Einheit inch
<code>pch</code>	(Zahl 1-25) bzw. "(Buchstabe)"	Art der Datenpunkt-Symbole. Dabei steht etwa 16 für den ausgefüllten Punkt. Für andere Werte vgl. <code>?points</code> und Abb. 10.5. Wird ein Buchstabe angegeben, dient dieser als Symbol der Datenpunkte
<code>xpd</code>	NA, TRUE, FALSE	Graphikelemente können nur in der Plot-Region eingefügt werden (Voreinstellung: FALSE, sog. Clipping). TRUE: gesamten Figure-Region steht zur Verfügung, NA: gesamte Device-Region (vgl. Abschn. 10.1.1, Fußnote 2).

`par()` Funktion, mit der sich die Einstellungen für ein Device separat bestimmen lassen. Die aktuell für das aktive Graphik-Device gültigen Einstellungen für diese Argumente lassen sich durch `par("<Arg1>", "<Arg2>", ...)` als Liste ausgeben, d. h. durch Nennung der relevanten Argumente ohne Zuweisung von Werten. Ohne weitere Argumente gibt `par()` die aktuellen Werte für alle Parameter aus.

Abbildung 10.5 veranschaulicht die mit `lty` und `pch` einstellbaren Linientypen und Datenpunkt-Symbole (vgl. Abschn. 10.5 für das Einfügen von Elementen

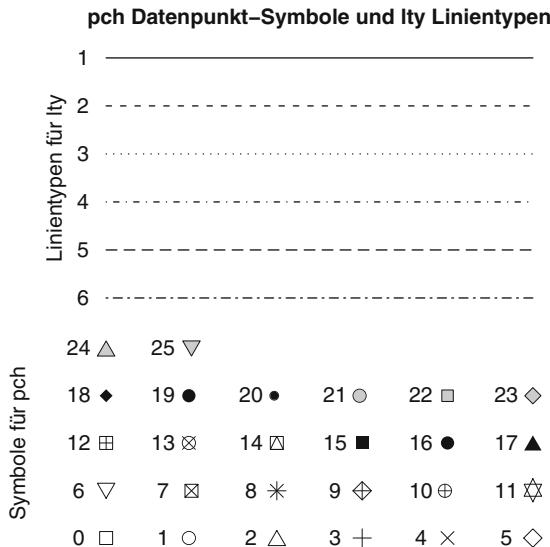


Abb. 10.5 Datenpunkt-Symbole und Linientypen zur Verwendung für die Argumente `pch` und `lty` von Graphikfunktionen

in ein Diagramm). Symbole 21–25 sind ausgefüllte Datenpunkte, deren Füllfarbe in Zeichenfunktionen über das Argument `bg="<Farbe>"` definiert wird, während `col="<Farbe>"` die Farbe des Randes bezeichnet (vgl. Abschn. 10.3.2).

```
# Koordinaten der Datenpunkte
> matX <- matrix(rep(1:6, 11),           ncol=11)
> matY <- matrix(rep(0:10, each=6),      ncol=11)

# leeres Diagramm mit der richtigen Skalierung
> plot(0:6, seq(0, 10, length.out=length(0:6)), type="n", axes=FALSE,
+       xlab=NA, ylab=NA, main="pch Datenpunkt-Symbole und lty Linientypen")

# Symbole für pch mit grauem Hintergrund für 21-25
> points(matX[1:26], matY[1:26], pch=0:25, bg="gray", cex=2)
# Linien für lty
> matpoints(matX[ , 6:11], matY[ , 6:11], type="l", lty=6:1, lwd=2,
+            col="black")

# erklärender Text - Nummer der pch Symbole und lty Linientypen
> text(matX[1:26]-0.3,  matY[1:26],      label=0:25)
> text(rep(0.7, 6),        matY[1, 6:11],    label=6:1)
> text(0, 7.5, label="Linientypen für lty", srt=90, cex=1.2)
> text(0, 2.0, label="Symbole für pch",      srt=90, cex=1.2)
```

Anstatt die in Tabelle 10.2 genannten Argumente direkt beim Aufruf von Graphikfunktionen mit anzugeben, können sie durch `par()` festgelegt werden. Die mit `par(<Option>=<Wert>)` geänderten Parameter sind Einstellungen für das aktive

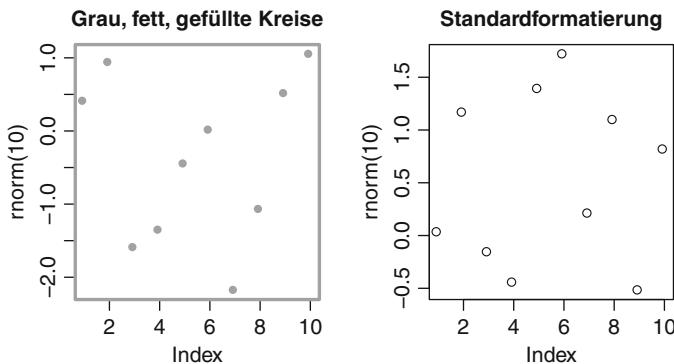


Abb. 10.6 Verwendung von `par()` zur Diagrammformatierung

Graphik-Device. Sie gelten für alle folgenden Ausgaben in dieses Device bis zur nächsten expliziten Änderung, oder bis ein neues Device aktiviert wird. Werden auf diese Weise Parameter verändert, enthält der auf der Konsole nicht sichtbare Rückgabewert von `par()` die alten Einstellungen in Form einer Liste, die auch direkt wieder an `par()` übergeben werden kann. Auf diese Weise lassen sich Einstellungen temporär ändern und dann wieder auf den Ursprungswert zurücksetzen (Abb. 10.6).

```
> op <- par(col="red", lwd=2, pch=16) # Werte ändern und alte speichern
> plot(rnorm(10), main="Rot, fett, gefüllte Kreise")
> par(op)                      # Parameter auf ursprüngliche Werte zurücksetzen
> plot(rnorm(10), main="Standardformatierung")
```

10.3.2 Farben spezifizieren

Häufig ist es sinnvoll, Diagrammelemente farblich von anderen abzusetzen, etwa um die Zusammengehörigkeit von Punkten innerhalb von Datenreihen deutlich und verschiedene Datenreihen leichter voneinander unterscheidbar zu machen. Aber auch Text- und Hintergrundfarben können in Diagrammen frei gewählt werden. Zu diesem Zweck lassen sich Farben in unterschiedlicher Form an die entsprechenden Funktionsargumente (meist `col`) übergeben:

- Als Farbname, z. B. "green" oder "blue", vgl. `colors()` für mögliche Werte.
- Als natürliche Zahl, die als Index für die derzeit aktive Farbpalette interpretiert wird. Eine Farbpalette ist dabei ein vordefinierter Vektor von Farben, der mit `palette()` ausgegeben werden kann. Die voreingestellte Palette beginnt mit den Farben "black", "red", "green3", "blue" – der Index 2 entspräche also der Farbe Rot. Die Palette kann gewechselt werden, indem an `palette(<Vektor>)` ein Vektor mit Farbnamen übergeben wird. Der (auf der Konsole unsichtbare) Rückgabewert enthält die ersetzte Palette und kann für einen nur temporären Wechsel der Palette in einem Vektor zwischengespeichert und später wieder an `palette()` übergeben werden.

- Im Hexadezimalformat, wobei die Intensitäten der Monitor-Grundfarben Rot, Grün und Blau in der Form "#RRGGBB" mit Werten für RR, GG und BB im Bereich von 00 bis FF angegeben werden. "#FF0000" entspräche Rot, "#00FF00" Grün.

Die Funktion `col2rgb(<Farbe>)` wandelt Farbnamen, Palettenindizes und Farben im Hexadezimalformat in einen Spaltenvektor um, der die Intensitäten der Monitor-Grundfarben Rot, Grün und Blau im Wertebereich von 0–255 enthält. Namen und Hexadezimalzahlen müssen dabei in Anführungszeichen gesetzt werden. Da die Spezifizierung von Farben im Hexadezimalformat nicht sehr intuitiv ist, stellt R verschiedene Funktionen bereit, mit denen Farben auf einfachere Art selbst definiert werden können. Diese Funktionen geben dann die bezeichnete Farbe im Hexadezimalformat aus.

- In der `rgb(red=<Rot>, green=<Grün>, blue=<Blau>)` Funktion können die Intensitäten der Monitor-Grundfarben Rot, Grün und Blau mit Zahlen im Wertebereich von 0–1 angegeben werden.⁶ So gibt etwa `rgb(0, 1, 1)` die Farbe "#00FFFF" (Cyan) aus. Von `col2rgb()` erzeugte Vektoren müssen deshalb vor ihrer Verwendung in `rgb()` zunächst normiert, d. h. durch 255 geteilt und zudem transponiert werden, z. B. `rgb(t(col2rgb("red")/255))`.
- Analog erzeugt `hsv(h=<Farbton>, s=<Sättigung>, v=<Helligkeit>)` (Hue, Saturation, Value) Farben, die mit Zahlen im Wertebereich von 0–1 für Farbton, Sättigung und Helligkeit definiert werden.⁷ So entspricht etwa `hsv(0.1666, 1, 1)` der Farbe "#FFFF00" (Gelb). Die Funktion `rgb2hsv(r=<Rot>, g=<Grün>, b=<Blau>)` rechnet von RGB-Farben in HSV-Werte um.
- Die `hcl(h=<Farbton>, c=<Sättigung>, l=<Helligkeit>)` (Hue, Chroma, Luminance) Funktion erzeugt Farben im CIE Luv Koordinatensystem, das auf gleiche perzeptuelle Unterschiedlichkeit von im Farbraum gleich weit entfernten Farben abzielt. Dabei ist `h` ein Winkel im Farbkreis im Bereich von 0–360°, `c` die Sättigung, deren Höchstwert vom Farbton und der Luminanz abhängt und schließlich `l` die Luminanz im Bereich von 0–100.
- Die `gray(level=<Grauwert>)` Funktion akzeptiert eine Zahl im Wertebereich von 0–1, die als Helligkeit einer achromatischen, also tonfreien Farbe mit identischen RGB-Werten interpretiert wird. So erzeugt etwa `gray(0.5)` die graue Farbe "#808080".

Bereits von R festgelegte Farben können auch aus anderen als der voreingestellten Palette stammen, so gibt etwa die `rainbow(n=<Anzahl>)` Funktion `n` viele Farben entsprechend ihrer Reihenfolge im Spektrum als Vektor von Farben im Hexadezimalformat zurück. Die Funktionen `heat.colors()`, `terrain.colors()`,

⁶ Ein vierter Wert zwischen 0 und 1 kann den Grad des sog. alpha-Blendings für simulierte Transparenz definieren. Niedrige Werte stehen für sehr durchlässige, hohe Werte für opaque Farben (vgl. Abb. 9.1). Diese Art von Transparenz wird nur von manchen Graphik-Devices unterstützt, etwa von `pdf()` oder `png()`.

⁷ Für weitere Funktionen zur Verwendung verschiedener Farbräume vgl. das Paket `colorspace` (Ihaka et al., 2009).

`topo.colors()`, `cm.colors()` und für Graustufen `gray.colors()` arbeiten analog, geben aber andere Farben aus. Die genannten Funktionen können auch dazu eingesetzt werden, die aktive Palette zu ändern, indem ihre Ausgabe an die `palette()` Funktion übergeben wird, z.B. mit `orgPal <- palette(rainbow(10))`.

10.3.3 Achsen formatieren

Ob durch die Darstellung von Datenpunkten in einem Diagramm automatisch auch Achsen generiert werden sollen, kontrollieren in High-Level-Graphikfunktionen die Argumente `xaxt` für die *x*-Achse, `yaxt` für die *y*-Achse und `axes` für beide Achsen gleichzeitig. Während für `xaxt` und `yaxt` der Wert "n" übergeben werden muss, um die Ausgabe der entsprechenden Achse zu unterdrücken, akzeptiert das Argument `axes` die Werte TRUE und FALSE. Achsen können auch separat einem Diagramm hinzugefügt werden, wobei sich Lage, Beschriftung und Formatierung der Achsenmarkierungen festlegen lassen (vgl. Abschn. 10.5.7).

Die Argumente `xlim=⟨Vektor⟩` und `ylim=⟨Vektor⟩` von High-Level-Funktionen legen den durch die Achsen abgedeckten Wertebereich in Form eines Vektors mit dem kleinsten und größten Achsenwert fest. Fehlen diese Argumente, wird jeder Bereich automatisch anhand der darzustellenden Daten bestimmt. Bei expliziten Angaben für `xlim` oder `ylim` bewirkt die Voreinstellung `xaxs="r"` der `par()` Funktion, dass die Achsen auf beiden Seiten um 4% über den angegebenen Wertebereich hinausgehen. Um dies zu verhindern, ist separat `par(xaxs="i")` aufzurufen. Die Achsenbezeichnungen können über die Argumente `xlab="⟨Name⟩"` und `ylab="⟨Name⟩"` gewählt werden. Für logarithmisch skalierte Achsen muss separat die `par()` Funktion aufgerufen und deren Argument `xlog` bzw. `ylog` auf TRUE gesetzt werden. Die Orientierung der Achsenmarkierungen legt das Argument `las` von `par()` fest.

10.4 Säulen- und Punktdiagramme

Mit `barplot()` erstellte Säulendiagramme eignen sich zur Darstellung von Kennwerten von Variablen, die getrennt für verschiedene Gruppen berechnet wurden. Dazu zählen etwa absolute oder relative Häufigkeiten von Gruppenzugehörigkeiten oder der jeweilige Mittelwert einer Variable in verschiedenen Stichproben. Der Kennwert jeder Gruppe wird dabei durch eine Säule repräsentiert, deren Höhe bzw. Länge seine Größe widerspiegelt.⁸

⁸ Für graphisch aufwendiger gestaltete Säulendiagramme vgl. die `barp()` Funktion aus dem Paket `plotrix` (Lemon et al., 2010).

10.4.1 Einfache Säulendiagramme

Sollen Kennwerte einer Variable getrennt für Gruppen dargestellt werden, die sich hinsichtlich der Ausprägungen eines einzelnen Faktors aufteilen lassen, lautet die Grundform von `barplot()`:

```
> barplot(height=<Vektor>, horiz=FALSE)
```

Diese und weitere mögliche Argumente tragen folgende Bedeutung:

- Unter `height` ist ein Vektor einzutragen, wobei jedes seiner Elemente den Kennwert für jeweils eine Bedingung repräsentiert und damit die Höhe einer Säule festlegt.
- `horiz` bestimmt, ob vertikale Säulen oder horizontale Balken gezeichnet werden. In letzterem Fall wird von einem Balkendiagramm gesprochen. Der (auf der Konsole nicht sichtbare) Rückgabewert von `barplot()` enthält die x -Koordinaten der eingezeichneten Säulen bzw. die y -Koordinaten der Balken.
- `space` legt den Abstand zwischen den Säulen fest, in der Voreinstellung beträgt er 0.2.
- `names.arg` nimmt einen Vektor von Zeichenketten an, der die Beschriftung der Säulen definiert.
- `ylim` bezeichnet den Wertebereich der y -Achse in Form eines Vektors `c(<Minimum>, <Maximum>)`. Ist `Minimum` größer als Null, muss ggf. `xpd=FALSE` gesetzt werden, um Säulen nicht unterhalb der x -Achse zeichnen zu lassen.
- `xpd` kontrolliert, ob Säulen unterhalb der x -Achse erscheinen, auch wenn diese nicht bei 0 beginnt. Voreinstellung ist `TRUE`.

Als Beispiel diene die Häufigkeitsauszählung des Ergebnisses mehrerer simulierter Würfe eines sechsseitigen Würfels (Abb. 10.7).

```
> dice <- sample(1:6, 100, replace=TRUE)
> (dTab <- table(dice))
dice
 1  2  3  4  5  6
17 23 14 20 14 12

> barplot(dTab, ylim=c(0, 30), xlab="Augenzahl", ylab="N", col="red",
+           main="Absolute Häufigkeiten")
```

Um nicht die absoluten, sondern relativen Häufigkeiten abzubilden, müssen letztere zunächst mit `prop.table()` berechnet werden.

```
> barplot(prop.table(table(dice)), ylim=c(0, 0.3), xlab="Augenzahl",
+           ylab="relative Häufigkeit", col="blue",
+           main="Relative Häufigkeiten")
```

10.4.2 Gruppierte und gestapelte Säulendiagramme

Gruppierte Säulendiagramme stellen Kennwerte von Variablen getrennt für Gruppen dar, die sich aus der Kombination zweier Faktoren ergeben. Zu diesem Zweck kann die Zusammengehörigkeit einer aus mehreren Säulen bestehenden Gruppe grafisch durch ihre räumliche Nähe innerhalb der Gruppe und die gleichzeitig größere Distanz zu anderen Säulengruppen kenntlich gemacht werden. Eine weitere Möglichkeit besteht darin, jede Einzelsäule nicht homogen, sondern als Stapel mehrerer Segmente darzustellen (Abb. 10.8).

- Für gruppierte oder gestapelte Säulendiagramme werden die Daten an `height` in Form einer Matrix übergeben, deren Werte die Säulenhöhen bzw. Balkenlängen festlegen.
- Das Argument `beside` von `barplot()` kontrolliert, welche Darstellungsart gewählt wird: auf `TRUE` gesetzt bewirkt es Säulengruppen, mit dem Wert `FALSE` gestapelte Säulen.

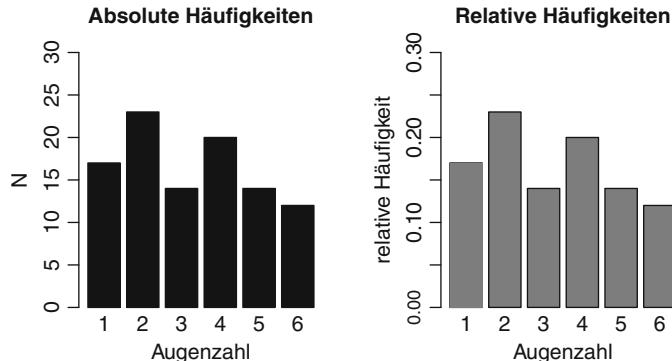


Abb. 10.7 Säulendiagramme

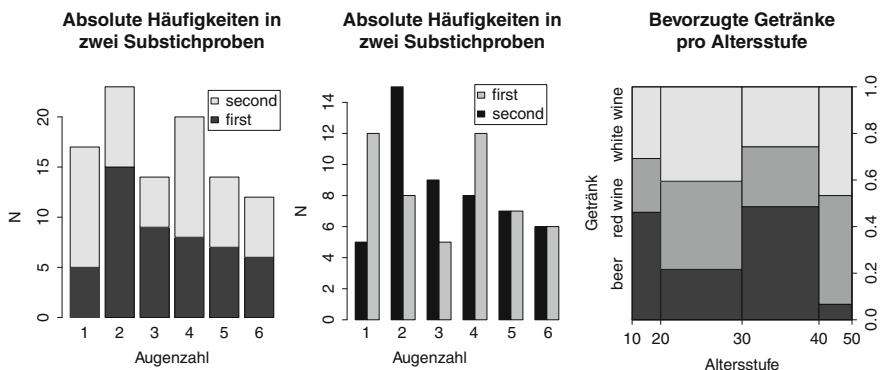


Abb. 10.8 Gestapeltes und gruppiertes Säulendiagramm absoluter sowie Spineplot bedingter relativer Häufigkeiten

- Ist `beside=FALSE`, definiert jede Spalte der Datenmatrix die innere Zusammensetzung einer Säule, indem die einzelnen Werte einer Spalte die Höhe der Segmente bestimmen, aus denen die Säule besteht. Bei `beside=TRUE` definiert eine Spalte der Datenmatrix eine Säulengruppe, deren jeweilige Höhen durch die Einzelwerte in der Spalte festgelegt sind.
- Bei gruppierten Säulendiagrammen muss `space` mit einem Vektor definiert werden. Das erste Element stellt den Abstand innerhalb der Gruppen dar, das zweite jenen zwischen den Gruppen. Voreinstellung ist der Vektor `c(0, 1)`.
- `names.arg` nimmt einen Vektor von Zeichenketten an, der die Beschriftung der Säulengruppen definiert.
- `legend.text` kontrolliert, ob eine Legende eingefügt wird, Voreinstellung ist `FALSE`. Auf `TRUE` gesetzt erscheint eine Legende, die auf den Zeilennamen der Datenmatrix basiert und sich auf die Bedeutung der Säulen innerhalb einer Gruppe bzw. auf die Segmente einer Säule bezieht. Alternativ können die Einträge der Legende als Vektor von Zeichenketten angegeben werden.

```
> roll11 <- dice[1:50] # erste Serie von Würfelwürfen
> roll12 <- dice[51:100] # zweite Serie
> rollAll <- rbind(table(roll11), table(roll12))
> rownames(rollAll) <- c("first", "second"); rollAll
   1 2 3 4 5 6
first 5 15 9 8 7 6
second 12 8 5 12 7 6
```

Die in jeder der sechs Gruppen vorhandenen zwei Säulen sollten farblich getrennt werden, um die Zugehörigkeit zur ersten bzw. zweiten Substichprobe deutlich zu machen. Dies geschieht, indem an `col` ein Vektor mit zwei Farbnamen übergeben wird, den R intern so häufig recycled, wie es Säulengruppen gibt.

```
> barplot(rollAll, beside=FALSE, legend.text=TRUE, xlab="Augenzahl",
+           ylab="N", main="Absolute Häufigkeiten in zwei Substichproben")

> barplot(rollAll, ylim=c(0, 15), col=c("red", "green"), beside=TRUE,
+           legend.text=TRUE, xlab="Augenzahl", ylab="N", main="Absolute
+           Häufigkeiten in zwei Substichproben")
```

Eine Verallgemeinerung gestapelter Säulendiagramme, mit denen die relativen Anteile von Ausprägungen einer kategorialen Variable in Abhängigkeit von einer anderen Variable dargestellt werden können, erzeugt die `spineplot()` Funktion (Abb. 10.8).⁹

```
> spineplot(x=<Vektor>, y=<Faktor>, breaks=<Grenzen>,
+             xaxlabels="<Bezeichnungen>", yaxlabels="<Bezeichnungen>")
```

Die Daten in `x` und `y` werden als an denselben Beobachtungsobjekten erhobene Werte interpretiert. Im Fall von `x` sind dies entweder Ausprägungen einer kategorialen (`x` ist ein Faktor) oder einer quantitativen Variable (`x` ist ein numerischer Vektor),

⁹ Für weitere Varianten vgl. `cdplot()` und `mosaicplot()`.

während y ein Faktor sein muss. Das Diagramm setzt sich aus nahtlos nebeneinander stehenden Säulen zusammen, deren Anzahl sowie Breite durch x und deren Aufteilung in Segmente durch y definiert ist: sofern x ein Faktor ist, stellt das Diagramm für jede seiner Ausprägungen eine Säule dar, die aus so vielen Segmenten besteht, wie y Ausprägungen hat. Die bedingten relativen Häufigkeiten der Stufen von y in der Stufe von x werden dabei über die Flächenanteile innerhalb der Säule visualisiert. Die Breiten der Säulen repräsentieren die relativen Häufigkeiten der Stufen von x , so dass insgesamt die Verteilungen beider Variablen im Diagramm abzulesen sind.

Ist x eine quantitative Variable, wird ihr Wertebereich zunächst entweder automatisch in disjunkte Intervalle eingeteilt oder anhand eines für `breaks` zu übergebenden Vektors mit den Intervallgrenzen. Statt zweier Faktoren kann für x und y zum einen eine Formel der Form $\langle y \rangle \sim \langle x \rangle$ genannt werden. Zum anderen lassen sich die Daten in Gestalt einer zweidimensionalen Kreuztabelle der gemeinsamen Häufigkeiten von x und y übergeben – diese Kreuztabelle ist gleichzeitig der auf der Konsole nicht sichtbare Rückgabewert von `spineplot()`. Die Bezeichnungen für die durch x und y definierten Bedingungen ergeben sich aus deren Faktorstufen, können aber auch explizit durch Vektoren aus Zeichenketten für `xaxlabels` und `yaxlabes` genannt werden.

Die folgenden Daten sollen das Ergebnis einer Umfrage simulieren, in der Personen unterschiedlichen Alters ihre Präferenz für ein alkoholisches Getränk abgeben.

```
> nSubj  <- 100                                     # Anzahl VPn
> age     <- sample(18:45, nSubj, replace=TRUE)      # Alter
> drinks <- c("beer", "red wine", "white wine")    # mögliche Getränke
> pref    <- factor(sample(drinks, nSubj, replace=TRUE)) # Präferenzen

# Intervallgrenzen zur Diskretisierung von age
> xRange <- round(range(age), -1) + c(-10, 10)
> lims   <- seq(xRange[1], xRange[2], by=10)
> spineplot(x=age, y=pref, xlab="Altersstufe", ylab="Getränk",
+            breaks=lims, main="Bevorzugte Getränke pro Altersstufe")
```

10.4.3 Cleveland Dotchart

Die Funktion `dotchart()` dient der Darstellung von Rohdaten einzelner Beobachtungsobjekte aber auch von aggregierten Kennwerten von Variablen. Dies können etwa absolute oder relative Häufigkeiten von Gruppenzugehörigkeiten oder die jeweiligen Mittelwerte einer Variable in verschiedenen Stichproben sein. Jeder Wert wird dabei durch einen Punkt repräsentiert, dessen x -Koordinate seine Größe widerspiegelt und dessen y -Koordinate das Beobachtungsobjekt codiert. Das Ergebnis von `dotchart()` ist analog zu einem horizontalen Balkendiagramm, wobei statt der Balken lediglich Punkte eingezeichnet werden.¹⁰

¹⁰ Das Paket `Hmisc` stellt mit `dotchart2()` eine ähnliche Funktion mit erweiterten Möglichkeiten bereit.

```
> dotchart(x=<Daten>, labels=<'Namen'>, groups=<Faktor>, gdata=<Daten>)
```

Für x ist der Datenvektor einzugeben. Über das Argument `labels` lassen sich mittels eines Vektors aus Zeichenketten derselben Länge wie x die Bezeichnungen der Datenpunkte angeben. Sollen Daten aus verschiedenen, durch die Kombination zweier Faktoren gebildeten Gruppen dargestellt werden, sind die Daten in Form einer Matrix zu übergeben, deren Werte die x -Koordinaten der Punkte festlegen. Dabei definiert jede Spalte der Datenmatrix eine Punktgruppe, deren Punkte vertikal nahe beieinander gezeichnet werden, wohingegen die durch verschiedene Spalten definierten Punkte stärker räumlich getrennt sind.

Stellen die Daten Kennwerte von Gruppen dar, die sich durch die Kombination von mehr als zwei Faktoren ergeben, können sie auch als Vektor x unter gleichzeitiger Angabe von `groups` übergeben werden. Für `groups` ist dann ein Faktor derselben Länge wie x zu nennen, der die Gruppenzugehörigkeit jedes Wertes definiert – auf diese Weise lassen sich auch Daten ungleich großer Gruppen darstellen. Zusätzlich zu den in x enthaltenen Werten lassen sich für `gdata` weitere Daten in Form eines Vektors mit so vielen Elementen angeben, wie Gruppen vorhanden sind. Die Werte von `gdata` werden als zu jeweils einer Gruppe gehörig interpretiert und als einzelner Vergleichswert eingezeichnet. Diese Möglichkeit ist z. B. geeignet, um neben den Rohdaten einer Gruppe gleichzeitig auch einen aggregierten Kennwert der Daten mit darzustellen (Abb. 10.9).

```
> nSubj    <- 5                                # Gruppengröße
> group1   <- rnorm(nSubj, 20, 2)              # AV in Gruppe 1
> group2   <- rnorm(nSubj, 25, 2)              # AV in Gruppe 2
> DV       <- c(group1, group2)                # kombinierte AV
> IV       <- factor(rep(1:2, each=nSubj))      # Gruppenzugehörigkeit
> groupMs  <- tapply(DV, IV, mean)            # Gruppenmittel
> dotchart(DV, gdata=groupMs, pch=20, color=rep(c("red", "blue",

```

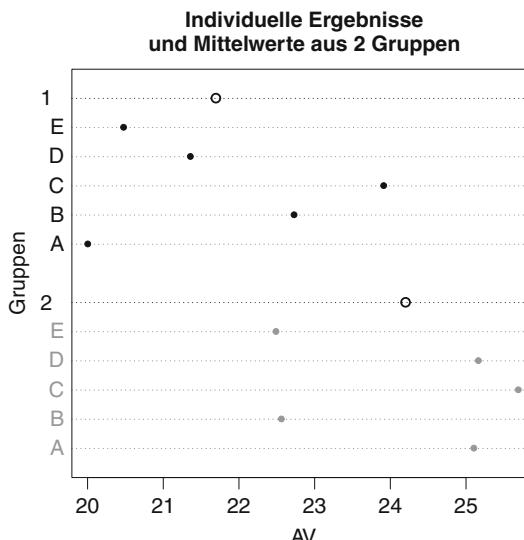


Abb. 10.9 Dotchart von individuellen Messwerten in zwei Gruppen samt zugehöriger Mittelwerte

```
+       "green"), each=nSubj), gcolor="black", groups=IV,
+       labels=rep(LETTERS[1:nSubj], 3), xlab="AV", ylab="Gruppen",
+       main="Individuelle Ergebnisse und Mittelwerte aus 2 Gruppen")
```

10.5 Elemente einem bestehenden Diagramm hinzufügen

Ein bereits erstelltes Diagramm lässt sich nachträglich durch zusätzliche Elemente erweitern, die durch Low-Level-Graphikfunktionen erzeugt werden (Tabelle 10.3). Auch einige High-Level-Funktionen besitzen das Argument `add=TRUE`, durch das ihre Ausgabe dem derzeit aktiven Graphik-Device hinzugefügt wird, ohne dass dessen Inhalte gelöscht würden. In diesem Fall kann meist durch das Argument `at=<Position>` bestimmt werden, an welcher Stelle der Zeichnungsfläche die zusätzlichen Daten erscheinen sollen. Ist kein `add` Argument vorhanden, bewirkt die vorhergehende Ausführung von `par(new=TRUE)`, dass das Ergebnis des folgenden High-Level-Befehls im schon geöffneten Graphik-Device erscheint. Später eingegebene Elemente werden immer über bereits bestehende gezeichnet, ältere Inhalte werden also durch neuere, sich an derselben Stelle befindliche übermalt.

Die Verwendung von Low-Level-Funktionen setzt voraus, dass bereits ein Diagramm mit einer High-Level-Funktion erstellt wurde. Um diesen Schritt zu überspringen und ein Diagramm ausschließlich aus Low-Level-Funktionen aufzubauen, kann aber auch ein Graphik-Device mit `windows()`; `plot.new()` geöffnet und zum Einfügen von Elementen vorbereitet werden.

10.5.1 Koordinaten in einem Diagramm identifizieren

Die meisten nachträglich einzufügenden Diagrammelemente erfordern für ihre Positionierung die Angabe von (x, y) -Koordinaten. Häufig ergeben sich diese aus den Daten, mitunter soll ein Element aber auch frei plaziert werden. Eine Alternative zur Bestimmung der dafür geeigneten Koordinaten durch Versuch und Irrtum bietet die Verwendung der `locator()` Funktion, nachdem ein Diagrammfenster erstellt wurde und noch geöffnet ist.

Tabelle 10.3 Mögliche Diagrammelemente und die sie hinzufügenden Funktionen

Diagrammelement	hinzufügende Low-Level-Graphikfunktion
Punkte	<code>points()</code> , <code>matpoints()</code>
Geradenabschnitte	<code>lines()</code> , <code>matlines()</code> , <code>segments()</code> , <code>abline()</code>
Gitter, Pfeile	<code>grid()</code> , <code>arrows()</code>
Polygone	<code>box()</code> , <code>rect()</code> , <code>polygon()</code>
interpolierte Punkte	<code>approx()</code> , <code>spline()</code> , <code>smooth.spline()</code> , <code>xspline()</code>
Funktionsgraphen	<code>curve()</code>
Text	<code>title()</code> , <code>legend()</code> , <code>text()</code> , <code>mttext()</code>
Achsen	<code>axis()</code>

```
> locator(n=<Anzahl>)
```

Das Argument n legt fest, wie viele Koordinaten zu bestimmen sind. Durch Ausführen von locator() ändert sich der Mauszeiger zu einem Kreuz, sobald er über der Diagrammfläche plaziert wird. Mit einem Klick der linken Maustaste werden die Koordinaten der Mausposition zwischengespeichert, währenddessen ist die Konsole für Eingaben blockiert. Der Vorgang kann mehrfach wiederholt werden und endet entweder, nachdem n Punkte gespeichert wurden oder über ein sich durch Klicken der rechten Maustaste öffnendes Kontextmenü. Daraufhin gibt locator() eine Liste mit zwei Vektoren zurück, die x- und y-Koordinaten der angeklickten Punkte enthalten.

```
> plot(rnorm(10))
> locator(n=5)
```

10.5.2 Punkte

```
> points(x=<x-Koordinaten>, y=<y-Koordinaten>, type=<Option>)
> matpoints(x=<Matrix>, y=<Matrix>, type=<Option>)
```

Ähnlich wie plot() fügt points() einem geöffneten Diagramm Punkte hinzu, indem über separate Vektoren jeweils deren x- und y-Koordinaten angegeben werden (Abb. 10.10). Fehlt einer der beiden Vektoren, wird der verbleibende als jener der y-Koordinaten interpretiert und die zugehörigen x-Koordinaten aus dem jeweiligen Index der Elemente gebildet. Das Argument type akzeptiert dieselben Werte wie das gleichlautende Argument von plot(). Analog zu matplot() können mit matpoints() für x- und y-Koordinaten separate Matrizen angegeben und so gleichzeitig mehrere Datenreihen spezifiziert werden.

```
> xA <- seq(-15, 15, length.out=200)
> yA <- sin(xA) / xA                                # Sinc-Funktion
> plot(xA, yA, type="l", xlab="x", ylab="sinc(x)",
+       main="Punkte und Linien einfügen", lwd=1.6)

> xB <- seq(-15, 15, length.out=30)
> yB <- sin(xB) / xB
> points(xB, yB, col="red", pch=20)
```

10.5.3 Linien

```
> lines(x=<x-Koordinaten>, y=<y-Koordinaten>, type=<Option>)
> matlines(x=<Matrix>, y=<Matrix>, type=<Option>)
```

Datenpunkte verbindende Linien werden mit lines() analog zu Punkten erstellt. Dabei geben die Vektoren x und y die Koordinaten der zu verbindenden Punk-

te an, das Argument `type` bestimmt wie in `plot()` die genaue Art der Linien.¹¹ Die `matlines()` Funktion arbeitet wie `matpoints()`, x - und y -Koordinaten können also für mehrere Datenreihen gleichzeitig in Form jeweils einer Matrix an die Argumente `x` und `y` übergeben werden (Abb. 10.10).

```
> yC <- sin(pi * xA) / (pi * xA) # normierte Sinc-Funktion
> lines(xA, yC, col="blue", type="l", lwd=1.6)

> abline(a=(y-Achsenabschnitt), b=(Steigung),
+         h=(y-Koordinate), v=(x-Koordinate), coef=(Vektor))
```

Mit `abline()` werden Geradenabschnitte in ein Diagramm gezeichnet, die auf unterschiedliche Art spezifizierbar sind. Geraden können über die Gleichung $y = b \cdot x + a$ mit den Parametern `a` für den Schnittpunkt mit der y -Achse und `b` für die Steigung beschrieben werden. Diese beiden Parameter akzeptiert alternativ auch das Argument `coef` in Form eines Vektors mit zwei Elementen. Ein über die gesamte Breite der Zeichnungsfläche gehender horizontaler Geradenabschnitt kann zudem über das Argument `h` in seiner y -Koordinate bezeichnet werden, ein vertikaler entsprechend über das Argument `v` in seiner x -Koordinate. Es lassen sich mehrere horizontale oder vertikale Geradenabschnitte mit nur einem Aufruf von `abline()` erzeugen, indem sowohl `h` als auch `v` gleichzeitig verwendet und für sie Vektoren von Koordinaten angegeben werden (Abb. 10.10).

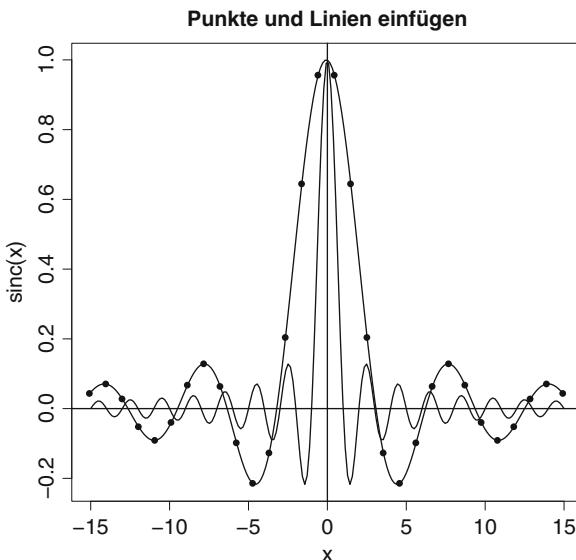


Abb. 10.10 Einfügen von Diagrammelementen: Punkte und Linien

¹¹ Das Aussehen von Linienenden sowie das Verhalten von sich treffenden Endpunkten bestimmen die Argumente `lend` und `ljoin` von `par()`.

```
> abline(h=0, v=0, col="green", lwd=1.6)

> grid(nx=NULL, ny=ny)
```

Die Argumente `nx` und `ny` bestimmen, wie viele Elemente in ein mit `grid()` einzufügendes Gitter senkrecht zur x - und y -Achse eingezeichnet werden. Auf `NULL` gesetzt bildet das Gitter die Fortsetzung der Wertemarkierungen der zugehörigen Achse. Sollen keine Gitterelemente senkrecht zu einer Achse gezeichnet werden, ist das entsprechende Argument auf `NA` zu setzen. Für quadratische Gitterfelder muss beim Erstellen des Diagramms z. B. mit `plot()` das Seitenverhältnis mit dem Argument `asp=1` kontrolliert werden.

Im folgenden Diagramm wird das Ergebnis einer Regression graphisch veranschaulicht, insbesondere die Residuen als Differenz der tatsächlichen zu den vorhergesagten Werten (Abb. 10.11).

```
> height  <- rnorm(20, 175, 7)                      # Prädiktor
> weight  <- 0.5*height + 10 + rnorm(20, 0, 4)       # Kriterium
> model   <- lm(weight ~ height)                   # Regression
> pred    <- predict(model)                         # Vorhersage
> plot(weight ~ height, asp=1, lwd=2, col="blue", pch=16,
+       main="Gitter, Segmente und Pfeile einfügen")

> abline(model, lwd=2)                                # Regressionsgerade
> grid()
```

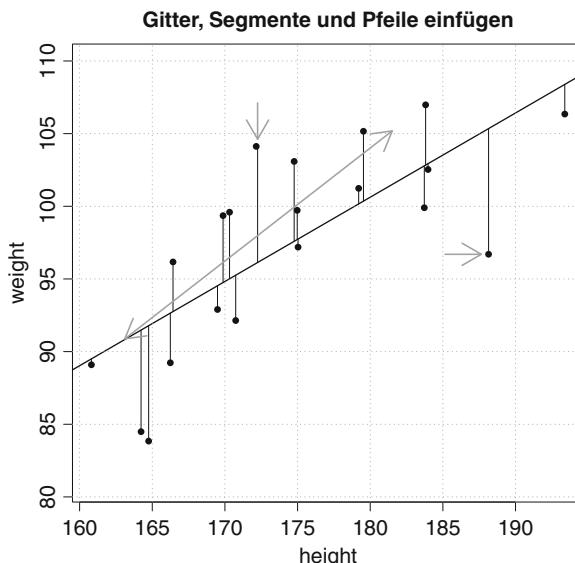


Abb. 10.11 Einfügen von Diagrammelementen: Gitter, Liniensegmente und Pfeile

Einzelne Liniensegmente können mit `segments()` eingezeichnet werden. Dazu werden in der Funktion die (x, y) -Koordinaten der Endpunkte der Segmente angegeben.

```
> segments(x0=<x Start>, y0=<y Start>, x1=<x Ziel>, y1=<y Ziel>)
```

Unter `x0` und `y0` sind die Koordinaten des Startpunkts zu nennen, von dem ausgehend das Segment gezeichnet wird. Unter `x1` und `y1` wird der Zielpunkt des Segments mit seinen Koordinaten angegeben. Wenn mehrere Linien zu zeichnen sind, lassen sich die Koordinaten auch als Vektoren eingeben.

```
> segments(x0=height, y0=pred, x1=height, y1=weight, col="gray")
```

Ähnlich wie Liniensegmente können auch Pfeile mit der `arrows()` Funktion in ein Diagramm eingezeichnet werden.

```
> arrows(x0=<x Start>, y0=<y Start>, x1=<x Ziel>, y1=<y Ziel>,
+         length=0.25, angle=30, code=2)
```

Unter `x0` und `y0` werden die Koordinaten des Startpunkts eingegeben, von dem ausgehend der Pfeil gezeichnet wird. Unter `x1` und `y1` wird der Zielpunkt des Pfeils mit seinen Koordinaten angegeben. Wenn mehrere Pfeile eingefügt werden sollen, können die Koordinaten auch als Vektoren eingegeben werden. Das Argument `length` kontrolliert die Länge der Pfeilspitzen in der Einheit inch, ihr Winkel wird über `angle` in Grad festgelegt. In der Voreinstellung `code=2` wird eine Pfeilspitze am Zielpunkt gezeichnet, mit `code=1` am Startpunkt, mit `code=3` an beiden Enden (Abb. 10.11).

```
> arrows(x0=c(height[1]-3,   height[3]), y0=c(weight[1], weight[3]+3),
+         x1=c(height[1]-0.5, height[3]), y1=c(weight[1], weight[3]+0.5),
+         col="red", lwd=2)

> arrows(x0=height[4]+0.1*(height[8]-height[4]),
+         y0=weight[4]+0.1*(weight[8]-weight[4]),
+         x1=height[4]+0.9*(height[8]-height[4]),
+         y1=weight[4]+0.9*(weight[8]-weight[4]),
+         code=3, col="red", lwd=2)
```

10.5.4 Polygone

Einem Diagramm kann mit `box()` an verschiedenen Stellen ein rechteckiger Rahmen hinzugefügt werden.

```
> box(which="plot", lty=<Linientyp>, lwd=<Linienstärke>, col=<Farbe>")
```

Das Argument `which` bestimmt, ob der Rahmen um die Zeichnungsfläche ("plot") oder das gesamte Diagramm ("figure") gezeichnet werden soll. Andere mögliche Werte sind "inner" und "outer", die in der Voreinstellung identische Ergebnisse zu "figure" erzielen und sich nur bei geänderten Seitenrändern anders verhalten. Wie üblich legen `lty`, `lwd` und `col` jeweils Linientyp, -stärke und -farbe fest.

Die `rect()` Funktion erzeugt beliebig dimensionierte Rechtecke, die sich auf der Zeichnungsfläche frei plazieren lassen.

```
> rect(xleft=(x-Koord. links),      ybottom=(y-Koord. unten),
+       xright=(x-Koord. rechts),    ytop=(y-Koord. oben), border=NULL)
```

Die Argumente `xleft`, `ybottom`, `xright` und `ytop` akzeptieren dafür Vektoren, die jeweils die Koordinaten der linken (x), unteren (y), rechten (x) und oberen Seiten (y) der zu zeichnenden Rechtecke enthalten. Soll kein Rahmen um ein Rechteck gezogen werden, ist `border=NA` zu setzen. Auch die Argumente `col` und `lty` lassen sich in der üblichen Bedeutung für die Füllfarbe und den Linientyp der Umrandung des Rechtecks verwenden (Abb. 10.12; Hoffman, 2000).

```
> n      <- 7                      # Anzahl an Zeilen und Spalten
> len   <- 1/n                     # Kantenlänge eines Quadrats

# Farbverlauf im RGB Farbsystem erstellen (Blau-Anteil ist überall 0)
> colsR  <- rep(seq(0.9, 0.2, length.out=n), each=n)  # Rot-Anteil
> colsG  <- rep(seq(0.9, 0.2, length.out=n), n)        # Grün-Anteil
> cols   <- rgb(colsR, colsG, 0)                      # Farben in RGB

# Koordinaten der Quadrate festlegen
> xLeft   <- rep(seq(0,     1-len, by=len), n)          # x-Koord. links
> yBot    <- rep(seq(0,     1-len, by=len), each=n)       # y-Koord. unten
> xRight  <- rep(seq(len,   1,      by=len), n)          # x-Koord. rechts
> yTop    <- rep(seq(len,   1,      by=len), each=n)       # y-Koord. oben
# zunächst ein leeres Diagramm erzeugen, dann Rechtecke einzeichnen
> op <- par(oma=c(1, 1, 1, 1), mar=c(0, 0, 1, 0))        # Ränder ändern
> plot(c(0,1), c(0, 1), axes=FALSE, type="n", asp=1, main="Farbverlauf")
```

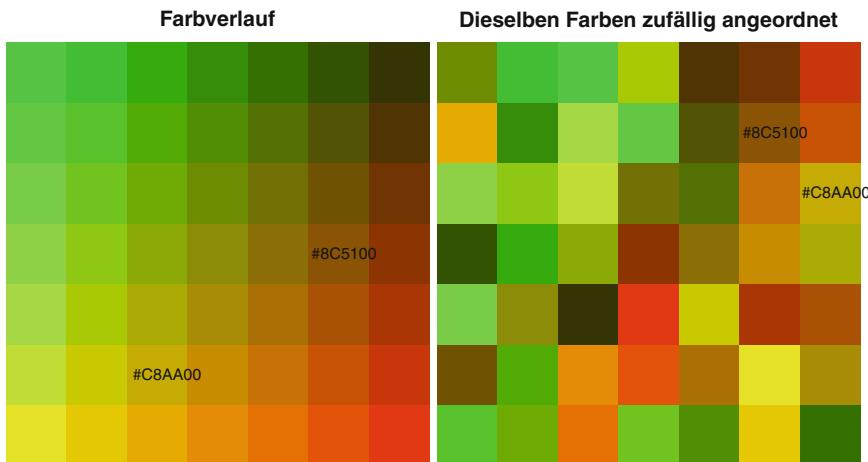


Abb. 10.12 Mit `rect()` erzeugte Rechtecke zur Veranschaulichung eines Farbphänomens: beide Graphiken sind aus denselben Quadraten zusammengesetzt, die jedoch – auch abgesehen von den farbigen Mach-Bändern links – in den beiden Konfigurationen unterschiedlich aussehen. Zwei jeweils links und rechts identische Farben sind mit ihrem Hexadezimalwert bezeichnet

```

> rect(xLeft, yBot, xRight, yTop, border=NA, col=cols)

# zwei Quadrate mit ihren hexadezimalen Farbwerten beschriften
> idx      <- c(10, 27)
> xText    <- xLeft[idx] + (xRight[idx] - xLeft[idx])/2
> yText    <- yBot[idx]  + (yTop[idx]  - yBot[idx])/2
> text(xText, yText, label=cols[idx])

# für zweites Diagramm Farben in zufällige Reihenfolge bringen
> shuffled <- sample(seq(along=cols), length(cols), replace=FALSE)
> idxS     <- c(which(shuffled == idx[1]), which(shuffled == idx[2]))
> plot(c(0, 1), c(0, 1), axes=FALSE, type="n", asp=1,
+       main="Dieselben Farben zufällig angeordnet")

> rect(xLeft, yBot, xRight, yTop, border=NA, col=cols[shuffled])

# die zwei Quadrate mit ihren hexadezimalen Farbwerten beschriften
> xTextS   <- xLeft[idxS] + (xRight[idxS] - xLeft[idxS])/2
> yTextS   <- yBot[idxS]  + (yTop[idxS]  - yBot[idxS])/2
> text(xTextS, yTextS, label=cols[idx])
> par(op)           # stelle ursprüngliche Ränder wieder her

```

Die `polygon()` Funktion erzeugt Vielecke beliebiger Gestalt, deren Eckpunkte über ihre jeweiligen (x, y) -Koordinaten zu definieren sind und in Form von Vektoren an `x` und `y` übergeben werden können. Das Polygon wird geschlossen, indem **R** den ersten und letzten Punkt miteinander verbindet. Soll kein Rahmen gezogen werden, ist `border=NA` zu setzen. Mittels eng gesetzter Eckpunkte können auch runde Formen durch Polygone approximiert werden.

```
> polygon(x=<x-Koordinaten>, y=<y-Koordinaten>, border=NULL)
```

Das folgende Beispiel illustriert die Beziehung zwischen der Fläche unter der Kurve der Dichtefunktion einer Normalverteilung und der Wahrscheinlichkeit von Intervallen, wie sie sich an der zugehörigen Verteilungsfunktion als Differenz der Funktionswerte der Intervallgrenzen ablesen lässt. Im Diagramm werden beide Funktionen gleichzeitig unter Verwendung unterschiedlicher `y`-Achsen dargestellt (Abb. 10.13).

```

> mu      <- 0                                # Erwartungswert
> sigma   <- 3                                # Streuung
> xLims   <- c(mu-4*sigma, mu+4*sigma)        # Grenzen x-Achse
> myX     <- seq(xLims[1], xLims[2], length.out=100)    # x-Werte
> myY     <- dnorm(myX, mu, sigma)            # Werte der Dichtefunktion
> selX    <- seq(mu-sigma, mu+sigma, length.out=100)    # [mu +- sigma]
> selY    <- dnorm(selX, mu, sigma)           # Werte im Intervall [mu +- sigma]
> cdf     <- pnorm(myX, mu, sigma)            # Werte der Verteilungsfunktion

> par(mar=c(5, 4, 4, 5)) # Graphik leer mit breitem rechten Rand öffnen
> plot(myX, myY, type="n", xlim=xLims-c(-2, 2), xlab=NA, ylab=NA,
+       main="Dichtefunktion und Verteilungsfunktion N(0, 3)")

```

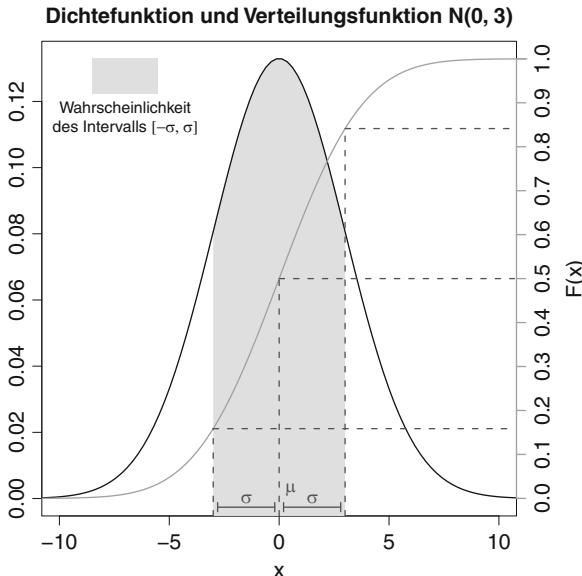


Abb. 10.13 Diagrammelemente einfügen: Rechtecke, Polygone, Achsen, Text und Symbole

```
# Rahmen, Dichtefunktion, Fläche über Intervall [mu +- sigma] zeichnen
> box(which="plot", col="gray", lwd=2)
> polygon(c(selX, rev(selX)), c(selY, rep(-1, length(selX))),
+           border=NA, col="lightgray")

> points(myX, myY, type="l", lwd=2)

# Verteilungsfunktion einzeichnen
> par(new=TRUE)          # folgendes Diagramm in bestehendes zeichnen
> plot(myX, cdf, xlim=xLims-c(-2, 2), type="l", lwd=2, col="blue",
+       xlab="x", ylab=NA, axes=FALSE)

# Achsen und Hilfslinien einzeichnen
> axis(side=4, at=seq(0, 1, by=0.1), col="blue")
> segments(x0=c(mu-sigma, mu, mu+sigma), y0=c(-1, -1, -1),
+           x1=c(mu-sigma, mu, mu+sigma), y1=c(pnorm(mu-sigma, mu, sigma),
+             pnorm(mu, mu, sigma), pnorm(mu+sigma, mu, sigma)),
+           lwd=2, col=c("darkgreen", "red", "darkgreen"), lty=2)

> segments(x0=c(mu-sigma, mu, mu+sigma), y0=c(pnorm(mu-sigma, mu, sigma),
+           pnorm(mu, mu, sigma), pnorm(mu+sigma, mu, sigma)),
+           x1=xLims[2]+10, y1=c(pnorm(mu-sigma, mu, sigma),
+             pnorm(mu, mu, sigma), pnorm(mu+sigma, mu, sigma)),
+           lwd=2, col=c("darkgreen", "red", "darkgreen"), lty=2)

> arrows(x0=c(mu-sigma+0.2, mu+sigma-0.2), y0=-0.02, x1=c(mu-0.2, mu+0.2),
+         y1=-0.02, code=3, angle=90, length=0.05, lwd=2, col="darkgreen")
```

```
# zusätzliche Beschriftungen
> mtext(text="F(x)", side=4, line=3)
> rect(-8.5, 0.92, -5.5, 1.0, col="lightgray", border=NA)
> text(-7.2, 0.9, labels="Wahrscheinlichkeit")
> text(-7.1, 0.86, expression(des ~ ~ Intervalls ~ ~ group("[",
+     list(-sigma, sigma), "]")))
>
> text(mu-sigma/2, 0,      expression(sigma), col="darkgreen", cex=1.2)
> text(mu+sigma/2, 0,      expression(sigma), col="darkgreen", cex=1.2)
> text(mu+0.5,     0.02,   expression(mu),    col="red",       cex=1.2)
```

10.5.5 Funktionsgraphen

```
> curve(expr=<Funktion>, from=<Zahl>, to=<Zahl>, n=101, add=FALSE)
```

Die High-Level-Funktion `curve()` dient dazu, Graphen von beliebigen Funktionen mit einer Veränderlichen zu erstellen. Dabei kann die darzustellende Funktion entweder über ihren Namen (z. B. `dnorm` oder `cos`) oder symbolisch als Funktionsgleichung mit `x` als Variable spezifiziert werden (z. B. `x^2 + 10`). In letzterem Fall wird also dieselbe Notation benutzt, wie um aus einem bestehenden Vektor `x` einen neuen Vektor mit den Funktionswerten zu generieren. In welchem Wertebereich die Funktion ausgewertet werden soll, bestimmen die Argumente `from` und `to`. Dabei legt das Argument `n` fest, an wie vielen gleichabständigen Stellen in diesem Bereich Funktionswerte bestimmt und eingezeichnet werden sollen. Eine höhere Anzahl bedeutet hier eine feinere Darstellung des Graphen. Um den Funktionsgraphen dem aktuell aktiven Graphik-Device hinzuzufügen, ohne dessen Inhalte zu löschen, ist `add=TRUE` zu setzen (Abb. 10.14 und 11.1). Als auf der Konsole unsichtbaren Rückgabewert liefert `curve()` eine Liste mit den x - und y -Koordinaten der gezeichneten Punkte.

```
> mu      <- 0
> sigma   <- 2
> curve(dnorm, from=-7, to=7, col="blue", lwd=2)
> curve((1/(sigma*sqrt(2*pi))) * exp(-0.5*((x-mu)/sigma)^2)), add=TRUE,
+       lwd=2, lty=2)
```

10.5.6 Text und mathematische Formeln

```
> title(main=<'Titel'>, sub=<'Untertitel'>)
```

Der Diagrammtitel lässt sich in High-Level-Graphikfunktionen meist über das Argument `main="<Name>"`, ein Untertitel über `sub="<Name>"` hinzufügen. Soll der Diagrammtitel nachträglich festgelegt werden, kann die separat aufzurufende Low-Level-Funktion `title()` Verwendung finden, die ihrerseits `main` und `sub` als Argumente besitzt. Die sog. Escape-Sequenz `\n` dient innerhalb einer Zeichenkette als Symbol für den Zeilenwechsel (Abb. 10.14).

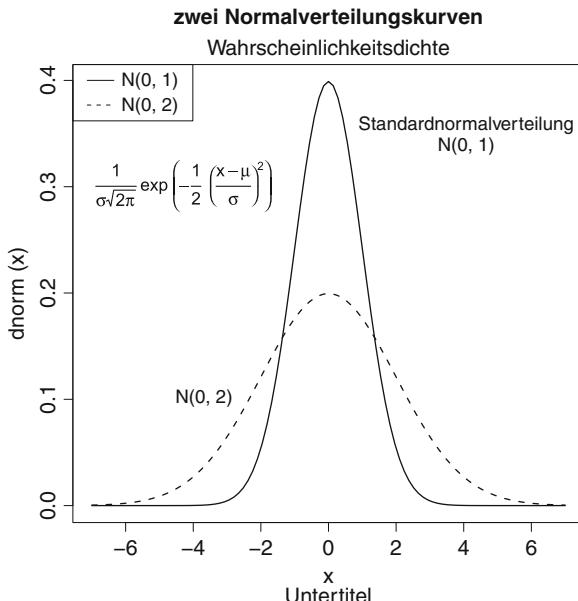


Abb. 10.14 Diagrammelemente einfügen: Funktionsgraphen, Text, Legende und mathematische Formeln

```
> title(main="zwei Normalverteilungskurven", sub="Untertitel")
```

Eine Legende zur Erläuterung der verwendeten Symbole erstellt die `legend()` Funktion.

```
> legend(x=(x-Koordinate), y=(y-Koordinate), legend="",  
+         col=(Farben), lty=(Linientypen), lwd=(Linienstärken),  
+         pch=(Symbole))
```

Die Legende wird entweder über die Angabe von (x, y) -Koordinaten für die Argumente `x` und `y` positioniert, oder durch Nennung eines der Schlüsselwörter "bottomright", "bottom", "bottomleft", "left", "topleft", "top", "topright", "right" oder "center" für `x`. Der Legendentext selbst ist als Vektor von Zeichenketten an `legend` zu übergeben, wobei jedes Element dieses Vektors einen Legendeneintrag definiert. Welches Aussehen dem Symbol neben einem Eintrag verliehen wird, kontrollieren die Argumente `col` (Farbe), `lty` (Linientyp), `lwd` (Linienstärke) und `pch` (Symbol), vgl. Abschn. 10.3.1. Für diese Argumente ist jeweils ein Vektor derselben Länge wie `legend` einzugeben, wobei NA Einträge anzeigen, dass die definierte Eigenschaft nicht auf das zugehörige Legendensymbol zutrifft. Sind in einem Diagramm etwa sowohl zwei Punkt-Reihen als auch zwei Linien enthalten, würde die Kombination von `pch=c(19, 20, NA, NA)` und `lty=c(NA, NA, 1, 2)` bewirken, dass die ersten beiden Legendeneinträge mit den Symbolen 19 und 20 dargestellt werden, die letzten beiden Einträge mit den Linientypen 1 und 2.

```
> legend(x="topleft", legend=c("N(0, 1)", "N(0, 2)"),
+         col=c("blue", "black"), lty=c(1, 2))
```

Mit der `text()` Funktion lässt sich allgemein Text an beliebiger Stelle in die Zeichnungsfläche eines Diagramms einfügen.

```
> text(x=<x-Koordinaten>, y=<y-Koordinaten>, adj=<Positionskorrektur>,
+       label=<Name>, srt=<Rotationswinkel>)
```

Zunächst sind mit den Argumenten `x` und `y` die Koordinaten des Textes festzulegen. Sollen mehrere Textelemente gleichzeitig eingefügt werden, sind hier Vektoren zu übergeben. In der Voreinstellung beziehen sich die Koordinaten auf den Mittelpunkt des Textes, was jedoch über `adj` veränderbar ist: durch einen Vektor mit zwei Elementen im Intervall $[0, 1]$ kann der Bezugspunkt der (x, y) -Koordinaten vom linken unteren Textrand (`c(0, 0)`) zum rechten oberen Textrand (`c(1, 1)`) verschoben werden, Voreinstellung ist `c(0.5, 0.5)` für die Textmitte. Die Texte selbst müssen dem Argument `label` in Form eines Vektors von Zeichenketten übergeben werden. Eine Rotation des Textes um den mit `adj` definierten Drehpunkt erlaubt das Argument `srt`, das eine Winkelangabe in Grad erwartet.

```
> text(x=3.5, y=0.35, label="Standardnormalverteilung\nN(0, 1)")
> text(x=-3.5, y=0.1, label="N(0, 2)")
```

Sollen Textelemente nicht innerhalb der Zeichnungsfläche, sondern an deren äußere Ränder geschrieben werden, muss `mtext()` eingesetzt werden.

```
> mtext(text=<Name>, side=<Nummer>, line=<Nummer>)
```

Während das Argument `text` die darzustellenden Texte in Form eines Vektors von Zeichenketten akzeptiert, bestimmt `side` mit einem Vektor der Zahlen 1–4, an welcher Seite der Text erscheinen soll. Die 1 steht dabei für unten, 2 für links, 3 für oben (Voreinstellung) und 4 für rechts. Die Orientierung des Texts ist immer parallel zum Rand, an dem der Text steht. Das Argument `line` legt in Form eines Vielfachen der Linienhöhe fest, wie weit außen der Text dargestellt wird, wobei 0 dem Rand der Zeichnungsfläche entspricht.

```
> mtext(text="Wahrscheinlichkeitsdichte", side=3)
```

In allen Funktionen zum Einfügen von Text können mit Hilfe einer an das Textsatzsystem \TeX angelehnten Syntax auch jedwede Art von Symbolen (griechische Buchstaben, mathematische Sonderzeichen, etc.) und mathematische Formeln definiert werden (Ligges, 2002; Murrell und Ihaka, 2000). Zu diesem Zweck wird eine Formel in Textform an die Funktion `expression((Ausdruck))` übergeben und das Ergebnis in den Funktionen zum Einfügen von Text eingesetzt. `(Ausdruck)` enthält dabei etwa Text (ohne Anführungszeichen), lateinische Umschreibungen griechischer Buchstaben (`theta`), Summenzeichen (`sum(x[i], i==1, n)`) Brüche (`frac((Zähler), (Nenner))`) oder Wurzelsymbole (`sqrt((Radikand), \Wurzelexponent))`). Zeilenumbrüche mit der `\n` Escape-Sequenz sind dagegen nicht möglich – stattdessen müssen mehrere Zeilen durch mehrere `text(x, y, expression())` Befehle mit entsprechend unterschiedlichen Koordinaten realisiert

werden. Eine Einführung in die Verwendung enthält die Hilfeseite `?plotmath`, weitere Veranschaulichungen zeigt `demo(plotmath)`.

```
> text(-4, 0.3, expression(frac(1, sigma*sqrt(2*pi)) ~ ~
+                         exp * bgroup("(, -frac(1, 2) ~ ~
+                         bgroup("(, frac(x-mu, sigma), ")^2, ")"))))
```

10.5.7 Achsen

Ob durch die Darstellung von Datenpunkten in einem Diagramm automatisch auch Achsen generiert werden sollen, kontrollieren in High-Level-Graphikfunktionen die Argumente `xaxt` für die x -Achse, `yaxt` für die y -Achse und `axes` für beide Achsen gleichzeitig. Während für `xaxt` und `yaxt` der Wert "n" übergeben werden muss, um die Ausgabe der entsprechenden Achse zu unterdrücken, akzeptiert das Argument `axes` die Werte `TRUE` und `FALSE`.

Sollen Achsen mit feinerer Kontrolle über den Wertebereich sowie über Aussehen und Lage der Wertemarkierungen erstellt werden, empfiehlt es sich, ihre automatische Generierung zunächst mit `axes=FALSE` zu unterdrücken. Nach dem initialen Erstellen des Diagramms können dann mit dem Befehl `axis()` samt seiner Argumente zur Formatierung und Positionierung Achsen hinzugefügt werden (Abb. 10.15).

```
> axis(side=<Nummer>, at=<Markierungen>, labels=<'Wertebeschriftungen'>,
+      pos=<Position>)
```

Achsen lassen sich an allen Diagrammseiten darstellen, was über das Argument `side` kontrolliert wird. Mögliche Werte sind 1 (unten, x -Achse), 2 (links, y -Achse),

Trigonometrische Funktionen

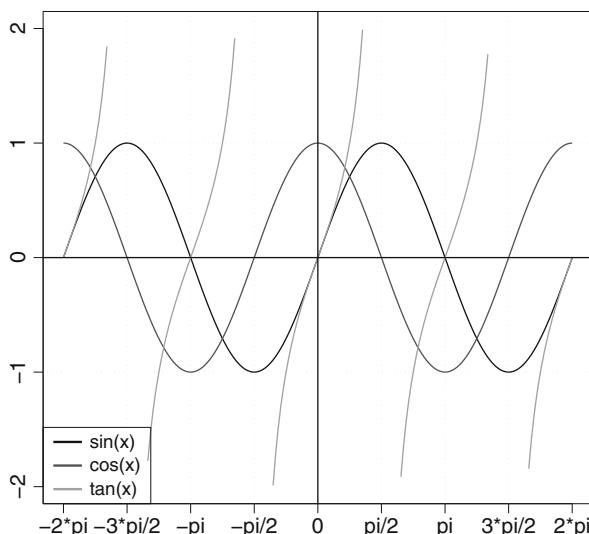


Abb. 10.15 Diagrammachsen mit `axis()` anpassen

3 (oben, alternative x -Achse) und 4 (rechts, alternative y -Achse). Die Wertemarierungen (Tickmarks) der Achse lassen sich über `at` in Form eines Vektors festlegen.¹² Das Argument `labels` bestimmt die Beschriftung dieser Markierungen und erwartet einen numerischen Vektor oder einen Vektor von Zeichenketten. Die Orientierung dieser Markierungen legt das Argument `las` von `par()` fest. Soll die Position der Achse nicht an den Diagrammrändern liegen, kann sie auch in Form einer Koordinate über das Argument `pos` definiert werden. Ist die Achse eine horizontale (`side=1` oder 3), wird der Wert für `pos` als y -Koordinate der Achse interpretiert, andernfalls (`side=2` oder 4) als deren x -Koordinate.

```
> vec <- seq(from=-2*pi, to=2*pi, length.out=200)
> mat <- cbind(sin(vec), cos(vec), tan(vec))
> mat <- ifelse(abs(mat) > 2, NA, mat)           # Werte > 2 auf NA setzen
> matplot(vec, mat, lwd=2, col=c(12, 14, 17), type="l", lty=1, xaxt="n",
+           xlab=NA, ylab=NA, main="Trigonometrische Funktionen")

> xTicks <- seq(from=-2*pi, to=2*pi, by=pi/2)
> xLabels <- c("-2*pi", "-3*pi/2", "-pi", "-pi/2", "0", "pi/2", "pi",
+               "3*pi/2", "2*pi")

> axis(side=1, at=xTicks, labels=xLabels)
> abline(h=c(-1, 0, 1), v=seq(from=-3*pi/2, to=3*pi/2, by=pi/2),
+          col="gray", lty=3)

> abline(h=0, v=0, lwd=2)
> legend(x="bottomleft", legend=c("sin(x)", "cos(x)", "tan(x)"),
+          lty=1, col=c(12, 14, 17))
```

10.5.8 Fehlerbalken

Fehlerbalken werden zusätzlich zu Kennwerten von Variablen vor allem in Säulen- und Streudiagrammen eingezeichnet, um die Variabilität der Daten auszudrücken. Als Maß für die Variabilität kann dabei u. a. die Breite eines statistischen Konfidenzintervalls für einen Parameter (z. B. den Erwartungswert) oder ein deskriptives Maß wie die Streuung oder der Standardschätzfehler verwendet werden.¹³

10.5.8.1 Fehlerbalken aus Liniensegmenten oder Pfeilen bilden

Um Fehlerbalken in ein Diagramm einzufügen, können entweder mit `arrows()` erstellte Pfeile „zweckentfremdet“ oder aber die notwendigen Bestandteile mit `segments()` in Form einzelner Liniensegmente zusammengestellt werden (vgl.

¹² Unterbrochene Achsen können mit der `axis.break()` Funktion aus dem `plotrix` Paket eingezeichnet werden.

¹³ Für Konfidenzellipsen als Maß für die Variabilität zweidimensionaler Daten vgl. das `ellipse` Paket (Murdoch und Chow, 2007).

Abschn. 10.5.3). Pfeilen lässt sich das Aussehen von Fehlerbalken geben, indem an beiden Enden Pfeilspitzen gezeichnet werden (code=3), für deren Winkel 90° zu wählen (angle=90) und deren Länge zu verkürzen ist (length=0.1).

Beispiel seien die Werte einer Variable in zwei Gruppen. Ihre Mittelwerte sollen in einem Säulendiagramm abgetragen und die Konfidenzintervalle für den Erwartungswert als Fehlerbalken eingezeichnet werden (Abb. 10.16).

```
> nSubj      <- 20          # Simulation der AV-Daten ohne Gruppeneffekt
> DV         <- c(rnorm(nSubj, 30, 6), rnorm(nSubj, 20, 6),
+                  rnorm(nSubj, 25, 6), rnorm(nSubj, 15, 6))

> IV         <- factor(rep(1:4, each=nSubj))    # Gruppenzugehörigkeit
> groupNs   <- tapply(DV, IV, length)        # Gruppengrößen
> groupMs   <- tapply(DV, IV, mean)          # Gruppenmittel
> groupSds  <- tapply(DV, IV, sd)            # Gruppenstreuungen
> barsX     <- barplot(height=groupMs, xaxt="n", xlab="Gruppe", ylim=c(0, 40),
+                  ylab="Mittelwert", main="Mittelwerte und Konfidenzintervalle")

> axis(side=1, at=1:4, labels=LETTERS[1:4])    # Gruppenbezeichnungen
```

Nachdem das Säulendiagramm gezeichnet wurde, werden die Fehlerbalken eingefügt. Die x -Koordinaten innerhalb jedes der senkrechten Segmente sind jeweils konstant, während innerhalb der (mit arrows() nicht manuell zu zeichnenden) waagerechten Segmente die y -Koordinaten jeweils konstant sind. Die x -Koordinate jedes Segments muss mit der x -Koordinate des Mittelpunkts der zugehörigen Säule übereinstimmen. Diese Mittelpunkte werden von barplot() beim Aufruf zurückgegeben und lassen sich so in einem Vektor (oder bei einem gruppierten Säulendiagramm in einer Matrix) speichern. Die Höhe der Säulen liefert den vertikalen Mittelpunkt der Fehlerbalken und ergibt sich direkt aus den Daten, ebenso die y -Koordinaten der Fehlerbalken als Grenzen des Konfidenzintervalls für den geschätzten statistischen Kennwert.

```
# Breite des 95%-Konfidenzintervalls für den Erwartungswert
> ciWidths <- qt(0.975, df=groupNs-1) * groupSds / sqrt(groupNs)
```

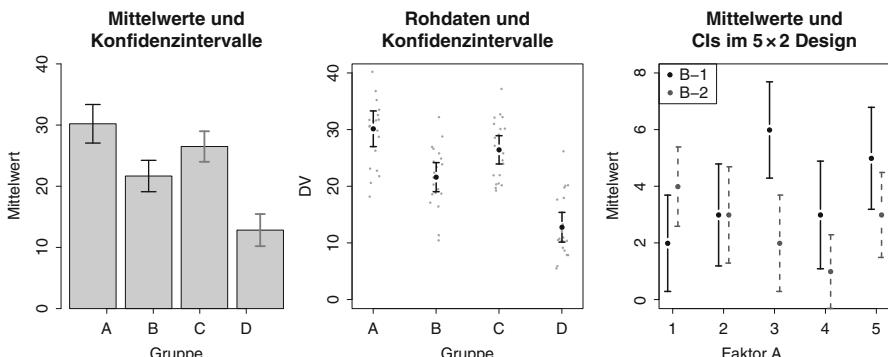


Abb. 10.16 Fehlerbalken als Liniensegmente, Pfeile oder mit plotCI() einfügen

```

> ciLo      <- groupMs - ciWidths           # untere Grenze
> ciHi      <- groupMs + ciWidths           # obere Grenze

# die ersten beiden Fehlerbalken mit arrows()
> xWidth <- 0.1                           # horizontale Breite der Fehlerbalken
> arrows(x0=barsX[1:2], y0=ciLo[1:2], x1=barsX[1:2], y1=ciHi[1:2],
+         code=3, angle=90, length=xWidth, col="blue")

# die verbleibenden beiden Fehlerbalken aus jeweils 3 Liniensegmenten
# senkrechte Segmente
> segments(x0=barsX[3:4], y0=ciLo[3:4],
+            x1=barsX[3:4], y1=ciHi[3:4], col="red")

# untere waagerechte Segmente
> segments(x0=barsX[3:4]+xWidth, y0=ciLo[3:4],
+            x1=barsX[3:4]-xWidth, y1=ciLo[3:4], col="red")

# obere waagerechte Segmente
> segments(x0=barsX[3:4]+xWidth, y0=ciHi[3:4],
+            x1=barsX[3:4]-xWidth, y1=ciHi[3:4], col="red")

```

10.5.8.2 Fehlerbalken mit dem Paket gplots

Für die Darstellung von Fehlerbalken enthält das Paket `gplots` (Warnes, 2009) mit `plotCI()` eine spezialisierte Funktion, die komfortabler zu benutzen ist, als Fehlerbalken aus Segmenten selbst zusammenzusetzen.¹⁴ Das vertikale Zentrum der Fehlerbalken wird als Punkt gezeichnet, so dass es nicht notwendig ist, zusätzlich Säulen oder Punkte zur Veranschaulichung des Parameters zu zeichnen, dessen Variabilität über den Fehlerbalken dargestellt wird (Abb. 10.16).

```

> plotCI(x=<x-Koordinaten>, y=<y-Koordinaten>, uiw=<Zahl>, liw=uiw,
+         err="y", add=FALSE)

```

Unter `x` und `y` werden die x - und y -Koordinaten des jeweiligen Mittelpunkts der Fehlerbalken angegeben. `uiw` bezeichnet die Höhe des oberen Teils der Fehlerbalken vom Mittelpunkt aus. Mit `liw` wird analog die Entfernung des unteren Endes der Fehlerbalken vom Mittelpunkt definiert. In der Voreinstellung ist `liw=uiw` für Fehlerbalken, die symmetrisch um ihren Mittelpunkt sind. Mit `err` wird definiert, ob die Balken in der Senkrechten (Voreinstellung "y") oder Waagerechten ("x") gezeichnet werden. Sollen die Fehlerbalken einem bereits erstellten Diagramm hinzugefügt werden, ist `add=TRUE` zu setzen, andernfalls wird ein neues Diagramm erzeugt.

```

> stripchart(DV ~ IV, method="jitter", xaxt="n", xlab="Gruppe",
+            ylim=c(0, 40), main="Rohdaten und Konfidenzintervalle",
+            col="darkgray", pch=16, vert=TRUE)

```

¹⁴ Vergleiche dazu auch die Funktionen `errbar()` aus dem `Hmisc` und `plotCI()` aus dem `plotrix` Paket.

```
> library(gplots)                                # für plotCI()
> plotCI(x=groupMs, uiw=ciWidths, col="blue", lwd=2, pch=19, add=TRUE)
> axis(side=1, at=1:4, labels=LETTERS[1:4])      # Gruppenbezeichnungen
```

Es können auch simultan Fehlerbalken für mehrere Gruppen dargestellt werden, die ähnlich einem gruppierten Säulendiagramm aufgebaut sind. Dabei wird die Gruppierung durch die Wahl der x -Koordinaten kontrolliert. Hier soll die Streuung die Länge der Fehlerbalken bestimmen.

```
> means1  <- c(2, 3, 6, 3, 5)                  # Mittelwerte in Gruppe 1
> sds1    <- c(1.7, 1.8, 1.7, 1.9, 1.8)       # Streuungen in Gruppe 1
> means2  <- c(4, 3, 2, 1, 3)                  # Mittelwerte in Gruppe 2
> sds2    <- c(1.4, 1.7, 1.7, 1.3, 1.5)        # Streuungen in Gruppe 2
> xOff    <- 0.1                               # horizontaler Offset zwischen Werten einer Gruppe
> plotCI(y=c(means1, means2), x=c((1:5)-xOff, (1:5)+xOff), uiw=c(sds1,
+     sds2), xlab="Faktor A", ylab="Mittelwert", ylim=c(0, 8),
+     main="Mittelwerte und CIs im 5x2 Design", col=rep(c("blue",
+     "red"), each=5), lty=rep(c(1, 2), each=5), pch=19, type="p")

> legend(x="topleft", legend=c("B-1", "B-2"), pch=c(19, 19),
+         col=c("blue", "red"))
```

10.6 Verteilungsdiagramme

Verteilungsdiagramme dienen dazu, sich einen Überblick über die Lage und Verteilungsform der in einer Stichprobe erhobenen Daten zu verschaffen. Sie eignen sich damit auch zur Überprüfung der Daten auf Ausreißer oder unplausible Werte, die etwa aus Eingabefehlern herrühren. Dies kann entweder anhand summarischer Kennwerte oder aber durch Darstellung von Einzelwerten, ggf. in vergrößerter Form, geschehen.

10.6.1 Histogramm und Schätzung der Dichtefunktion

Für Stichproben stetiger Variablen, die eine Vielzahl unterschiedlicher Werte enthalten, kann ein Histogramm als Sonderform eines Säulendiagramms für die Darstellung der empirischen Häufigkeitsverteilung verwendet werden. Histogramme stellen nicht die Häufigkeit einzelner Werte, sondern die von Wertebereichen (disjunkten Intervallen) anhand von Säulen dar, zwischen denen kein Zwischenraum gelassen wird (Abb. 10.17).¹⁵

```
> hist(x=(Vektor), breaks=(Anzahl bzw. Vektor), freq=FALSE)
```

¹⁵ Für den Vergleich der Verteilung einer Variable in zwei Bedingungen stellt die `histbackback()` Funktion aus dem `Hmisc` Paket die zugehörigen Histogramme Rücken-an-Rücken angeordnet simultan in einem Diagramm dar.

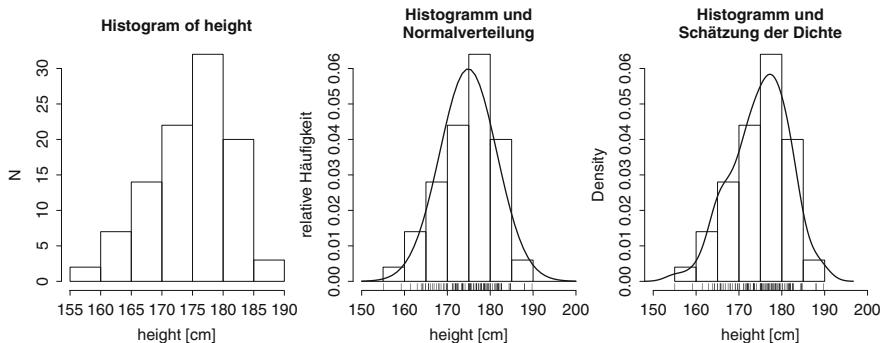


Abb. 10.17 Histogramm, zusätzlich mit Einzelwerten und Normalverteilung bzw. mit Schätzung der Dichtefunktion

Die Daten sind in Form eines Vektors `x` zu übergeben. Die Intervallgrenzen werden über das Argument `breaks` festgelegt, wobei mit einer einzelnen Zahl deren Anzahl oder mit einem Vektor deren genaue Lage vorgegeben werden kann.¹⁶ In der Voreinstellung wird beides nach einem in der Hilfe beschriebenen Algorithmus entsprechend den Daten in `x` gewählt. In der Voreinstellung werden absolute Häufigkeiten angezeigt, relative Häufigkeiten sind mit `freq=FALSE` möglich. Der auf der Konsole nicht sichtbare Rückgabewert von `hist()` enthält in Form einer Liste u. a. Angaben zur Lage und Besetzung der verwendeten Intervalle.

```
> height <- rnorm(100, 175, 7) # Körpergröße von 100 VPn
> hist(height, xlab="height [cm]", ylab="N")
```

Für die individuelle Wahl der Klassengrenzen empfiehlt es sich, zunächst den Wertebereich der Daten auszuwerten. Intervallgrenzen in regelmäßigen Abständen können dann z. B. mit `seq()` generiert werden. Werte, die genau auf einer Grenze liegen, werden immer der unteren Klasse zugeordnet, die Klassen sind also nach unten offene und nach oben geschlossene Intervalle.

```
# darzustellender Wertebereich
> fromTo <- c(round(min(height), -1)-10, round(max(height), -1)+10)
> limits <- seq(from=fromTo[1], to=fromTo[2], by=5) # Intervallgrenzen
> hist(height, freq=FALSE, xlim=fromTo, xlab="height [cm]",
+       ylab="relative Häufigkeit", breaks=limits,
+       main="Histogramm und Normalverteilung")
```

Als weitere Information lässt sich im Anschluss an den Aufruf von `hist()` mit der `rug()` Funktion auch die Lage der Einzelwerte mit darstellen. Dies geschieht in Form senkrechter Striche entlang der Abszisse, wobei jeder Strich für einen Wert steht. Um mehrfach vorkommende Werte in Form separater Striche als solche sichtbar zu machen, sollte die Funktion mit `jitter()` gekoppelt werden. Diese Funktion

¹⁶ Wird die Anzahl der Klassengrenzen genannt, behandelt R diesen Wert nur als Vorschlag, nicht als zwingend.

verändert die Werte um einen zufälligen kleinen Betrag, wodurch sich die Lage der Striche horizontal zufällig leicht verschiebt. Dies führt zu dickeren Strichen als Repräsentation mehrfach vorkommender Werte.

```
> rug(jitter(height))
```

Soll über das Histogramm zum Vergleich eine theoretisch vermutete Dichtefunktion gelegt werden, kann diese etwa mit `lines()` oder `curve()` hinzugefügt werden. Dabei muss das Histogramm entweder relative Häufigkeiten darstellen oder aber die Skalierung der theoretischen Verteilungskurve angepasst werden.

```
# lege Dichtefunktion einer Normalverteilung über Histogramm
> xNormal <- seq(from=fromTo[1], to=fromTo[2], length.out=50)
> yNormal <- dnorm(xNormal, mean=mean(height), sd=sd(height))
> lines(xNormal, yNormal, lwd=2, col="blue")
```

Die Wahl der Intervallgrenzen hat einen starken Einfluss auf die Form von Histogrammen empirischer Daten. Aufgrund dieser Abhängigkeit von einem willkürlich festzulegenden Parameter eignen sich Histogramme oft nicht sehr gut, um sich einen Eindruck von der empirischen Verteilung einer Variable zu verschaffen.

Als Alternative lässt sich mit `density(x=<Vektor>)` auf Basis einer im Vektor `x` gespeicherten Stichprobe von Werten die Dichtefunktion der zugehörigen Variable schätzen und graphisch darstellen. Die Berechnung der Schätzung kann über eine Reihe zusätzlicher Argumente kontrolliert werden, vgl. `?density`. Über `plot(density(<Vektor>))` wird die Schätzung der Dichtefunktion graphisch in einem separaten Diagramm veranschaulicht, während `lines(density(<Vektor>))` die geschätzte Dichtefunktion dem aktuell aktiven Graphik-Device hinzufügt. Soll dies in einem Histogramm geschehen, ist darauf zu achten, dort für die Darstellung relativer Häufigkeiten das Argument `freq=FALSE` zu setzen.

```
> hist(height, freq=FALSE, xlim=fromTo, xlab="height [cm]",
+       main="Histogramm und Schätzung der Dichte")

> lines(density(height), lwd=2, col="blue")
> rug(jitter(height))
```

10.6.2 Stamm-Blatt-Diagramm

Das Stamm-Blatt-Diagramm ist eine Mischung aus einer Darstellung der Häufigkeiten einzelner Werte und einem Histogramm. Seine Ausgabe erfolgt nicht in einem Graphik-Device, sondern in Textform auf der Konsole. Die Werte werden dafür zunächst wie bei einem Histogramm in disjunkte Intervalle eingeteilt, die (bis auf die äußersten beiden) dieselbe Breite besitzen. Diese Intervalle bilden den sog. Stamm und werden durch die Ziffernfolge repräsentiert, mit der alle Werte im Intervall beginnen. Geht das erste Intervall etwa von 120 bis (ausschließlich) 130, würde der Wert des Stamms 12 sein. Die sog. Blätter werden dann jeweils durch jene Werte gebildet, die in dasselbe Intervall fallen und durch die auf den Stamm folgende Ziffer

repräsentiert. Dabei werden die Werte zunächst auf die Stelle gerundet, die auf den Stamm folgt. Insgesamt repräsentiert also jedes Blatt einen Wert der Stichprobe.

```
> stem(x=<Vektor>, scale=1, width=80, atom=1e-08)
```

Unter x wird der Datenvektor eingetragen. Mit scale kann die Anzahl der Intervalle in Form eines Skalierungsfaktors verändert werden. width legt mit Werten über 10 fest, wie viele Blätter maximal gezeigt werden, wobei diese Anzahl width-10 beträgt. Bei Werten für width unter 10 werden keine Blätter angezeigt – es wird nur vermerkt, ob mehr als width Werte im Intervall liegen und ggf. wie viele dies sind. Unter atom wird die Genauigkeit der Unterscheidung zwischen den einzelnen Werten definiert. Per Voreinstellung wird bis zur achten Nachkommastelle unterschieden.

```
> stem(rnorm(100, 175, 7))
The decimal point is 1 digit(s) to the right of the |
```

```
15 | 8
16 | 12344
16 | 5556678888999999
17 | 001111122222222223334444
17 | 555555555666666677777788889999999
18 | 00001223333444
18 | 5666899
```

Aus der Konstruktion des Diagramms folgt, dass die Kombination des Stammes mit einem zugehörigen Blatt einen Wert von x repräsentiert, weswegen sich alle Werte der Stichprobe aus dem Diagramm (bis auf die Rundung) rekonstruieren lassen. Ist der Stamm 15 und ein Blatt 8, wird etwa der Wert 158 dargestellt – in Abhängigkeit von der Lage der Dezimalstelle, über die in der Diagrammüberschrift informiert wird. Wenn für den Stamm 16 die Blätter 1, 2, 3, 4 und 4 vorhanden sind, werden also die Werte 161, 162, 163, 164, 164 repräsentiert.

10.6.3 Box-Whisker-Plot

Ein Boxplot (auch Box-Whisker-Plot) stellt die Lage und Verteilung empirischer Daten durch die gleichzeitige Visualisierung verschiedener Kennwerte dar. Der Median wird dabei durch eine schwarze horizontale Linie innerhalb einer Box gekennzeichnet, deren untere Grenze sich auf Höhe des ersten und deren obere Grenze sich auf Höhe des dritten Quartils befindet. Innerhalb des so gebildeten Rechtecks liegen damit die mittleren 50% der Werte, seine Länge spiegelt den Interquartilabstand wider. Jenseits der Box erstrecken sich nach oben und unten dünne Striche (Whiskers), deren Enden jeweils den extremsten Wert angeben, der noch keinen Ausreißer darstellt. Als Ausreißer werden dabei in der Voreinstellung solche Werte betrachtet, die um mehr als das Anderthalbfache des Interquartilabstands unter oder über der Box liegen. Solche Ausreißer werden schließlich durch Kreise gekennzeichnet.

Boxplots sind insbesondere dazu geeignet, Symmetrie bzw. Schiefe unimodaler Verteilungen zu beurteilen. Zudem lassen sich Lage und Verteilung einer Variable für mehrere Gruppen getrennt vergleichen, indem die zugehörigen Boxplots nebeneinander in ein Diagramm gezeichnet werden (Abb. 10.18).

```
> boxplot(x=<Vektor>, range=<Zahl>, notch=FALSE, horizontal=FALSE)
```

Bei einem einzelnen Boxplot wird unter x der Datenvektor eingegeben. Statt eines Datenvektors kann für x eine Formel der Form $\langle AV \rangle \sim \langle Faktor \rangle$ übergeben werden, wobei $\langle Faktor \rangle$ dieselbe Länge wie der Vektor $\langle AV \rangle$ besitzt und für jeden von dessen Werten codiert, aus welcher Bedingung er stammt. Hiermit werden getrennt für die von $\langle Faktor \rangle$ definierten Gruppen Boxplots nebeneinander in einem Diagramm dargestellt. Dasselbe Ergebnis lässt sich erzielen, indem an x eine Matrix übergeben wird, deren Spalten die Gruppen definieren, für die jeweils ein Boxplot dargestellt wird. Über das Argument range wird die Definition eines Ausreißers als Vielfaches des Interquartilabstands kontrolliert. Das Argument notch bestimmt, ob die Box tailliert gezeichnet werden soll. Für horizontal verlaufende Boxen ist horizontal=TRUE zu setzen. Der auf der Konsole nicht sichtbare Rückgabewert enthält in Form einer Liste Angaben zu den dargestellten statistischen Kennwerten.

```
> nSubj    <- 40                                # Gruppengröße
> nGroups  <- 3                                 # Anzahl Gruppen
> DV       <- rnorm(nGroups*nSubj, mean=100, sd=15) # AV-Daten

# Gruppenzugehörigkeit
> IV <- factor(rep(c("Control", "Group A", "Group B"), each=nSubj))
> boxplot(DV ~ IV, ylab="Score", col=c("red", "blue", "green"),
+           main="Boxplots der Scores in 3 Gruppen")
```

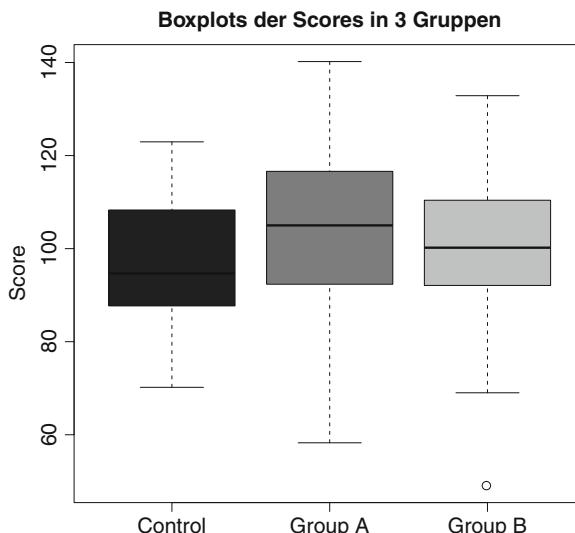


Abb. 10.18 Boxplots getrennt nach Gruppen

10.6.4 stripchart()

Ein mit `stripchart()` erstelltes eindimensionales Streudiagramm eignet sich zur Veranschaulichung der empirischen Verteilung quantitativer Variablen, wenn der Stichprobenumfang gering ist. Statt wie ein Boxplot Daten summarisch anhand ihrer wichtigsten Verteilungsparameter zu illustrieren, stellt ein Stripchart alle vorkommenden Werte selbst dar. Zu diesem Zweck wird jeder Einzelwert als Punkt auf einer horizontalen Achse repräsentiert, wobei der Wert die x -Koordinate des Punkts bestimmt (Abb. 10.19).

```
> stripchart(x=<Vektor>, method="overplot", vertical=FALSE, add=FALSE,
+             at=<Position>)
```

Die im Diagramm einzutragenden Daten werden in Form eines Vektors für x übergeben. In der Voreinstellung "overplot" bewirkt das Argument `method`, dass mehrfach vorkommende Werte durch dasselbe Symbol repräsentiert werden. Die so erstellte Graphik liefert damit keinen Aufschluss darüber, wie oft ein bestimmter Wert vorkommt. Um auch dies zu erreichen, gibt es zwei Methoden, die über das Argument `method` kontrolliert werden. Auf "jitter" gesetzt werden die Werte durch Symbole mit derselben x -Koordinate, aber einem zufälligen vertikalen Versatz dargestellt. Durch "stack" werden die Symbole eines mehrfach vorkommenden Wertes vertikal gestapelt. Um im Diagramm die Rolle von x - und y -Achse zu vertauschen, kann das Argument `vertical=TRUE` gesetzt werden.

Die Verteilung einer quantitativen Variable kann auch für mehrere Gruppen gleichzeitig in einem Stripchart veranschaulicht werden. Hierfür gibt es einerseits die Möglichkeit, mit `add=TRUE` das Ergebnis eines `stripchart()` Aufrufs einem schon bestehenden Diagramm hinzuzufügen. In diesem Fall kontrolliert das Argument `at` die vertikale Position der Achse, auf der die Symbole einzzeichnen sind. Beim ersten Aufruf von `stripchart()` wird die Achse auf einer Höhe von 1 eingezeichnet. Andererseits können die Daten auch als Formel $\langle AV \rangle \sim \langle Faktor \rangle$ übergeben werden, wobei $\langle Faktor \rangle$ dieselbe Länge wie der Vektor $\langle AV \rangle$ besitzt und

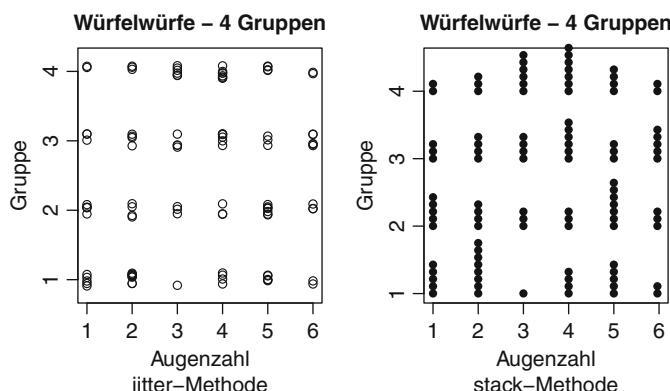


Abb. 10.19 Stripcharts mit verschiedenen Methoden zur Darstellung einzelner Werte

für jeden von dessen Werten codiert, aus welcher Bedingung er stammt. In diesem Fall wird für jede Stufe von `(Faktor)` jeweils ein Stripchart der zugehörigen Daten im Diagramm eingezeichnet, wobei unterschiedliche Faktorstufen durch vertikal getrennte Achsen kenntlich gemacht werden.

```
> nRolls  <- 25                                # Anzahl Würfe pro Gruppe
> nGroups <- 4                                  # Anzahl Gruppen
> dice    <- sample(1:6, nGroups*nRolls, replace=TRUE)  # Würfelwürfe
> group   <- factor(rep(1:nGroups, each=nRolls))  # Gruppenzugehörigkeit
> stripchart(dice ~ group, xlab="Augenzahl", ylab="Gruppe",
+             pch=1, col="blue", main="Würfelwürfe - 4 Gruppen",
+             sub="jitter-Methode", method="jitter")

> stripchart(dice ~ group, xlab="Augenzahl", ylab="Gruppe",
+             pch=16, col="red", main="Würfelwürfe - 4 Gruppen",
+             sub="stack-Methode", method="stack")
```

10.6.5 Quantil-Quantil-Diagramme

Viele statistische Auswertungsverfahren setzen voraus, dass die zu analysierenden Variablen in einer Population eine bestimmte Verteilung aufweisen, etwa normalverteilt sind oder der Verteilung in einer anderen Population gleichen. Inwieweit die empirischen Werte mit dieser Annahme verträglich sind, kann mit einem Q-Q-Diagramm (Quantil-Quantil Darstellung) heuristisch abgeschätzt werden. Es vergleicht die empirischen Quantile mit jenen Quantilen, die sich unter der Annahme einer bestimmten Verteilung ergeben. Hierfür werden die erwarteten Quantile als x - und die tatsächlichen Quantile als y -Koordinaten von Punkten in einem Diagramm verwendet. Sind erwartete und tatsächliche Quantile identisch, fallen die Punkte bei gleicher Achsenkalierung auf die Winkelhalbierende. Unterscheiden sich die Verteilungen der Daten lediglich durch ihre Skalierung, fallen die Punkte auf eine Gerade. Unterschiede zwischen den Quantilen als Zeichen unterschiedlicher Verteilungen werden entsprechend durch Abweichungen von dieser Referenzgeraden deutlich (Abb. 10.20).

Zwei Funktionen müssen zum Erstellen des Diagramms mit den Werten und der Referenzgeraden aufgerufen werden. Um die Punkte einzuziehen, ist im allgemeinen Fall die `qqplot()` Funktion aufzurufen, mit der zwei Verteilungen miteinander verglichen werden können. Für den besonders häufigen Fall eines Vergleichs empirischer Daten mit einer Standardnormalverteilung existiert `qnorm()`. Die Referenzgerade wird hier durch `qqline()` in das Diagramm eingetragen.

```
> qqplot(x=<Vektor>, y=<Vektor>, datax=FALSE)
> qnorm(           y=<Vektor>, datax=FALSE)
> qqline(          y=<Vektor>, datax=FALSE)
```

Für x und y sind bei `qqplot()` Vektoren einzutragen, deren Quantile miteinander zu vergleichen sind. Bei `qnorm()` zum Vergleich mit einer Standardnormalverteilung entfällt die Angabe des ersten Datenvektors, da die x -Koordinaten der Punkte von

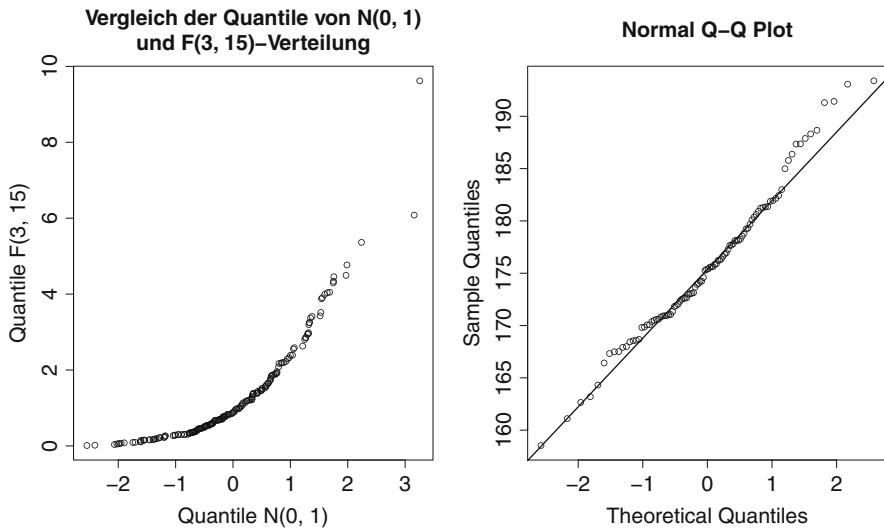


Abb. 10.20 Quantil-Quantil Darstellung zur Überprüfung von Normalverteiltheit

den theoretisch erwarteten Quantilen gebildet werden. Mit `datax=FALSE` wird in der Voreinstellung festgelegt, dass die Quantile der empirischen Werte auf der y-Achse abgetragen werden.

```
> vec1 <- rnorm(200)                      # normalverteilte AV simulieren
> vec2 <- rf(200, df1=3, df2=15)        # F-verteilte AV simulieren
> qqplot(vec1, vec2, xlab="Quantile N(0, 1)", ylab="Quantile F(3, 15)",
+         main="Vergleich der Quantile von N(0, 1) und
+         F(3, 15)-Verteilung")

> height <- rnorm(100, 175, 7)
> qnorm(height)
> qqline(height, col="red", lwd=2)
```

10.6.6 Empirische kumulative Häufigkeitsverteilungen

Für empirische Werte einer Variable lässt sich die kumulative Häufigkeitsverteilung durch Anwendung der `ecdf()` Funktion ermitteln, die ihrerseits eine Funktion erzeugt (vgl. Abschn. 2.11.6). Die graphische Darstellung dieser Funktion ist möglich, indem sie an `plot()` übergeben wird (Abb. 10.21).

```
> vec      <- round(rnorm(10), 1)
> myStep  <- ecdf(vec)
> plot(myStep, main="Empirische kumulative Häufigkeitsverteilung")
> curve(pnorm, add=TRUE, col="gray", lwd=2) # Vergleich mit Standard-NV
```

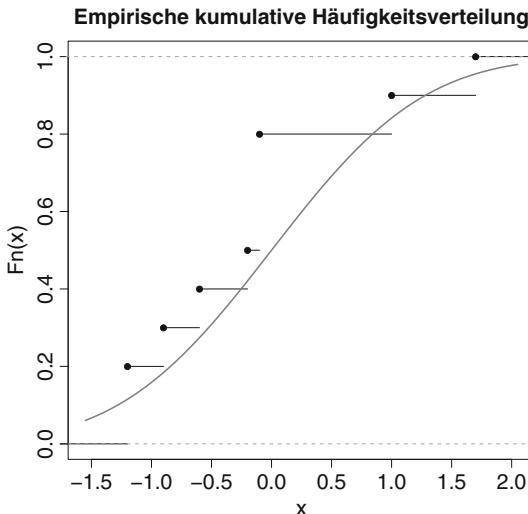


Abb. 10.21 Vergleich von empirischen kumulierten Häufigkeiten mit der Verteilungsfunktion der Standardnormalverteilung

10.6.7 Kreisdiagramm

Das Kreis- oder auch Tortendiagramm ist eine weitere Möglichkeit, die Werte einer diskreten Variable grob zu veranschaulichen. Hier repräsentiert die Größe eines farblich hervorgehobenen Kreissektors (als stilisiertes Tortenstück) den Anteil des zugehörigen Wertes an der Summe aller Werte.¹⁷ Da die korrekte Einschätzung von Flächeninhalten bzw. Sektorgrößen deutlich schwerer fällt als etwa der Vergleich von Linienlängen, sind Kreisdiagramme oft schlecht ablesbar und werden daher nicht häufig im wissenschaftlichen Kontext verwendet (Abb. 10.22).

```
> pie(x=<Vektor>, labels=<'Namen'>, col=<'Farben'>)
```

Für x ist ein Datenvektor mit nicht negativen Werten anzugeben. Sollen die einzelnen Sektoren mit einer Bezeichnung versehen werden, können diese als Vektor aus Zeichenketten für das Argument `labels` übergeben werden. Die Farbe der Sektoren kontrolliert das Argument `col`, das einen Vektor aus Farbnamen akzeptiert.

```
> dice <- sample(1:6, 100, replace=TRUE)    # Würfelwürfe
> dTab <- table(dice)                      # absolute Häufigkeiten
> pie(dTab, col=c("blue", "red", "yellow", "pink", "green", "orange"),
+      main="Relative Häufigkeiten beim Würfeln")
```

Um die Kreissektoren zu beschriften, wird die `text()` Funktion verwendet. Die zur Plazierung notwendigen (x, y) -Koordinaten ergeben sich aus den relativen Häufigkeiten der Würfelergebnisse. Dabei ist dafür zu sorgen, dass die Beschriftung

¹⁷ Für eine graphisch aufwendigere Darstellung von Kreisdiagrammen vgl. `pie3D()` aus dem `plotrix` Paket.

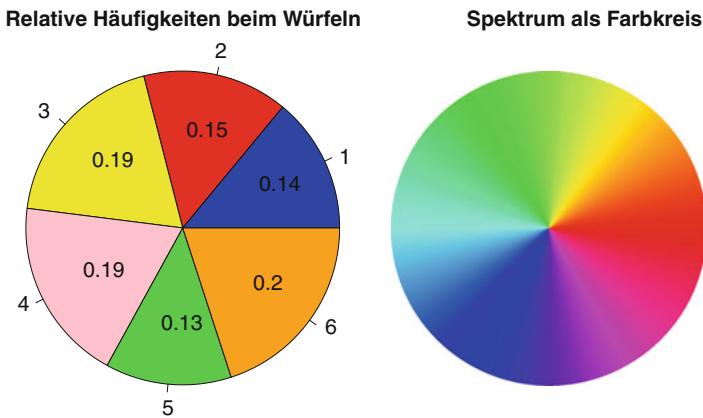


Abb. 10.22 Kreisdiagramm als Mittel zur Darstellung von Kategorienhäufigkeiten und der Spektralfarben mit `rainbow()`

immer in der Mitte eines Sektors liegt. Die Kreismitte hat die Koordinaten $(0, 0)$, der Radius des Kreises beträgt 1. Die Beschriftungen liegen mit ihrem Mittelpunkt also auf einem Kreis mit Radius 0.5.

```
> dTabFreq <- prop.table(dTab)          # relative Häufigkeiten
> textRad  <- 0.5                      # Radius für Beschriftungen
> angles   <- dTabFreq * 2 * pi        # Sektorgrößen als Winkel
> csAngles <- cumsum(angles)           # Winkel der Sektorgrenzen
> csAngles <- csAngles - angles/2     # Winkel der Sektormitten
> textX    <- textRad * cos(csAngles)  # x-Koordinaten für Text
> textY    <- textRad * sin(csAngles)  # y-Koordinaten für Text
> text(x=textX, y=textY, label=dTabFreq) # Sektorbefchriftungen
```

Mit einer großen Zahl einzelner Sektoren lässt sich mit `rainbow()` das Spektrum als Farbkreis darstellen. Dabei verhindert das Argument `border=NA` das Einzeichnen von Sektorgrenzen.

```
> pie(rep(1, 1000), labels="", border=NA, col=rainbow(1000),
+      main="Spektrum als Farbkreis")
```

10.6.8 Gemeinsame Verteilung zweier Variablen

Die in Abschn. 10.2 vorgestellte `plot()` Funktion eignet sich dafür, die gemeinsame Verteilung von zwei Variablen in Form eines Streudiagramms zu untersuchen.¹⁸ Stammen die Daten aus verschiedenen Gruppen, kann die Gruppenzugehörigkeit über die Farbe oder den Typ der Datenpunktsymbole gekennzeichnet werden. Hierfür lässt sich die Eigenschaft von Faktoren ausnutzen, dass die Stufen intern über natürliche Zahlen repräsentiert sind, die sich mit `unclass()` ausgeben lassen und damit als Indizes dienen können (Abb. 10.23).

¹⁸ Für die Darstellung von sehr vielen Wertepaaren vgl. insbesondere S. 341.

```

> nSubj <- 200                                # Anzahl VPn
> x      <- rnorm(nSubj, 100, 15)             # Daten Variable 1
> y      <- 0.5*x + rnorm(nSubj, 0, 10)        # Daten Variable 2

# Gruppierungsfaktor -> zwei Gruppen
> group <- factor(rep(LETTERS[1:2], each=nSubj/2))

# gemeinsame Verteilung mit verschiedenen Datenpunktsymbolen
# in den Gruppen
> plot(x, y, pch=c(4, 16)[unclass(group)], lwd=2,
+       col=c("black", "darkgray")[unclass(group)],
+       main="Gemeinsame Verteilung getrennt nach Gruppen")

# Legende einfügen
> legend(x="topleft", legend=c("Gruppe A", "Gruppe B"),
+          pch=c(4, 16), col=c("black", "darkgray"))

```

Weiterhin kann es hilfreich sein, ebenfalls die Streuungsellipse der gemeinsamen Verteilung einzuziehen, wie sie durch deren Hauptkomponenten, also durch die Eigenwerte und Eigenvektoren der Kovarianzmatrix der Variablen definiert wird (vgl. Abschn. 2.9.5 und 9.2). Die Ellipse hat ihren Mittelpunkt im Zentroid der Daten, ihre Hauptachsen sind durch die Eigenvektoren gegeben, wobei die Länge jeder Halbachse gleich der Wurzel aus dem zugehörigen Eigenwert ist (Abb. 10.23).¹⁹

```

> mat      <- cbind(x, y)                      # Datenmatrix
> ctr      <- colMeans(mat)                     # Mittelwerte

```

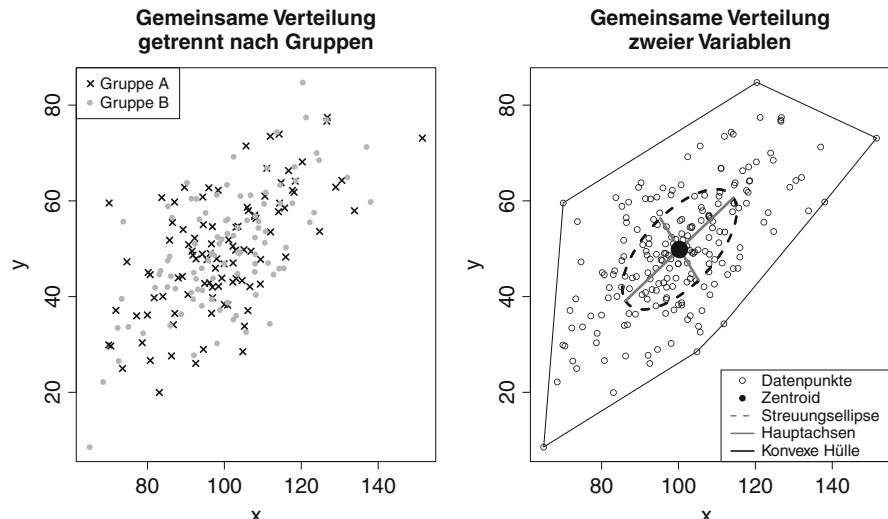


Abb. 10.23 Gemeinsame Verteilung zweier Variablen getrennt nach Gruppen. Gesamtdaten mit Streuungsellipse, deren Hauptachsen und der konvexen Hülle

¹⁹ Für Konfidenzellipsen im inferenzstatistischen Sinn vgl. das `ellipse` Paket.

```

> eigVal <- eigen(cov(mat))$values           # Eigenwerte
> eigVec <- eigen(cov(mat))$vectors         # Eigenvektoren

# mit der Wurzel der Eigenwerte skalierte Eigenvektoren
> eigScl <- sweep(eigVec, 2, sqrt(eigVal), "*")

# Hauptachsen der Ellipse: Zentroid +/- skalierte Eigenvektoren
> xMat <- rbind(ctr[1] + eigScl[1, ], ctr[1] - eigScl[1, ])
> yMat <- rbind(ctr[2] + eigScl[2, ], ctr[2] - eigScl[2, ])

# Koordinaten der Ellipse: zunächst unrotiert
> angles <- seq(0, 2*pi, length.out=200)
> ellBase <- cbind(sqrt(eigVal[1])*cos(angles), sqrt(eigVal[2])*sin(angles))

# rotiere mit Matrix der standardisierten Eigenvektoren
> ellRot <- eigVec %*% t(ellBase)

# gemeinsame Verteilung der Variablen darstellen
> plot(mat[, 1], mat[, 2], xlab="x", ylab="y",
+       main="Gemeinsame Verteilung zweier Variablen")

# in Zentroid verschobene Streuungsellipse einzeichnen
> lines((ellRot+ctr)[1, ], (ellRot+ctr)[2, ], lwd=2, col="blue")

# Hauptachsen einzeichnen
> matlines(xMat, yMat, lty=1, lwd=2, col="green")

# Zentroid markieren
> points(ctr[1], ctr[2], pch=4, col="red", cex.lab=1.5, lwd=3)

```

Die konvexe Hülle der gemeinsamen Verteilung zweier Variablen ist definiert als kleinstes convexes Polygon, in dem, bzw. auf dessen Rand alle Datenpunkte liegen. Ihr mit `chull()` ermittelbarer Rand markiert die extremsten Datenpunkte der Verteilung und kann damit bei der Identifikation von Ausreißern hilfreich sein (Abb. 10.23).

```
> chull(x=<Vektor>, y=<Vektor>)
```

Unter `x` und `y` sind die Ausprägungen der ersten und zweiten Variable jeweils als Vektor einzutragen. Wird nur ein Vektor angegeben, werden seine Elemente als `y`-Koordinaten interpretiert und die `x`-Koordinate jedes Datenpunkts gleich dem Index des zugehörigen Vektorelements gesetzt. Die Ausgabe liefert die Indizes der Randpunkte, durch die das Polygon definiert wird, im Uhrzeigersinn.

```

> hullIdx <- chull(x, y)           # Indizes Randpunkte der konvexen Hülle
> polygon(x[hullIdx], y[hullIdx]) # konvexe Hülle einzeichnen

# Legende einfügen
> legend(x="bottomright", legend=c("Datenpunkte", "Zentroid",
+ "Streuungsellipse", "Hauptachsen", "konvexe Hülle"),
+ pch=c(1, 4, NA, NA, NA), lty=c(NA, NA, 1, 1, 1),
+ col=c("black", "red", "blue", "green", "black"))

```

10.7 Datenpunkte interpolieren

R verfügt über verschiedene vorbereitete Möglichkeiten, zwischen gegebenen Datenpunkten im zweidimensionalen Raum zu interpolieren. Hierbei werden Datenpunkte generiert, die horizontal und vertikal zwischen den bestehenden plaziert sind und diese im Sinne einer zugrundeliegenden Funktion ergänzen.

10.7.1 Lineare Interpolation und polynomiale Glätter

Die `approx()` Funktion stellt die Möglichkeit bereit, zwischen Datenpunkten linear zu interpolieren:

```
> approx(x=<x-Koordinaten>, y=<y-Koordinaten>,
+         method=<Interpolationsmethode>, n=<Anzahl>)
```

Die Koordinaten der Datenpunkte sind für x und y in Form von Vektoren der selben Länge zu übergeben. Mit dem Argument `method` wird kontrolliert, wie die Interpolation erfolgt – "linear" bewirkt eine lineare Interpolation, wie sie den mit `plot(..., type="l")` erzeugten Liniensegmenten entspricht. Mit `method="constant"` erhalten alle interpolierten Punkte die y-Koordinate des in horizontaler Richtung vorangehenden Datenpunkts. Das Ergebnis ähnelt dann jenem von `plot(..., type="s")`. Wie viele Punkte hinzugefügt werden sollen, kann über das Argument `n` festgelegt werden. Die Ausgabe von `approx()` besteht aus einer Liste mit zwei Komponenten, den x- und y-Koordinaten der interpolierten Punkte. Sie kann direkt etwa an `points()` übergeben werden.

```
> xOne      <- 1:9
> yOne      <- rnorm(9)
> ptsLin    <- approx(xOne, yOne, method="linear")          # linear
> ptsConst  <- approx(xOne, yOne, method="constant")        # konstant

# linkes Diagramm
> plot(xOne, yOne, pch=19, cex=2, main="Datenpunkte interpolieren")
> points(ptsLin, pch=16, col="red", lwd=1.5)
> points(ptsConst, pch=22, col="blue", lwd=1.5)
> legend(x="bottomleft", c("Daten", "linear", "konstant"),
+         pch=c(19, 16, 22), col=c("black", "red", "blue"))
```

Um (x, y) -Streudiagramme durch glatte Kurven zu approximieren, existieren verschiedene Glätter, die auf lokal gewichteten Polynomen basieren – darunter die in `lowess()` und `loess.smooth()` implementierten (Abb. 10.24).²⁰

```
> lowess(x=<x-Koordinaten>, y=<y-Koordinaten>, f=<Spannweite>)
> loess.smooth(x=<x-Koordinaten>, y=<y-Koordinaten>, span=<Spannweite>)
```

²⁰ Weitere Glätter lassen sich mit `supsmu()` und `smooth()` anwenden.

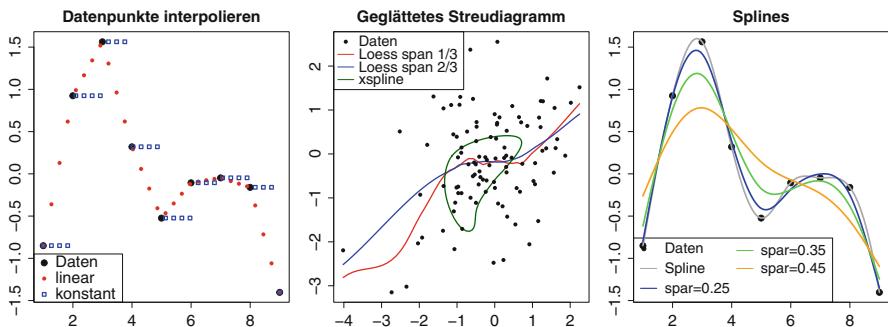


Abb. 10.24 Lineare und konstante Interpolation von Daten, polynomiale Glätter sowie unterschiedlich glatte Splines

Die Koordinaten der Datenpunkte sind für x und y in Form von Vektoren derselben Länge zu übergeben. Das Argument f von `lowess()` gibt ebenso wie das Argument span von `loess()` die Spannweite des Glätters als Anteil der Punkte an, deren Position in jedem geglätteten Wert mit berücksichtigt wird. Größere Anteile bewirken dabei glattere Interpolationen (Voreinstellung ist 2/3). Beide Funktionen geben eine Liste mit zwei Komponenten zurück, den x- und y-Koordinaten der interpolierten Punkte.

```
# mittleres Diagramm
> xTwo <- rnorm(100)
> yTwo <- 0.4 * xTwo + rnorm(100, 0, 1)
> ptsL1 <- loess.smooth(xTwo, yTwo, span=1/3)           # weniger glatt
> ptsL2 <- loess.smooth(xTwo, yTwo, span=2/3)           # glatter
> plot(xTwo, yTwo, xlab=NA, ylab=NA, pch=16,
+       main="Geglättetes Streudiagramm")

> points(ptsL1, lwd=2, col="red", type="l")
> points(ptsL2, lwd=2, col="blue", type="l")
```

10.7.2 Splines

Splines sind parametrisierte Kurven, die sich dafür eignen, zwischen Werten glatt zu interpolieren bzw. Daten durch glatte Kurven zu approximieren, wenn keine theoretisch motivierte Funktion zur Verfügung steht. Kubische Splines können mit `spline()` und `smooth.spline()` erzeugt werden: während die von `spline()` ermittelten Punkte dabei auf einer Kurve durch alle vorhandenen Datenpunkte liegen, ist dies für die von `smooth.spline()` berechnete Kurve nicht der Fall. Hier lässt sich die Kurve über Argumente in ihrer Glattheit, d. h. in dem Ausmaß ihrer Variation kontrollieren, durch die auch bestimmt wird, wie nah sie an den Datenpunkten liegt (`spar`, Smoothing Parameter). Das Ergebnis von `smooth.spline()` ist ein Objekt, mit dem über `predict()` die interpolierten y-Koordinaten für neue x-Koordinaten erzeugt werden können.

Die `xspline()` Funktion stellt weitere Splines bereit, mit denen nicht nur Funktionswerte interpoliert, sondern allgemein über sog. Kontrollpunkte beliebige, auch geschlossene und sich überschneidende Formen angenähert werden können (Abb. 10.24). Sofern gewünscht zeichnet `xspline()` das Ergebnis direkt in ein Diagramm ein.²¹

```
> ord <- order(xTwo)                      # für geordnete x-Koordinaten
> idx <- seq(8, 88, by=20)                # wähle 4 Spline-Kontrollpunkte
> xspline(xTwo[ord][idx], yTwo[ord][idx], c(1, -1, -1, 1, 1),
+           border="darkgreen", lwd=2, open=FALSE)

> legend(x="topleft", c("Daten", "Loess span 1/3", "Loess span 2/3",
+ "xspline"), pch=c(19, NA, NA, NA), lty=c(NA, 1, 1, 1),
+ col=c("black", "red", "blue", "darkgreen"))

# rechtes Diagramm
> plot(xOne, yOne, pch=19, cex=1.5, main="Splines")    # Daten
> ptsSpline <- spline(xOne, yOne, n=201)                 # kubischer Spline

# Smoothing Splines unterschiedlicher Glattheit
> smSpline1 <- smooth.spline(xOne, yOne, spar=0.25)     # recht variabel
> smSpline2 <- smooth.spline(xOne, yOne, spar=0.45)     # mittel variabel
> smSpline3 <- smooth.spline(xOne, yOne, spar=0.45)     # recht glatt

# neue x-Koordinaten, auf die Splines angewendet werden
> ptsX      <- seq(1, 9, length.out=201)
> ptsSmSpl1 <- predict(smSpline1, ptsX)
> ptsSmSpl2 <- predict(smSpline2, ptsX)
> ptsSmSpl3 <- predict(smSpline3, ptsX)

# Linien einzeichnen
> lines(ptsSpline, col="darkgray", lwd=2)
> matlines(x=ptsX, y=cbind(ptsSmSpl1$y, ptsSmSpl2$y, ptsSmSpl3$y),
+           col=c("blue", "green", "orange"), lty=1, lwd=2)

> legend(x="bottomleft", c("data", "spline", "spar=0.3", "spar=0.4",
+ "spar=0.5"), pch=c(19, NA, NA, NA, NA), lty=c(NA, 1, 1, 1, 1),
+ col=c("black", "darkgray", "blue", "green", "orange"))
```

10.8 Multivariate Daten visualisieren

In unterschiedlichen Situationen kann es erstrebenswert sein, multivariate Daten graphisch darzustellen: so kann eine Abhängige Variable in ihrer Ausprägung von mehr als einer numerischen Variablen abhängen, etwa im Kontext einer multiplen linearen Regression. Oder aber mehrere Abhängige Variablen werden gleichzeitig erhoben, um ihre gemeinsame Verteilung zu analysieren. Schließlich hängt etwa im

²¹ Für weitere Spline-Typen vgl. `help(package="splines")`.

Rahmen mehrfaktorieller Varianzanalysen eine Abhängige Variable von der Kombination verschiedener qualitativer Faktoren ab. Die Visualisierung solcher Daten muss dann alle beteiligten Komponenten berücksichtigen. Da Diagramme nur zweidimensional sein können, ergibt sich dabei das Problem, dass die räumliche Lage eines Datenpunkts auf der Zeichnungsfläche zwar zwei Komponenten repräsentieren kann, mehr jedoch nicht.

Für dreidimensionale Daten existieren als Ausweg verschiedene Diagrammtypen, die sich darin unterscheiden, auf welche Weise die dritte Komponente grafisch codiert wird: so wird versucht, einen räumlichen Tiefeneindruck zu erzeugen und die dritte Komponente als z -Koordinate, d. h. als Höhe eines Datenpunkts über einer Ebene zu repräsentieren (dreidimensionale Streudiagramme und Gitterflächen). Oder die dritte Komponente wird nicht räumlich, sondern farblich bzw. durch andere graphische Unterscheidungsmerkmale symbolisiert, etwa durch Höhenlinien oder die Größe der Datenpunktssymbole. Bei mehr als drei Variablen kann auf die direkte Veranschaulichung aller Komponenten zugunsten einer aufgeteilten Graphik verzichtet werden, in der eine Serie uni- oder bivariater Diagramme nebeneinander für alle Stufen bzw. Stufenkombinationen weiterer Variablen angeordnet ist.

10.8.1 Höhenlinien und variable Datenpunktssymbole

Den Diagrammtypen, die Höhenlinien oder Gitter zur Visualisierung der z -Koordinate verwenden, ist die Art der Angabe von Koordinaten gemein. Sie benötigen zum einen zwei Vektoren, die die Werte auf der x - und y -Achse festlegen. Als drittes Argument erwarten die Funktionen eine Matrix, die Werte für jede Kombination der übergebenen x - und y -Koordinaten enthält und deswegen aus so vielen Zeilen wie x - und so vielen Spalten wie y -Koordinaten besteht. Die Werte dieser Matrix definieren die z -Koordinate als dritte Komponente für jedes (x, y) -Koordinatenpaar.

Höhenlinien symbolisieren die z -Koordinate wie topographische Karten durch die Zugehörigkeit eines Punkts zu einer Region der Diagrammfläche, die durch eine geschlossene Höhenlinie definiert wird. Dieses Vorgehen ist mit einer Vergrößerung der Daten verbunden, da Wertebereiche in Kategorien zusammengefasst werden.

```
> contour(x=(x-Koordinaten), y=(y-Koordinaten), z=(z-Koordinaten),
+           nlevels=(Anzahl Höhenlinien), levels=(Kategorien),
+           labels="(Namen)", drawlabels=TRUE)
```

Für x und y muss jeweils ein Vektor mit den x - bzw. y -Koordinaten der Datenpunkte übergeben werden. Die Matrix z definiert die z -Koordinaten in der o. g. Form. Welche Kategorien Verwendung finden, kann über die Argumente `levels` und `nlevels` kontrolliert werden. Für `levels` ist ein Vektor aus Kategorienbezeichnungen zu übergeben. In diesem Fall ist die Verwendung von `nlevels` nicht mehr notwendig, da sich die Zahl der Kategorien aus der Länge von `levels` ergibt. Andernfalls ist für `nlevels` die gewünschte Anzahl an Kategorien zu nennen. Das Argument `drawlabels` bestimmt, ob die Kategorienbezeichnungen im Diagramm eingetragen werden.

Im folgenden Beispiel soll die Dichtefunktion von zwei eindimensionalen, gemeinsam normalverteilten Variablen mit positiver Korrelation dargestellt werden (Abb. 10.25). Um die Dichte zu berechnen, wird eine eigene Funktion definiert (vgl. Abschn. 11.1), da R anders als im Fall einer eindimensionalen Normalverteilung mit `dnorm()` hier keine Funktion bereitstellt.²² Mit `outer()` lässt sich die Funktion für alle (x, y) -Koordinatenpaare unterteilt in drei Schritte berechnen: die Dichte ergibt sich nämlich in der Form $a \cdot e^{b \cdot c}$, wovon nur c von x und y abhängt – a und b bestimmen sich bereits aus den Erwartungswerten und Streuungen sowie aus der Korrelation der beiden Variablen.

```
> muX      <- 1          # Erwartungswert Normalverteilung entlang x-Achse
> muY      <- 3          # Erwartungswert Normalverteilung entlang y-Achse
> sX       <- 1          # Streuung Normalverteilung entlang x-Achse
> sY       <- 1          # Streuung Normalverteilung entlang y-Achse
> rhoXY    <- 0.6        # Korrelation der beiden Variablen
> rangeX   <- 2.5        # betrachteter Wertebereich von x (in Std.-Abw.)
> rangeY   <- 2.5        # betrachteter Wertebereich von y (in Std.-Abw.)
> N         <- 50         # Anzahl der x- und der y-Koordinaten

# x-Koordinaten
> myX <- seq(from=muX-rangeX*sX, to=muX+rangeX*sX, length.out=N)

# y-Koordinaten
> myY <- seq(from=muY-rangeY*sY, to=muY+rangeY*sY, length.out=N)
> aa  <- 1 / (2 * pi * sX * sY * sqrt(1-rhoXY^2))    # a-Komponente
> bb  <- -1 / (2 * (1-rhoXY^2))                         # b-Komponente

# Funktion, um c aus x und y zu erzeugen
> genC <- function(x, y) {
+   ((x-muX)^2/sX^2)-(2*rhoXY*(x-muX)*(y-muY)/(sX*sY))+((y-muY)^2/sY^2) }

# erzeuge c aus allen Kombinationen von x- und y-Koordinaten
> cc <- outer(myX, myY, "genC")
```

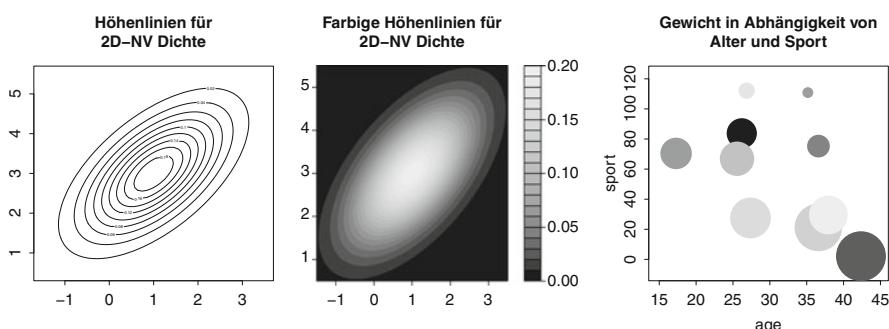


Abb. 10.25 Visualisierungsmöglichkeiten für dreidimensionale Daten: Höhenlinien und variable Datenpunktsymbole

²² Alternativ wäre die `dmvnorm()` Funktion aus dem `mvtnorm` Paket verwendbar.

```
# Dichte an allen Kombinationen von x- und y-Koordinaten
> myZ <- aa * exp(bb*cc)
> contour(myX, myY, myZ, main="Höhenlinien für 2D-NV Dichte")
```

Die durch die Höhenlinien definierten Regionen können mit der `filled.contour()` Funktion auch eingefärbt werden, wobei die Farben aus einem Farverlauf stammen, dessen Zuordnung zu Kategorien auf der rechten Seite des Diagramms in einer Legende erläutert wird (Abb. 10.25). Der Aufruf gleicht dem für `contour()` stark, jedoch kann über das zusätzliche Argument `color.palette` eine eigene Farbpalette spezifiziert werden.

```
> filled.contour(myX, myY, myZ, main="Farbige Höhenlinien")
```

Mit der `symbols()` Funktion lassen sich zwei – typischerweise quantitative – Variablen durch die räumliche Lage von Datenpunktssymbolen gemeinsam mit weiteren Variablen durch die Gestaltung dieser Symbole graphisch repräsentieren (Abb. 10.25).²³

```
> symbols(x=<Vektor>, y=<Vektor>, add=FALSE, circles=<Vektor>,
+           boxplots=<Nx5 Matrix>, inch=TRUE, fg=<Farben>, bg=<Farben>)
```

Für `x` und `y` muss jeweils ein Vektor mit den *x*- bzw. *y*-Koordinaten der Datenpunkte genannt werden, alternativ ist auch eine Formel der Form $\langle y \rangle \sim \langle x \rangle$ möglich. Welche Datenpunktssymbole an diesen Koordinaten erscheinen, richtet sich danach, für welches weitere Argument Daten übergeben werden: `circles` erwartet einen Vektor derselben Länge wie `x` und `y` mit positiven Werten für die Größe der als Datenpunktssymbol dienenden Kreise. Der größte Wert entspricht in der Voreinstellung `inch=TRUE` einer Symbolgröße von 1 in, die Größe der übrigen Symbole richtet sich nach dem Verhältnis der zugehörigen Werte zum Maximum. Wird an `inch` stattdessen ein numerischer Wert übergeben, definiert er die Maximalgröße der Symbole. Um an jedem Koordinatenpaar einen Boxplot zu zeichnen, muss an `boxplots` eine Matrix mit 5 Spalten und so vielen Zeilen übergeben werden, wie `x` und `y` Elemente besitzen. Die Werte in den Spalten stehen dabei für die Breite und Höhe der Boxen, für die Länge der unteren und oberen Striche (Whiskers) sowie für den Median. Für weitere Symbole vgl. `?symbols`. Die Farben der Symbole legt `fg`, die der von ihnen umschlossenen Flächen `bg` fest.

Das Beispiel soll anhand eines (sicher unrealistischen) Modells den Zusammenhang zwischen den negativ korrelierten Prädiktoren Alter und Sport (in Minuten pro Woche) und dem Körpergewicht als AV simulieren.

```
> nSubj <- 10
> age <- rnorm(nSubj, 30, 8)
> sport <- abs(-0.25*age + rnorm(nSubj, 60, 30))
> weight <- -0.3*age -0.4*sport + 100 + rnorm(nSubj, 0, 3)

# reskaliere Werte für das Körpergewicht auf das Intervall [0.2, 1]
wScale <- (weight-min(weight)) * (0.8 / abs(diff(range(weight)))) + 0.2
```

²³ Für ähnliche Funktionen vgl. `stars()` und `sunflowerplot()`.

```
> symbols(age, sport, circles=wScale, inch=0.6, fg=NULL, bg=rainbow(nSubj),
+           main="Gewicht in Abhängigkeit von Alter und Sport")
```

10.8.2 Dreidimensionale Gitter und Streudiagramme

Die `persp()` Funktion erzeugt Diagramme mit Tiefeneindruck, in der die Höhenwerte zu einer Gitterfläche verbunden sind. Der Augpunkt, d. h. die Perspektive, aus der die simulierte dreidimensionale Szene gezeigt wird, ist frei wählbar (Abb. 10.26).

```
persp(x=(x-Koordinaten), y=(y-Koordinaten), z=(z-Koordinaten),
+      theta=0, phi=15, r=sqrt(3))
```

Die Argumente `x`, `y` und `z` haben dieselbe Bedeutung wie in Höhenlinien-Diagrammen. Die Argumente `theta`, `phi` und `r` bestimmen die Blickrichtung auf das Diagramm in Form von Polarkoordinaten, die innerhalb einer gedachten Kugel mit dem Ursprung des Koordinatensystems im Zentrum definiert sind. Dabei bezeichnet `theta` den Azimuth (den Längengrad) und `phi` die Höhe über dem Äquator, also den Breitengrad (Elevation). Die Entfernung zum Diagramm als Kugelradius kontrolliert `r`. Die Funktion verfügt über weitere Argumente, u. a. zur Kontrolle der perspektivischen Verzerrung und zur räumlichen Kompression der `z`-Achse.

```
> persp(myX, myY, myZ, xlab="x", ylab="y", zlab="Dichte", theta=5,
+       phi=35, main="Dichte einer 2D Normalverteilung")
```

Die Verallgemeinerung eines mit `plot()` erstellten konventionellen Streudiagramms auf dreidimensionale Daten ist etwa mit der `scatterplot3d()` Funktion aus dem gleichlautenden Paket möglich.

```
> scatterplot3d(x=(x-Koordinaten), y=(y-Koordinaten), z=(z-Koordinaten))
```

Im Unterschied zu `contour()` und `persp()` müssen die Koordinaten hier in Form dreier Vektoren an die Argumente `x`, `y` und `z` übergeben werden, in denen jeweils

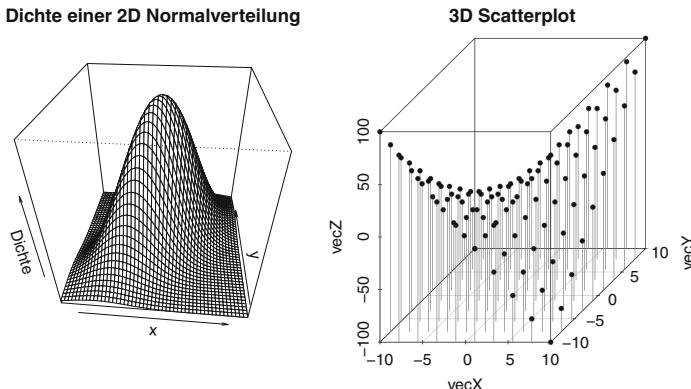


Abb. 10.26 Visualisierungsmöglichkeiten für dreidimensionale Daten: Gitter und Streudiagramm

die x -, y - und z -Koordinaten einzelner Punkte enthalten sind. Die Funktion liefert als Rückgabewert ein Objekt, mit dessen Hilfe der Graphik Diagrammelemente hinzugefügt werden können – ein Verfahren, das von dem in Abschn. 10.5 für zweidimensionale Diagramme beschriebenen abweicht (Abb. 10.26).

```
> library(scatterplot3d)
> vecX <- rep(seq(-10, 10, length.out=10), 10)
> vecY <- rep(seq(-10, 10, length.out=10), each=10)
> vecZ <- vecX*vecY
> sp3 <- scatterplot3d(vecX, vecY, vecZ, type="h", main="3D Scatterplot")

# dem Diagramm Datenpunkte hinzufügen
> sp3$points3d(vecX, vecY, vecZ, col="red", pch=16)
```

Mit Funktionen aus dem Paket `rgl` (Adler und Murdoch, 2010) erstellte Diagramme – etwa `plot3d()`, `hist3d()` oder `persp3d()` als Pendants zu den konventionellen Funktionen `plot()`, `hist()` und `persp()`, erlauben durch den Einsatz der Graphikbibliothek OpenGL eine interaktive Manipulation²⁴: durch Anklicken der Diagrammfläche lässt sich die Perspektive auf das Diagramm beliebig ändern, indem die linke Maustaste gedrückt gehalten und die Maus bewegt wird.

```
> plot3d(x=<x-Koordinaten>, y=<y-Koordinaten>, z=<z-Koordinaten>)
```

Die x -, y und z -Koordinaten werden wie bei `scatterplot3d()` als Vektoren übergeben.

```
> library(rgl)
> plot3d(vecX, vecY, vecZ, col="blue", type="s")
```

Das `rgl` Paket enthält viele Zusatzfunktionen, mit denen der dreidimensionale Eindruck eines Diagramms über die Plazierung von Lichtquellen und die Manipulation von simulierten Oberflächenbeschaffenheiten gesteigert werden kann. Darüber hinaus bringt es Funktionen zum Einfügen aller in Abschn. 10.5 für zweidimensionale Diagramme beschriebenen Graphikelemente mit, vgl. `help(package="rgl")`. Durch `demo(rgl)` sowie insbesondere `example(persp3d)` erhält man einen Überblick über die zahlreichen Gestaltungsmöglichkeiten, die das Paket eröffnet.

10.8.3 Simultane Darstellung mehrerer Diagramme gleichen Typs

Die `coplot()` Funktion (Conditioning Plot) stellt mehrere Diagramme desselben Typs getrennt nach mehreren Gruppen simultan dar und eignet sich damit zur Veranschaulichung einer Variable, die in mehreren (Kombinationen von) Bedingungen erhoben wurde. Die Zugehörigkeit der in einer Region dargestellten Daten zu einer durch eine Bedingungskombination definierten Substichprobe wird am Diagrammrand graphisch veranschaulicht (Abb. 10.27).

²⁴ Auch die Pakete `iplots` und `rggobi` stellen Funktionen bereit, die Graphiken interaktiv in Echtzeit verändern, sie etwa an geänderte Ursprungsdaten anpassen.

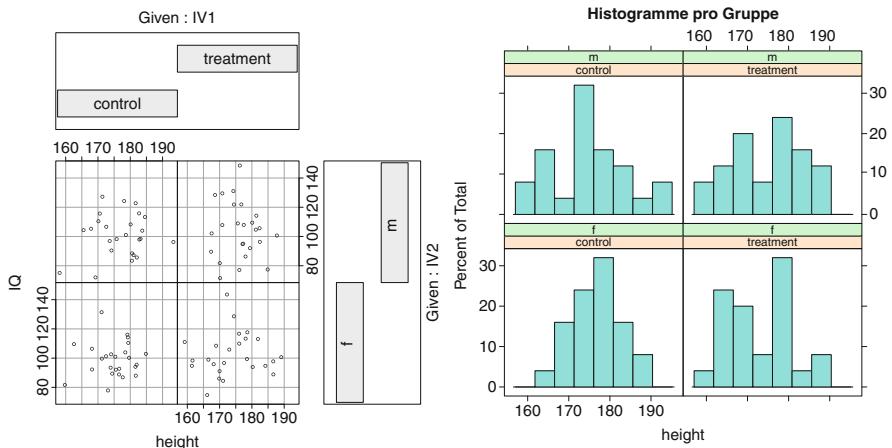


Abb. 10.27 Darstellung multivariater Daten mit `coplot()` und `histogram()`

```
> coplot((y-Koordinaten) ~ (x-Koordinaten) | <Faktor>, data=<Datensatz>,
+         panel=(Zeichenfunktionsname), rows=(Anzahl), columns=(Anzahl))
```

Als erstes Argument ist eine Formel anzugeben, die wie in der `plot()` Funktion zunächst y- und x-Koordinaten der darzustellenden Datenpunkte nennt. Als Besonderheit folgt dann hinter dem `|` Zeichen ein Faktor, der die Gruppen definiert, für die separate Diagramme erstellt werden sollen. Die Unterteilung kann auch durch die Kombination mehrerer Faktoren definiert sein, die dann in der Form `<Faktor1> * <Faktor2>` hinter dem `|` Zeichen einzufügen sind. Stammen die in der Formel verwendeten Variablen aus einem Datensatz, ist dieser unter `data` zu nennen. In der Voreinstellung werden die Werte als Punkte dargestellt. Dies kann über das Argument `panel` jedoch geändert werden, das den Namen einer Low-Level-Zeichenfunktion ohne Anführungszeichen erwartet – etwa `lines`, um Linien zu zeichnen. Über die Argumente `rows` und `columns` kann die Anzahl der Zeilen und Spalten manuell bestimmt werden, andernfalls legt R für sie Werte auf Basis der Anzahl und Stufen der Gruppierungsfaktoren fest.

```
> nSubj    <- 25                                # Zellbesetzung
> P        <- 2                                  # Anzahl Gruppen UV1
> Q        <- 2                                  # Anzahl Gruppen UV2
> IQ       <- rnorm(P*Q*nSubj, mean=100, sd=15) # AV1
> height   <- rnorm(P*Q*nSubj, 175, 7)        # AV2

# Gruppenzugehörigkeiten bzgl. UV1 und UV2
> IV1    <- factor(rep(c("control", "treatment"), each=Q*nSubj))
> IV2    <- factor(rep(c("f", "m"), P*nSubj))
> myDf   <- data.frame(IV1, IV2, IQ, height)
> coplot(IQ ~ height | IV1*IV2, data=myDf)      # Streudiagramm
```

Das Paket `lattice` stellt eine Vielzahl von Funktionen bereit, die ähnlich wie `coplot()` Diagramme erzeugen, die Daten getrennt nach Gruppen simultan in

mehreren Subdiagrammen darstellen (Abb. 10.27).²⁵ Diese Darstellungsart wird auch als Trellis-Diagramm bezeichnet (Cleveland, 1993) und erlaubt es, alle hier bereits dargestellten Diagrammtypen zu verwenden, etwa Streudiagramme mit `xyplot()`, Säulendiagramme mit `barchart()`, Cleveland Dotcharts mit `dotplot()`, Boxplots mit `bwplot()`, Stripcharts mit `stripplot()` oder Histogramme mit `histogram()`. Der Aufruf all dieser Funktionen erfolgt immer in folgender Grundform:

```
> xyplot((y-Koordinaten) ~ (x-Koordinaten) | <Faktor>, data=<Datensatz>)
```

Die Formel definiert auf dieselbe Weise wie bei `coplot()`, nach welchen Gruppen getrennt die separaten Subdiagramme erstellt werden sollen. Auch hier ist es möglich, Gruppen durch die Kombination mehrerer Faktoren zu bilden, indem `\~>Faktor1 * <Faktor2>` hinter dem `|` Zeichen verwendet wird. Trellis-Diagramme verfügen über eine Reihe weiterer Argumente, durch die sie sich weitgehend anpassen lassen, Details enthalten die zugehörigen Hilfe-Seiten.

```
> library(lattice)
> histogram(IQ ~ height | IV1*IV2, data=myDf, main="Histogramme pro Gruppe")
```

10.8.4 Matrix aus Streudiagrammen

Enthält ein Datensatz viele Variablen, können ihre paarweise gebildeten gemeinsamen Verteilungen in einer Matrix aus Streudiagrammen simultan veranschaulicht werden, wie sie `pairs()` erzeugt. Die Matrix enthält jeweils so viele Zeilen und Spalten, wie Variablen vorhanden sind, wobei sich in einer Zelle (i, j) das Streudiagramm der gemeinsamen Verteilung der i -ten und j -ten Variable befindet. Auf der Hauptdiagonale (Zellen (i, i)) werden in der Voreinstellung die Variablennamen ausgegeben. An der Hauptdiagonale gespiegelte Zellen $((i, j)$ und (j, i)) stellen die selbe gemeinsame Verteilung dar, allerdings mit vertauschten Achsen (Abb. 10.28).

```
> pairs(x=<Matrix>, labels=<Variablennamen>, panel=<Zeichenfunktion>,
+        lower.panel=<Zeichenfunktion>, upper.panel=<Zeichenfunktion>,
+        diag.panel=<Zeichenfunktion>)
```

Für `x` ist eine spaltenweise aus Variablen gebildete Matrix (oder ein Datensatz) anzugeben, deren paarweise gemeinsame Verteilungen dargestellt werden sollen. Ergeben sich die Variablennamen nicht aus den Spaltennamen von `x`, können sie als Vektor für `labels` genannt werden. `panel` erwartet den Namen einer Zeichenfunktion, deren Ergebnis in den Zellen außerhalb der Diagonale der erzeugten Graphik-Matrix abgebildet wird. Die Funktion muss als Argumente zwei Vektoren (Spalten

²⁵ `lattice` setzt wie das eine vergleichbare Funktionalität bietende Paket `ggplot2` nicht auf dem graphischen Basissystem von R auf, sondern nutzt das in der Standardinstallation von R bereits enthaltene `grid` Paket. Damit ergeben sich bzgl. der Feinkontrolle von Graphiken einige Abweichungen zu den Darstellungen in Abschn. 10.3 und 10.5, für Details vgl. Sarkar (2008).

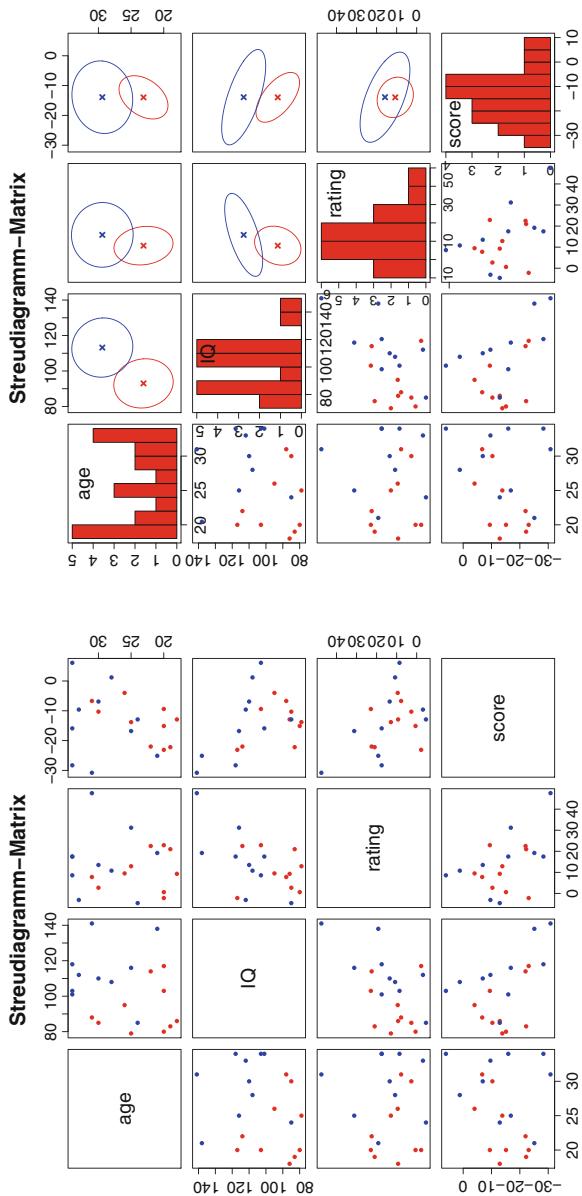


Abb. 10.28 Matrizen von paarweisen Streudiagrammen mehrerer Variablen mit `pairs()`

von `x`) verarbeiten können und darf kein neues Diagramm öffnen, muss also eine Low-Level-Funktion sein. Voreinstellung ist `points` für ein Streudiagramm. Sollen die Zellen oberhalb und unterhalb der Hauptdiagonale der Graphik-Matrix unterschiedliche Diagrammtypen zeigen, können für `lower.panel` und `upper.panel` ebenfalls separate Zeichenfunktionen angegeben werden, die denselben Bedingungen wie jene für `panel` unterliegen. In der Voreinstellung enthalten die Zellen auf der Hauptdiagonale jeweils den Variablenamen der zugehörigen Zeile bzw. Spalte. Durch Angabe einer Zeichenfunktion für `diag.panel`, die als Argument die Daten der Variable *i* erhält, lassen sich jedoch auch hier Diagramme einfügen (Voreinstellung ist `NULL`).

Im Beispiel seien VPn in zwei Gruppen aufgeteilt und an ihnen vier Abhängige Variablen erhoben worden. Die Streudiagramme sollen die Gruppenzugehörigkeit über die Farbe der Datenpunktssymbole kenntlich machen (vgl. Abschn. 10.6.8).

```
> nSubj <- 20                                # Anzahl Versuchspersonen
> P      <- 2                                  # Anzahl Gruppen
> group <- rep(c("CG", "T"), each=nSubj/P)    # Gruppenzugehörigkeit

# Abhängige Variablen
> age     <- sample(18:35, nSubj, replace=TRUE)
> IQ      <- round(rnorm(nSubj, rep(c(100, 115), each=nSubj/P), 15))
> rating  <- round(0.4*IQ - 30 + rnorm(nSubj, 0, 10), 1)
> score   <- round(-0.3*IQ + 0.7*age + rnorm(nSubj, 0, 8), 1)
> (mvDf  <- data.frame(group, age, IQ, rating, score)) # Datensatz ...

# Matrix aus Streudiagrammen darstellen
> pairs(mvDf[c("age", "IQ", "rating", "score")]),
+       main="Streudiagramm-Matrix", pch=16,
+       col=c("red", "blue")[unclass(mvDf$group)])
```

Da sich nur Low-Level Zeichenfunktionen für die `panel` Argumente eignen, ist häufig der Umweg über selbst erstellte Funktionen notwendig, um bestimmte Graphiken in der Matrix darzustellen (vgl. Abschn. 11.1): für High-Level-Funktionen muss dafür zunächst `par(new=TRUE)` aufgerufen werden, damit sie selbst kein neues Graphik-Device öffnen (vgl. Abschn. 10.9.2).

Hier soll dies zunächst für in der Hauptdiagonale darzustellende Histogramme geschehen. Zusätzlich sollen die für beide Gruppen getrennten Streuungsellipsen der gemeinsamen Verteilung in den Zellen oberhalb der Hauptdiagonale gezeigt werden (Abb. 10.23, vgl. Abschn. 10.6.8).

```
# Funktion, um Histogramm darzustellen, ohne neue Graphik zu öffnen
> myHist <- function(x, ...) { par(new=TRUE); hist(x, ..., main="") }

# Funktion, um Streuungsellipsen getrennt für beide Gruppen zu zeichnen
> myEll <- function(x, y, ...) {
+   # trenne Beobachtungen in Variablen x und y nach Gruppen
+   xLL <- tapply(x, mvDf$group, "[")
+   yLL <- tapply(y, mvDf$group, "[")
+
+   # verbinde Variablen getrennt nach Gruppen zu einer Datenmatrix
```

```

+ matCG <- cbind(xLL$CG, yLL$CG)
+ matT  <- cbind(xLL$T, yLL$T)
+
+ # berechne für jede Gruppe Zentroid und Cholesky-Dekomposition
+ ctrCG <- colMeans(matCG)
+ ctrT  <- colMeans(matT)
+ RRcg  <- chol(cov(matCG))
+ RRt   <- chol(cov(matT))
+
+ # berechne Ellipse: zunächst Winkel
+ angles <- seq(0, 2*pi, length.out=200)
+
+ # rotierte Ellipse im Ursprung
+ ellCG <- t(RRcg %*% rbind(cos(angles), sin(angles)))
+ ellT  <- t( RRt %*% rbind(cos(angles), sin(angles)))
+
+ # in Zentroid verschobene Ellipse
+ ellCGctr <- sweep(ellCG, 2, ctrCG, "+")
+ ellTctr  <- sweep(ellT, 2, ctrT, "+")
+
+ # Ellipse für jede Gruppe inkl. Zentroid zeichnen
+ points(ctrCG[1], ctrCG[2], col="red", pch=4, lwd=2)
+ lines(ellCGctr, col="red")
+
+ points(ctrT[1], ctrT[2], col="blue", pch=4, lwd=2)
+ lines(ellTctr, col="blue")
}

> pairs(mvDf[c("age", "IQ", "rating", "score")], diag.panel=myHist,
+        upper.panel=myEll, main="Streudiagramm-Matrix", pch=16,
+        col=c("red", "blue")[unclass(mvDf$group)])

```

10.9 Mehrere Diagramme in einem Graphik-Device darstellen

Die in den Abschn. 10.8.3 und 10.8.4 vorgestellten Funktionen verdeutlichen, dass auch mehr als ein Diagramm in ein Graphik-Device gezeichnet werden kann. Sollen nicht nur Diagramme gleichen Typs, sondern auch unterschiedliche Graphiken in einem Device dargestellt werden, lässt sich dessen Fläche manuell in mehrere rechteckige Bereiche aufteilen, die dann mit jeweils einem Diagramm gefüllt werden können. Auf diese Weise simultan dargestellte Diagramme können den Vergleich von Ergebnissen in verschiedenen Teilstichproben erleichtern, aber auch genutzt werden, um dieselben Daten parallel auf verschiedene Arten zu visualisieren.

10.9.1 layout()

Um n Diagramme in einem Graphik-Device darzustellen, kann dessen Fläche mit der `layout()` Funktion unter Zuhilfenahme einer Matrix `mat` aufgeteilt werden,

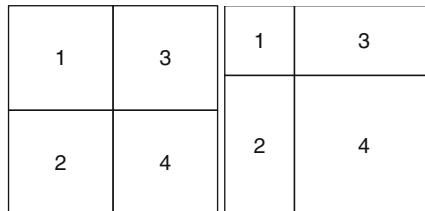


Abb. 10.29 Mit `layout()` eingeteilte Regionen eines Graphik-Devices

die aus den Zahlen $0, 1, \dots, n$ gebildet ist. Die Ergebnisse der sich an `layout()` anschließenden n High-Level-Funktionen zum Erstellen von Diagrammen werden automatisch den durch `mat` definierten Regionen zugewiesen, statt jeweils die gesamte Fläche des aktiven Devices zu überschreiben (Abb. 10.29 und 10.30).

```
> layout(mat=<Matrix>, widths=<Vektor>, heights=<Vektor>)
```

`mat` bestimmt die Einteilung der Device-Fläche in Regionen, indem ihre Zellen die der gedanklich analog in Planquadrate eingeteilten Device-Fläche symbolisieren. Enthält eine Zelle von `mat` dabei die 0, wird die zugehörige Region beim Befüllen mit Grafiken übersprungen und bleibt leer. Dann ist die Anzahl n der dargestellten Diagramme kleiner als die Zahl der Zellen von `mat`, also der ursprünglichen Plan-

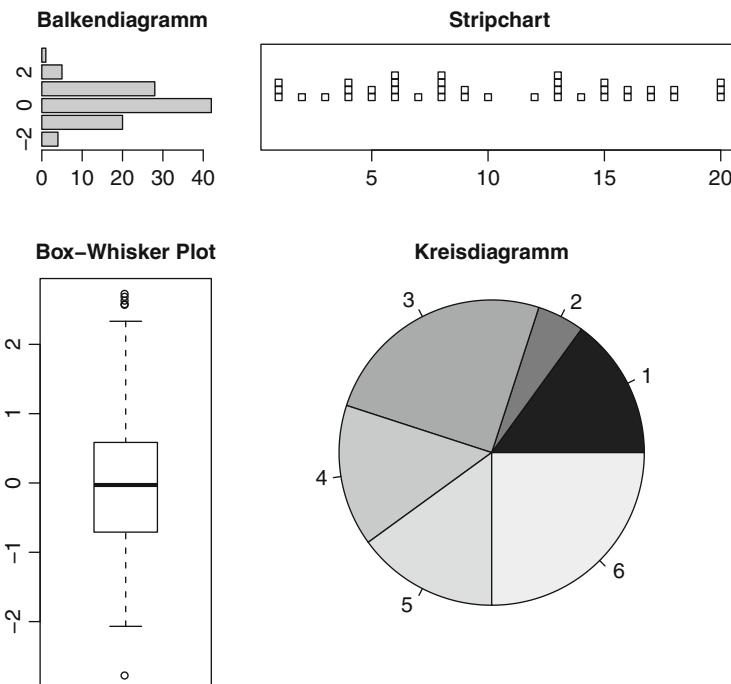


Abb. 10.30 Verschiedene Diagramme in einem Diagrammfenster mittels `layout()`

quadrate. Über das Argument `widths` ist die Breite der Spalten und über `heights` die Höhe der Zeilen veränderbar – in der Voreinstellung erfolgt die Aufteilung gleichmäßig (Abb. 10.31). Breite und Höhe können zum einen als relative Größe in Form eines Vektors von ganzzahligen Gewichten ausgedrückt werden, die dann an der Summe aller Gewichte relativiert werden. Zum anderen können die Maße auch als absolute Werte mit der Funktion `1cm(<Zahl>)` in der Einheit cm eingegeben werden. Um die n Planquadrate durch Umrandung sichtbar zu machen, dient der Befehl `layout.show(n)`.

```
> (mat1 <- matrix(1:4, 2, 2))    # vier Regionen derselben Größe definieren
[,1] [,2]
[1,]   1   3
[2,]   2   4

> layout(mat1)                  # Device-Fläche teilen
> layout.show(4)                # Regionen anzeigen

# dieselben Regionen mit unterschiedlicher Größe
> layout(mat1, widths=c(1, 2), heights=c(1, 2))    # Aufteilung 1/3, 2/3
> layout.show(4)                # Regionen anzeigen

> barplot(table(round(rnorm(100))), horiz=TRUE, main="Balkendiagramm")
> boxplot(rt(100, df=5), main="Box-Whisker Plot")
> stripchart(sample(1:20, 40, replace=TRUE), method="stack",
+             main="Stripchart")

> pie(table(sample(1:6, 20, replace=TRUE)), main="Kreisdiagramm")
```

Benachbarte Planquadrate der Device-Fläche lassen sich zusammenfassen, indem den zugehörigen Matrixelementen dieselbe Zahl zugeordnet wird (Abb. 10.31). Auch hier ist die Anzahl n der dargestellten Diagramme kleiner als die Zahl der Zellen von `mat`.

```
# Zellen der ersten Zeile zusammengefasst, Region links unten leer
> (mat2 <- matrix(c(1, 0, 1, 2), 2, 2))
[,1] [,2]
[1,]   1   1
[2,]   0   2

> layout(mat2)
> stripchart(sample(1:20, 40, replace=TRUE), method="stack",
+             main="Stripchart")

> barplot(table(round(rnorm(100))), main="Säulendiagramm")
```

10.9.2 `par(mfrow, mfcol, fig)`

Ein weiterer Weg Device-Flächen zu unterteilen, besteht in der Verwendung der Argumente `mfrow` oder `mfcol` von `par()`.

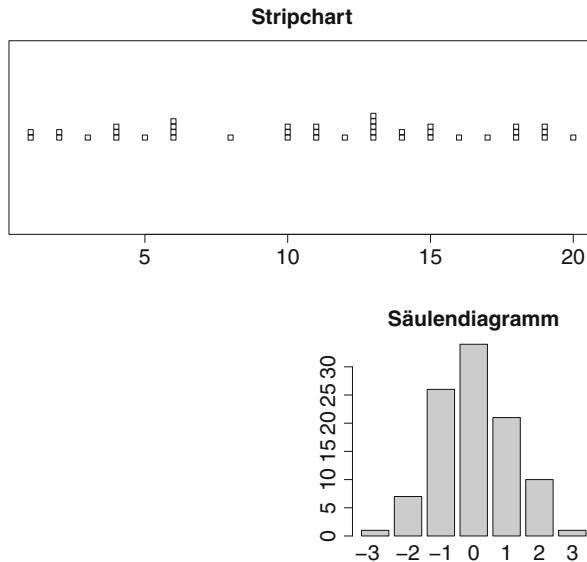


Abb. 10.31 Mehrere, teils zusammengefasste, teils leere Regionen in einem Diagrammfenster mittels `layout()`

```
> par(mfrow=c(Anzahl Zeilen), (Anzahl Spalten))
> par(mfcoll=c(Anzahl Zeilen), (Anzahl Spalten))
```

An `mfrow` oder `mfcoll` ist ein Vektor mit zwei Elementen zu übergeben, die die Anzahl der Zeilen und Spalten definieren, aus denen sich dann die einzelnen Regionen der Device-Fläche ergeben. Diese Regionen besitzen dieselbe Größe und lassen sich nicht weiter zusammenfassen. Der Unterschied zwischen beiden Argumenten betrifft die Reihenfolge, in der die entstehenden Regionen mit Diagrammen gefüllt werden: mit `mfrow` werden durch sich anschließende High-Level-Funktionen nacheinander zunächst die Zeilen, bei `mfcoll` entsprechend nacheinander die Spalten gefüllt. Sind auf der Diagrammfläche in einem rechteckigen Layout letztlich `(Anzahl)` viele verschiedene Diagramme unterzubringen, kann alternativ auch die Funktion `n2mfrow(nr.plots=(Anzahl))` zur Erstellung eines geeigneten Vektors verwendet werden, der dann an `mfrow` zu übergeben ist.

```
> windows(width=7, height=4) # Diagrammgröße ändern
> par(mfrow=c(1, 2)) # Diagrammfenster aufteilen
> boxplot(rt(100, df=5), xlab=NA, notch=TRUE, main="Box-Whisker Plot")
> plot(rnorm(10), pch=16, xlab=NA, ylab=NA, main="Streudiagramm")
```

Als Alternative kann ebenfalls mit `par()` vor einem High-Level-Befehl zum Erstellen eines Diagramms die rechteckige Figure-Region innerhalb der Device-Fläche definiert werden, in die das Diagramm gezeichnet wird.

```
> par(fig=(Vektor), new=TRUE)
```

Die Elemente des Vektors `fig` stellen dabei die Koordinaten der Rechteckkanten in der Reihenfolge links, rechts, unten, oben dar und müssen Werte im Bereich von 0–1 besitzen. Der für die untere und obere Kante eingetragene Wert gilt als deren jeweilige y -Koordinate, wobei 0 die Koordinate des unteren und 1 die des oberen Device-Randes ist. Der für die linke und rechte Kante eingetragene Wert gilt als deren jeweilige x -Koordinate, wobei 0 die Koordinate des linken und 1 die des rechten Device-Randes ist. Das Argument `new=TRUE` legt fest, dass der folgende Graphikbefehl dem aktiven Device hinzugefügt werden soll. Indem `par(fig)` mehrfach alternierend mit High-Level-Graphikfunktionen aufgerufen wird, lassen sich mehrere Diagramme in einem Graphik-Device plazieren (Abb. 10.32).

```
# Simulation von 1000 mal 10 Bernoulli-Experimenten
> resBinom <- rbinom(1000, size=10, prob=0.3)

# Ergebnis in Faktor umwandeln, um Kategorien hinzufügen zu können,
# die möglich sind aber nicht auftauchen
> facBinom <- factor(resBinom, levels=0:10)

# Häufigkeitsverteilung bestimmen und als senkrechte Linien darstellen
> tabBinom <- table(facBinom)
> par(fig=c(0, 1, 0.20, 1))
> plot(tabBinom, type="h", bty="n", xlim=c(0, 10), xlab=NA,
+       ylab="Häufigkeit", main="Ergebnisse von 1000*10 Bernoulli
+       Experimenten (p=0.3)")

# Werte zusätzlich als Punkte einzeichnen
```

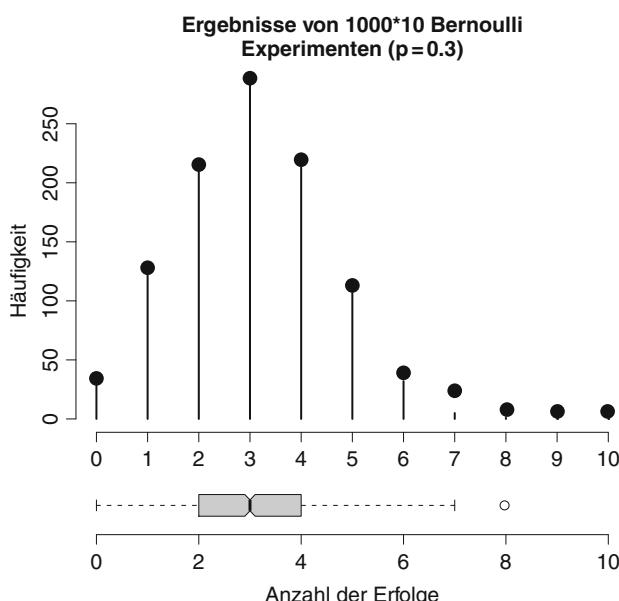


Abb. 10.32 Anpassen der Größe verschiedener Regionen mit `par(fig)`

```
> points(names(tabBinom), tabBinom, pch=16, cex=2, col="red")

# Im unteren Diagrammbereich Boxplot einzeichnen
> par(fig=c(0, 1, 0, 0.35), bty="n", new=TRUE)
> boxplot(resBinom, horizontal=TRUE, ylim=c(0, 10), notch=TRUE,
+           col="gray", xlab="Anzahl der Erfolge")
```

10.9.3 split.screen()

Eine dritte Methode, um die Device-Fläche in Regionen zu unterteilen und mit separaten Diagrammen zu füllen, stellt die `split.screen()` Funktion bereit, die in Kombination mit den Funktionen `screen()` und `close.screen()` zu verwenden ist.

```
> split.screen(figs=(2-Vektor oder Nx4-Matrix), screen=<Nummer>)
```

Das Hauptargument `figs` erwartet einen Vektor mit zwei Elementen oder eine Matrix mit vier Spalten, die beide die Aufteilung der Device-Fläche definieren können. Ein Vektor legt durch seine beiden Elemente die Zahl der Zeilen und Spalten fest, wie dies auch in `par(mfrow=...)` geschieht. Die Fläche wird in diesem Fall gleichmäßig unter den Regionen aufgeteilt. Dagegen wird jede der in einer für `figs` übergebenen vierspaltigen Matrix enthaltenen Zeilen als Definition jeweils einer rechteckigen Region interpretiert. Die Werte in einer Zeile stellen dabei die Koordinaten der Rechteckkanten in der Reihenfolge links, rechts, unten, oben dar und müssen im Bereich von 0–1 liegen. Der für die untere und obere Kante eingetragene Wert gilt als deren jeweilige y -Koordinate, wobei 0 die Koordinate des unteren und 1 die des oberen Device-Randes ist. Der für die linke und rechte Kante eingetragene Wert gilt als deren jeweilige x -Koordinate, wobei 0 die Koordinate des linken und 1 die des rechten Device-Randes ist. Auf diese Weise können auch sich überschneidende oder die Diagrammfläche nicht vollständig ausfüllende Regionen definiert werden (Abb. 10.33).

Die mit `split.screen()` gebildeten Regionen sind intern bei 1 beginnend durchnumeriert und können über ihre Nummer angesprochen werden. Dies ist in zwei Fällen notwendig: zum einen kann eine Teilfläche durch erneuten Aufruf von `split.screen(screen=<Nummer>)` weiter unterteilt werden, wenn dabei an das Argument `screen` die Nummer der zu unterteilenden Fläche übergeben wird. Zum anderen muss mit der `screen(n=<Nummer>)` Funktion vor jedem folgenden Befehl zur Diagrammerstellung die Nummer der Fläche genannt werden, in die das Diagramm gezeichnet werden soll.

Eine Teilfläche sollte vollständig bearbeitet werden, ehe die nächste Region ausgewählt wird. Zwar ist es prinzipiell möglich, eine bereits verlassene Teilfläche wieder zu aktivieren, allerdings mit dem Risiko unerwünschter Effekte. Der Inhalt einer mit `split.screen()` erzeugten Teilfläche kann mit `erase.screen(n=<Nummer>)` wieder gelöscht werden.

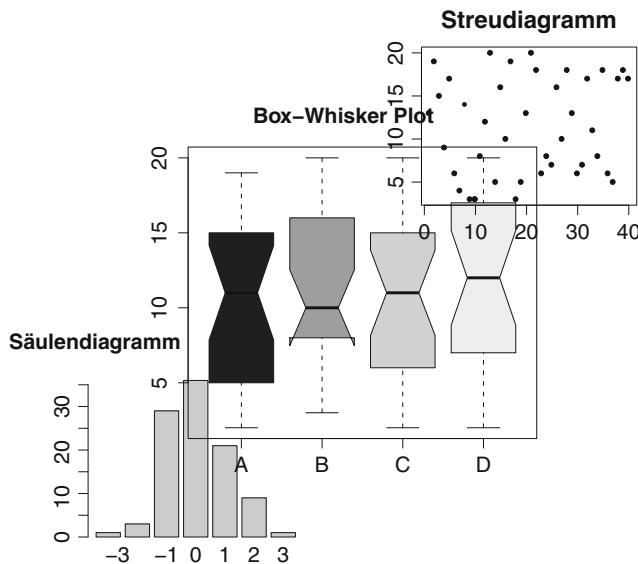


Abb. 10.33 Sich überlappende Regionen mit `screen()` erzeugen

Um eine mit `split.screen()` erzeugte Teilfläche abzuschließen und damit ein weiteres Hinzufügen von Elementen zu verhindern, dient die `close.screen(n=<Nummer>, all.screens=FALSE)` Funktion. Mit ihr können über das Argument `all.screens=TRUE` auch alle Teilflächen simultan gesperrt werden. Dies sollte etwa geschehen, wenn alle Regionen eines Graphik-Devices mit Diagrammen gefüllt und keine weiteren Änderungen gewünscht sind.

```
# 3 Regionen definieren
> splitMat <- rbind(c(0.0, 0.5, 0.0, 0.5), c(0.15, 0.85, 0.15, 0.85),
+                      c(0.5, 1.0, 0.5, 1.0))

> split.screen(splitMat)           # Diagrammfläche unterteilen
> screen(1)                      # in Region 1 zeichnen
> barplot(table(round(rnorm(100))), main="Säulendiagramm")
> screen(2)                      # in Region 2 zeichnen
> boxplot(sample(1:20, 100, replace=TRUE) ~ factor(rep(LETTERS[1:4],
+ each=25)), col=rainbow(4), notch=TRUE, main="Box-Whisker Plot")

> screen(3)                      # in Region 3 zeichnen
> plot(sample(1:20, 40, replace=TRUE), pch=20, xlab=NA, ylab=NA,
+       main="Streudiagramm")

> close.screen(all.screens=TRUE)    # Bearbeitung abschließen
```


Kapitel 11

R als Programmiersprache

Wie eingangs dieses Textes erwähnt, bietet R nicht nur Mittel zur numerischen und graphischen Datenanalyse, sondern ist gleichzeitig eine Programmiersprache, die dieselbe Syntax wie die bisher behandelten Auswertungen verwendet. Das seinerseits sehr umfangreiche Thema der Programmierung mit R soll in den folgenden Abschnitten nur soweit angedeutet werden, dass einfache Funktionen schnell selbst erstellt und nützliche Sprachkonstrukte wie z. B. Kontrollstrukturen verwendet werden können. Eine angemessene Behandlung des Themas, insbesondere auch der zahlreichen Debugging-Funktionen zur Fehlersuche in eigenen Programmen, sei der hierauf spezialisierten Literatur überlassen (Chambers, 2008; Ligges, 2009).

11.1 Eigene Funktionen erstellen

Funktionen sind eine Zusammenfassung von Befehlen auf Basis von beim Aufruf mitgelieferten Eingangsinformationen, den sog. *Funktionsargumenten*. Ebenso wie etwa Matrizen als Objekte der Klasse `matrix` können Funktionen als Objekte der Klasse `function` erstellt werden, indem über `function()` eine Funktion definiert und das Ergebnis einem Objekt zugewiesen wird. Selbst erstellte Funktionen haben denselben Status und dieselben Möglichkeiten wie mit R mitgelieferte Funktionen, stellen also etwa keine in ihrer Funktionalität eingeschränkte Makros dar.

```
> <Name> <- function(<Argument1>=<Voreinstellung>, <Arg2>=<Voreinst.>, ...) {  
+   <Befehl1>  
+   <Befehl2>  
+   ...  
+ }
```

Aus welchen Befehlen eine Funktion besteht, gibt R aus, wenn der Funktionsname ohne runde Klammern als Befehl eingegeben wird.¹ Bei `sd()` etwa ist erkennbar,

¹ Bei von R mitgelieferten Funktionen bestehen die Befehle allerdings häufig nur aus dem Aufruf von internen, nicht unmittelbar einsehbaren Methoden. Über den Aufruf des spezifischen Methodennamens (vgl. Abschn. 11.1.5), aber auch über den Quelltext von R ist ihre Analyse jedoch meist möglich (vgl. auch `?getS3method`).

dass nach einigen Fallunterscheidungen mit `if()` letztlich `var()` aufgerufen und die Wurzel aus dem Ergebnis gezogen wird.

```
> sd
function (x, na.rm=FALSE)
{
  if (is.matrix(x))
    apply(x, 2, sd, na.rm=na.rm)
  else if (is.vector(x))
    sqrt(var(x, na.rm=na.rm))
  else if (is.data.frame(x))
    sapply(x, sd, na.rm=na.rm)
  else sqrt(var(as.vector(x), na.rm=na.rm))
}

<environment: namespace:stats>
```

11.1.1 Funktionskopf mit Argumentliste

Zunächst ist bei einer Funktion der sog. *Funktionskopf* innerhalb der runden Klammern () zu definieren, in dem durch Komma getrennt jene sog. *formalen* Argumente benannt werden, die eine Funktion als Eingangsinformation akzeptiert. Die hier benannten Argumente stehen innerhalb des Funktionsrumpfes (s. u.) als Objekte zur Verfügung. Auch ein leerer Funktionskopf ist möglich, wenn die folgenden Befehle nicht von äußeren Informationen abhängen.

Jedem formalen Argument kann auf ein Gleichheitszeichen folgend ein Wert zugewiesen werden, den das Argument als Voreinstellung (Default) annimmt, sofern es beim Aufruf der Funktion nicht explizit genannt wird. Fehlt ein solcher Default-Wert, muss beim Aufruf der Funktion zwingend ein Wert für das Argument übergeben werden. Argumente mit Voreinstellung sind beim Aufruf dagegen optional (vgl. Abschn. 1.2.5).

Argumente können von jeder Klasse, also auch ihrerseits Funktionen sein – eine Möglichkeit, die etwa bei `apply()` oder `curve()` Verwendung findet. Anders als in vielen Programmiersprachen ist es nicht notwendig, im Funktionskopf explizit anzugeben, was für eine Klasse ein Objekt haben muss, das für ein Argument bestimmt ist. Eine Prüfung, ob beim Funktionsaufruf ein für die weiteren Berechnungen geeignetes Objekt für ein Argument übergeben wurde, sollte im Funktionsrumpf erfolgen.

Das Argument ... besitzt eine besondere Bedeutung: beim Aufruf der Funktion hierfür übergebene Werte können im Funktionsrumpf unter dem Namen ... verwendet werden. Dabei kann es sich um mehrere, durch Komma getrennte Werte handeln, die sich gemeinsam mittels `(Objekt) <- list(...)` in einem Objekt speichern und weitergeben lassen. Das Argument bietet sich besonders dann an, wenn im Funktionsrumpf eine Funktion verwendet wird, bei der noch nicht feststeht, ob und wenn ja wie viele Argumente sie ihrerseits benötigt, wenn die selbst definierte Funktion später ausgeführt wird.

11.1.2 Funktionsrumpf mit Befehlen

Auf den Funktionskopf folgt eingeschlossen in geschweifte Klammern {} der sich ggf. über mehrere Zeilen erstreckende sog. *Funktionsrumpf*, der aus einer Reihe von Befehlen und durch # eingeleiteten Kommentaren besteht.²

Die zum Funktionsrumpf gehörenden Befehle haben Zugriff auf alle in einer R-Sitzung zuvor erstellten Variablen. Variablen, die innerhalb eines Funktionsrumpfes definiert werden, stehen dagegen nur innerhalb der Funktion zur Verfügung, verfallen also nach dem Aufruf der Funktion.³

Werden innerhalb des Funktionsrumpfes Argumente aus dem Funktionskopf verwendet, ist das sog. *Lazy-Evaluation*-Prinzip zu beachten, nach dem der Inhalt von Argumenten erst mit ihrer ersten Verwendung ausgewertet wird. Im folgenden laute der Funktionskopf `function(var1, var2=mean(var1))`. Wird nun beim Aufruf der Funktion für var2 kein Wert übergeben, führt R die Berechnung `mean(var1)` als voreingestellten Wert für var2 erst dann aus, wenn var2 zum ersten Mal im Funktionsrumpf auftaucht. Jede in vorherigen Befehlen des Funktionsrumpfes ggf. vorgenommene Änderung an var1 würde sich dann im Wert von var2 niederschlagen. Die Funktionsrumpfe { var2 } und { var1 <-var1^2; var2 } hätten damit unterschiedliche Ergebnisse für var2 zur Folge. Bei der Verwendung von Zuweisungen in Voreinstellungen von Argumenten ist deshalb Sorgfalt geboten.

Das Lazy-Evaluation-Prinzip ermöglicht es auch, dass die Voreinstellung eines Arguments auf Objekte Bezug nimmt, die erst im Funktionsrumpf erstellt werden.

```
# Voreinstellung für xLims ist ein erst im Rumpf erstelltes Objekt
> myPlot <- function(x, y, xLims=xRange) {
+   xRange <- round(range(x), -1) + c(-10, 10)
+   # Auswertung von Argument xLims erfolgt erst im nächsten Befehl
+   plot(x, y, xlim=xLims)
}
```

Die Ausgabe der im Funktionsrumpf beherbergten Befehle erscheint nicht auf der Konsole, die Arbeitsschritte der Funktion sind also beim Funktionsaufruf nicht sichtbar. Für Ausgaben auf der Konsole müssen innerhalb des Funktionsrumpfes deshalb die Funktionen `print()` und `cat()` verwendet werden.

11.1.3 Rückgabewert

Das von einer Funktion zurückgegebene Ergebnis ist die Ausgabe des letzten Befehls im Funktionsrumpf. Soll die Rückgabe eines Wertes dagegen explizit erfolgen,

² Findet die vollständige Funktionsdefinition in einer Zeile Platz, sind die den Rumpf einfassenden geschweiften Klammern optional.

³ Dies sind sog. *lokale* Variablen, sie existieren in einer beim Funktionsaufruf eigens erstellten Umgebung, vgl. Abschn. 1.3.1. Für das in diesem Kontext relevante, aber komplexe Thema der Regeln für die Gültigkeit von Variablen (sog. *Scoping*) vgl. Chambers (2008) sowie Ligges (2009).

ist dieser in den `return(<Wert>)` Befehl einzuschließen. Dieser Befehl beendet die Ausführung der Funktion, auch wenn im Funktionsrumpf weitere Auswertungsschritte folgen – eine Situation, die auftreten kann, wenn sich der Befehlsfluss unter Verwendung von Kontrollstrukturen verzweigt (vgl. Abschn. 11.2.1).

Soll die Funktion keinen Wert zurückgeben, etwa weil ihre einzige Aufgabe das Erstellen von Graphiken ist, ist als letzte Zeile der Befehl `return(invisible(NULL))` zu verwenden. Mit `return(invisible(<Wert>))` gibt die Funktion `<Wert>` zurück, der jedoch nach ihrem Aufruf nicht auf der Konsole ausgegeben wird – ein etwa bei `barplot()` verwendetes Vorgehen.

Mehrere Werte lassen sich zurückgeben, indem sie zuvor im Funktionsrumpf in einem geeigneten Objekt zusammengefasst werden, etwa einem Vektor, einer Matrix, einer Liste oder einem Datensatz.

11.1.4 Eigene Funktionen verwenden

Nach ihrer Definition sind eigens erstellte Funktionen auf dieselbe Weise verwendbar wie die von R selbst oder von Zusatzpaketen zur Verfügung gestellten. Dies ist bei der Datenanalyse u. a. dann häufig einsetzbar, wenn an Befehle wie etwa `apply()` oder `curve()` Funktionen übergeben werden können. So lässt sich mit `curve()` zunächst keine Dichtefunktion einer normalverteilten Variable mit Erwartungswert 100 darstellen, da an `curve()` nur die Funktion `dnorm`, nicht aber deren Argument `mean` übergeben werden kann. Abhilfe können hier selbst erstellte Funktionen schaffen (Abb. 11.1).

```
# Funktionsdefinitionen für Normalverteilungen mit unterschiedlichem sd
> dnorm100.5 <- function(x) { dnorm(x, mean=100, sd=5) }
> dnorm100.7.5 <- function(x) { dnorm(x, mean=100, sd=7.5) }
> dnorm100.10 <- function(x) { dnorm(x, mean=100, sd=10) }
> dnorm100.15 <- function(x) { dnorm(x, mean=100, sd=15) }

# Diagramm mit einem Funktionsgraphen erstellen
> curve(dnorm100.5, from=50, to=150, col="green", lwd=2,
+         main="Normalverteilungen", xlab=NA, ylab="Dichte")

# weitere Funktionsgraphen und Legende hinzufügen
> curve(dnorm100.7.5, from=50, to=150, col="red", lwd=2, add=TRUE)
> curve(dnorm100.10, from=50, to=150, col="blue", lwd=2, add=TRUE)
> curve(dnorm100.15, from=50, to=150, col="black", lwd=2, add=TRUE)
> legend(x="topleft", legend=c("N(100, 5)", "N(100, 7.5)", "N(100, 10)",
+         "N(100, 15)", lwd=2, col=c("green", "red", "blue", "black")))
```

Funktionen müssen für ihre Verwendung nicht unbedingt in einem Objekt gespeichert werden. Analog zur Indizierung eines nicht als Objekt gespeicherten Vektors mit `c(1, 2, 3)[1]` lassen sich auch Funktionen direkt verwenden – man bezeichnet sie dann als sog. *anonyme*, weil namenlose Funktionen.

```
> (function(arg1, arg2) { arg1^2 + arg2^2 })(-3, 4)
[1] 25
```

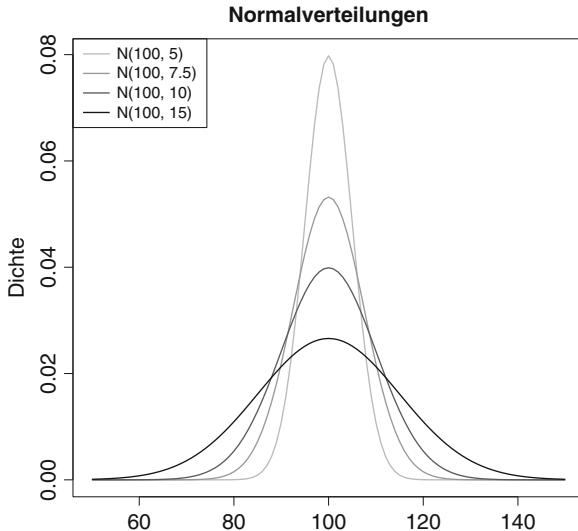


Abb. 11.1 Normalverteilungen mit $\mu = 100$ und unterschiedlichen Streuungen

Anonyme Funktionen lassen sich auch überall dort verwenden, wo eine Funktion als Argument zu übergeben ist – etwa bei `apply()`, um für jede Spalte einer Matrix ihre euklidische Länge zu berechnen.

```
> mat <- matrix(round(rnorm(16, 100, 15)), 4, 4)
> apply(mat, 2, function(x) { sqrt(sum(x^2)) })
[1] 218.9201 203.0049 205.1682 209.4517
```

11.1.5 Generische Funktionen

Generische Funktionen stellen eine Möglichkeit dar, mit der R objektorientiertes Programmieren unterstützt.⁴ Funktionen werden als *generisch* (oder auch als *polymorph*) bezeichnet, wenn sie sich abhängig von der Klasse der übergebenen Argumente automatisch unterschiedlich verhalten. Es existieren dann mehrere Varianten (sog. *Methoden*) einer Funktion, die alle unter demselben allgemeinen Namen ange- sprochen werden können. Funktionen, für die mehrere Methoden existieren, werden auch als *überladen* bezeichnet. Sie können ebenfalls unter einem eigenen, eindeutigen Namen aufgerufen werden, der nach dem Muster `<Funktionsname>. <Klasse>` aufgebaut ist, wobei sich `<Klasse>` auf die Klasse des ersten übergebenen Arguments bezieht. So existiert etwa für `plot()` die Methode `plot.lm()` für den Fall, dass das erste Argument ein mit `lm()` angepasstes lineares Modell ist. Ebenso verhält sich z. B. `summary()` für numerische Vektoren anders als für Faktoren.

⁴ Es handelt sich bei der hier vorgestellten Technik um das sog. S3-Paradigma – in Abgrenzung zum flexibleren, aber auch komplizierteren S4-Paradigma.

Die für eine Funktion verfügbaren Methoden nennt `methods(<Funktionsname>)`, auf die auch die mit `?<Funktionsname>` aufrufbare Hilfeseite im Abschnitt Usage verweist. Die generische Eigenschaft von Funktionen bleibt dem Anwender meist verborgen, da die richtige der unterschiedlichen Methoden automatisch in Abhängigkeit von der Klasse der übergebenen Argumente aufgerufen wird, ohne dass man dafür den passenden spezialisierten Funktionsnamen nennen müsste. Selbst erstellte Funktionen können im S3-Paradigma generisch gemacht werden, indem sie im Funktionsrumpf als letzten Befehl einen Aufruf von `UseMethod()` enthalten.

```
> UseMethod(generic="⟨Funktionsname⟩")
```

Für das Argument `generic` ist als Zeichenkette der Name der Funktion zu nennen, die generisch werden soll.

In einem zweiten Schritt sind alle gewünschten Methoden der mit generic bezeichneten Funktion zu erstellen, deren Namen nach dem <Funktionsname>.(<→Klasse>) Muster aufgebaut sein müssen. Für Argumente aller Klassen, die nicht explizit durch eine eigene Methode Berücksichtigung finden, muss eine Methode mit dem Namen <Funktionsname>.default angelegt werden, wenn die Funktion auch in diesem Fall arbeiten soll.

Als Beispiel wird eine Funktion `info()` erstellt, die eine wichtige Information über das übergebene Objekt ausgibt und dabei unterscheidet, ob es sich um einen numerischen Vektor (Klasse `numeric`), eine Matrix (Klasse `matrix`) oder einen Datensatz (Klasse `data.frame`) handelt.

Die verschiedenen Methoden einer generischen Funktion können im Prinzip gänzlich andere Argumente als diese erwarten. In der Praxis empfiehlt es sich jedoch, diese Freiheit nur dahingehend zu nutzen, dass jede Methode zunächst alle Argumente der generischen Funktion ebenfalls in derselben Reihenfolge mit denselben Voreinstellungen besitzt. Zusätzlich können Methoden danach weitere Argumente besitzen, die sich je nach Methode unterscheiden.

11.2 Kontrollstrukturen

Kontrollstrukturen ermöglichen es, die Abfolge von Befehlen gezielt zu steuern. Sie machen etwa die Ausführung von Befehlen von Bedingungen abhängig und verzweigen so den Befehlsfluss oder wiederholen in sog. *Schleifen* dieselben Befehle, solange bestimmte Nebenbedingungen gelten. Kontrollstrukturen tauchen häufig in selbst definierten Funktionen auf, lassen sich jedoch auch in gewöhnlichen Befehlsabfolgen verwenden.

11.2.1 Bedingungen zur Fallunterscheidung prüfen

Sollen Befehle im Rahmen einer Fallunterscheidung in Abhängigkeit davon ausgeführt werden, ob eine Bedingung gilt, ist die `if()` Funktion zu verwenden.

```
> if(<logischer Ausdruck>) {
+   <Befehle>
+ }
```

Als Argument erwartet `if(<Ausdruck>)` einen Ausdruck, der sich zu einem einzelnen Wahrheitswert TRUE oder FALSE auswerten lässt. Dies können etwa logische Vergleiche sein, wobei im Fall eines auf diese Weise erzeugten Vektors von Wahrheitswerten nur dessen erstes Element berücksichtigt wird. Im folgenden, durch `{}` eingeschlossenen Block, sind jene Befehle aufzuführen, die nur dann ausgewertet werden sollen, wenn `<Ausdruck>` gleich TRUE ist.⁵

```
> (x <- round(rnorm(1, 100, 15)))
[1] 115

> if(x > 100) { cat("x is", x, "(greater than 100)\n") }
x is 115 (greater than 100)
```

Eine Verzweigung des Befehlsflusses kann für den Fall, dass `<Ausdruck>` gleich FALSE ist, auch einen alternativen Block von Befehlen vorsehen, der sich durch das Schlüsselwort `else` eingeleitet an den auf `if()` folgenden Block anschließt.

⁵ Mit `if(FALSE){ <Befehle> }` können damit schnell viele Befehlszeilen von der Verarbeitung ausgeschlossen werden, ohne diese einzeln mit # auskommentieren zu müssen. Die ausgeschlossenen Zeilen müssen dabei jedoch nach wie vor syntaktisch korrekt, können also keine Kommentare im engeren Sinne sein.

```
> if(<Ausdruck>) {
+   <Befehle>
+ } else {
+   <Befehle>
+ }
```

Hier ist zu beachten, dass sich das Schlüsselwort `else` in derselben Zeile wie die schließende Klammer `}` des `if()` Blocks befindet, und nicht separat in der auf die Klammer folgenden Zeile stehen kann.

```
> x <- round(rnorm(1, 100, 15))
> if(x > 100) {
+   cat("x is", x, "(greater than 100)\n")
+ } else {
+   cat("x is", x, "(100 or less)\n")
+ }
x is 83 (100 or less)
```

Innerhalb der auf `if()` oder `else` folgenden Befehlssequenz können wiederum `if()` Abfragen stehen und so zu einer verschachtelten Prüfung einer Bedingung führen, die mehr als zwei mögliche Ausprägungen annehmen kann.

```
> (day <- sample(c("Mon", "Tue", "Wed"), 1))
[1] "Wed"

> if(day == "Mon") {
+   print("The day is Monday")
+ } else {
+   if(day == "Tue") {
+     print("The day is Tuesday")
+   } else {
+     print("The day is Wednsday")
+   }
+ }
[1] "The day is Wednsday"
```

Die `switch()` Anweisung ist für eben solche Situationen gedacht, in denen eine den Befehlfluss steuernde Nebenbedingung mehr als zwei Ausprägungen besitzen kann und für jede dieser Ausprägungen anders verfahren werden soll. Sie ist meist übersichtlicher als eine ebenfalls immer mögliche verschachtelte `if()` Abfrage. Lässt sich die Bedingung zu einem ganzzahligen Wert im Bereich von 1 bis zur Anzahl der möglichen Befehle auswerten, hat die Syntax die folgende Form:

```
> switch(EXPR=<Ausdruck>, <Befehl1>, <Befehl2>, ...)
```

Als erstes Argument ist der Ausdruck zu übergeben, von dessen Ausprägung abhängen soll, welcher Befehl ausgeführt wird – häufig ist dies lediglich ein Objekt. Es folgen durch Komma getrennt mögliche Befehle. EXPR muss in diesem Fall den auszuführenden Befehl numerisch bezeichnen, bei einer 1 würde der erste Befehl ausgewertet, bei einer 2 der zweite, usw. Sollen für einen Wert von EXPR mehrere Befehle ausgeführt werden, sind diese in `{}` einzuschließen.

```
> (val <- sample(1:3, 1))
[1] 3

> switch(val, print("val is 1"), print("val is 2"), print("val is 3"))
[1] "val is 3"
```

Ist EXPR dagegen eine Zeichenkette, hat die Syntax diese Gestalt:

```
> switch(EXPR=<Ausdruck>, <Wert1>=<Befehl1>, <Wert2>=<Befehl2>, ...
+           <Befehl>)
```

Auf EXPR folgt hier durch Komma getrennt eine Reihe von (nicht in Anführungszeichen stehenden) möglichen Werten mit einem durch Gleichheitszeichen angeschlossenen zugehörigen Befehl. Schließlich besteht die Möglichkeit, einen Befehl ohne vorher genannte Ausprägung für all jene Situationen anzugeben, in denen EXPR keine der zuvor explizit genannten Ausprägungen besitzt – andernfalls ist das Ergebnis in einem solchen Fall NULL.

```
> myCalc <- function(op, vals) {
+   switch(op,
+     plus = vals[1] + vals[2],
+     minus = vals[1] - vals[2],
+     times = vals[1] * vals[2],
+     vals[1] / vals[2])
+ }

> (vals <- round(rnorm(2, 1, 10)))
[1] -5 -7

> myCalc("minus", vals)
[1] 2

> myCalc("XX", vals)
[1] 0.7142857
```

11.2.2 Schleifen

Schleifen dienen dazu, eine Folge von Befehlen mehrfach ausführen zu lassen, ohne diese Befehle für jede Ausführung neu notieren zu müssen. Wie oft eine Befehlssequenz durchlaufen wird, kann dabei von Nebenbedingungen und damit von zuvor durchgeföhrten Auswertungen abhängig gemacht werden.⁶

⁶ Anders als in kompilierten Programmiersprachen wie etwa C sind Schleifen in R als interpretierter Sprache oft ineffizient. Als Grundregel sollten sie deswegen bei der Auswertung größerer Datenmengen nach Möglichkeit vermieden und durch sog. *vektorierte* Befehle ersetzt werden, die mehrere, als Vektor zusammengefasste Argumente gleichzeitig bearbeiten.

```
> for(<Variable> in <Vektor>) {
+   <Befehle>
+ }
```

Die `for()` Funktion besteht aus zwei Teilen: zum einen, in runde Klammern () eingeschlossen, dem *Kopf* der Schleife, zum anderen, darauf folgend in {} eingeschlossen, der zu wiederholenden Befehlssequenz – dem sog. *Rumpf*. Im Kopf der Schleife ist `<Variable>` ein Objekt, das im Rumpf verwendet werden kann. Es nimmt nacheinander die in `<Vektor>` enthaltenen Werte an, oft ist dies eine numerische Sequenz. Für jedes Element von `<Vektor>` wird der Schleifenrumpf einmal ausgeführt, wobei der Wert von `<Variable>` wie beschrieben nach jedem Durchlauf der Schleife wechselt.

```
> ABC <- c("Alfa", "Bravo", "Charlie", "Delta")
> for(i in ABC) { print(i) }
[1] "Alfa"
[1] "Bravo"
[1] "Charlie"
[1] "Delta"
```

Mit Schleifen lassen sich Simulationen erstellen, mit denen etwa die Robustheit statistischer Verfahren bei Verletzung ihrer Voraussetzungen untersucht werden kann. Im folgenden Beispiel wird zu diesem Zweck zunächst eine Grundgesamtheit von 100000 Zufallszahlen einer normalverteilten Variable simuliert. Daraufhin werden die Zahlen quadriert und logarithmiert, also einer nichtlinearen Transformation unterzogen. In einer Schleife werden aus dieser Grundgesamtheit 1000 mal 2 Gruppen von je 20 Zahlen zufällig ohne Zurücklegen gezogen und mit einem *t*-Test für unabhängige Stichproben sowie mit einem *F*-Test auf Varianzhomogenität verglichen. Da beide Stichproben aus derselben Grundgesamtheit stammen, ist in beiden Tests die Nullhypothese richtig, die Voraussetzung der Normalverteiltheit dagegen verletzt. Ein robuster Test sollte in dieser Situation nicht wesentlich häufiger fälschlicherweise signifikant werden, als durch das nominelle α -Niveau vorgegeben. Während der *t*-Test im Beispiel in der Tat robust ist, reagiert der *F*-Test sehr liberal – er fällt weit häufiger signifikant aus, als das nominelle α -Niveau erwarten ließe.

```
> src      <- log((rnorm(100000, 0, 1))^2)          # Grundgesamtheit
> alpha    <- 0.05                                     # nominelles alpha
> nTests   <- 1000                                    # Anzahl simulierter Tests
> n        <- 20                                      # Gruppengröße
> sigVecT <- numeric(nTests)                         # für Ergebnisse t.test()
> sigVecV <- numeric(nTests)                         # für Ergebnisse var.test()

> for (i in seq(along=numeric(nTests))) {
+   group1     <- sample(src, n, replace=FALSE)         # Stichprobe 1
+   group2     <- sample(src, n, replace=FALSE)         # Stichprobe 2
+   resT       <- t.test(group1, group2)                # t-Test
+   resV       <- var.test(group1, group2)               # Varianz-Test
+   sigVecT[i] <- as.logical(resT$p.value < alpha)    # t signifikant?
+   sigVecV[i] <- as.logical(resV$p.value < alpha)    # Var. signifikant?
+ }
```

```
> cat("Erwartete Anzahl signifikanter Tests:", alpha*nTests, "\n")
Erwartete Anzahl signifikanter Tests: 50
```

```
> cat("Signifikante t-Tests:", sum(sigVecT), "\n")
Signifikante t-Tests: 46
```

```
> cat("Signifikante Tests auf Varianzhomogenität:", sum(sigVecV), "\n")
Signifikante Tests auf Varianzhomogenität: 197
```

Im konkreten Beispiel ließe sich die `for()` Schleife auch vermeiden, indem stärker R-eigene Konzepte zur häufigen Wiederholung derselben Arbeitsschritte sowie Möglichkeiten zur Vektorisierung genutzt werden. Häufig ist insbesondere letzteres aus Effizienzgründen erstrebenswert. Schleifen können dann gut vermieden werden, wenn die Berechnungen in den einzelnen Schleifendurchläufen voneinander unabhängig sind, wenn – wie hier – ein Durchlauf also keine Werte benötigt, die in vorherigen Durchläufen berechnet wurden.

```
# wiederhole nTests mal für beide Stichproben das Ziehen von je n Personen
> group1 <- replicate(nTests, sample(src, n, replace=FALSE))
> group2 <- replicate(nTests, sample(src, n, replace=FALSE))
> groups <- rbind(group1, group2)      # füge Daten zu Matrix zusammen

# berechne Teststatistiken für t-Tests: zunächst Streuungsschätzungen
> estSigDiffs <- apply(groups, 2, function(x) {
+   sqrt(1/n) * sqrt(var(x[1:n]) + var(x[(n+1):(2*n)])) } )

# berechne alle Mittelwertsdifferenzen
> meanDiffs <- apply(groups, 2, function(x) {
+   mean(x[1:n]) - mean(x[(n+1):(2*n)]) } )

# vektorisierte Befehle, um t-Werte und p-Werte zu erhalten
> tVals <- meanDiffs / estSigDiffs      # t-Werte
> pVals <- 1-pt(tVals, (2*n)-2)        # p-Werte
> sum(pVals < alpha)                  # Anzahl signifikanter Tests
[1] 55
```

Während bei `for()` durch die Länge von `(Vektor)` festgelegt ist, wie häufig die Schleife durchlaufen wird, kann dies bei `while()` durch Berechnungen innerhalb der Schleife gesteuert werden.

```
> while(<Ausdruck>) { <Befehle> }
```

Der in `{}` eingefasste Schleifenrumpf wird immer wieder durchlaufen, solange der zu einem einzelnen logischen Wert auswertbare `<Ausdruck>` gleich TRUE ist. Typischerweise ändern Berechnungen im Schleifenrumpf `<Ausdruck>`, so dass dieser FALSE ergibt, sobald ein angestrebtes Ziel erreicht wird. Es ist zu gewährleisten, dass dies auch irgendwann der Fall ist, andernfalls handelt es sich um eine sog. Endlosschleife, die den weiteren Programmablauf blockiert und mit der ESC Taste abgebrochen werden müsste.

```
# Modulo-Funktion nur für positive Werte
```

```
> myMod <- function(x, y) {
+   if((x < 0) | (y < 0)) { return(invisible(NULL)) }      # x, y positiv?
+   while(abs(x) >= abs(y)) { x <- x-y }
+   print(x)
+ }
> myMod(37, 10)
[1] 7

> myMod(5, 17)
[1] 5
```

Um die Ausführung von Schleifen innerhalb des Schleifenrumpfes zu steuern, stehen die Schlüsselwörter `break` und `next` zur Verfügung. Durch `break` wird die Ausführung der Schleife abgebrochen, noch ehe das hierfür im Schleifenkopf definierte Kriterium erreicht ist. Mit `next` bricht nur der aktuelle Schleifendurchlauf ab und geht zum nächsten Durchlauf über, ohne ggf. auf `next` folgende Befehle innerhalb des Schleifenrumpfes auszuführen. Die Schleife wird also fortgesetzt, die auf `next` folgenden Befehle dabei aber einmal übersprungen.

```
> for(i in 1:10) {
+   if((i %% 2) != 0) { next }
+   if((i %% 8) == 0) { break }
+   print(i)
+ }
[1] 2
[1] 4
[1] 6
```

Eine durch `repeat` eingeleitete Schleife wird solange ausgeführt, bis sie explizit durch `break` abgebrochen wird. Der Schleifenrumpf muss also eine Bedingung überprüfen und eine `break` Anweisung beinhalten, um eine Endlosschleife zu vermeiden.

```
> repeat { (Befehle) }
> i <- 0
> repeat {
+   i <- i+1
+   if((i %% 4) == 0) { break }
+   print(i)
+ }
[1] 1
[1] 2
[1] 3
```

Literaturverzeichnis

- Adler, D. and Murdoch, D. (2010). rgl: 3D visualization device system (OpenGL) [Software]. URL <http://rgl.neoscientists.org/> (R package version 0.90)
- Agresti, A. (2007). *An Introduction to Categorical Data Analysis* (2. Aufl.). New York, NY: Wiley.
- Andres, J. (1996). Das allgemeine lineare Modell. In E. Erdfelder, R. Mausfeld, T. Meiser and G. Rudinger (Hrsg.), *Handbuch Quantitative Methoden* (S. 185–200). Weinheim: Beltz.
- Backhaus, K., Erichson, B., Plinke, W. and Rolf, W. (2008). *Multivariate Analysemethoden* (12. Aufl.). Heidelberg: Springer.
- Bates, D. and Mächler, M. (2010). Matrix: Sparse and Dense Matrix Classes and Methods [Software]. URL <http://CRAN.R-project.org/package=Matrix> (R package version 0.999375-37)
- Bates, D., Mächler, M. and Dai, B. (2009). lme4: Linear mixed-effects models using S4 classes [Software]. URL <http://lme4.r-forge.r-project.org/> (R package version 0.999375-32)
- Bengtsson, H. and Riedy, J. (2009). R.matlab: Read and write of MAT files together with R-to-Matlab connectivity [Software]. URL <http://www.braju.com/R/> (R package version 1.26)
- Bernaards, C. A. and Jennrich, R. I. (2005). Gradient Projection Algorithms and Software for Arbitrary Rotation Criteria in Factor Analysis. *Educational and Psychological Measurement*, 65, 676–696. URL <http://www.stat.ucla.edu/research/gpa/>
- Bliese, P. (2008). multilevel: Multilevel Functions [Software]. URL <http://CRAN.R-project.org/package=multilevel> (R package version 2.3)
- Bortz, J. (2005). *Statistik für Human- und Sozialwissenschaftler* (6. Aufl.). Heidelberg: Springer.
- Bortz, J., Lienert, G. A. and Boehnke, K. (2008). *Verteilungsfreie Methoden in der Biostatistik* (3. Aufl.). Heidelberg: Springer.
- Bronstein, I. N., Semendjajew, K. A., Musiol, G. and Mühlig, H. (2008). *Taschenbuch der Mathematik* (7. Aufl.). Frankfurt am Main: Harri Deutsch.
- Büning, H. and Trenkler, G. (1994). *Nichtparametrische statistische Methoden* (2. Aufl.). Berlin: Water de Gruyter.
- Canty, A. and Ripley, B. D. (2009). boot: Bootstrap R (S-Plus) Functions [Software]. URL <http://CRAN.R-project.org/package=boot> (R package version 1.2-41)
- Carr, D., Lewin-Koh, N. and Mächler, M. (2009). hexbin: Hexagonal binning routines [Software]. URL <http://CRAN.R-project.org/package=hexbin> (R package version 1.20.0)
- Chambers, J. M. (2008). *Software for Data Analysis: Programming with R*. New York, NY: Springer. URL <http://stat.stanford.edu/~jmc4/Rbook/>
- Champely, S. (2009). pwr: Basic functions for power analysis [Software]. URL <http://CRAN.R-project.org/package=pwr> (R package version 1.1.1)
- Chasalow, S. (2009). combinat: combinatorics utilities [Software]. URL <http://CRAN.R-project.org/package=combinat> (R package version 0.0-7)
- Cleveland, W. S. (1993). *Visualizing Data*. Summit, NJ: Hobart Press.
- Cowlishaw, M. F. (2008). *Decimal Arithmetic FAQ Part 1 – General Questions*. URL <http://speleotrove.com/decimal/decifaq1.html>

- Dalgaard, P. (2008). *Introductory Statistics with R* (2. Aufl.). London, UK: Springer. URL <http://www.biostat.ku.dk/~pd/ISwR.html>
- Davison, A. C. and Hinkley, D. V. (1997). *Bootstrap Methods and Their Applications*. Cambridge, UK: Cambridge University Press.
- Dorai-Raj, S. (2009). binom: Binomial Confidence Intervals For Several Parameterizations [Software]. URL <http://CRAN.R-project.org/package=binom> (R package version 1.0-5)
- Eddelbuettel, D. (2009). random: True random numbers using random.org [Software]. URL <http://www.random.org/> (R package version 0.2.1)
- Everitt, B. S. (2005). *An R and S-PLUS Companion to Multivariate Analysis*. New York, NY: Springer. URL <http://biostatistics.iop.kcl.ac.uk/publications/everitt/>
- Everitt, B. S. and Hothorn, T. (2006). *A Handbook of Statistical Analysis Using R*. Boca Raton, FL: Chapman & Hall/CRC. URL <http://CRAN.R-project.org/web/packages/HSAUR/>
- Faraway, J. J. (2004). *Linear Models with R*. Boca Raton, FL: Chapman & Hall/CRC. URL <http://www.maths.bath.ac.uk/~jjf23/LMR/>
- Faraway, J. J. (2006). *Extending the Linear Model with R: Generalized Linear, Mixed Effects and Nonparametric Regression Models*. Boca Raton, FL: Chapman & Hall/CRC. URL <http://www.maths.bath.ac.uk/~jjf23/ELM/>
- Faria, J. C., Grosjean, P. and Jelihovschi, E. (2010). Tinn-R – GUI/Editor for R environment [Software]. URL <http://sourceforge.net/projects/tinn-r/> (Version 2.3.4.4)
- Paul, F., Erdfelder, E., Lang, A.-G. and Buchner, A. (2007). G*Power 3: A Flexible Statistical Power Analysis Program for the Social, Behavioral, and Biomedical Sciences. *Behavior Research Methods*, 39 (2), 175–191. URL <http://www.psycho.uni-duesseldorf.de/abteilungen/aap/gpower3/>
- Fischer, G. (2008). *Lineare Algebra* (16. Aufl.). Wiesbaden: Vieweg+Teubner.
- Fox, J. (2002). *An R and S-PLUS Companion to Applied Regression*. Thousand Oaks, CA: Sage. URL <http://socserv.socsci.mcmaster.ca/jfox/Books/Companion/>
- Fox, J. (2005). The R Commander: A Basic-Statistics Graphical User Interface to R. *Journal of Statistical Software*, 14 (9), 1–42. URL <http://www.jstatsoft.org/v14/i09/>
- Fox, J. (2009). car: Companion to Applied Regression [Software]. URL <http://CRAN.R-project.org/package=car> (R package version 1.2–16)
- Fox, J., Kramer, A. and Friendly, M. (2009). sem: Structural Equation Models [Software]. URL <http://CRAN.R-project.org/package=sem> (R package version 0.9–19)
- François, R. (2006). *R Graph Gallery*. URL <http://addictedtor.free.fr/graphiques/>
- Friedl, J. E. F. (2006). *Mastering Regular Expressions* (3. Aufl.). Sebastopol, CA: O'Reilly. URL <http://regex.info/>
- Friedrichsmeier, T., Ecochard, P. and d'Hardemare, A. (2009). RKWward Data Analysis Tool [Software]. URL <http://rkward.sourceforge.net/> (Version 0.5.2)
- Gamer, M., Lemon, J. and Fellows, I. (2009). irr: Various Coefficients of Interrater Reliability and Agreement [Software]. URL <http://CRAN.R-project.org/package=irr> (R package version 0.82)
- Genz, A., Bretz, F., Miwa, T., Mi, X., Leisch, F., Scheipl, F. et al. (2009). mvtnorm: Multivariate Normal and t Distributions [Software]. URL <http://CRAN.R-project.org/package=mvtnorm> (R package version 0.9–9)
- Goldberg, D. (1991). What Every Computer Scientist Should Know About Floating-Point Arithmetic. *ACM Computing Surveys*, 23(1), 5–48. URL http://docs.sun.com/source/806-3568/ngc_goldberg.html
- Grosjean, P. (2006). R GUI Projects. URL http://www.sciviews.org/_rgui/
- Groß, J. (2008). nortest: Tests for Normality [Software]. URL <http://CRAN.R-project.org/package=nortest> (R package version 1.0)
- Härdle, W. and Simar, L. (2007). *Applied Multivariate Statistical Analysis* (2. Aufl.). Berlin: Springer.
- Harrell Jr, F. E. (2009a). Hmisc: Harrell Miscellaneous [Software]. URL <http://CRAN.R-project.org/package=Hmisc> (R package version 3.7-0)

- Harrell Jr, F. E. (2009b). rms: Regression Modeling Strategies [Software]. URL <http://CRAN.R-project.org/package=rms> (R package version 2.1-0)
- Hartung, J., Elpelt, B. and Klösener, K.-H. (2005). *Statistik: Lehr- und Handbuch der angewandten Statistik* (14. Aufl.). München: Oldenbourg.
- Hays, W. L. (1994). *Statistics* (5. Aufl.). Belmont, CA: Wadsworth Publishing.
- Heiberger, R. M. and Neuwirth, E. (2009). *R Through Excel*. New York, NY: Springer.
- Helbig, M., Theus, M. and Urbanek, S. (2005). JGR: Java GUI for R. *Statistical Computing and Graphics Newsletter*, 16 (2), 9–12.
- Hendrickx, J. (2008). perturb: Tools for evaluating collinearity [Software]. URL <http://CRAN.R-project.org/package=perturb> (R package version 2.03)
- Hoffman, D. D. (2000). *Visual Intelligence: How We Create What We See*. New York, NY: W. W. Norton & Company.
- Honaker, J., King, G. and Blackwell, M. (2009). Amelia: II: A Program for Missing Data [Software]. URL <http://gking.harvard.edu/amelia/> (R package version 1.2–1.5)
- Hornik, K. (2009). The R FAQ [Software-Handbuch]. URL <http://CRAN.R-project.org/doc/FAQ/>
- Hothorn, T., Bretz, F. and Genz, A. (2001). On Multivariate t and Gauß Probabilities in R. *R News*, 1 (2), 27–29. URL <http://CRAN.R-project.org/doc/Rnews/>
- Hothorn, T., Bretz, F. and Westfall, P. (2008). Simultaneous Inference in General Parametric Models. *Biometrical Journal*, 50 (3), 346–363.
- Hothorn, T., Hornik, K., van de Wiel, M. A. and Zeileis, A. (2008). Implementing a Class of Permutation Tests: The coin Package. *Journal of Statistical Software*, 28 (8), 1–23. URL <http://www.jstatsoft.org/v28/i08/>
- Ihaka, R. and Gentleman, R. (1996). R: A Language for Data Analysis and Graphics. *Journal of Computational and Graphical Statistics*, 5 (3), 299–314.
- Ihaka, R., Murrell, P. and Zeileis, A. (2009). colorspace: Color Space Manipulation [Software]. URL <http://CRAN.R-project.org/package=colorspace> (R package version 1.0–1)
- Jarek, S. (2009). mvnormtest: Normality test for multivariate variables [Software]. (R package version 0.1–1.7)
- Jurečková, J. and Picek, J. (2006). *Robust statistical methods with R*. Boca Raton, FL: Chapman & Hall/CRC. URL <http://www.fp.vslib.cz/kap/picek/robust/>
- Keele, L., Tingley, D., Yamamoto, T. and Imai, K. (2009). mediation: R Package for Causal Mediation Analysis [Software]. URL <http://CRAN.R-project.org/package=mediation> (R package version 2.1)
- Kirk, R. E. (1995). *Experimental Design – Procedures for the Social Sciences* (3. Aufl.). Pacific Grove, CA: Brooks/Cole.
- Komsta, L. and Novomestky, F. (2007). moments: Moments, cumulants, skewness, kurtosis and related tests [Software]. URL <http://www.komsta.net/> (R package version 0.11)
- Kuhn, M. and Weston, S. (2009). odfWeave: Sweave processing of Open Document Format (ODF) files [Software]. URL <http://CRAN.R-project.org/package=odfWeave> (R package version 0.7.10)
- Leisch, F. (2002). Sweave: Dynamic generation of statistical reports using literate data analysis. In W. Härdle & B. Rönz (Hrsg.), *Compstat 2002 — Proceedings in Computational Statistics* (S. 575–580). Heidelberg: Physica. URL <http://www.stat.uni-muenchen.de/~leisch/Sweave/>
- Lemon, J., Bolker, B. et al. (2010). plotrix: Various plotting functions [Software]. URL <http://CRAN.R-project.org/package=plotrix> (R package version 2.8.4)
- Ligges, U. (2002). R Help Desk: Automation of Mathematical Annotation in Plots. *R News*, 2 (3), 32–34. URL <http://CRAN.R-project.org/doc/Rnews/>
- Ligges, U. (2003). R Help Desk: Package Management. *R News*, 3 (3), 37–39. URL <http://CRAN.R-project.org/doc/Rnews/>
- Ligges, U. (2009). Programmieren mit R (3. Aufl.). Berlin: Springer. URL <http://www.statistik.tu-dortmund.de/~ligges/PmitR/>

- Ligges, U. and Mächler, M. (2003). Scatterplot3d – an R Package for Visualizing Multivariate Data. *Journal of Statistical Software*, 8 (11), 1–20. URL <http://www.jstatsoft.org/v08/i11/>
- Maindonald, J. and Braun, W. J. (2007). *Data Analysis and Graphics Using R: An Example-Based Approach* (2. Aufl.). Cambridge, UK: Cambridge University Press. URL <http://wwwmaths.anu.edu.au/~johnm/r-book.html>
- Maindonald, J. and Braun, W. J. (2009). DAAG: Data Analysis And Graphics [Software]. URL <http://CRAN.R-project.org/package=DAAG> (R package version 1.01)
- Marchetti, G. M. and Drton, M. (2010). ggm: Graphical Gaussian Models [Software]. URL <http://CRAN.R-project.org/package=ggm> (R package version 1.0.4)
- Mardia, K. V., Kent, J. T. and Bibby, J. M. (1980). *Multivariate Analysis*. London, UK: Academic Press.
- Maxwell, S. E. and Delaney, H. D. (2004). *Designing Experiments and Analyzing Data: A Model Comparison Perspective* (2. Aufl.). Mahwah, NJ: Lawrence Erlbaum.
- Meyer, D., Zeileis, A. and Hornik, K. (2010). vcd: Visualizing Categorical Data [Software]. URL <http://CRAN.R-project.org/package=vcd> (R package version 1.2–8)
- Miller, A. J. (2002). *Subset Selection in Regression* (2. Aufl.). Boca Raton, FL: Chapman & Hall/CRC.
- Muenchen, R. A. (2008). *R for SAS and SPSS Users*. New York, NY: Springer, URL <http://RforSASandSPSSusers.com/>
- Murdoch, D. and Chow, E. D. (2007). ellipse: Functions for drawing ellipses and ellipse-like confidence regions [Software]. URL <http://CRAN.R-project.org/package=ellipse> (R package version 0.3–5)
- Murrell, P. (2005). *R Graphics*. Boca Raton, FL: Chapman & Hall/CRC. URL <http://www.stat.auckland.ac.nz/~paul/RGraphics/rgraphics.html>
- Murrell, P. and Ihaka, R. (2000). An Approach to Providing Mathematical Annotations in Plots. *Journal of Computational and Graphical Statistics*, 9 (3), 582–599.
- Neuwirth, E., Heiberger, R. M., Ritter, C., Pieterse, J. and Volkering, J. (2010). RExcelInstaller: Integration of R and Excel (use R in Excel, read/write XLS files) [Software]. URL <http://CRAN.R-project.org/package=RExcelInstaller> (R package version 3.0–19)
- Nordhausen, K., Sirkia, S., Oja, H. and Tyler, D. E. (2010). ICSNP: Tools for Multivariate Nonparametrics [Software]. URL <http://CRAN.R-project.org/package=ICSNP> (R package version 1.0–7)
- Ogle, D. H. and Spangler, G. R. (2009). FSA: Fisheries Stock Analysis [Software]. URL <http://www.rforge.net/FSA/> (R package version 0.1–3)
- Pinheiro, J. C. and Bates, D. M. (2000). *Mixed-Effects Models in S and S-PLUS*. New York, NY: Springer.
- Pinheiro, J. C., Bates, D. M., DebRoy, S., Sarkar, D. and the R Core Team. (2009). nlme: Linear and Nonlinear Mixed Effects Models [Software]. URL <http://CRAN.R-project.org/package=nlme> (R package version 3.1–96)
- Raiche, G. and Magis, D. (2009). nFactors: Parallel Analysis and Non Graphical Solutions to the Cattell Scree Test [Software]. URL <http://CRAN.R-project.org/package=nFactors> (R package version 2.3.1)
- R Core Members, DebRoy, S., Bivand, R. et al. (2010). foreign: Read Data Stored by Minitab, S, SAS, SPSS, Stata, Systat, dBase, ... [Software]. URL <http://CRAN.R-project.org/package=foreign> (R package version 0.8–39)
- R Development Core Team. (2009a). CRAN Task Views. URL <http://CRAN.R-project.org/web/views/>
- R Development Core Team. (2009b). R: A Language and Environment for Statistical Computing [Software-Handbuch]. Vienna, Austria. URL <http://www.r-project.org/>
- R Development Core Team. (2009c). R: Data Import/Export [Software-Handbuch]. Vienna, Austria. URL <http://CRAN.R-project.org/doc/manuals/R-data.html>
- R Development Core Team. (2009d). R: Installation and Administration [Software-Handbuch]. Vienna, Austria. URL <http://CRAN.R-project.org/doc/manuals/R-admin.html>

- Revelle, W. (2009). psych: Procedures for Psychological, Psychometric, and Personality Research [Software]. URL <http://CRAN.R-project.org/package=psych> (R package version 1.0–85)
- Ripley, B. D. (2001). Using Databases with R. *R News*, 1 (1), 18–20. URL <http://CRAN.R-project.org/doc/Rnews/>
- Ripley, B. D. (2009). RODBC: ODBC Database Access [Software]. URL <http://CRAN.R-project.org/package=RODBC> (R package version 1.3–1)
- Ripley, B. D. and Hornik, K. (2001). Date-Time Classes. *R News*, 1 (2), 8–11. URL <http://CRAN.R-project.org/doc/Rnews/>
- Ripley, B. D. and Murdoch, D. (2009). *R for Windows FAQ*. URL <http://CRAN.R-project.org/bin/windows/base/rw-FAQ.html>
- Ritz, C. and Streibig, J. C. (2009). *Nonlinear Regression with R*. New York, NY: Springer.
- Sarkar, D. (2002). Lattice: An Implementation of Trellis Graphics in R. *R News*, 2 (2), 19–23. URL <http://CRAN.R-project.org/doc/Rnews/>
- Sarkar, D. (2008). *Lattice: Multivariate Data Visualization with R*. New York, NY: Springer. URL <http://lmdvr.r-forge.r-project.org/>
- Schlittgen, R. (2005). *Das Statistiklabor; Einführung und Benutzerhandbuch*. Berlin: Springer.
- Sing, T., Sander, O., Beerenwinkel, N. and Lengauer, T. (2009). ROCR: Visualizing the performance of scoring classifiers [Software]. URL <http://rocr.bioinf.mpi-sb.mpg.de/> (R package version 1.0–4)
- Strang, G. (2003). *Lineare Algebra*. Berlin: Springer.
- Suter, H.-P. (2010). xlsReadWrite: Natively read and write Excel files [Software]. URL <http://www.swissr.org/software/xlsReadWrite> (R package version 1.5.1)
- Temple Lang, D., Swayne, D., Wickham, H. and Lawrence, M. (2009). rggobi: Interface between R and GGobi [Software]. URL <http://CRAN.R-project.org/package=rggobi> (R package version 2.1.14)
- TIBCO Software Inc. (2008). Spotfire S+ [Software]. URL <http://www.insightful.com/> (Version 8.1)
- Urbanek, S. and Wichertey, T. (2009). ipplots: iPlots - interactive graphics for R [Software]. URL <http://www.ipplots.org/> (R package version 1.1–3)
- van Buuren, S. and Groothuis-Oudshoorn, K. (2010). MICE: Multivariate Imputation by Chained Equations in R. *Journal of Statistical Software*, forthcoming.
- Venables, W. N. and Ripley, B. D. (2002). *Modern Applied Statistics with S* (4. Aufl.). New York, NY: Springer. URL <http://www.stats.ox.ac.uk/pub/MASS4/>
- Venables, W. N., Smith, D. M. and the R Development Core Team. (2009). An introduction to R [Software-Handbuch]. Vienna, Austria. URL <http://CRAN.R-project.org/doc/manuals/R-intro.html>
- Verzani, J. (2005). *Using R for Introductory Statistics*. Boca Raton, FL: Chapman & Hall/CRC. URL <http://wiener.math.csi.cuny.edu/UsingR/>
- Wahlbrink, S. (2009). eclipse Plug-In for R: StatET [Software]. URL <http://www.walware.de/goto/statet> (Version 0.8.1)
- Warnes, G. R. (2009). gplots: Various R programming tools for plotting data [Software]. URL <http://CRAN.R-project.org/package=gplots> (R package version 2.7.4)
- Wickham, H. A. (2007). Reshaping Data with the Reshape Package. *Journal of Statistical Software*, 21 (12), 1–20. URL <http://www.jstatsoft.org/v21/i12/>
- Wickham, H. A. (2009). *ggplot2: Elegant Graphics for Data Analysis*. New York, NY: Springer. URL <http://had.co.nz/ggplot2>
- Wickham, H. A. (2010). ggplot2: An implementation of the Grammar of Graphics [Software]. URL <http://CRAN.R-project.org/package=ggplot2> (R package version 0.8.7)
- Williams, G. (2010). rattle: A graphical user interface for data mining in R using GTK [Software]. URL <http://CRAN.R-project.org/package=rattle> (R package version 2.5.24)
- Würtz, D. and Chalabi, Y. (2009). timeDate: Rmetrics – Chronological and Calendrical Objects [Software]. URL <http://www.rmetrics.org/> (R package version 2110.87)
- Zeileis, A. (2005). CRAN Task Views. *R News*, 5 (1), 39–40. URL <http://CRAN.R-project.org/doc/Rnews/>

Sachverzeichnis

!, 22
!=, 22
", 21
", 108
\$, 119, 124
&, 22, 35
&&, 22, 36
' , 21, 108
(), 19, 20, 402
*, 12, 164
+, 9, 12, 164
-, 12, 164
->, 19
. . . 20, 164
. . . 402
.First(), 10
.GlobalEnv, 18
.Last(), 10
.Last.value, 20, 154
.RData, 11
.Rhistory, 11
.Rprofile, 10
/, 12, 164
:, 44, 164
;, 6
<, 22
<-, 19
<=, 22
=, 19
==, 22
>, 6, 22
>=, 22
? , 14
[[]], 117, 124
[] , 26, 70, 91, 125
#, 6, 19, 153
% , 108
%*% , 79
%/%, 12
%%, 12
%in%, 42, 132, 164
%o%, 57
\, 109

- Anova(), 266, 273, 284, 288, 290–291, 294
 anova(), 237, 252, 267, 285, 289, 293, 326,
 328
 anova.mlm(), 326
 anpassen, *siehe* Optionen
 Ansari-Bradley-Test, 240
 ansari.test(), 240
 any(), 36
 Anzahl, *siehe* Vektor
 aov(), 249, 269, 280, 286, 290–291
 aperm(), 91
 append(), 28
 apply(), 73, 143
 approx(), 381
 apropos(), 15
 Arbeitsverzeichnis, *siehe* Verzeichnis
 args(), 13
 Arithmetik, 11
 Array, 90
 array, 90
 array(), 90
 arrows(), 358
 as.(Datentyp)(), 21
 as.(Klasse)(), 17
 as.data.frame(), 124
 as.Date(), 114
 as.list(), 124
 as.matrix(), 124
 as.POSIXct(), 115
 as.POSIXlt(), 115
 asin(), 12
 assocstats(), 191
 Assoziativität, 12
 atan(), 12
 atan2(), 12
 attach(), 126
 attr(), 18
 Attribut, 17
 äußeres Produkt, 57
 attributes(), 17
 Ausführungsreihenfolge, *siehe* Assoziativität
 Auswahl, *siehe* Daten
 ave(), 65
 axis(), 365
 axis.break(), 366
- B**
 Balkendiagramm, *siehe* Graphik
 barchart(), 390
 barp(), 348
 barplot(), 348
 Bartlett-Test, 241
 bartlett.test(), 241
- Batch-Modus, 152
 beenden, 9
 Befehlshistorie, 10
 Befehlskript, *siehe* Skript
 Betrag, 12
 binom, 173
 binom.test(), 172
 Binomialkoeffizient, 43
 Binomialtest, 172, 199
 Binomialverteilung, *siehe* Verteilung
 Biostatistik, *siehe* nonparametrische Methoden
 boot, 171
 Bootstrap-Verfahren, 171
 Bowker-Test, 211
 box(), 358
 Boxplot, *siehe* Graphik
 boxplot(), 373
 break, 412
 bwplot(), 390
 by(), 146
- C**
 c(), 25
 cancor(), 56
 car, 39, 233, 241, 266, 284, 288, 290–291,
 294, 339
 cat(), 109, 403
 cbind(), 72, 128, 140
 cdplot(), 351
 ceiling(), 12
 character, 21
 χ^2 -Test
 Auftrittswahrscheinlichkeiten, 184
 feste Verteilung, 179
 Gleichheit von Verteilungen, 183
 Normalverteilung, 180, 181
 Pearson- χ^2 -Test, 181
 Unabhängigkeit, 182
 Verteilungsklasse, 180
 χ^2 -Verteilung, *siehe* Verteilung
 chisq.test(), 179, 182
 chol(), 89
 Cholesky-Zerlegung, *siehe* Matrix
 choose(), 43
 chull(), 380
 class(), 17
 close.screen(), 399
 Clusteranalyse, 309
 cm.colors(), 348
 cmdscale(), 323
 cmh_test(), 192
 cnvrt.coords(), 335
 Cochran-Armitage-Trend-Test, 185

- Cochran-Mantel-Haenszel-Test, 192
Cochran-*Q*-Test, 210
`coefficients()`, 220
Cohens κ , 194
`coin`, 171, 185, 192, 199, 210, 214
`col()`, 69
`col2rgb()`, 347
`colMeans()`, 73
`colnames()`, 127
`colors()`, 346
`colorspace`, 347
`colSums()`, 73
`combinat`, 42
`combn()`, 43
`comment()`, 18
`complete.cases()`, 148
`complex`, 21
`confint()`, 222
`conflicts()`, 19
Connection, 156
`contour()`, 384
`contr.sum()`, 248
`contr.treatment()`, 247
`contrasts()`, 247
`coplot()`, 388
`cor()`, 56, 75, 189
`cor.test()`, 190, 217
`cos()`, 12
`cov()`, 55, 75, 189
`cov.wt()`, 75
Cramérs V , 191
CRAN, 15
`crossprod()`, 79
`cumprod()`, 50
`cumsum()`, 50
`curve()`, 362
`cut()`, 64
`cut2()`, 64
`cv.lm()`, 228
- D**
- DAAG, 122, 228
`data()`, 16
`data.frame`, 121
`data.frame()`, 122, 140
`data.matrix()`, 124
Date, 114
Daten
 auswählen, 48
 diskretisieren, 64
 doppelte Werte, 40, 147
 eingeben, 153
 ersetzen, 38
fehlende Werte, 12, 100, 148, 169
 ausschließen, 103
 codieren, 101
 ersetzen, 101
 fallweiser Ausschluss, 104
 identifizieren, 101
 Indexvektor, 38
 multiple Imputation, 101
 paarweiser Ausschluss, 105
 sortieren, 106
 umcodieren, 102
 zählen, 102
getrennt nach Gruppen auswerten, 64
Import / Export, 153
 andere Programme, 157
 Datenbank, 160
 Editor, 154
 Konsole, 153
 Online, 156
 R-Format, 157
 Textformat, 155–157
 Zwischenablage, 155–156
interpolieren, *siehe* Graphik
Kennwerte, *siehe* deskriptive Kennwerte
Qualität, v, 100, 147, 369
recodieren, 38
skalieren, 34
speichern, 153
wiederholen, 46
zentrieren, *siehe* Matrix
 z -Transformation, 33
Zufallsauswahl, 48
Datenbank, *siehe* Daten
Datensatz, 121
 Aufbau, 122, 133, 134
 Dimensionierung, 123
 Dokumentation, 16
 doppelte Werte, 147
 erzeugen, 122
 fehlende Werte, 148
 Funktionen anwenden, 143, 145–146
 indizieren, 124
 Konvertierung, 124
 Long-Format, 134, 135, 262, 266, 280, 286
 Pakete, 16, 122
 sortieren, 149
 Suchpfad, 126
 teilen, 138
 Teilmengen auswählen, 130
 umformen, 133, 134
 Variablen hinzufügen, 128
 Variablen löschen, 129
 Variablennamen, 127

- verbinden, 139
 Verlust einer Dimension, 125
 Wide-Format, 134, 135, 262, 266, 284
Datentyp, 17, 20
 Hierarchie, 21
 mischen, 28, 67, 117, 121
 umwandeln, 21
`dbinom()`, 166
`dchisq()`, 166
`demo()`, 335, 388
`density()`, 371
`deparse()`, 113
 Designmatrix, 163, 220, 247
 deskriptive Kennwerte, 49
 getrennt nach Gruppen berechnen, 64
`det()`, 86
`detach()`, 16, 127
 Determinante, *siehe* Matrix
 Determinationskoeffizient, *siehe* Regression
`dev.copy()`, 338
`dev.cur()`, 337
`dev.list()`, 336
`dev.next()`, 337
`dev.off()`, 337
`dev.prev()`, 337
`dev.set()`, 337
 Deviance, *siehe* Regression
 Dezimalstellen, 11, *siehe* runden
 Dezimalteil, 12
 Dezimaltrennzeichen, 11, 158, 159
`df()`, 166
`diag()`, 79
 diagonalisieren, *siehe* Matrix
 Diagramm, *siehe* Graphik
`diff()`, 50
 Differenzen, 50
`difftime`, 114–115
`difftime()`, 114
`dim()`, 68, 123
 Dimensionierung, *siehe* Matrix
`dimnames()`, 127
`dir()`, 152
 Diskriminanzanalyse, 309
`dist()`, 82
 Distanzmaß, *siehe* Abstand
 Division, 12
`dmvnorm()`, 385
`dnorm()`, 166
`do.call()`, 144
 Dokumentation, *siehe* Literatur
 doppelte Werte, *siehe* Daten
`dotchart()`, 352
`dotchart2()`, 352
`dotplot()`, 390
`double`, 21
 Download, 4
`drop`, 61, 71, 125
`drop1()`, 231, 273
`dt()`, 166
 Dummy-Codierung, 246, 273, 296
`dump()`, 157
`duplicated()`, 40, 147
- E**
`e`, 12
`ecdf()`, 98, 376
`edit()`, 154
 Editor, 3, 151
 Effekt-Codierung, *siehe* Dummy-Codierung
 Effizienz, 2, 28, 67, 70, 78, 122, 126, 409
`eigen()`, 87
 Eigenwerte und -vektoren, *siehe* Matrix
 Eingabe, *siehe* Daten
 Einstellungen, 9
`ellipse`, 366, 379
`else`, 407
 ENTWEDER-ODER, 22
 Environment, *siehe* Umgebung
`erase.screen()`, 398
`errbar()`, 368
`Error()`, 262, 280, 286
 Escape-Sequenz, 21, 109
`eval()`, 113
`example()`, 15
 Excel, 157–158, 160
`exists()`, 19
`exp()`, 12
`expand.grid()`, 44, 63
 Exponentialfunktion, 12
`expression()`, 364
 Extremwerte, 51
- F**
`F`, *siehe* FALSE
`F`-Verteilung, *siehe* Verteilung
`F`-Wert, *siehe* 2×2 Kontingenztafeln
`factanal()`, 318
`factor`, 57
`factor()`, 58, 61
`factorial()`, 12, 50
 Faktor, 57
 Faktorenanalyse, 317
 Fakultät, 12, 50
 Fallunterscheidung, 407
`FALSCH`, 22
`FALSE`, 22
 FAQ, 4

- Farben, *siehe* Graphik
fehlende Werte, *siehe* Daten
Fehlerbalken, *siehe* Graphik
`filled.contour()`, 386
`fisher.test()`, 185, 187
Fishers exakter Test
 Gleichheit von Verteilungen, 186
 Unabhängigkeit, 185
Fishers z -Transformation, 218
`fisherz()`, 218
`fisherz2r()`, 218
`fitted()`, 220, 236
`fix()`, 154
Fleiss' κ , 194
Fligner-Killeen-Test, 241
`fligner.test()`, 241
`floor()`, 12
`for()`, 410
`foreign`, 159
Format-String, 108
`formatC()`, 107
Formel, 163
 multivariat, 310
`formula`, 163
Friedman-Test, 208
`friedman.test()`, 209
`FSA`, 211
`ftable()`, 96
`function`, 401
`function()`, 401
Funktion
 überladen, 405
 anonym, 404
 Argumente, 13–14, 402
 aufrufen, 13, 404
 erstellen, 401
 Funktionsgraph, *siehe* Graphik
 generisch, 405
 getrennt nach Gruppen anwenden, 64
 Kopf, 402
 Methode, 405
 polymorph, 405
 Quelltext, 401
 Rückgabewert, 403
 Rumpf, 403
 unbenannt, 404
Funktionsschreibweise, *siehe* Präfix-Form
- G**
`gamma()`, 50
 Γ -Funktion, 50
gemischte Modelle, 265
Genauigkeit, *siehe* Zahlen
- generische Funktion, *siehe* Funktion
`geometric.mean()`, 52
geordnete Paare, 57
Geschwindigkeit, *siehe* Effizienz
`getwd()`, 9
`ggm`, 56
`ggplot2`, 335, 390
`ginv()`, 81
`gl()`, 63
Gleichverteilung, *siehe* Verteilung
`glht()`, 254, 256, 276, 297
`glm()`, 234
`glob2rx()`, 112
Globbing-Muster, *siehe* Zeichenketten
Goodman und Kruskals γ , 192
`GPArotation`, 318
`gplots`, 368
`graphics.off()`, 337
Graphik
 Achsen, 348, 365
 Ausgabe, 335
 Balkendiagramm, 348
 Basissystem, 335
 Beschriftung, 362, 364–365
 α -Blending, 347
 Boxplot, 372
 Cleveland Dotchart, 352
 Clipping, 335, 344
 3D, 383
 Datenpunkte identifizieren, 342
 Datenpunktssymbole, 344
 Device, 335
 unterteilen, 393
 verwalten, 336–337
 3D Gitter-Diagramm, 387
 3D Streudiagramm, 387
 Einstellungen, 343
 Elemente hinzufügen, 354
 Farben, 346
 Fehlerbalken, 366, 368
 Fenster öffnen, 336
 Formate, 338
 formatieren, 343
 Funktionsgraph, 362
 gemeinsame Verteilung, 378
 Gerade, 355
 Gitter, 357
 Glättter, 381
 grid-System, 390
 Höhenlinien, 384, 386
 High-Level-Funktion, 335, 336, 354, 396
 Histogramm, 369
 interaktiv, 335, 388

- interpolieren, 381–382
- konvexe Hülle, 380
- Koordinaten identifizieren, 354
- Koordinatensysteme, 335
- kopieren, 338
- Kreisdiagramm, 377
- kumulierte Häufigkeiten, 376
- Legende, 363
- Linien, 355
- Liniendiagramm, 339, 342
- Liniensegmente, 358
- Linientypen, 344
- Low-Level-Funktion, 335, 354
- mathematische Formeln, 364
- mehrere gleichartige Diagramme, 388
- mehrere Graphiken in einem Device, 393
- multivariate Daten, 383
- Optionen, 343
- Pfeil, 358
- Polygon, 360
- Punktdiagramm, 352
- Punkte, 355
- Quantil-Quantil-Diagramm, 375
- Ränder, 335
- Rahmen, 358
- Rechtecke, 358
- Regionen, 335
- Säulendiagramm, 348, 350
- Scree-Plot, 316
- Seitenverhältnis, 357
- speichern, 337
- Splines, 382
- Stamm-Blatt-Diagramm, 371
- Streudiagramm, 339, 342
- Streudiagramm-Matrix, 390
- Streuungsellipse, 378
- Stripchart, 374
- Symbole, 364
- Text, 364
- Titel, 362
- Tortendiagramm, 377
- Transparenz, 347
- Trellis-Diagramm, 390
- graphische Benutzeroberfläche, 3
- `gray()`, 347
- `gray.colors()`, 348
- `grep()`, 111
- `grep1()`, 112
- `grid`, 390
- `grid()`, 357
- Groß- und Kleinschreibung, 18
- Grundrechenarten, 11
- Gruppierungsfaktor, *siehe* Faktor
- `gsub()`, 112, 160
- GUI, *siehe* graphische Benutzeroberfläche
- H**
- `harmonic.mean()`, 52
- Häufigkeiten
 - absolute, 91
 - bedingte relative, 95
 - Iterationen, 94
 - Kreuztabelle, 94
 - kumulierte relative, 98
 - natürliche Zahlen, 93
 - Prozentrang, 98
 - Randkennwerte, 97
 - relative, 92
- Hauptkomponentenanalyse, 311
- `hcl()`, 347
- `head()`, 123
- `heat.colors()`, 347
- `help()`, 14
- `help.search()`, 15
- `help.start()`, 14
- herunterladen, *siehe* Download
- `hexbin`, 341
- `hexbin()`, 341
- Hilfe, 14
- `hist()`, 369
- `hist3d()`, 388
- `histbackback()`, 369
- `histogram()`, 390
- Histogramm, *siehe* Graphik
- `history()`, 10
- `Hmisc`, 18, 64, 101, 159–160, 192, 217, 335, 352, 368–369
- Homepage, 4
- Homogenität, 264
- Hotelling-Lawley-Spur, 326, 328, 330
- Hotellings T^2
 - eine Stichprobe, 325
 - zwei Stichproben, 327
- `HotellingsT2()`, 325, 327
- `hsv()`, 347
- I**
- `i`, 13
- `I()`, 122, 165
- `ICC()`, 192
- `icc()`, 192
- `ICSNP`, 325, 327
- `identify()`, 342
- `if()`, 407
- `ifelse()`, 39
- Imputation, *siehe* Daten

- i**
 - independence_test(), 185
 - Indexvektor, *siehe* Vektor
 - indizieren, *siehe* Datensatz, Matrix, Vektor
 - Inf, 12
 - install.packages(), 15
 - Installation, 5, *siehe* Pakete
 - installed.packages(), 16
 - interaction.plot(), 270
 - integer, 21
 - Inter-Rater-Übereinstimmung, 192
 - Cohens κ , 194
 - Fleiss' κ , 194
 - Kendalls W , 197
 - Krippendorffs α , 198
 - prozentuale Übereinstimmung, 193
 - Interquartilabstand, 54
 - intersect(), 41
 - Intra-Klassen-Korrelation, 192
 - inverse.rle(), 94
 - invertieren, *siehe* Matrix
 - invisible(), 404
 - iplots, 335, 388
 - IQR(), 54
 - irr, 192
 - is.(Datentyp)(), 21
 - is.(Klasse)(), 17
 - is.(Speicherart)(), 21
 - is.element(), 41, 132
 - is.finite(), 13
 - is.infinite(), 13
 - is.na(), 13, 101
 - is.nan(), 13
 - is.null(), 13
 - isoMDS(), 325
 - isTRUE(), 22, 36
 - Iterationen, *siehe* Häufigkeiten
 - Iterationslängentest, *siehe* Runs Test
- J**
 - jitter(), 341, 370
 - jpeg(), 338
- K**
 - kappa(lm-Modell), 233
 - kappa2(), 194–196
 - kappam.fleiss(), 194
 - kendall(), 197
 - Kendalls τ , 189
 - Kendalls W , 197
 - Kern, *siehe* Matrix
 - Klammern, 12, 19
 - Klasse, 17
 - knots(), 99
- L**
 - label(), 18
 - Länge
 - Norm, 81
 - Vektor, 17
 - Zeichenkette, 26
 - lapply(), 143
 - latente Klassen, 317
- Kolmogorov-Smirnov-Test**
 - Anpassungstest, 176
 - Gleichheit von Verteilungen, 203
 - Lilliefors-Schranken, 178
 - Normalverteiltheit, 176
- Kombination**, *siehe* Kombinatorik
- Kombinatorik**, 42
- Kommentar**, 6, 20, 153, 407
- komplexe Zahlen**, *siehe* Zahlen
- Kondition κ** , *siehe* Matrix
- Konsole**, 6
- Konstanten**, 11
- Kontingenzkoeffizient**, 191
- Kontingenztafel**, *siehe* Häufigkeiten
- Kontrast**, 250, *siehe* Varianzanalyse
- Kontrollstrukturen**, 407
- konvexe Hülle**, *siehe* Graphik
- Korrelation**, 56
 - kanonische, 56
 - Kendalls τ , 189
 - Korrelationsmatrix, 75
 - multiple Korrelation, 223
 - Partialkorrelation, 56, 227
 - Rangkorrelation, 189
 - Spearmans ρ , 189
 - Test, 217
- Kovarianz**, 55
 - Kovarianzmatrix, 75
 - Rangkovarianz, 189
 - unkorrigierte, 55, 75
- Kovarianzanalyse**, 292
 - Einzelvergleiche (Kontraste), 297–299
 - graphische Darstellung, 293, 296
- Kreuztabelle**, *siehe* Häufigkeiten
- kripp.alpha()**, 198
- Krippendorffs α** , 198
- Kruskal-Wallis-H-Test**, 207
- kruskal.test()**, 207
- ks.test()**, 176, 203
- kumulierte relative Häufigkeiten**, *siehe* Häufigkeiten
- kumulierte Summe**, *siehe* Summe
- kumulierte Produkt**, *siehe* Produkt
- kurtosis()**, 55

- lattice**, 335, 389, 390
layout(), 393
layout.show(), 395
Lazy Evaluation, 403
lcm(), 395
leere Menge, 12
legend(), 363
length(), 17, 26, 117, 123
LETTERS, 29
letters, 29
levels(), 59, 60
Levene-Test, 241
levene.test(), 241
library(), 16
lillie.test(), 178
Lilliefors-Test, *siehe* Kolmogorov-Smirnov-Test
Lineare Algebra, *siehe* Matrix
lineare Gleichungssysteme lösen, 81
lineare Strukturgleichungsmodelle, 317–318
lineares Modell, 163, 246
lines(), 355
Link-Funktion, 234
list, 117
list(), 117
Liste, 117
Literatur, 4
lm(), 219, 229, 252, 273, 295, 309, 326, 328
lm.ridge(), 219
lme4, 265
load(), 157
loadhistory(), 11
locator(), 354
loess.smooth(), 381
log(), 12
log-Likelihood, 236
log10(), 12
log2(), 12
Logarithmus, 12
logical, 21
logical(), 25
logische Operatoren, *siehe* Vergleiche
logische Vergleiche, *siehe* Vergleiche
logische Werte, *siehe* Wahrheitswerte
logistische Regression, *siehe* Regression
Logit-Funktion, 234
logLik(), 236
Long-Format, *siehe* Datensatz
lower.tri(), 70
lowess(), 381
lqs(), 219
ls(), 18, 20
- M**
mad(), 55
mahalanobis(), 83
Mahalanobisdistanz, 83
Mahalanobistransformation, 82
Mailing-Liste, 4
Mann-Whitney-U-Test, 207
MANOVA, *siehe* Varianzanalyse
manova(), 328, 330, 332
mantelhaen.test(), 192
mapply(), 145
Maskierung, *siehe* Namen
MASS, 81, 81, 219, 325
match(), 111
MATLAB, 158
matlines(), 355
matplot(), 342
matpoints(), 355
Matrix
 Addition, 79
 Algebra, 79
 Cholesky-Zerlegung, 89
 Datentypen, 67
 Determinante, 86
 Diagonalelemente, 79
 diagonalisieren, 88
 Diagonalmatrix, 79
 Dimensionierung, 68
 Dreiecksmatrix, 70
 Eigenwerte und -vektoren, 87, 311
 Einheitsmatrix, 79
 Elemente ändern, 70, 72
 Elemente auswählen, 70, 71
 erstellen, 66, 72
 Funktionen anwenden, 73
 in Matrix umwandeln, 68
 indizieren, 70, 71
 Inverse, 81
 Kern, 81
 Kondition κ , 89, 233
 mit Kennwert verrechnen, 74
 Multiplikation, 79
 Norm, 81
 Pseudoinverse, 81
 Randkennwerte, 73
 Rang, 87
 Singulärwertzerlegung, 89, 316
 Skalarprodukt, 79
 sortieren, 76
 Spaltenindex, 69
 Spaltennorm, 81
 Spektralzerlegung, 88
 Spur, 86

- transponieren, 78
umwandeln, 68
verbinden, 72
Verlust einer Dimension, 71
Wurzel, 88
Zeilenindex, 69
zentrieren, 74
Matrix, 81
matrix, 66
matrix(), 66
Mauchly-Test, 264
mauchly.test(), 264
max(), 51
Maximum, 51
McNemar-Test, 213
mcnemar.test(), 213
mean(), 52
Median, 52
Median der absoluten Abweichungen vom Median, 55
median(), 52
median_test(), 199
mediation, 221
Mediationsanalyse, *siehe* Regression
Mengen, 40–41
merge(), 140
Methode, *siehe* Funktion
methods(), 406
Metrik, *siehe* Abstand
mh_test(), 214
mice, 101
min(), 51
Mindeststichprobengröße, *siehe* Stichprobengröße
Minimum, 51
Missing Values, *siehe* Daten
Mittelwert, 52
Mittelwertstabelle, 65
mittlere absolute Abweichung vom Median, 55
Modalwert, 53
mode(), 17, 20
model.matrix(), 163, 220
model.tables(), 250, 269
Modus, *siehe* Datentyp
moments, 55
mood.test(), 240
Mood-Test, 240
mosaicplot(), 351
mshapiro(), 178, 325
mtext(), 364
multcomp, 254, 256, 276, 297
Multidimensionale Skalierung, 323
multilevel, 192, 221
multiple Imputation, *siehe* Daten
multiple Korrelation, *siehe* Korrelation
Multiplikation, 12
multivariate Varianzanalyse, *siehe* Varianzanalyse
multivariate Verfahren, 309
multivariate z -Transformation, 82
mvnormtest, 178, 325
mvtnorm, 83, 326, 385
- N**
n2mfrow(), 396
NA, 12, 101
na.fail(), 169
na.omit(), 103, 105, 148, 169
Namen
 Konflikte, 18
 Maskierung, 18
 Objektnamen, 18–20
names(), 31, 120, 127
NaN, 12
nchar(), 26
ncol(), 68, 123
next, 412
nFactors, 321
nlevels(), 59
nlme, 265
nls(), 219
nonparametrische Methoden, 171
Norm, 81
norm(), 81
Normalverteilung, *siehe* Verteilung
nortest, 178, 181
Notation, 3
 Versuchsdesigns, 246
nrow(), 68, 123
NULL, 12
Null(), 81
numeric, 21
numeric(), 25
- O**
Objekt, 17
 lösen, 20
 Objektnamen, *siehe* Namen
 versteckt, 20
Objektorientierung, 405
ODBC, 160
odbcConnectExcel(), 160
odbcGetInfo(), 160
Odds, 234
Odds Ratio, 185, 235
ODER, 22, 35, 36
odfWeave, 3

`oneway.test()`, 248
 OpenOffice, 3, 157
 Operatoren, 11
 Optimierung, *siehe* Effizienz
 Optionen, 10
`options()`, 10, 11, 169, 222, 247–248
`order()`, 29, 76, 149
`ordered()`, 61
 Ordner, *siehe* Verzeichnis
 orthogonale Projektion, 85
`outer()`, 57

P

`pairs()`, 390
`pairwise.t.test()`, 258
 Pakete, 15
`palette()`, 346
`par()`, 343, 395
`parse()`, 113
 Partialkorrelation, *siehe* Korrelation
`paste()`, 108
`pbinom()`, 167
`pbirthday()`, 167
`pchisq()`, 167
`pcor()`, 56
`pdf()`, 338, 347
 Pearson- χ^2 -Test, *siehe* χ^2 -Test
`pearson.test()`, 181
`permn()`, 42
 Permutation, *siehe* Kombinatorik
`persp()`, 387
`persp3d()`, 388
`perturb`, 233
`pf()`, 167
 φ -Koeffizient, 191
 π , 12
`pi`, 12
`pie()`, 377
`pie3D()`, 377
 Pillai-Bartlett-Spur, 330
`plot()`, 98, 225, 261, 339
`plot.design()`, 251, 270
`plot.new()`, 354
`plot3d()`, 388
`plotCI()`, 368
`plotrix`, 348, 366, 368, 377
`pmatch()`, 111
`pmax()`, 51
`pmin()`, 51
`png()`, 347
`pnorm()`, 167
`points()`, 355
`polygon()`, 360

Polymorphie, *siehe* Funktion
 Positiver Vorhersagewert, *siehe* 2×2 Kontingenztafeln
`POSIXct`, 115
`POSIXlt`, 115
 Potenz, 12
 Power, 299
 t -Test, 301–303
 Binomialtest, 300
 einfaktorielle Varianzanalyse, 304
`power.anova.test()`, 304
`power.t.test()`, 303
 Prävalenz, *siehe* 2×2 Kontingenztafeln
 Präzision, *siehe* 2×2 Kontingenztafeln
 Präfix-Form, 13
`prcomp()`, 312
`predict()`, 225, 236
`princomp()`, 316
`print()`, 20, 403
`prod()`, 50
 Produkt, 50
 Produkt-Moment-Korrelation, *siehe* Korrelation
 programmieren, 401
 Projektion, *siehe* orthogonale Projektion
 Prompt, 6
`prop.table()`, 92
`prop.test()`, 184
`prop.trend.test()`, 185
 Protokoll, 6
 Prozentrang, *siehe* Häufigkeiten
 Pseudoinverse, *siehe* Matrix
`psych`, 52, 122, 192, 218
`pt()`, 167
`pwr`, 303

Q

`q()`, 9
 Q -Test, *siehe* Cochran- Q -Test
`qbinom()`, 168
`qchisq()`, 168
`qf()`, 168
`qnorm()`, 168
`qqline()`, 375
`qqnorm()`, 375
`qqplot()`, 375
`qr()`, 87
 QR -Zerlegung, 87
`qt()`, 168
 Quadratsummen-Typen, *siehe* Varianzanalyse
 Quadratwurzel, 12
 Quantil, 53

- `quantile()`, 53
`Quartil`, 53
`quartz()`, 336
- R**
`R.matlab`, 158
 R^2 , *siehe* Regression
Rückgabewert, *siehe* Funktion
`rWG`-Koeffizienten, 192
`rainbow()`, 347
`random`, 47
Rang, *siehe* Matrix, Vektor
Range, 51
`range()`, 51
Rangkorrelation, *siehe* Korrelation
`rank()`, 30
Rate der korrekten Klassifikation, *siehe* 2×2 Kontingenztafeln
`rbind()`, 72, 139
`rbinom()`, 48
`rchisq()`, 48
`rcorr()`, 217
`rcorr.cens()`, 192
`read.dta()`, 160
`read.spss()`, 159
`read.table()`, 155
`read.xls()`, 158
`read.xport()`, 160
Recall, *siehe* 2×2 Kontingenztafeln
recodieren, *siehe* Daten
`rect()`, 359
Recycling, *siehe* Vektor
Regression
 R^2 , 223
Determinationskoeffizient, 223
Deviance, 235
Diagnostik, 224
einfache lineare, 218
graphische Darstellung, 220, 230, 236
Kreuzvalidierung, 227
logistische, 233
Mediation, 221
Modell auswählen, 231
Modell verändern, 230
Modellvergleich, 252
Multikollinearität, 232
multiple lineare, 228
multivariate, 309
nichtlineare, 219
Regressionsanalyse, 221
Residuen, 220, 235
Ridge-Regression, 219
robuste, 219
- Sobel-Test, 221
standardisierte, 220
Stepwise-Verfahren, 233
Toleranz, 233
Varianzanalyse, *siehe* lineares Modell
Varianzinflationsfaktor, 233
Voraussetzungen, 224
Vorhersage, 225
- Regressionsanalyse, *siehe* Regression
regulärer Ausdruck, *siehe* Zeichenketten
Relevanz, *siehe* 2×2 Kontingenztafeln
`relevel()`, 62
`recode()`, 39
`reorder()`, 62
`rep()`, 46
`repeat`, 412
`replace()`, 39
`replicate()`, 144
`require()`, 16
`reshape`, 136
`reshape()`, 136
`residuals()`, 220
`return()`, 404
`rev()`, 29
`RExcelInstaller`, 158
`rf()`, 48
`rgb()`, 347
`rgb2hsv()`, 347
`rggobi`, 335, 388
`rgl`, 388
Ridge-Regression, *siehe* Regression
`rle()`, 94
`rlm()`, 219
`rm()`, 20
`rms`, 234
`rmvnorm()`, 83
`RNGkind()`, 47
`rnorm()`, 48
Robustheit, 410
ROC-Kurve, 188
`ROCR`, 188
`RODBC`, 160
`round()`, 12
`row()`, 69
`rowMeans()`, 73
`rownames()`, 128
`rowSums()`, 73
Roys Maximalwurzel, 330
`Rprofile.site`, 10
`rt()`, 48
`rug()`, 370

runden, 12
`runif()`, 48
 Runs Test, 174

S

`S`, 1
`S+`, 1
 S3-Paradigma, 405
 S4-Paradigma, 405
`sammon()`, 325
`sample()`, 47
`sapply()`, 143
 SAS, 160
`sasxport.get()`, 160
 Säulendiagramm, *siehe* Graphik
`save()`, 157
`save.image()`, 157
`savehistory()`, 11
`scale()`, 33
`scan()`, 153, 156
`scatterplot()`, 339
`scatterplot3d()`, 230, 387
`scatterplot3d()`, 230, 387
 Schiefe, 55
 Schleifen, 409
 Schlüsselwörter, 18
 Scoping Regeln, *siehe* Variable
 Scree-Plot, *siehe* Graphik
`screen()`, 398
`sd()`, 55
`search()`, 16, 18
`segments()`, 358
`sem`, 318
 Sensitivität, *siehe* 2 × 2 Kontingenztafeln
`seq()`, 45
`sequence()`, 45
`set.seed()`, 47
`setdiff()`, 41
`setequal()`, 41
`setRepositories()`, 15
`setwd()`, 10
 Shapiro-Wilk-Test, 178, 325
`shapiro.test()`, 178
`shell()`, 6, 152
`sign()`, 12
 Sign-Test, *siehe* Vorzeichen-Test
`sim.(Typ)()`, 122
 Simulation, 122, 410
`sin()`, 12
 Singulärwertzerlegung, *siehe* Matrix
`sink()`, 6
 Skalarprodukt, *siehe* Matrix
`skewness()`, 55

Skript, 151
`smooth()`, 381
`smooth.spline()`, 382
`sobel()`, 221
 Sobel-Test, *siehe* Regression
 Somers' *d*, 192
`somers2()`, 192
`sort()`, 29
 sortieren, *siehe* Datensatz, Matrix, Vektor
`source()`, 152, 157
 Spannweite, *siehe* Range
 Spearmans ρ , 189
 speichern, *siehe* Workspace
 Spektralzerlegung, *siehe* Matrix
 Spezifität, *siehe* 2 × 2 Kontingenztafeln
`spineplot()`, 351
`spline()`, 382
 Splines, *siehe* Graphik
`split()`, 138
`split.screen()`, 398
`sprintf()`, 108
 SPSS, 18, 158, 271
`spss.get()`, 159
`sqlFetch()`, 160
`sqlQuery()`, 160
`sqlTables()`, 160
`sqrt()`, 12
SSCP-Matrix, 79
`stack()`, 133
`stars()`, 386
 starten, 6
 Stata, 160
`stata.get()`, 160
`stem()`, 372
`step()`, 231
`stepfun()`, 99
 Stepwise-Verfahren, *siehe* Regression
 Stichprobengröße, 299
 t-Test, 303
 einfaktorielle Varianzanalyse, 304
`str()`, 18, 123
 Streudiagramm, *siehe* Graphik
 Streuung, 55
 Streuungsellipse, *siehe* Graphik
`stripchart()`, 374
`stripplot()`, 390
`strptime()`, 115
`strReverse()`, 110
`strsplit()`, 110
 Stuart-Maxwell-Test, 214
 Stufenfunktion, 98
`sub()`, 112
`subset()`, 131

- `substring()`, 109
`Subtraktion`, 12
`Suchpfad`, 18, *siehe* Datensatz
`sum()`, 50
`summary()`, 50, 59, 97, 123, 211, 221, 250, 255, 267, 314, 330
`Summe`, 50
`Summenscore`, 34
`sunflowerplot()`, 341, 386
`supsmu()`, 381
`svd()`, 89
`Sweave()`, 3
`sweep()`, 74
`switch()`, 408
`symbols()`, 386
`symmetry_test()`, 210, 214
`Sys.Date()`, 114
- T**
`T`, *siehe* TRUE
`t()`, 78
 t -Test
 eine Stichprobe, 242
 zwei abhängige Stichproben, 245
 zwei unabhängige Stichproben, 243
 t -Verteilung, *siehe* Verteilung
`t.test()`, 242, 243, 245
Tabellenkalkulation, 157
`table()`, 91
`tabulate()`, 93
`tail()`, 123
`tan()`, 12
`tapply()`, 65
Task Views, 15, 113, 309, 317
Tastatatkürzel, 9, 151
`termplot()`, 224
`terms()`, 165
`terrain.colors()`, 347
Test auf Zufälligkeit, *siehe* Runs Test
Text, *siehe* Graphik, Zeichenkette
`text()`, 364
`textConnection()`, 156
`timeDate`, 113
`title()`, 362
`tolower()`, 109
`topo.colors()`, 348
`toString()`, 107
`toupper()`, 109
tranchieren, *siehe* runden
`transform()`, 129
transponieren, *siehe* Matrix
Treatment-Kontrast, *siehe* Kontrast
trigonometrische Funktionen, 12
- `TRUE`, 22
`trunc()`, 12
`TukeyHSD()`, 260
`typeof()`, 21
- U**
Uhrzeit, 115
Umgebung, 18, 403
unbenannte Funktion, *siehe* Funktion
`unclass()`, 59
UND, 22, 35, 36
undefiniert, 12
unendlich, 12
`union()`, 41
`unique()`, 40, 147
`unlist()`, 121
`unstack()`, 134
`update()`, 230
`update.packages()`, 16
`upper.tri()`, 70
`UseMethod()`, 406
- V**
`var()`, 54, 75
`var.test()`, 239
Variable
 Gültigkeit, 403
 kategorial, *siehe* Faktor
 Label, 18
 latente, 317
 Lebensdauer, 403
 lokal, 403
 neue aus bestehenden bilden, 34
Varianz, 54
 unkorrigierte, 54, 75
Varianzanalyse
 assoziierte einfaktorielle, 275, 278
 balanciertes Design, 268
 Blockeffekt, 262, 280, 286
 dreifaktoriell
 abhängige Gruppen (RBF- pqr), 283
 Split-Plot (SPF- $p \cdot qr$), 290
 Split-Plot (SPF- $pq \cdot r$), 291
 unabhängige Gruppen (CRF- pqr), 270
 einfaktoriell, 246
 abhängige Gruppen (RB- p), 261, 266
 unabhängige Gruppen (CR- p), 248–249, 252
 fester Effekt, 262
 genestetes Design, 164, 263
Kontraste
 a-priori, 254, 275
 paarweise t -Tests, 258
 post-hoc (Scheffé), 257, 278

- simultane Konfidenzintervalle (Tukey), 259
- zweifaktoriell, 275, 278
- MANOVA, 329
- Modellvergleich, 252
- multivariat, 329
 - einfaktoriell, 329
 - mehr faktoriell, 332
- orthogonales Design, 268, 271
- Quadratsummen-Typen, 232, 271
- Random-Faktor, 262, 280, 286
- Regression, *siehe* lineares Modell
- Voraussetzungen, 251
- Zirkularität, 262, 264, 283, 287
 - ϵ -Korrektur, 264, 266, 268, 283, 288
 - Mauchly-Test, 264, 266
- zweifaktoriell
 - abhängige Gruppen (RBF- pq), 279
 - Split-Plot (SPF- $p \cdot q$), 286
 - unabhängige Gruppen (CRF- pq), 268
- Varianzhomogenität
 - F-Test, 239
 - Ansari-Bradley-Test, 240
 - Bartlett-Test, 241
 - Fligner-Killeen-Test, 241
 - Levene-Test, 241
 - Mood-Test, 240
- vcov, 191
- vcov(), 220
- Vektor, 17, 25
 - Anzahl Elemente, 26
 - Datentypen, 28
 - Elemente ändern, 28
 - Elemente auslassen, 27
 - Elemente auswählen, 26
 - Elemente benennen, 30
 - Elemente ersetzen, 42
 - Elemente löschen, 31
 - elementweise Rechnung, 31
 - erstellen, 25
 - Fälle zählen, 36
 - Indexvektor
 - logisch, 37
 - numerisch, 27, 38
 - indizieren, 26
 - Kriterien prüfen, 35
 - Länge, 26
 - Norm, 81
 - Rang, 30
 - Recycling, 32
 - Reihenfolge, 29
 - Skalarprodukt, *siehe* Matrix
 - sortieren, 29
- unbenannt, 27
- Vergleiche, 35
- verkürzen, 31
- verlängern, 28
- verrechnen, 31
- zammenfügen, 26
- zyklische Verlängerung, 32
- Verallgemeinerte Lineare Modelle, 234
- Vergleiche, 23
- Verteilung
 - F-Verteilung, 48, 165
 - χ^2 -Verteilung, 48, 165
 - t-Verteilung, 48, 165
 - Binomialverteilung, 48, 165
 - Dichtefunktion, 166
 - Dichtefunktion schätzen, 371
 - Gleichverteilung, 48, 165
 - Hypergeometrische Verteilung, 165
 - kritischer Wert, 168, 173
 - Nonzentralitätsparameter, 301, 304–305
 - Normalverteilung, 48, 165
 - χ^2 -Test, 180
 - Anderson-Darling-Test, 178
 - Kolmogorov-Smirnov-Test, 176
 - Lilliefors-Test, 178
 - Shapiro-Wilk-Test, 178
 - p-Wert, 167
 - Poisson-Verteilung, 165
 - Quantilfunktion, 168
 - Verteilungsfunktion, 167
 - Wahrscheinlichkeitsfunktion, 166
 - Wilcoxon-Vorzeichen-Rang-Verteilung, 165
- verteilungsfreie Methoden, *siehe* nonparametrische Methoden
- Verzeichnis
 - Arbeitsverzeichnis, 9
 - Pfadangabe, 151
- Verzweigung, 407
- vif(), 233
- vignette(), 16
- Vorzeichen, 12
- Vorzeichen-Test, 199
- W**
- WAHR, 22
- Wahrheitswerte, 21–22
- Wald-Wolfowitz-Test, 203
- weighted.mean(), 52
- Wettquotient, 234
- which(), 38
- which.max(), 51
- which.min(), 51

- `while()`, 411
Wide-Format, *siehe* Datensatz
`wilcox.test()`, 200, 205, 207
Wilcoxon-Test
 Rangsummen-Test, 205
 Vorzeichen-Rang-Test, 200
 zwei abhängige Stichproben, 207
Wilcoxon-Vorzeichen-Rang-Verteilung, *siehe*
 Verteilung
Wilks' Λ , 330
`windows()`, 336
Winkelfunktionen, *siehe* trigonometrische
 Funktionen
`with()`, 126
Wölbung, 55
Workspace, 10, 18
`write.foreign()`, 159–160
`write.table()`, 156
`write.xls()`, 158
Wurzel, *siehe* Quadratwurzel
- X**
`x11()`, 336
`xlsReadWrite`, 158
`xor()`, 22
`xspline()`, 383
`xtabs()`, 96
`xyplot()`, 390
- Z**
Zahlen
 Exponentialschreibweise, 11
 ganze, 21
 Genauigkeit, 23
 Gleitkommazahlen, 21, 23
 komplexe, 13, 21
 reelle, 21
 Zahlenfolgen erstellen, 44
- Zeichenketten, 21
 ausführen, 113
 Globbing-Muster, 112
 Groß- / Kleinbuchstaben, 109
 in Zeichenketten umwandeln, 107
 Länge, 26
 nach Muster erstellen, 107
 Platzhalter, 112
 regulärer Ausdruck, 111
 umkehren, 110
 Umwandlung in Faktor, 123, 155
 verbinden, 109
 Wildcards, 112
 Zeichenfolge ersetzen, 110, 112
 Zeichenfolge extrahieren, 109
 Zeichenfolge finden, 111
 Zeilenumbruch, 109
 zerlegen, 110
Zeit, *siehe* Uhrzeit
Zirkularität, *siehe* Varianzanalyse
zufällige Reihenfolge, 49
Zufallsauswahl, *siehe* Daten
Zufallseffekt, *siehe* Varianzanalyse
Zufallsvariablen, 165
Zufallszahlen, 46–47
Zusammenhangsmaße, 191
 φ -Koeffizient, 191
 r_{WG} -Koeffizienten, 192
 Cramér's V , 191
 Goodman und Kruskals γ , 192
 Intra-Klassen-Korrelation, 192
 Kontingenzkoeffizient, 191
 Somers' d , 192
Zusatzpakete, *siehe* Pakete
Zuweisung, 19
Zwischenablage, 155–156, 158
zyklische Verlängerung, *siehe* Vektor