

Grundlagen der Datenanalyse mit R

Eine anwendungsorientierte Einführung

Daniel Wollschläger

29. September 2024

Vorwort

Dieses Buch liefert eine an wissenschaftlichen Anwendungen orientierte Einführung in die Datenauswertung mit R. R ist eine freie Umgebung zur umfassenden statistischen Analyse und grafischen Darstellung von Datensätzen, die befehlsoorientiert arbeitet. Der vorliegende Text soll allen den Einstieg in R erleichtern, die in erster Linie grundlegende statistische Auswertungsverfahren anwenden möchten und keine Vorkenntnisse mit Programmen ohne grafische Benutzeroberfläche besitzen.

Das Buch stellt die Umsetzung grafischer und deskriptiver Datenauswertung, nonparametrischer Verfahren, (verallgemeinerter) linearer Modelle und multivariater Methoden vor. Die Auswahl der behandelten statistischen Verfahren orientiert sich an den Anforderungen der Psychologie, deckt aber auch die wichtigsten Auswertungsmethoden anderer Human- und Sozialwissenschaften sowie der klinischen Forschung ab.

Struktur und Lesereihenfolge

Das Buch besteht aus vier großen Teilen:

- I Kapitel 1–4 befassen sich mit den zum Einlesen und Verarbeiten von Daten notwendigen Grundlagen. Kapitel 1 dient der Einführung in den Umgang mit R sowie in die Syntax der Befehlssteuerung. In Kapitel 2 werden grundlegende Datenstrukturen gemeinsam mit Methoden zur deskriptiven Datenauswertung behandelt. Kapitel 3 erläutert den Import / Export und die Aufbereitung von Datensätzen. Arbeitstechniken für eine zuverlässige und reproduzierbare Datenauswertung bespricht Kapitel 4.
- II Kapitel 5–13 behandeln die Anwendung verschiedener statistischer Modelle und Methoden. Kapitel 5 stellt Hilfsmittel für die schließende Statistik bereit. Diese wird in Kapitel 6 (lineare Regression), 7 (*t*-Tests und Varianzanalysen), 8 (Regression für kategoriale Daten), 9 (Survival-Analyse), 10 (klassische nonparametrische Tests), 11 (bootstrap und Permutationstests), 12 (multivariate Methoden) und 13 (Vorhersagegüte prädiktiver Modelle) behandelt.
- III Kapitel 14–15 stellen vor, wie Diagramme erstellt werden können. Kapitel 14 erläutert die allgemeinen Grundlagen sowie die relevanten Funktionen des Basisumfangs von R. Kapitel 15 vermittelt den Umgang mit dem beliebten Zusatzpaket **ggplot2**.
- IV Kapitel 16–17 beschreiben fortgeschrittene Techniken, insbesondere für maßgeschneiderte Auswertungen. Dazu zählen numerische Methoden in Kapitel 16 sowie der Einsatz von R als Programmiersprache in Kapitel 17.

Die Lesereihenfolge muss der Reihenfolge der Kapitel nicht unbedingt folgen. Während Teil I universell für die meisten Anwendungen wichtig ist und daher zuerst gelesen werden sollte, lassen sich die Inhalte von Teil II und III auch unabhängig voneinander nach Bedarf kombinieren. Teil IV ist optional und besonders für erfahrene Nutzer gedacht.

Hinweise

Der Fokus des Buchs liegt auf der Umsetzung von Methoden zur Datenauswertung mit R, nicht aber auf der Vermittlung statistischer Grundlagen. Von diesen wird hier angenommen, dass die Leser mit ihnen vertraut sind. Auf Literatur zu den behandelten Verfahren wird jeweils hingewiesen.

Um die Ergebnisse der R-eigenen Auswertungsfunktionen besser nachvollziehbar zu machen, wird ihre Anwendung an vielen Stellen durch manuelle Kontrollrechnungen begleitet. Die eigene Umsetzung soll zudem zeigen, wie auch Testverfahren, für die zunächst keine vorbereiteten Funktionen vorhanden sind, prinzipiell selbst verwirklicht werden können.

Änderungen in der zweiten Auflage

Das Buch vertieft nun das Thema der Verarbeitung von Zeichenketten sowie von Datumsangaben (Abschn. 2.14, 2.15) und beinhaltet eine umfassendere Darstellung der Diagnostik und Kreuzvalidierung von Regressionsmodellen (Abschn. 6.5, 13.1). Die Auswertung varianzanalytischer Fragestellungen berücksichtigt jetzt die Schätzung von Effektstärken (Abschn. 7.3–7.8). Als Tests auf gleiche Variabilität werden zusätzlich jene nach Fligner-Killeen sowie nach Mood und Ansari-Bradley besprochen (7.2.3, 10.4). Das neue Kap. 11 führt in die Anwendung von bootstrapping und Permutationstests ein. Bei multivariaten Verfahren ist die Diskriminanzanalyse ebenso hinzugekommen wie eine Behandlung des allgemeinen linearen Modells (Abschn. 12.8, 12.9). Schließlich geht der Text nun auf Möglichkeiten zur Darstellung von Bitmap-Grafiken ein (Abschn. 14.5.10) und beschreibt detaillierter, welche Möglichkeiten für Funktionsanalyse und debugging R bietet (Abschn. 17.4).

Änderungen in der dritten Auflage

Abschnitt 3.1.3 behandelt ausführlicher den Datenaustausch mit Datenbanken. Der neue Abschn. 3.2 stellt vor, wie man mit Dateien und Pfaden arbeitet. Hinweise auf Erweiterungen der linearen Regression liefert Abschn. 6.6. Regressionsmodelle für kategoriale Daten und Zähldaten sind nun in Kap. 8 zusammengefasst. Das neue Kap. 9 führt in die Analyse von Ereigniszeiten mit Kaplan-Meier, Cox Regression und parametrischen Modellen ein. ROC-Kurven und AUC werden nun in Abschn. 10.2.7 beschrieben. Abschnitt 11.1.3 zeigt ein Beispiel für stratifiziertes bootstrapping, Abschn. 11.1.6 demonstriert den wild bootstrap für lineare Modelle. Der Abschnitt zur Kreuzvalidierung linearer Modelle wurde zu Kap. 13 erweiter. Wie man mit dem Paket `ggplot2` Diagramme erstellt, erläutert Kap. 15.

Änderungen in der vierten Auflage

Abschnitt 1.3 zur Arbeit mit Zusatzpaketen ist ausführlicher und klarer strukturiert. Abschnitte 2.14.3 und 2.14.4 beschreiben neu in R integrierte Funktionen zum Manipulieren und Suchen von Zeichenketten. Hinweise zur Prüfung der Datenqualität enthält der neue Abschn. 4.3. Informationen zu Dokumenten mit eingebetteten R-Auswertungen gibt Abschn. 4.2. Abschnitt 11.2 erweitert die Resampling-Verfahren um parametrisches bootstrapping. Der Abschnitt zum Erstellen von Diagrammen mit `ggplot2` wurde deutlich erweitert zum neuen Kap. 15. Das ebenfalls neue Kap. 16 gibt einen Überblick über allgemeine numerische Methoden.

Änderungen in der fünften Auflage

Das Hauptaugenmerk lag neben einer allgemeinen Aktualisierung und punktuellen Ergänzung darauf, *data science* orientierte Inhalte zu stärken. Dazu wurde Kap. 3 stark überarbeitet,

insbesondere für die Darstellung der Datenaufbereitung mit dem Zusatzpaket `dplyr` (Abschn. 3.4). Kapitel 4 fokussiert auf zuverlässige und reproduzierbare Datenauswertungen mit dem erweiterten Abschn. 4.2 zu R-Markdown Dokumenten und dem neuen Abschn. 4.4 mit Hinweisen für bessere Reproduzierbarkeit. Abschnitt 17.5.2 zeigt, wie sich umfangreiche Auswertungen beschleunigen lassen, indem man sie auf mehrere Prozessoren verteilt und parallel bearbeitet.

Änderungen in der sechsten Auflage

In der vorliegenden sechsten Auflage bezieht sich das Buch auf Version 4.4.1 von R.

- Neben vielen Detailänderungen wurde die Auswahl verwendeter Zusatzpakete angepasst, neuere Literatur berücksichtigt, außerdem wurden Online-Quellen aktualisiert.
- 2.5.6 Werte ersetzen mit `case_match()` sowie `replace()`
- Berücksichtigung zahlreicher Veränderungen in den Paketen `dplyr` sowie `ggplot2`.
- Ergänzungen im Kapitel `ggplot2`
- Pipe-Syntax `|>` im Basisumfang von R
- Quarto
- Ergänzungen zum Einlesen aus Excel-Tabellen
- Ergänzungen zur replizierbaren Datenauswertungen
- Landschaft der IDEs
- Abschnitt Tests auf Normalverteilung
- PCA Darstellung Biplot
- Basisumfang von R: Erweiterung der Syntax vieler Funktionen um Möglichkeit Modellformel anzugeben
- Einheitlich `emmeans` für moderierte Regression, simple effects, korrigierte Mittelwerte in ANCOVA

Korrekturen, Ergänzungen und Anregungen sind herzlich willkommen. Die verwendeten Daten sowie alle Befehle des Buches und ggf. notwendige Berichtigungen erhalten Sie online:

<http://www.dwoll.de/r/>

Danksagung

Mein Dank gilt den Personen, die an der Entstehung des Buches in frühen und späteren Phasen mitgewirkt haben: Abschnitte 1.1 bis 1.2.3 entstanden auf der Grundlage eines Manuskripts von Dieter Heyer und Gisela Müller-Plath, denen ich für die Überlassung des Textes danken möchte. Zahlreiche Korrekturen und viele Verbesserungsvorschläge wurden von Federico Marini, Philipp Mildenberger, Julian Etzel, Andri Signorell, Ulrike Groemping, Wolfgang Ramos, Erwin Grüner, Johannes Andres, Sabrina Flindt und Susanne 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,

Martha und Nike Jores sowie von Vincent van Houten begleitet. Iris Ruhmann, Clemens Heine, Niels Peter Thomas und Alice Blanck vom Springer Verlag möchte ich herzlich für die freundliche Kooperation und Betreuung der Veröffentlichung danken.

Zuvorderst ist den Entwicklern von R, dem CRAN-Team sowie den Autoren der zahlreichen Zusatzpakete besonderer Dank und Anerkennung dafür zu zollen, dass sie in freiwillig geleisteter Arbeit eine hervorragende Umgebung zur statistischen Datenauswertung geschaffen haben, deren mächtige Funktionalität hier nur zu einem Bruchteil vorgestellt werden kann.

Mainz,
März 2020

Daniel Wollschläger
contact@dwoll.de

Inhaltsverzeichnis

1	Erste Schritte	1
1.1	Vorstellung	1
1.1.1	Pro und Contra R	1
1.1.2	Typografische Konventionen	3
1.1.3	R installieren	4
1.1.4	Grafische Benutzeroberflächen	5
1.1.5	Weiterführende Informationsquellen und Literatur	5
1.2	Grundlegende Elemente	7
1.2.1	R Starten, beenden und die Konsole verwenden	7
1.2.2	Einstellungen	11
1.2.3	Umgang mit dem workspace	12
1.2.4	Einfache Arithmetik	13
1.2.5	Funktionen mit Argumenten aufrufen	16
1.2.6	Hilfe-Funktionen	16
1.2.7	Empfehlungen und typische Fehlerquellen	17
1.3	Zusatzpakete verwenden	18
1.3.1	Zusatzpakete installieren: Grundlagen	18
1.3.2	Zusatzpakete installieren: Erweiterte Optionen	19
1.3.3	Zusatzpakete laden	20
1.3.4	Hinweise zum Arbeiten mit Zusatzpaketen	21
1.4	Datenstrukturen: Klassen, Objekte, Datentypen	22
1.4.1	Objekte benennen	23
1.4.2	Zuweisungen an Objekte	24
1.4.3	Objekte ausgeben	25
1.4.4	Objekte anzeigen lassen, umbenennen und entfernen	25
1.4.5	Datentypen	26
1.4.6	Logische Werte, Operatoren und Verknüpfungen	28
2	Elementare Dateneingabe und -verarbeitung	31
2.1	Vektoren	31
2.1.1	Vektoren erzeugen	31
2.1.2	Elemente auswählen und verändern	32
2.1.3	Datentypen in Vektoren	35
2.1.4	Elemente benennen	35
2.1.5	Elemente löschen	36
2.2	Logische Operatoren	37
2.2.1	Vektoren mit logischen Operatoren vergleichen	37
2.2.2	Logische Indexvektoren	39
2.3	Mengen	41
2.3.1	Doppelt auftretende Werte finden	41

Inhaltsverzeichnis

2.3.2	Mengenoperationen	42
2.3.3	Kombinatorik	43
2.4	Systematische und zufällige Wertefolgen erzeugen	46
2.4.1	Numerische Sequenzen erstellen	46
2.4.2	Wertefolgen wiederholen	48
2.4.3	Zufällig aus einer Urne ziehen	48
2.4.4	Zufallszahlen aus bestimmten Verteilungen erzeugen	49
2.5	Daten transformieren	50
2.5.1	Werte sortieren	50
2.5.2	Werte in zufällige Reihenfolge bringen	51
2.5.3	Teilmengen von Daten auswählen	52
2.5.4	Daten umrechnen	53
2.5.5	Neue aus bestehenden Variablen bilden	56
2.5.6	Werte ersetzen oder recodieren	56
2.5.7	Kontinuierliche Variablen in Kategorien einteilen	59
2.6	Gruppierungsfaktoren	59
2.6.1	Ungeordnete Faktoren	60
2.6.2	Faktoren kombinieren	61
2.6.3	Faktorstufen nachträglich ändern	62
2.6.4	Geordnete Faktoren	65
2.6.5	Reihenfolge von Faktorstufen kontrollieren	66
2.6.6	Faktoren nach Muster erstellen	68
2.6.7	Quantitative in kategoriale Variablen umwandeln	68
2.7	Deskriptive Kennwerte numerischer Daten	70
2.7.1	Summen, Differenzen und Produkte	70
2.7.2	Extremwerte	71
2.7.3	Mittelwert, Median und Modalwert	72
2.7.4	Robuste Maße der zentralen Tendenz	74
2.7.5	Prozentrang, Quartile und Quantile	75
2.7.6	Varianz, Streuung, Schiefe und Wölbung	76
2.7.7	Diversität kategorialer Daten	77
2.7.8	Kovarianz und Korrelation	78
2.7.9	Robuste Streuungsmaße und Kovarianzschatzer	79
2.7.10	Kennwerte getrennt nach Gruppen berechnen	80
2.7.11	Funktionen auf geordnete Paare von Werten anwenden	82
2.8	Matrizen	83
2.8.1	Datentypen in Matrizen	83
2.8.2	Dimensionierung, Zeilen und Spalten	84
2.8.3	Elemente auswählen und verändern	86
2.8.4	Weitere Wege, Elemente auszuwählen und zu verändern	88
2.8.5	Matrizen verbinden	89
2.8.6	Matrizen sortieren	90
2.8.7	Randkennwerte berechnen	91
2.8.8	Beliebige Funktionen auf Matrizen anwenden	92
2.8.9	Matrix zeilen- oder spaltenweise mit Kennwerten verrechnen	92
2.8.10	Kovarianz- und Korrelationsmatrizen	93
2.9	Arrays	95

Inhaltsverzeichnis

2.10 Listen	97
2.10.1 Komponenten auswählen und verändern	98
2.10.2 Komponenten hinzufügen und entfernen	101
2.10.3 Listen mit mehreren Ebenen	101
2.11 Datensätze	103
2.11.1 Datentypen in Datensätzen	106
2.11.2 Elemente auswählen und verändern	106
2.11.3 Namen von Variablen und Beobachtungen	108
2.11.4 Datensätze in den Suchpfad einfügen	108
2.12 Häufigkeiten bestimmen	110
2.12.1 Einfache Tabellen absoluter und relativer Häufigkeiten	110
2.12.2 Iterationen zählen	112
2.12.3 Absolute und (bedingte) relative relative Häufigkeiten in Kreuztabellen	112
2.12.4 Randkennwerte von Kreuztabellen	115
2.12.5 Datensätze aus Häufigkeitstabellen erstellen	115
2.12.6 Kumulierte relative Häufigkeiten und Prozentrang	116
2.13 Fehlende Werte behandeln	117
2.13.1 Fehlende Werte codieren und identifizieren	117
2.13.2 Fehlende Werte ersetzen und umcodieren	119
2.13.3 Behandlung fehlender Werte bei der Berechnung einfacher Kennwerte	120
2.13.4 Behandlung fehlender Werte in Matrizen	120
2.13.5 Behandlung fehlender Werte beim Sortieren von Daten	122
2.13.6 Behandlung fehlender Werte in inferenzstatistischen Tests	123
2.13.7 Multiple Imputation	123
2.14 Zeichenketten verarbeiten	123
2.14.1 Objekte in Zeichenketten umwandeln	124
2.14.2 Zeichenketten erstellen und ausgeben	124
2.14.3 Zeichenketten manipulieren	127
2.14.4 Zeichenfolgen finden	129
2.14.5 Zeichenfolgen extrahieren	130
2.14.6 Zeichenfolgen ersetzen	132
2.14.7 Zeichenketten als Befehl ausführen	133
2.15 Datum und Uhrzeit	134
2.15.1 Datumsangaben erstellen und formatieren	134
2.15.2 Uhrzeit	135
2.15.3 Mit Datum und Uhrzeit rechnen	137
3 Daten importieren, exportieren, aufbereiten und aggregieren	139
3.1 Daten importieren und exportieren	139
3.1.1 Datentabellen im Textformat	139
3.1.2 R-Objekte	142
3.1.3 Daten mit anderen Programmen austauschen	143
3.1.4 Daten in der Konsole einlesen	150
3.1.5 Unstrukturierte Textdateien nutzen	151
3.2 Dateien verwalten	152
3.2.1 Dateien auswählen	152
3.2.2 Dateipfade manipulieren	154

Inhaltsverzeichnis

3.2.3 Dateien verändern	154
3.3 Datensätze aufbereiten und aggregieren	155
3.3.1 Verkettung von Befehlen mit der pipe	156
3.3.2 Variablen umbenennen	157
3.3.3 Teilmengen von Daten auswählen	158
3.3.4 Doppelte und fehlende Werte ausschließen	161
3.3.5 Variablen entfernen, hinzufügen und transformieren	163
3.3.6 Datensätze sortieren	164
3.3.7 Datensätze aufteilen	165
3.3.8 Datensätze zeilen- oder spaltenweise verbinden	166
3.3.9 Datensätze zusammenführen	167
3.3.10 Organisationsform einfacher Datensätze ändern	170
3.3.11 Organisationsform komplexer Datensätze ändern	172
3.3.12 Daten getrennt nach Gruppen auswerten und aggregieren	176
3.3.13 Funktionen auf Variablen anwenden	179
3.3.14 Funktionen für mehrere Variablen anwenden	181
3.4 Datensätze aufbereiten und aggregieren mit dplyr	182
3.4.1 Besonderheiten	183
3.4.2 Variablen umbenennen	184
3.4.3 Teilmengen von Daten auswählen	184
3.4.4 Doppelte und fehlende Werte ausschließen	187
3.4.5 Variablen entfernen, hinzufügen und transformieren	188
3.4.6 Datensätze sortieren	189
3.4.7 Datensätze zeilen- oder spaltenweise verbinden	190
3.4.8 Datensätze zusammenführen	191
3.4.9 Organisationsform komplexer Datensätze ändern	194
3.4.10 Datensätze getrennt nach Gruppen auswerten und aggregieren	195
3.4.11 Funktionen auf Gruppen von Variablen anwenden	198
3.4.12 Häufigkeiten bestimmen	203
4 Zuverlässige und reproduzierbare Datenauswertung	206
4.1 Befehlssequenzen im Editor bearbeiten	206
4.2 Dokumente erstellen	207
4.2.1 Grundprinzip	207
4.2.2 Arbeitsschritte	209
4.2.3 Aufbau eines Quelldokuments	210
4.2.4 Beispiel	211
4.2.5 Quarto	214
4.3 Datenqualität prüfen	216
4.4 Reproduzierbare Auswertungen sicherstellen	217
4.4.1 Potentielle Probleme und Maßnahmen	217
4.4.2 Allgemeine Empfehlungen	219
5 Hilfsmittel für die Inferenzstatistik	221
5.1 Wichtige Begriffe inferenzstatistischer Tests	221
5.2 Lineare Modelle formulieren	222

Inhaltsverzeichnis

5.3	Funktionen von Zufallsvariablen	225
5.3.1	Dichtefunktion	225
5.3.2	Verteilungsfunktion	226
5.3.3	Quantilfunktion	227
6	Lineare Regression	229
6.1	Test des Korrelationskoeffizienten	229
6.2	Einfache lineare Regression	230
6.2.1	Deskriptive Modellanpassung	231
6.2.2	Regressionsanalyse	233
6.3	Multiple lineare Regression	236
6.3.1	Deskriptive Modellanpassung und Regressionsanalyse	236
6.3.2	Modell verändern	238
6.3.3	Modelle vergleichen und auswählen	239
6.3.4	Moderierte Regression	242
6.4	Regressionsmodelle auf andere Daten anwenden	245
6.5	Regressionsdiagnostik	247
6.5.1	Extremwerte, Ausreißer und Einfluss	247
6.5.2	Verteilungseigenschaften der Residuen	250
6.5.3	Multikollinearität	254
6.6	Erweiterungen der linearen Regression	256
6.6.1	Robuste Regression	256
6.6.2	Penalisierte Regression	257
6.6.3	Nichtlineare Zusammenhänge	261
6.6.4	Abhängige Fehler bei Messwiederholung oder Clusterung	261
6.6.5	Beta-Regression für natürliche Anteile	262
6.6.6	Regressionsmodelle für mehrere Verteilungsparameter	262
6.7	Partialkorrelation und Semipartialkorrelation	262
7	<i>t</i>-Tests und Varianzanalysen	265
7.1	Tests auf Normalverteilung	265
7.2	Tests auf Varianzhomogenität	266
7.2.1	<i>F</i> -Test auf Varianzhomogenität für zwei Stichproben	266
7.2.2	Levene-Test für mehr als zwei Stichproben	267
7.2.3	Fligner-Killeen-Test für mehr als zwei Stichproben	268
7.3	<i>t</i> -Tests	269
7.3.1	<i>t</i> -Test für eine Stichprobe	269
7.3.2	<i>t</i> -Test für zwei unabhängige Stichproben	270
7.3.3	<i>t</i> -Test für zwei abhängige Stichproben	273
7.4	Einfaktorielle Varianzanalyse (CR- <i>p</i>)	274
7.4.1	Auswertung mit <code>oneway.test()</code>	274
7.4.2	Auswertung mit <code>aov()</code>	276
7.4.3	Auswertung mit <code>anova()</code>	277
7.4.4	Effektstärke schätzen	278
7.4.5	Voraussetzungen grafisch prüfen	278
7.4.6	Einzelvergleiche (Kontraste)	279

Inhaltsverzeichnis

7.5	Einfaktorielle Varianzanalyse mit abhängigen Gruppen (RB- <i>p</i>)	286
7.5.1	Univariat formuliert auswerten und Effektstärke schätzen	287
7.5.2	Zirkularität der Kovarianzmatrix prüfen	290
7.5.3	Multivariat formuliert auswerten mit Anova()	292
7.5.4	Multivariat formuliert auswerten mit anova()	293
7.5.5	Einzelvergleiche und alternative Auswertungsmöglichkeiten	294
7.6	Zweifaktorielle Varianzanalyse (CRF- <i>pq</i>)	294
7.6.1	Auswertung und Schätzung der Effektstärke	295
7.6.2	Quadratsummen vom Typ I, II und III	298
7.6.3	Bedingte Haupteffekte testen	302
7.6.4	Beliebige a-priori Kontraste	303
7.6.5	Beliebige post-hoc Kontraste nach Scheffé	307
7.6.6	Marginale Paarvergleiche nach Tukey	308
7.7	Zweifaktorielle Varianzanalyse mit zwei Intra-Gruppen Faktoren (RBF- <i>pq</i>)	309
7.7.1	Univariat formuliert auswerten und Effektstärke schätzen	309
7.7.2	Zirkularität der Kovarianzmatrizen prüfen	314
7.7.3	Multivariat formuliert auswerten	315
7.7.4	Einzelvergleiche (Kontraste)	316
7.8	Zweifaktorielle Varianzanalyse mit Split-Plot-Design (SPF- <i>p · q</i>)	316
7.8.1	Univariat formuliert auswerten und Effektstärke schätzen	317
7.8.2	Voraussetzungen und Prüfen der Zirkularität	319
7.8.3	Multivariat formuliert auswerten	321
7.8.4	Einzelvergleiche (Kontraste)	322
7.8.5	Erweiterung auf dreifaktorielles SPF- <i>p · qr</i> Design	323
7.8.6	Erweiterung auf dreifaktorielles SPF- <i>pq · r</i> Design	324
7.9	Kovarianzanalyse	326
7.9.1	Test der Effekte von Gruppenzugehörigkeit und Kovariate	326
7.9.2	Beliebige a-priori Kontraste	332
7.9.3	Beliebige post-hoc Kontraste nach Scheffé	333
7.10	Power, Effektstärke und notwendige Stichprobengröße	334
7.10.1	Binomialtest	334
7.10.2	<i>t</i> -Test	336
7.10.3	Einfaktorielle Varianzanalyse	339
8	Regressionsmodelle für kategoriale Daten und Zähldaten	343
8.1	Logistische Regression	344
8.1.1	Modell für dichotome Daten anpassen	344
8.1.2	Modell für binomiale Daten anpassen	346
8.1.3	Anpassungsgüte	347
8.1.4	Vorhersage, Klassifikation, Kalibrierung und Anwendung auf neue Daten	349
8.1.5	Signifikanztests für Parameter und Modell	352
8.1.6	Andere Link-Funktionen	354
8.1.7	Modelle für overdispersion	355
8.1.8	Mögliche Probleme bei der Modellanpassung	355
8.2	Ordinale Regression	356
8.2.1	Modellanpassung	358
8.2.2	Anpassungsgüte	359

8.2.3	Signifikanztests für Parameter und Modell	359
8.2.4	Vorhersage, Klassifikation und Anwendung auf neue Daten	361
8.3	Multinomiale Regression	362
8.3.1	Modellanpassung	364
8.3.2	Anpassungsgüte	364
8.3.3	Signifikanztests für Parameter und Modell	364
8.3.4	Vorhersage, Klassifikation und Anwendung auf neue Daten	366
8.4	Regression für Zähldaten	367
8.4.1	Poisson-Regression	367
8.4.2	Ereignisraten analysieren	369
8.4.3	Adjustierte Poisson-Regression und negative Binomial-Regression	370
8.4.4	Zero-inflated Poisson-Regression	372
8.4.5	Zero-truncated Poisson-Regression	374
8.5	Log-lineare Modelle	374
8.5.1	Modell	375
8.5.2	Modellanpassung mit <code>loglm()</code>	376
8.5.3	Modellanpassung mit <code>glm()</code>	379
9	Survival-Analyse	381
9.1	Verteilung von Ereigniszeiten	381
9.2	Zensierte und gestutzte Ereigniszeiten	382
9.2.1	Zeitlich konstante Prädiktoren	383
9.2.2	Daten in Zählprozess-Darstellung	385
9.3	Kaplan-Meier-Analyse	388
9.3.1	Survival-Funktion schätzen	388
9.3.2	Survival, kumulative Inzidenz und kumulatives hazard darstellen	390
9.3.3	Log-Rank-Test auf gleiche Survival-Funktionen	390
9.4	Cox proportional hazards Modell	392
9.4.1	Anpassungsgüte und Modelltests	394
9.4.2	Survival-Funktion, baseline hazard und kumulatives hazard schätzen	395
9.4.3	Modelldiagnostik	397
9.4.4	Vorhersage und Anwendung auf neue Daten	401
9.4.5	Erweiterungen des Cox PH-Modells	402
9.5	Parametrische proportional hazards Modelle	403
9.5.1	Darstellung über die Hazard-Funktion	403
9.5.2	Darstellung als accelerated failure time Modell	404
9.5.3	Anpassung und Modelltests	405
9.5.4	Survival-Funktion schätzen	406
10	Klassische nonparametrische Methoden	409
10.1	Anpassungstests	409
10.1.1	Binomialtest	410
10.1.2	Test auf Zufälligkeit (Runs-Test)	411
10.1.3	Kolmogorov-Smirnov-Anpassungstest	413
10.1.4	χ^2 -Test auf eine feste Verteilung	415
10.1.5	χ^2 -Test auf eine Verteilungsklasse	417

Inhaltsverzeichnis

10.2 Analyse von gemeinsamen Häufigkeiten kategorialer Variablen	419
10.2.1 χ^2 -Test auf Unabhängigkeit	419
10.2.2 χ^2 -Test auf Gleichheit von Verteilungen	420
10.2.3 χ^2 -Test für mehrere Auftretenswahrscheinlichkeiten	421
10.2.4 Fishers exakter Test auf Unabhängigkeit	422
10.2.5 Fishers exakter Test auf Gleichheit von Verteilungen	424
10.2.6 Kennwerte von (2×2) -Konfusionsmatrizen	425
10.2.7 ROC-Kurve und AUC	427
10.3 Maße für Zusammenhang und Übereinstimmung	429
10.3.1 Spearmans ρ und Kendalls τ	429
10.3.2 Zusammenhang kategorialer Variablen	431
10.3.3 Inter-Rater-Übereinstimmung	433
10.4 Tests auf gleiche Variabilität	442
10.4.1 Mood-Test	442
10.4.2 Ansari-Bradley-Test	443
10.5 Tests auf Übereinstimmung von Verteilungen	444
10.5.1 Kolmogorov-Smirnov-Test für zwei Stichproben	445
10.5.2 Vorzeichen-Test	447
10.5.3 Wilcoxon-Vorzeichen-Rang-Test für eine Stichprobe	448
10.5.4 Wilcoxon-Rangsummen-Test / Mann-Whitney-U-Test	449
10.5.5 Wilcoxon-Test für zwei abhängige Stichproben	451
10.5.6 Kruskal-Wallis-H-Test für unabhängige Stichproben	452
10.5.7 Friedman-Rangsummen-Test für abhängige Stichproben	453
10.5.8 Cochran-Q-Test für abhängige Stichproben	455
10.5.9 Bowker-Test für zwei abhängige Stichproben	456
10.5.10 McNemar-Test für zwei abhängige Stichproben	458
10.5.11 Stuart-Maxwell-Test für zwei abhängige Stichproben	460
11 Resampling-Verfahren	462
11.1 Nonparametrisches Bootstrapping	462
11.1.1 Replikationen erstellen	463
11.1.2 Bootstrap-Vertrauensintervalle für μ	466
11.1.3 Bootstrap-Vertrauensintervalle für $\mu_2 - \mu_1$	468
11.1.4 Lineare Modelle: case resampling	469
11.1.5 Lineare Modelle: model-based resampling	471
11.1.6 Lineare Modelle: wild bootstrap	473
11.2 Parametrisches Bootstrapping	474
11.2.1 Bootstrap-Vertrauensintervalle für $\mu_2 - \mu_1$	475
11.2.2 Verallgemeinerte lineare Modelle	476
11.3 Permutationstests	477
11.3.1 Test auf gleiche Lageparameter in unabhängigen Stichproben	478
11.3.2 Test auf gleiche Lageparameter in abhängigen Stichproben	481
11.3.3 Test auf Unabhängigkeit von zwei Variablen	482
12 Multivariate Verfahren	484
12.1 Lineare Algebra	484
12.1.1 Matrix-Algebra	484

Inhaltsverzeichnis

12.1.2 Lineare Gleichungssysteme lösen	488
12.1.3 Norm und Abstand von Vektoren und Matrizen	488
12.1.4 Mahalanobistransformation und Mahalanobisdistanz	490
12.1.5 Kennwerte von Matrizen	492
12.1.6 Zerlegungen von Matrizen	494
12.1.7 Orthogonale Projektion	496
12.2 Hauptkomponentenanalyse	499
12.2.1 Berechnung	499
12.2.2 Dimensionsreduktion	503
12.2.3 Visualisierung	505
12.3 Faktorenanalyse	506
12.4 Multidimensionale Skalierung	512
12.5 Multivariate multiple Regression	514
12.6 Hotellings T^2	516
12.6.1 Test für eine Stichprobe	516
12.6.2 Test für zwei unabhängige Stichproben	518
12.6.3 Test für zwei abhängige Stichproben	519
12.6.4 Univariate Varianzanalyse mit abhängigen Gruppen (RB-p)	520
12.7 Multivariate Varianzanalyse (MANOVA)	521
12.7.1 Einfaktorielle MANOVA	521
12.7.2 Zweifaktorielle MANOVA	522
12.8 Diskriminanzanalyse	523
12.9 Das allgemeine lineare Modell	528
12.9.1 Modell der multiplen linearen Regression	528
12.9.2 Modell der einfaktoriellen Varianzanalyse	530
12.9.3 Modell der zweifaktoriellen Varianzanalyse	536
12.9.4 Parameterschätzungen, Vorhersage und Residuen	540
12.9.5 Hypothesen über parametrische Funktionen testen	542
12.9.6 Lineare Hypothesen als Modellvergleiche formulieren	542
12.9.7 Lineare Hypothesen testen	547
12.9.8 Beispiel: Multivariate multiple Regression	550
12.9.9 Beispiel: Einfaktorielle MANOVA	552
12.9.10 Beispiel: Zweifaktorielle MANOVA	556
13 Vorhersagegüte prädiktiver Modelle	559
13.1 Kreuzvalidierung linearer Regressionsmodelle	560
13.1.1 k -fache Kreuzvalidierung	560
13.1.2 Leave-One-Out Kreuzvalidierung	562
13.2 Kreuzvalidierung verallgemeinerter linearer Modelle	563
13.3 Bootstrap-Vorhersagefehler	564
14 Diagramme erstellen	567
14.1 Grafik-Devices	567
14.1.1 Aufbau und Verwaltung von Grafik-Devices	567
14.1.2 Grafiken speichern	569
14.2 Streu- und Liniendiagramme	570
14.2.1 Streudiagramme mit <code>plot()</code>	570

Inhaltsverzeichnis

14.2.2 Datenpunkte eines Streudiagramms identifizieren	571
14.2.3 Streudiagramme mit <code>matplotlib()</code>	572
14.3 Diagramme formatieren	573
14.3.1 Grafikelemente formatieren	573
14.3.2 Farben spezifizieren	576
14.3.3 Achsen formatieren	578
14.4 Säulen- und Punktdiagramme	579
14.4.1 Einfache Säulendiagramme	579
14.4.2 Gruppierte und gestapelte Säulendiagramme	580
14.4.3 Dotchart	584
14.5 Elemente einem bestehenden Diagramm hinzufügen	585
14.5.1 Koordinaten in einem Diagramm identifizieren	586
14.5.2 In beliebige Diagrammbereiche zeichnen	587
14.5.3 Punkte	588
14.5.4 Linien	589
14.5.5 Polygone	592
14.5.6 Funktionsgraphen	595
14.5.7 Text und mathematische Formeln	596
14.5.8 Achsen	598
14.5.9 Fehlerbalken	599
14.5.10 Rastergrafiken	602
14.6 Verteilungsdiagramme	604
14.6.1 Histogramm und Schätzung der Dichtefunktion	604
14.6.2 Stamm-Blatt-Diagramm	606
14.6.3 Boxplot	608
14.6.4 Stripchart	609
14.6.5 Quantil-Quantil-Diagramm	610
14.6.6 Empirische kumulierte Häufigkeitsverteilung	613
14.6.7 Kreisdiagramm	613
14.6.8 Gemeinsame Verteilung zweier Variablen	614
14.7 Multivariate Daten visualisieren	618
14.7.1 Höhenlinien und variable Datenpunktssymbole	618
14.7.2 Dreidimensionale Gitter und Streudiagramme	620
14.7.3 Matrix aus Streudiagrammen	622
14.7.4 Heatmap	624
14.8 Mehrere Diagramme in einem Grafik-Device darstellen	625
14.8.1 <code>layout()</code>	626
14.8.2 <code>par(mfrow, mfcol, fig)</code>	628
14.8.3 <code>split.screen()</code>	629
15 Diagramme mit <code>ggplot2</code>	632
15.1 Grundprinzip	632
15.1.1 Grundschicht	633
15.1.2 Diagramme speichern	634
15.2 Diagrammtypen	635
15.2.1 Punkt-, Streu- und Liniendiagramme	635
15.2.2 Säulendiagramm	636

Inhaltsverzeichnis

15.2.3 Histogramm	638
15.2.4 Boxplot	639
15.2.5 Quantil-Quantil-Diagramm	641
15.3 Bedingte Diagramme in Panels darstellen	643
15.4 Diagrammelemente hinzufügen	644
15.5 Diagramme formatieren	647
15.5.1 Elementposition kontrollieren	647
15.5.2 Achsen anpassen	649
15.5.3 Legende ändern	652
15.5.4 Farben, Datenpunktssymbole und Linientypen	654
15.5.5 Aussehen im Detail verändern	655
16 Numerische Methoden	657
16.1 Daten interpolieren und glätten	657
16.1.1 Lineare Interpolation	657
16.1.2 Splines	658
16.1.3 LOESS-Glättter	660
16.1.4 Nonparametrische Kerndichteschätzer	661
16.2 Nullstellen finden	662
16.3 Integrieren und differenzieren	664
16.3.1 Numerisch integrieren	664
16.3.2 Numerisch differenzieren	665
16.4 Numerisch optimieren	668
16.4.1 Maximum-Likelihood-Parameterschätzung	668
16.4.2 Allgemeine Optimierung	669
17 R als Programmiersprache	672
17.1 Kontrollstrukturen	672
17.1.1 Fallunterscheidungen	672
17.1.2 Schleifen	675
17.2 Funktionsaufrufe ohne Schleifen wiederholen	678
17.3 Eigene Funktionen erstellen	679
17.3.1 Funktionskopf	679
17.3.2 Funktionsrumpf	680
17.3.3 Funktionsargumente prüfen	681
17.3.4 Fehler behandeln	683
17.3.5 Rückgabewert und Funktionsende	684
17.3.6 Eigene Funktionen verwenden	684
17.3.7 Generische Funktionen	685
17.4 Funktionen analysieren	687
17.4.1 Quelltext fremder Funktionen begutachten	687
17.4.2 Funktionen zur Laufzeit untersuchen	688
17.5 Effizienz von Auswertungen steigern	691
17.5.1 Grundlegende Empfehlungen	691
17.5.2 Auswertungen parallelisieren	693
Literaturverzeichnis	697

Kapitel 1

Erste Schritte

1.1 Vorstellung

R ist eine freie und kostenlose Software-Umgebung zur statistischen Datenanalyse ([Ihaka & Gentleman, 1996](#); [R Core Team, 2020a](#)). R integriert eine Vielzahl von Möglichkeiten, um Daten 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 Denn in R bestehen Auswertungen aus einer Abfolge von Befehlen in Textform, die unter Einhaltung einer bestimmten Syntax selbst einzugeben sind. Jeder Befehl stellt dabei einen eigenen Auswertungsschritt dar, wobei eine vollständige Datenanalyse die Abfolge vieler solcher Schritte umfasst. So könnte man Daten zunächst aus einer Datei lesen, sie dann prüfen und geeignet transformieren, um dann eine Teilmenge von Beobachtungen auszuwählen und mit ihr einen statistischen Test durchzuführen, dessen Ergebnisse im Anschluss in einem Diagramm visualisiert werden.

1.1.1 Pro und Contra R

Während in Programmen, die über eine grafische Benutzeroberfläche gesteuert werden, die Auswahl von vorgegebenen Menüpunkten einen wesentlichen Teil der Arbeit ausmacht, ist es in R die aktive Produktion von Befehlausdrücken. Diese Eigenschaft ist gleichzeitig ein wesentlicher Vorteil wie auch eine Herausforderung beim Einstieg in die Arbeit mit R. Zu den Vorteilen von R zählen:

- Die befehlsgesteuerte Arbeitsweise stellt sicher, dasss einmal erstellte Auswertungen später leicht angepasst und erweitert werden können. Sie erhöht durch die Wiederverwendbarkeit von bewährten Befehlssequenzen für typische Arbeitsschritte auch langfristig die Effizienz und Zuverlässigkeit von Analysen.
- Die Möglichkeit zur Weitergabe von Befehlssequenzen mit den zugehörigen Datensätzen an Dritte kann die Auswertung für andere transparent und überprüfbar machen. Auf diese Weise lassen sich Auswertungen selbst und durch andere genau reproduzieren, wodurch die Auswertungsobjektivität steigt.

¹Genauer gesagt ist GNU R ursprünglich eine *open source* Implementierung der Sprache S ([Becker, Chambers & Wilks, 1988](#)). Der *open source* Programmen zugrundeliegende Quelltext ist frei erhältlich, zudem darf die Software frei genutzt, verbreitet und verändert werden. Genauereres erläutert der Befehl `licence()`. Die *R Foundation* ([R Foundation, 2024](#)) steuert die Weiterentwicklung von R. Wichtige Mitglieder der Nutzer- und Entwickler-Community haben sich im *R Consortium* ([R Consortium, 2024](#)) zusammengeschlossen.

- R-Auswertungen lassen sich in Dokumente einbetten, die beschreibenden Text, R-Befehle und die automatisch erzeugte Ausgabe dieser Befehle enthalten – inkl. Diagramme und Tabellen (Abschn. 4.2). Auswertungen lassen sich sogar als interaktive Web-Anwendungen veröffentlichen. Indem Hintergrundinformationen, Details zur Auswertung sowie Ergebnisse kombiniert werden, bietet R eine umfassende Plattform für transparente, reproduzierbare statistische Analysen (Abschn. 4.4).
- R ist kostenlos und sowohl für Windows als auch für MacOS und Linux frei verfügbar. R kann dadurch ohne Einschränkungen in jedem Kontext – beruflich, privat, in der Ausbildung – und unabhängig von den finanziellen Möglichkeiten von allen verwendet werden.
- Als freies open source Programm wird R beständig von vielen Personen evaluiert, weiterentwickelt und verbessert (vgl. [R Foundation for Statistical Computing, 2018](#) für eingesetzte Methoden der Qualitätssicherung). 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 bei Interesse analysiert und auf Richtigkeit kontrolliert werden.
- Dank seines modularen Aufbaus bietet R die Möglichkeit, die Basisfunktionalität für spezielle Anwendungszwecke durch eigenständig entwickelte Zusatzkomponenten (*Pakete*) zu erweitern, von denen viele tausend frei verfügbar sind (Abschn. 1.3).
- 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 geschrieben werden, sind dafür zunächst nur wenige Zusatzschritte zu erlernen (Kap. 17). Reine Anwender können sich schrittweise zu R Entwicklern weiterentwickeln, wobei R angepasst an die individuellen Anforderungen „mitwächst“.
- In den vergangenen Jahren hat sich eine sehr lebhafte, inhaltlich vielseitige und hilfsbereite Online-Community um R entwickelt, die in Email-Verteilern, Blogs, Online-Kursen und Video-Tutorials viel Unterstützung liefert (Abschn. 1.1.5).

R hält für Anwender allerdings auch Hürden bereit, insbesondere für Einsteiger, die nur Erfahrung mit Programmen besitzen, die über eine grafische Benutzeroberfläche bedient werden ([Muenchen, 2018](#)):

- 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 umfangend ausgewertet werden können.
- Im Gegensatz zu Programmen mit grafischer Benutzeroberfläche müssen Befehle aktiv erinnert werden. Es stehen keine Gedächtnisstützen i. S. von Elementen einer grafischen Umgebung zur Verfügung, die interaktiv mögliche Vorgehensweisen anbieten und zu einem Wiedererkennen führen könnten. Dies betrifft sowohl die Auswahl von geeigneten Analysen als auch die konkrete Umsetzung von Arbeitsschritten.

- Anders als in kommerzieller Statistik-Software gibt es in R keine festgelegte, standardisierte Vorgehensweise für einen bestimmten Auswertungsschritt. Stattdessen führen meist mehrere Ansätze zum selben Ergebnis. Dies gilt insbesondere durch die Funktionalität von Zusatzpaketen, die sich untereinander, aber auch mit dem Basisumfang von R überschneiden kann. Deshalb machen auch Bücher oder Online-Tutorials oft uneinheitliche Empfehlungen. Diese Flexibilität und Vielfalt kann zu Beginn für Verwirrung und Unsicherheit sorgen, welcher Weg zum Ziel am besten gewählt werden sollte.
- Während natürliche Sprache sehr fehlertolerant ist, müssen Befehle in R immer exakt richtig eingegeben werden. Dies kann zu Beginn frustrierend und auch zeitraubend sein, weil schon vermeintlich unwesentliche Fehler verhindern, dass Befehle ausgeführt werden (Abschn. 1.2.7). Im Fall falsch eingegebener Befehle liefert R aber Fehlermeldungen, die Rückschlüsse auf die Ursache erlauben. Moderne Entwicklungsumgebungen helfen durch visuelle Hinweise dabei, Syntaxfehler zu vermeiden (Abschn. 1.1.4).
- Bei der Analyse extrem großer Datenmengen besteht gegenwärtig das Problem, dass R Datensätze zur Bearbeitung im Arbeitsspeicher vorhalten muss, was die Größe von praktisch auswertbaren Datensätzen einschränkt. Hinweise zu Lösungsansätzen sowie zur Ausnutzung paralleler Rechnerarchitekturen finden sich in Abschn. 17.5.

Startschwierigkeiten sind beim Lernen von R zu erwarten, sollten jedoch niemanden entmutigen: Auch wenn dies zu Beginn häufige Fehler mit sich bringt, ist vielmehr ein spielerisch-exploratives Kennenlernen wichtig, um ein besseres Verständnis der Grundprinzipien zu entwickeln. Ein wichtiger Grundsatz ist dabei, sich zunächst inhaltlich zu überlegen, welche Teilschritte für eine konkrete Auswertung notwendig sind. Je konkreter die Teilschritte dieses gedanklichen Ablaufplans sind, desto einfacher lassen sich ihnen einzelne Befehlsbausteine zuordnen. Sind dann aus einfachen Auswertungen viele solcher Befehlsbausteine vertraut, lassen sie sich Schritt für Schritt zu komplexen Analysen zusammenstellen.

1.1.2 Typografische Konventionen

Zur besseren Lesbarkeit sollen zunächst einige typografische 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. 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.

Internet-URLs, Tastenkombinationen sowie Dateien und Verzeichnisse werden im Schrifttyp **Schreibmaschine** dargestellt, wobei die Unterscheidung zu R-Befehlen aus dem Kontext hervorgeht.

1.1.3 R installieren

Die Homepage des R-Projekts ist die zentrale Anlaufstelle für Nachrichten über die Entwicklung von R, für den *download* des Programms selbst, für Zusatzpakete sowie für frei verfügbare Literatur:

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

Die folgenden Ausführungen beziehen sich auf die Installation des R-Basisumfangs unter Windows 11, vgl. dazu auch die offizielle Installationsanleitung ([R Core Team, 2020c](#)) und die Windows-FAQ ([Ripley, Murdoch & Kalibera, 2024](#)).² Für die Installationsdatei des Programms folgt man auf der Projektseite dem Verweis **Download**, CRAN, der auf eine Übersicht von CRAN-servern verweist, von denen die Dateien erhältlich sind.³ Nach der Wahl eines CRAN-servers gelangt man über **Download and Install R / Download R for Windows: base** zum Verzeichnis mit der Installationsdatei **R-4.0.0-win.exe**, die auf dem eigenen Rechner zu speichern ist.⁴

Um R zu installieren, ist die gespeicherte Installationsdatei **R-<Version>-win.exe** auszuführen und den Aufforderungen des Setup-Assistenten zu folgen. Die Installation setzt voraus, dass man ausreichende Schreibrechte auf dem Computer besitzt. Wenn keine Änderungen am Installationsordner von R vorgenommen wurden, sollte R daraufhin im Verzeichnis **C:\Programme\R-<Version>** installiert sein, was sich in R mit **Sys.getenv("R_HOME")** prüfen lässt.

Unter MacOS verläuft die Installation analog, wobei den entsprechenden CRAN-links zur Installationsdatei zu folgen ist. Für einige Funktionen muss X11-Unterstützung über die XQuartz-Software zusätzlich installiert werden.⁵ R ist in allen gängigen Linux-Distributionen enthalten und kann dort am einfachsten direkt über den jeweils verwendeten Paketmanager installiert werden. Unter Ubuntu und darauf aufbauenden Distributionen heißt das Basispaket **r-base**, unter SUSE **R-base** und unter RedHat/Fedora **R**.

Eine vollständige virtualisierte R Umgebung bietet das *Rocker* Projekt für die Container-Software Docker ([Boettiger & Eddelbuettel, 2017](#)).⁶ Auf diesem Weg kann R auch online auf einer Cloud-Plattform gehostet werden. Schließlich erlaubt es die derzeit noch junge WebAssembly Technologie, R mittels JavaScript in eine HTML-Seite einzubetten und damit vollständig im Webbrower eines Nutzers ausführen zu lassen.⁷

² Abgesehen von der Oberfläche und abweichenden Pfadangaben bestehen nur unwesentliche Unterschiede zwischen der Arbeit mit R unter verschiedenen Betriebssystemen.

³ CRAN (*Comprehensive R Archive Network*) bezeichnet ein weltweites Netzwerk von *mirror servers* für R-Installationsdateien, Zusatzpakete und offizielle Dokumentation. Eine durchsuchbare und übersichtlichere Oberfläche mit weiteren Funktionen ist <https://r-universe.dev/>.

⁴ R Version 4.0.0 wurde im Frühjahr 2020 veröffentlicht. Neuere Versionen erscheinen ca. halbjährlich. Dabei sind leichte, für den Benutzer jedoch üblicherweise nicht merkliche Abweichungen zur in diesem Manuskript beschriebenen Arbeitsweise von Funktionen möglich.

⁵ <https://www.xquartz.org/>

⁶ <https://www.rocker-project.org/>

⁷ <https://www.r-wasm.org/>

1.1.4 Grafische Benutzeroberflächen

Obwohl es nicht zwingend notwendig ist, sollte zusätzlich zur Basis-Oberfläche von R eine komfortablere grafische Umgebung zur Befehlseingabe installiert werden. Eine solche alternative grafische Umgebung setzt dabei voraus, dass R selbst bereits vorhanden ist.

- Die auf R zugeschnittene Entwicklungsumgebung (*integrated development environment*, IDE) RStudio ([Posit PBC, 2021](#)) ist frei verfügbar, einfach zu installieren, läuft unter mehreren Betriebssystemen sowie Online als Server-Version mit einer konsistenten Oberfläche. RStudio erleichtert häufige Arbeitsschritte (Abschn. [1.2.1](#)) und unterstützt besonders gut die Möglichkeiten, Dokumente mit eingebetteten R-Auswertungen zu erstellen (Abschn. [4.2](#)).
- Von denselben Entwicklern stammt die noch sehr junge IDE Positron ([Posit PBC, 2024](#)), die den Fokus auf *data science*-orientierte Anwendungen mit R und der Programmiersprache Python legt. Sie basiert auf derselben Grundlage wie der kostenlose und plattformunabhängige Texteditor Visual Studio Code ([Microsoft, 2024](#)).
- Architect ([OpenAnalytics BVBA, 2023](#)) ist eine auf R zugeschnittene Variante der freien IDE Eclipse mit dem StatET plugin ([Wahlbrink, 2024](#)), die besonders für die Programmiersprache Java populär ist.
- Schließlich existieren Programme, die R-basierte Analysefunktionen über grafische Menüs und damit ohne Befehlseingabe nutzbar machen, darunter jamovi ([Love, Dropmann & Selker, 2024](#)). Diese Programme bieten derzeit jedoch nur Zugang zu einem Teil der Möglichkeiten von R.

1.1.5 Weiterführende Informationsquellen und Literatur

Häufige Fragen zu R sowie speziell zur Verwendung von R unter Windows werden in den FAQ (*frequently asked questions*) beantwortet ([Hornik & R Core Team, 2025](#); [Ripley et al., 2024](#)). Für individuelle Fragen existiert die Mailing-Liste *R-help*, deren Adresse auf der Projektseite unter *R Project / Get Involved: Mailing Lists* genannt wird. Bevor sie für eigene Hilfegesuche genutzt wird, sollte eine selbständige Recherche vorausgehen. Zudem sind die Hinweise des *posting-guide*⁸ zu beherzigen, wobei ein einfaches reproduzierbares Beispiel besonders wichtig ist (Abschn. [4.4](#)).⁹ Beides gilt auch für das hilfreiche Web-Forum StackOverflow.¹⁰ Kostenlose Online-Kurse werden etwa von edX angeboten.¹¹ Der mastodon.social bzw. fosstodon hashtag lautet `#rstats`.

Unter dem link *Documentation / Manuals* auf der Projektseite von R findet sich die vom R Entwickler-Team herausgegebene offizielle Dokumentation. Sie liefert einerseits einen umfassenden, aber sehr konzisen Überblick über R selbst ([Venables et al., 2020](#)) und befasst sich andererseits mit Spezialthemen wie dem Datenaustausch mit anderen Programmen (Abschn.

⁸<https://www.r-project.org/posting-guide.html>

⁹<https://stackoverflow.com/q/5963269>

¹⁰<https://stackoverflow.com/tags/R> – <https://stats.stackexchange.com/tags/R>

¹¹<https://www.edx.org/learn/r-programming/harvard-university-statistics-and-r>

3.1). Darüber hinaus existiert eine Reihe von Büchern mit unterschiedlicher inhaltlicher Ausrichtung:

- Eine empfehlenswerte Einführung in R anhand grundlegender statistischer Themen findet man etwa bei [Maindonald, Braun und Andrews \(2024\)](#). [Hothorn und Everitt \(2014\)](#) sowie [Venables und Ripley \(2002\)](#) behandeln den Einsatz von R für fortgeschrittene statistische Anwendungen.
- Für die Umsetzung spezieller statistischer Themen existieren eigene Bücher – etwa zu
 - Regressionsmodellen ([Fox & Weisberg, 2019](#); [Harrell Jr, 2015](#))
 - multivariaten Verfahren ([Zelterman, 2015](#))
 - gemischten linearen Modellen ([West, Welch & Gałecki, 2022](#); [Pinheiro & Bates, 2000](#))
 - Resampling-Verfahren ([Chihara & Hesterberg, 2022](#))
 - Zeitreihen ([Shumway & Stoffer, 2019, 2016](#); [Hyndman & Athanasopoulos, 2019](#))
 - Bayes-Methoden ([McElreath, 2020](#); [Kruschke, 2015](#))
 - räumlicher Statistik ([Pebesma & Bivand, 2023](#))
 - Meta-Analyse ([Harrer, Cuijpers, Furukawa & Ebert, 2021](#); [Schwarzer, Carpenter & Rücker, 2015](#)).
- Einen Fokus auf R im Umfeld von data science und Maschinen-Lernen legen [James, Witten, Hastie und Tibshirani \(2021\)](#) sowie [Wickham und Gromelund \(2023\)](#).
- Das Gebiet der Programmierung mit R wird von [Chambers \(2016\)](#); [Gillespie und Lovelace \(2017\)](#) sowie [Wickham \(2019a\)](#) thematisiert.
- [Murrell \(2018\)](#); [Chang \(2018\)](#) und [Unwin \(2015\)](#) erläutern detailliert, wie Diagramme mit R erstellt werden können.
- [Xie \(2015\)](#) und [Xie, Allaire und Gromelund \(2018\)](#) behandeln, wie sich mit R dynamische Dokumente erstellen lassen, die Berechnungen, Ergebnisse und Beschreibungen integrieren. Viele Beispiele liefern [Xie, Dervieux und Riederer \(2020\)](#).
- Wie interaktive Web-Anwendungen mit dem Paket `shiny` ([Chang, Cheng, Allaire, Xie & McPherson, 2020](#)) entwickelt werden können, zeigen [Sievert \(2020\)](#), [Wickham \(2020b\)](#) und die Online-Dokumentation unter <https://shiny.posit.co/>

1.2 Grundlegende Elemente

1.2.1 R Starten, beenden und die Konsole verwenden

Nach der Installation lässt sich R unter Windows über die bei der Installation erstellte Verknüpfung auf dem desktop bzw. im Startmenü starten. Hierdurch öffnen sich zwei Fenster: ein großes, das Programmfenster, und darin ein kleineres, die *Konsole*. Unter MacOS und Linux ist die Konsole das einzige sich öffnende Fenster.

Der Arbeitsbereich der Entwicklungsumgebung RStudio gliedert sich in vier Regionen (Abb. 1.1):¹²

- Links unten befindet sich die Konsole zur interaktiven Arbeit mit R. Befehle können direkt auf der Konsole eingegeben werden, außerdem erscheint die Ausgabe von R hier.
- Links oben öffnet sich ein Editor, in dem über das Menü *File* → *New File* → *R Script* ein eigenes Befehlsskript erstellt oder über *File* → *Open File* ein bereits vorhandenes Befehlsskript geöffnet werden kann. Ein Skript ist dabei eine einfache Textdatei mit einer Sammlung von nacheinander abzuarbeitenden Befehlen (Abschn. 4.1). Die Befehle eines Skripts können im Editor markiert und mit dem icon *Run* bzw. mit der Tastenkombination **Strg+Return** (unter MacOS **Cmd+Return**) an die Konsole gesendet werden.
- Rechts oben befinden sich zwei Tabs: *Environment* zeigt alle derzeit verfügbaren Objekte an – etwa Datensätze oder einzelne Variablen, die sich für Auswertungen nutzen lassen (Abschn. 1.2.3). *History* speichert als Protokoll eine Liste der schon aufgerufenen Befehle.
- Rechts unten erlaubt es *Files*, die Ordner der Festplatte anzuzeigen und zu navigieren. Im *Plots*-Tab öffnen sich die erstellten Diagramme, *Packages* informiert über die verfügbaren Zusatzpakete (Abschn. 1.3), und *Help* erlaubt den Zugriff auf das Hilfesystem (Abschn. 1.2.6).
- RStudio lässt sich über den Menüeintrag *Tools* → *Global Options* stark an die eigenen Präferenzen anpassen.

Auf der Konsole werden im interaktiven Wechsel von eingegebenen Befehlen und der Ausgabe von R die Verarbeitungsschritte vorgenommen.¹³ Hier erscheint nach dem Start unter einigen Hinweisen zur installierten R-Version hinter dem *Prompt*-Zeichen > ein Cursor. Der Cursor signalisiert, 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 oft zunächst die laufende Nummer des ersten in der jeweiligen Zeile angezeigten Objekts aufgeführt. Anschließend erscheint erneut ein Prompt-Zeichen, hinter dem neue Befehle eingegeben werden können.

¹²RStudio wird laufend weiterentwickelt. Es ist deshalb möglich, dass Aussehen und Funktionalität in Details von der folgenden Beschreibung abweichen.

¹³Für automatisierte Auswertungen s. Abschn. 4.1. Um die Ausgabe ganz oder als Protokoll-Kopie aller Vorgänge in eine Datei umzuleiten s. Abschn. 2.14.1. Befehle des Betriebssystems sind mit `shell("<Befehl>")` ausführbar, so können etwa die Netzwerkverbindungen mit `shell("netstat")` angezeigt werden.

Kapitel 1 Erste Schritte

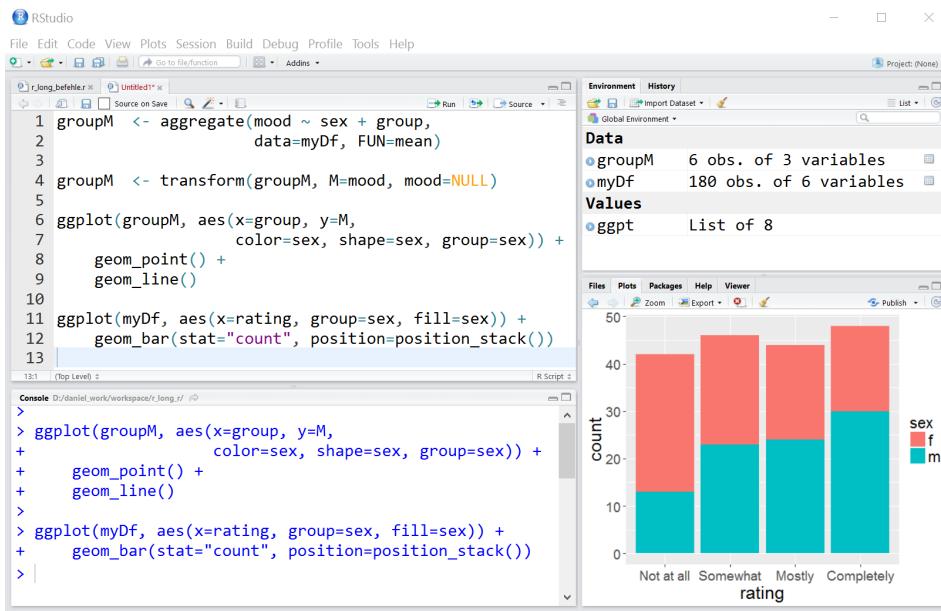


Abbildung 1.1: Oberfläche der Entwicklungsumgebung RStudio

```

> 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 ; zu trennen. Das Symbol # markiert den Beginn eines *Kommentars* und verhindert, dass der dahinter auftauchende Text in derselben Zeile als Befehl interpretiert wird.

War die Eingabe nicht korrekt, erscheint eine Fehlermeldung mit einer Beschreibung der Ursache. Abschnitt 1.2.7 erläutert typische Fehlerquellen. Befehle können auch Warnungen verursachen, die die Auswertung zwar nicht wie Fehler verhindern, aber immer daraufhin untersucht werden sollten, ob sie ein Symptom für falsche Berechnungen sind.

Das folgende Beispiel demonstriert eine kurze Auswertung in Form mehrerer aufeinanderfolgender Arbeitsschritte. 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 inkl. der von R erzeugten Ausgabe aussehen kann. Die Analyse bezieht sich auf den Datensatz **Duncan** aus dem Paket **carData** (Fox, Weisberg & Price, 2020), das zunächst zu installieren ist (Abschn. 1.3). Der Datensatz speichert Daten von Personen verschiedener Berufe. 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**). Jede Zeile enthält die Daten jeweils eines Berufs, die Daten jeweils einer Variable finden sich in den Spalten des Datensatzes.

```
> data(Duncan, package="carData") # lade Datensatz Duncan aus carData
> head(Duncan, n=4) # die ersten 4 Zeilen von Duncan
   type income education prestige
1 accountant prof      62       86      82
2 pilot      prof      72       76      83
3 architect  prof      75       92      90
4 author     prof      55       90      76
```

Zunächst sollen wichtige deskriptive Kennwerte von `income` in nach Gruppen getrennten *Boxplots* dargestellt und dabei auch die Rohdaten selbst abgebildet werden (Abb. 1.2). Es folgt die Berechnung der Gruppenmittelwerte für `education` und die Korrelationsmatrix der drei erhobenen Variablen. Als inferenzstatistische Auswertung schließt sich eine Varianzanalyse mit der Variable `prestige` und dem Gruppierungsfaktor `type` an. Im folgenden *t*-Test der Variable `education` sollen nur die Gruppen `bc` und `wc` berücksichtigt werden.

```
# nach Gruppen getrennte Boxplots und Rohdaten der Variable income
> boxplot(income ~ type, data=Duncan,
+           main="Anteil hoher Einkommen pro Berufsgruppe")

> stripchart(income ~ type, data=Duncan, pch=20, vert=TRUE, add=TRUE)

#####
# Gruppenmittelwerte von income und education getrennt nach type
> aggregate(cbind(income, education) ~ type, FUN=mean, data=Duncan)
  type    income   education
1 bc 23.76190 25.33333
2 prof 60.05556 81.33333
3 wc 50.66667 61.50000

#####
# Korrelationsmatrix der Variablen income, education, prestige
> with(Duncan, 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 von prestige in Gruppen type
> anova(lm(prestige ~ type, data=Duncan))
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 <- with(Duncan, (type == "bc") | (type == "wc"))

#####
# linksseitiger t-Test der Variable education
# für die zwei ausgewählten Berufsgruppen bc und wc
> t.test(education ~ type,
+         alternative="less", data=Duncan, 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
```

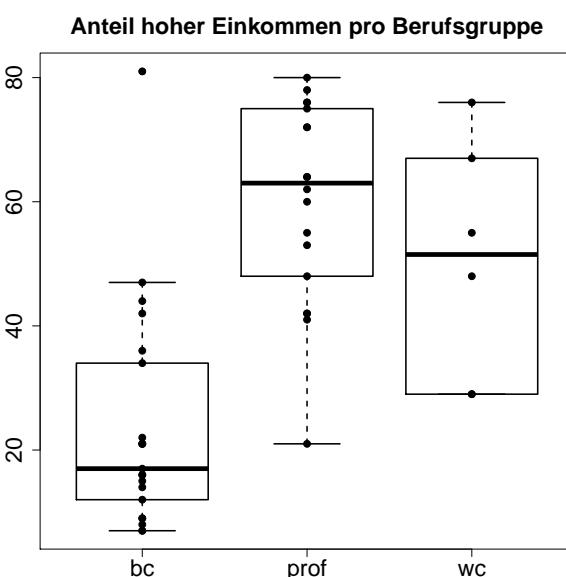


Abbildung 1.2: Daten des Datensatzes Duncan: Anteil der Angehörigen eines Berufs mit hohem Einkommen in Abhängigkeit vom Typ des Berufs

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 der Konsole 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 voran-

gehenden gehört. Falls dieses + ungewollt in der Konsole auftaucht, muss der Befehl entweder vervollständigt oder mit der ESC Taste (Windows) bzw. der Strg+c Tastenkombination (Linux) abgebrochen werden.

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

In der Konsole können Befehle mit der Maus markiert und (unter Windows) über die Tastenkombination Strg+c in die Zwischenablage hinein bzw. mit Strg+v aus der Zwischenablage heraus in die Befehlszeile hinein kopiert werden.

Wird der Anfang eines Befehlsnamens in der Konsole oder im Editor eingegeben, bietet RStudio mit einer kurzen Zeitverzögerung automatisch mögliche Vervollständigungen des begonnenen Befehls an. In der Konsole des Basisumfangs von R ist dafür zweimal die Tabulator-Taste zu drücken. Es existieren noch weitere vergleichbare Kurzbefehle, über die das RStudio Menü *Help: Keyboard Shortcuts* informiert.

R wird entweder über den Befehl q() in der Konsole, über den Menüpunkt *File: Quit Session* oder durch Schließen des Programmfensters beendet. Zuvor erscheint die Frage, ob man den *workspace*, also alle erstellten Daten, im aktuellen Arbeitsverzeichnis speichern möchte (Abschn. 1.2.2, 1.2.3).

1.2.2 Einstellungen

R wird immer in einem *Arbeitsverzeichnis* ausgeführt, das sich durch getwd() ausgeben lässt. In der Voreinstellung handelt es sich um das Heimverzeichnis des Benutzers, das man mit path.expand("~") erfährt. Alle während einer Sitzung gespeicherten Dateien werden, sofern nicht explizit anders angegeben, in 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 zu ändern, existieren folgende Möglichkeiten:

- Im Menü der R-eigenen Oberfläche unter Windows kann *Datei → Verzeichnis wechseln* gewählt und anschließend ein neues Arbeitsverzeichnis angegeben werden. Dieser Schritt ist dann nach jedem Start von R zu wiederholen. Eine analoge Funktion findet sich in RStudio im Menüpunkt *Session → Set Working Directory*.
- In der Konsole lässt sich das aktuelle Arbeitsverzeichnis mit setwd("Pfad") ändern, unter Windows also z.B. mit setwd("c:/work/r/") (s. Abschn. 3.2 für die möglichen Formen von Pfadangaben).

R wird mit einer Reihe von Voreinstellungen gestartet, die sich über selbst editierbare Textdateien steuern lassen, über die ?Startup Auskunft gibt ([R Core Team, 2020c](#)). Sollen etwa bestimmte Pakete in jeder Sitzung geladen werden (Abschn. 1.3), können die entsprechenden library() Befehle in die Datei Rprofile.site im etc/ Ordner des Programmord-

ners geschrieben werden.¹⁴ Gleiches gilt für befehlsübergreifende Voreinstellungen, die mit `getOption()` angezeigt und mit `options()` verändert werden können.

```
getOption("<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 auf der Konsole unsichtbar 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.

```
>getOption("width")      # aktuellen Wert für Option anzeigen
[1] 116

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

1.2.3 Umgang mit dem workspace

R speichert während einer Sitzung automatisch alle erstellten Daten in einem *Workspace* genannten Speicher und hält sie dort für die Dauer der Sitzung verfügbar. So können Daten nachträglich in externen Dateien abgespeichert und bei einer neuen Sitzung wieder genutzt werden (Abschn. 3.1.1, 3.1.2).

Auf bereits eingegebene Befehle kann auch bereits während derselben Sitzung erneut zugegriffen werden: Über die Pfeiltasten nach oben und unten lassen sich verwendete Befehle auf der Konsole wieder aufrufen. Diese Funktion wird im Folgenden als Befehlshistorie bezeichnet. Eine Übersicht der letzten Befehle liefert die Funktion `history()`, in RStudio finden sich die Befehle im Tab *History*.

Der erwähnte workspace trägt den Namen `.GlobalEnv`. Neben ihm existieren noch weitere Umgebungen, die abgekapselt voneinander ebenfalls Objekte speichern. Da ein Nutzer auf die Objekte aller Umgebungen zugreifen kann, tritt dieser Umstand nur selten explizit zutage, ist jedoch manchmal wichtig: Objekte, die in unterschiedlichen Umgebungen liegen, können denselben Namen tragen, ohne dass ihre Inhalte wechselseitig überschrieben würden (Abschn. 1.4.1). Die Umgebungen sind intern sequentiell geordnet. Die Reihenfolge, in der R die Umgebungen nach Objekten durchsucht, ist der *Suchpfad*, der von `search()` ausgegeben wird. Existieren mehrere Objekte desselben Namens in unterschiedlichen Umgebungen, bezeichnet der einfache Name das Objekt in der früheren Umgebung. Welche Objekte eine bestimmte Umgebung speichert, lässt sich auf der Konsole mit `ls()` feststellen, während RStudio dafür den *Environment* Tab besitzt.

```
ls(name="<Umgebung>", pattern="<Suchmuster>")
```

¹⁴Zudem kann jeder Benutzer eines Computers dazu die Datei `.Rprofile` in seinem Heimverzeichnis anlegen. In dieser Datei können auch die Funktionen namens `.First` bzw. `.Last` mit beliebigen Befehlen definiert werden, die dann beim Start als erstes bzw. beim Beenden als letztes ausgeführt werden (Abschn. 17.3). R auf diesem Weg individuell zu konfigurieren, senkt jedoch die Reproduzierbarkeit der Auswertungen (Abschn. 4.4).

In der Voreinstellung `ls()` werden die Objekte der derzeit aktiven Umgebung aufgelistet, bei einem Aufruf von der Konsole i. d. R. also jene aus `.GlobalEnv`, der ersten Umgebung im Suchpfad. Alternativ kann für `name` der Name einer Umgebung oder aber ihre Position im Suchpfad angegeben werden. Über `pattern` lässt sich ein Muster für den Objektnamen spezifizieren, so dass nur solche Objekte angezeigt werden, deren Name auf das Muster passt. Im einfachsten Fall könnte das Muster ein Buchstabe sein, den der Objektname enthalten muss, kompliziertere Muster sind über *reguläre Ausdrücke* möglich (Abschn. 2.14.4).

```
> ls()                                # nenne alle Objekte der aktuellen Umgebung
[1] "BCandWC" "Duncan"

# Objekte der Standard-Umgebung, die ein großes C im Namen tragen
> ls(".GlobalEnv", pattern="C")
[1] "BCandWC"
```

Es gibt mehrere, in ihren Konsequenzen unterschiedliche Wege, Befehle und Daten der aktuellen Sitzung zu sichern. Eine Kopie des aktuellen workspace sowie der Befehlshistorie wird etwa gespeichert, indem man die beim Beenden von R erscheinende diesbezügliche Frage bejaht. Als Folge wird – falls vorhanden – der sich 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 mit der Befehlshistorie (Datei `.Rhistory`). Der so gespeicherte workspace wird beim nächsten Start von R automatisch geladen.

Da es im Sinne der Reproduzierbarkeit der Auswertungsumgebung sinnvoller ist, immer mit einem leeren workspace zu starten und Objekte dann explizit zu erstellen (Abschn. 4.4), sollte die Frage nach einer Sicherung der Sitzung verneint werden. In RStudio verhindern dies entsprechende Einstellungen im Menü *Tools* → *Global Options* → *R General* → *Workspace* dauerhaft.

Der workspace kann in RStudio im *Environment* Tab über das Disketten-icon unter einem frei wählbaren Namen gespeichert werden, um früher angelegte Dateien nicht zu überschreiben. Über das Ordner-icon im *Environment* Tab lässt sich eine Workspace-Datei wieder laden. Dabei ist zu beachten, dass zuvor definierte Objekte von jenen Objekten aus dem geladenen workspace überschrieben werden, die denselben Namen tragen. Die manuell gespeicherten workspaces werden bei einem Neustart von R nicht automatisch geladen.

Um die Befehlshistorie unter einem bestimmten Dateinamen abzuspeichern, wählt man den *History* Tab, der ebenfalls ein Disketten-icon besitzt. In der Konsole stehen zum Speichern und Laden der Befehlshistorie die Funktionen `savehistory("<Dateiname>")` und `loadhistory("<Dateiname>")` zur Verfügung.

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 `?Syntax` und `?Arithmetic` aufzurufenden Hilfe-Seiten. Punktrechnung geht dabei vor Strichrechnung,

das Dezimaltrennzeichen ist unabhängig von den Ländereinstellungen des Betriebssystems immer der Punkt. Nicht ganzzahlige Werte werden in der Voreinstellung mit sieben relevanten Stellen ausgegeben, was mit dem Befehl `options(digits=<Anzahl>)` veränderbar ist.

Alternativ können Zahlen auch in wissenschaftlicher, also verkürzter Exponentialschreibweise ausgegeben werden.¹⁵ Dabei ist z. B. der Wert `2e-03` als $2 \cdot 10^{-3}$, also $\frac{2}{10^3}$, mithin 0.002 zu lesen. Auch die Eingabe von Zahlen ist in diesem Format möglich.

Tabelle 1.1: Arithmetische Funktionen, Operatoren und Konstanten

Name	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 ¹⁶)
<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>	runden (mit Argument <code>digits</code> zur Anzahl der Dezimalstellen) ¹⁷
<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(<Zahl>, base=<Basis>)</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 (Argument im Bogenmaß)
<code>factorial()</code>	Fakultät
<code>pi</code>	Kreiszahl π
<code>Inf</code> , <code>-Inf</code>	∞ , $-\infty$ (<i>infinity</i>)

¹⁵ Sofern diese Formatierung nicht mit `options(scipen=999)` ganz unterbunden wird. Allgemein kann dabei mit ganzzahlig positiven Werten für `scipen` (*scientific penalty*) die Schwelle erhöht werden, ab der R die wissenschaftliche Notation für Zahlen verwendet (vgl. `?options`).

¹⁶ Der Dezimalteil einer Dezimalzahl ergibt sich also als `<Zahl> %% 1`.

¹⁷ R runden in der Voreinstellung nicht nach dem vielleicht vertrauten Prinzip des kaufmännischen Rundens, sondern *unverzerrt* (Bronstein & Semendjajew, 2012). Durch negative Werte für `digits` kann auch auf Zehnerpotenzen gerundet werden. `signif()` runden auf eine bestimmte Anzahl signifikanter Stellen.

Tabelle 1.1: (Forts.)

NA	fehlender Wert (<i>not available</i>)
NaN	nicht definiert (<i>not a number</i>), verhält sich weitestgehend wie NA
NULL	leere Menge

Die in Tab. 1.1 aufgeführten Funktionen, Operatoren und Konstanten sind frei kombinierbar und können auch ineinander verschachtelt werden. Um die Reihenfolge der Auswertung eindeutig zu halten, sollten in Zweifelsfällen Klammern gesetzt werden.¹⁸

```
> 12^2 + 1.5*10
```

```
[1] 159
```

```
> sin(pi/2) + sqrt(abs(-4))
```

```
[1] 3
```

$\langle\text{Realteil}\rangle+\langle\text{Imaginärteil}\rangle i$ ist die Notation zur Eingabe komplexer Zahlen. $0+1i$ ist also die imaginäre Zahl i , $-1+0i$ hingegen die reelle Zahl -1 in komplexer Schreibweise. Den Realteil einer komplexen Zahl liefert `Re(⟨Zahl⟩)`, den Imaginärteil `Im(⟨Zahl⟩)`. Die Funktionen `Mod(⟨Zahl⟩)` und `Arg(⟨Zahl⟩)` geben die Polarkoordinaten in der komplexen Ebene aus, die komplexe Konjugierte ermittelt `Conj(⟨Zahl⟩)`. In der Voreinstellung rechnet R mit reellen Zahlen, aus diesem Grund erzeugt etwa `sqrt(-1)` die Ausgabe `NaN`, während `sqrt(-1+0i)` mit `0+1i` das richtige Ergebnis i liefert.

```
> exp(1)^((0+1i) * pi)           # komplexe Zahl (Eulersche Identität)
[1] -1+0i
```

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 (Abschn. 1.4.6).

```
is.infinite(⟨Zahl⟩)             # ist ⟨Zahl⟩ +/- unendlich?
is.finite(⟨Zahl⟩)              # ist ⟨Zahl⟩ gültige endliche Zahl?
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
```

Die Funktion `is.finite(⟨Objekt⟩)` liefert nur für solche Objekte `TRUE` zurück, die eine gültige Zahl sind, neben `-Inf` und `Inf` ergeben auch Zeichenketten, `NA`, `NaN` und `NULL` das Ergebnis `FALSE`.

¹⁸Für die zur Bestimmung der Ausführungsreihenfolge wichtige Assoziativität von Operatoren vgl. `?Syntax`.

1.2.5 Funktionen mit Argumenten aufrufen

Beim Aufruf von Funktionen in R sind die Werte, die der Funktion als notwendige Eingangs-information dienen, in runde Klammern einzuschließen: `<Funktionsname>(<Argumentliste>)`. Die Argumentliste besteht aus Zuweisungen an Argumente in der Form `<Argumentname>=<Wert>`. Es können je nach Funktion ein oder mehrere durch Komma getrennte Argumente angegeben werden, die ihrerseits obligatorisch oder nur optional sein können.¹⁹ Auch wenn eine Funktion keine Argumente besitzt, müssen die runden Klammern vorhanden sein, z. B. `q()`.²⁰

Argumente sind benannt und machen so 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(1.27)`. Weiterhin besteht die Möglichkeit, über das zusätzliche Argument `digits` die gewünschte Anzahl an Nachkomma-stellen zu bestimmen: `round(pi, digits=2)` rundet die Zahl π auf zwei Dezimalstellen. Das Argument `digits` ist optional, wird es nicht angegeben, kommt der auf 0 voreingestellte Wert (*default*) zur Rundung auf ganze Zahlen zum tragen.

Der Name von Argumenten muss nicht unbedingt vollständig angegeben werden, wenn eine Funktion aufgerufen wird – Kürzung auf den zur eindeutigen Identifizierung notwendigen Na-mensanfang reicht aus.²¹ 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 aufzuführen, insbeson-dere, wenn viele Argumente an die Funktion übergeben werden können.

1.2.6 Hilfe-Funktionen

R hat ein integriertes Hilfesystem, das auf verschiedene Arten genutzt werden kann: Zum einen ruft `help.start()` eine lokal gespeicherte Webseite auf, von der aus spezifische Hilfe-Seiten erreichbar sind. Zum anderen kann auf diese Seiten mit `help(<Befehlsname>)` zugegriffen werden, in Kurzform auch mit `?<Befehlsname>`. Operatoren müssen bei beiden Versionen in Anführungszeichen gesetzt werden, etwa `?"/"`. Die Hilfe-Funktion lässt sich in RStudio auch über den *Help* Tab erreichen. Weitere Hinweise zum Hilfesystem liefert `help()` ohne zusätzliche Argumente.

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.

¹⁹In diesem Text werden nur die wichtigsten Argumente der behandelten Funktionen vorgestellt, eine vollstän-dige Übersicht liefert jeweils `args(<Funktionsname>)` sowie die zugehörige Hilfe-Seite `?<Funktionsname>`.

²⁰In R sind Operatoren wie `+`, `-`, `*` oder `/` Funktionen, für die lediglich eine bequemere und vertrautere Kurz-schreibweise zur Verfügung steht. Operatoren lassen sich auch in der Präfix-Form benutzen, wenn sie in Anführungszeichen gesetzt werden. So ist `"/"(1, 10)` äquivalent zu `1/10`.

²¹Gleiches 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>, u="pairwise")` als Funktions-aufruf möglich.

Im Abschnitt **Usage** der Hilfe-Seiten 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 (Abschn. 17.3.7). Unter **Arguments** wird erläutert, welche notwendigen sowie optionalen Argumente die Funktion besitzt und welcher Wert für ein optionales Argument voreingestellt ist. Der Abschnitt **Value** erklärt, welche Werte die Funktion als Ergebnis zurückliefert. Weiterhin wird die Benutzung der Funktion im Abschnitt **Examples** mit Beispielen erläutert. `example(<Befehlsname>)` führt diese Beispiele samt ihrer Ergebnisse auf der Konsole vor.

Wenn der genaue Name einer gesuchten Funktion unbekannt ist, können die Hilfe-Seiten mit `help.search("<Stichwort>")` nach Stichworten gesucht werden. Das Ergebnis führt jene Funktionsnamen auf, in deren Hilfe-Seiten `<Stichwort>` vorhanden ist. Mit `apropos("<Stichwort>")` werden Funktionen ausgegeben, die `<Stichwort>` in ihrem Funktionsnamen tragen. Für weiterführende Recherchemöglichkeiten s. Abschn. 1.1.5.

1.2.7 Empfehlungen und typische Fehlerquellen

Übersichtlichkeit und Nachvollziehbarkeit sind entscheidende Gesichtspunkte beim Erstellen einer Abfolge von Befehlen, um ihre Korrektheit prüfen und Bausteine der Datenanalyse später wiederverwenden zu können. Befehlssequenzen sollten daher so geschrieben werden, dass sie ein einfaches Verständnis der Vorgänge gewährleisten – etwa durch folgende Maßnahmen:

- Leerzeichen zwischen Befehlsteilen und insbesondere zwischen Operatoren und Objektnamen verwenden.
- Objekte sinnvoll (inhaltlich aussagekräftig) benennen und dabei ein einheitliches Schema verwenden, z. B. `groupMean` (*CamelCase*) oder `group_mean` (*snake case*)
- Komplexe Berechnungen in einzelne Schritte aufteilen, deren Zwischenergebnisse separat geprüft werden können.
- Mit dem `#` Zeichen Kommentare einfügen. Kommentare erläutern Befehle und erklären Analysen. Sie dienen anderen Personen dazu, die Bedeutung der Befehle und das Ziel von Auswertungsschritten schnell zu erfassen. Auch für die erstellende Person selbst sind Kommentare hilfreich, wenn Befehle längere Zeit nach Erstellen geprüft oder für eine andere Analyse angepasst werden sollen.

Bei falsch eingegebenen Befehlen bricht R die Auswertung ab und liefert eine Fehlermeldung, die meist einen Rückschluss auf die Ursache erlaubt. Einige typische Fehlerquellen bei der Arbeit mit R sind die folgenden:

- Groß- und Kleinschreibung sind relevant. Dies gilt für Objekte, Funktionsnamen und deren Argumente.
- Kann R einen richtig geschriebenen Funktionsnamen nicht finden, stammt die Funktion vermutlich aus einem Zusatzpaket, das zunächst mit `library()` geladen werden muss (Abschn. 1.3).

- Allgemein sollte bei Fehlern geprüft werden, ob Funktionsnamen und Argumente richtig geschrieben sind. Diese sind in R sehr inkonsistent benannt, so existieren etwa die folgenden Funktionen, deren Namen sich jeweils aus zwei Bestandteilen zusammensetzen: `as.Date()`, `read.table()`, `seq_along()`, `TukeyHSD()`, `zapsmall()`. Zwei Argumente von `read.table()` lauten `row.names` und `colClasses`. Durch die Unterscheidung von Groß- und Kleinschreibung ist das Fehlerpotential hier sehr hoch. Eine Übersicht über die genaue Schreibweise von Argumenten erhält man mit `args(<Funktionsname>)` oder über die Hilfe `?<Funktionsname>`. In grafischen Entwicklungsumgebungen wie RStudio lassen sich Fehler vermeiden, wenn man die Möglichkeiten zur automatischen Vervollständigung angefangener Befehlsnamen verwendet (Abschn. 1.2.1).
- Das Dezimaltrennzeichen ist immer der Punkt (2.173), nicht das Komma.
- Mehrere Argumente von Funktionen sind durch ein Komma voneinander zu trennen.
- Zeichenketten müssen fast immer in Anführungszeichen stehen.
- Alle öffnenden Klammern müssen auch geschlossen werden, dabei ist besonders auf die richtige Position der schließenden Klammer und auf den richtigen Typ (eckig oder rund) zu achten. Auch hier hilft eine Entwicklungsumgebung wie RStudio, die für jede geöffnete Klammer automatisch eine schließende einfügt und ein Klammernpaar farblich hervorhebt.

Abschnitt 4.4 diskutiert weiterführende Empfehlungen für reproduzierbare Datenauswertungen.

1.3 Zusatzpakete verwenden

R lässt sich über eigenständig entwickelte Zusatzkomponenten modular erweitern, die in Form von *Paketen* inhaltlich spezialisierte Funktionen zur Datenanalyse mit vorgefertigten Datensätzen und eigener Dokumentation bündeln (Ligges, 2003). Einen ersten Überblick liefern die thematisch geordneten und redaktionell gepflegten *Task Views* auf CRAN. Diese Übersicht führt etwa unter *Epidemiology* (Jombart, Rolland & Gruson, 2024) oder *Survival Analysis* (Allignol & Latouche, 2020) viele für Sozialwissenschaft und medizinische Forschung relevante Pakete an.²²

1.3.1 Zusatzpakete installieren: Grundlagen

Während einige Pakete bereits in einer R Basisinstallation enthalten sind, aber aus Effizienzgründen im Bedarfsfall erst explizit geladen werden müssen, sind die meisten Zusatzpakete zunächst manuell zu installieren. Auf Rechnern mit Online-Zugriff lassen sich Zusatzpakete aus RStudio über das Menü *Tools* → *Install Packages* von einem CRAN-server installieren. In der Konsole ist dafür der Befehl `install.packages("<Paketname>")` vorgesehen.

²²Der Lesbarkeit halber werden in diesem Buch vorgestellte Pakete nur bei ihrer ersten Verwendung auch zitiert, bei späteren Erwähnungen wird nur ihr Name genannt. Über den im Index markierten Haupteintrag für ein Paket lässt sich die Zitation finden.

```
> install.packages("colorspace")          # installiere Paket colorspace
# Ausgabe gekürzt ...
```

Der Paketname ist in Anführungszeichen eingeschlossen als erstes Argument zu nennen. Durch Angabe eines Vektors von Paketnamen wie `c("<Paket1>", "<Paket2>", ...)` (Abschn. 2.1.1) lassen sich auch mehrere Pakete in einem Aufruf installieren. Ein Paket kann in dem Sinn von anderen Paketen abhängen, dass es auf deren Funktionalität aufbaut und deshalb voraussetzt, dass sie ebenfalls installiert sind. `install.packages()` berücksichtigt diese Abhängigkeiten und installiert alle notwendigen Pakete automatisch.

R installiert Zusatzpakete entweder in einem Unterverzeichnis der R-Installation (*system library*), in einem vom Systemadministrator definierten Verzeichnis (*site library*)²³ oder aber in einem Unterverzeichnis des Heimverzeichnisses des Benutzers (*user library*). Letzteres ist die Voreinstellung für nachträglich hinzugefügte Pakete. Den Pfad zu den genannten Verzeichnissen zeigt `.libPaths()` an, den Pfad zu einem konkreten Paket `find.package("<Paket>")`.²⁴ `remove.packages()` deinstalliert ein Paket wieder.

```
> .libPaths()                      # Speicherorte von Zusatzpaketen
[1] "C:/Users/dwollsch/AppData/Local/R/win-library/4.4"
[2] "C:/Program Files/R/R-4.4.0patched/library"

> find.package("colorspace") # wo ist Paket colorspace gespeichert?
[1] "C:/Users/dwollsch/AppData/Local/R/win-library/4.4/colorspace"
```

Alle installierten Pakete lassen sich in RStudio über das Menü *Tools* → *Check for Package Updates* aktualisieren, sofern eine neue Version auf den CRAN-servern vorhanden ist. Diesem Vorgehen entspricht auf der Konsole der Befehl `update.packages()`.

1.3.2 Zusatzpakete installieren: Erweiterte Optionen

Mit dem Argument `repos` von `install.packages()` können temporär, mit `setRepositories()` auch dauerhaft nicht-CRAN-server als Paketquelle verwendet werden. In RStudio lassen sich Paketquellen in *Tools* → *Global Options* → *Packages* → *Secondary repositories* eintragen. In diesem Kontext ist etwa das BioConductor-Projekt (Gentleman et al., 2004; Huber et al., 2015) mit Paketen vor allem zur Bioinformatik zu nennen. Für die Installation von auf GitHub gehosteten Paketen eignet sich `install_github()` aus dem Paket `remotes` (Hester et al., 2020).

Während unter Windows und MacOS in der Voreinstellung Binärversionen der Pakete installiert werden, lädt R unter Linux den Quellcode der Pakete herunter und kompiliert diese dann lokal. Für die Ubuntu-Distribution existiert mit `r2u`²⁵ ein Service, der Binärversionen bereit hält, deren Installation weit weniger Zeit benötigt. Unter Windows können Pakete ebenfalls

²³Dafür 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.

²⁴Bei der Installation einer neuen R-Version müssen zuvor manuell hinzugefügte Pakete erneut installiert werden, wenn es sich um einen großen Versionssprung handelt, z. B. von Version 4.0.0 zu 4.1.0 – nicht aber von Version 4.0.0 zu 4.0.1.

²⁵<https://eddelbuettel.github.io/r2u/>

über das Argument `type="source"` von `install.packages()` aus dem Quellcode installiert werden. Dies kann in seltenen Fällen erforderlich sein, wenn CRAN keine aktuelle Binärversion anbietet. Für die Installation auf Basis des Quellcodes ist es unter Windows notwendig, auch die Programmsammlung Rtools²⁶ zu installieren.

Abschnitt 17.5.2, S. 696 zeigt, wie sich die Installation von vielen Paketen gleichzeitig beschleunigen lässt, indem man die Aufgabe auf mehrere Rechenkerne parallel verteilt. Das Zusatzpaket `pak` (Csárdi & Hester, 2024) hilft ebenfalls dabei, Pakete schneller herunterzuladen und zu installieren.

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` gewählt werden. Die alphabetisch geordnete Liste führt alle verfügbaren Zusatzpakete inkl. einer kurzen Beschreibung auf. Durch Anklicken eines Paketnamens öffnet sich die zugehörige Download-Seite. Sie enthält eine längere Beschreibung sowie u. a. den Quelltext des Pakets (Dateiendung `.tar.gz`), eine zur Installation unter Windows geeignete Archivdatei (Dateiendung `.zip`) sowie Dokumentation im PDF-Format, die u. a. die Funktionen des Zusatzpakets erläutert. Unter den Überschriften `Depends:` und `Imports:` sind die Abhängigkeiten eines Pakets aufgeführt. Bei manueller Installation eines Pakets müssen auch die Installationsdateien seiner Abhängigkeiten (und wiederum derer Abhängigkeiten) heruntergeladen werden.

Nachdem die Archivdatei eines Pakets auf den Zielrechner übertragen ist, lässt sie sich in RStudio über *Tools → Install Packages → Install from: Package Archive File* installieren. Auf der Konsole dient dazu `install.packages()` mit dem lokalen Pfad zur Paketdatei als erstem Argument (Abschn. 3.2) sowie `repos=NULL` als weiterem Argument.

1.3.3 Zusatzpakete laden

Damit die Funktionen und Datensätze eines installierten Zusatzpakets auch zur Verfügung stehen, muss es bei jeder neuen R-Sitzung manuell mit `library(<Paketname>)` geladen werden.²⁷

```
> library(colorspace) # Funktionen Paket colorspace verfügbar machen
```

Die installierten und damit ladbaren Zusatzpakete lassen sich auf der Konsole mit folgenden Befehlen auflisten:

```
installed.packages()                      # installierte Pakete auflisten  
library()                                # installierte Pakete auflisten
```

²⁶ <https://cloud.r-project.org/bin/windows/Rtools/>

²⁷ Wird versucht, ein nicht installiertes Paket zu laden, erzeugt `library()` einen Fehler. Mit dem Argument `logical.return=TRUE` erzeugt `library()` nur eine Warnung und gibt ein später zur Fallunterscheidung verwendbares `FALSE` zurück (Abschn. 17.1.1). Auch `require()` warnt nur, wenn ein zu ladendes Paket nicht vorhanden ist.

`installed.packages()` und `library()` zeigen ohne Angabe von Argumenten alle installierten und damit ladbaren Pakete samt ihrer Versionsnummer an, die in RStudio auch im *Packages* Tab aufgelistet sind. Ist ein Paket geladen, finden sich die in ihm enthaltenen Objekte in einer eigenen Umgebung wieder, die den Namen `package:<Paketname>` trägt, was am von `search()` ausgegebenen Suchpfad zu erkennen ist (Abschn. 1.2.3).²⁸

Kurzinformationen zu einem ladbaren Paket, etwa die darin enthaltenen Funktionen, liefert `help(package="<Paketname>")`. Viele Pakete bringen darüber hinaus noch ausführlichere Dokumentation mit, die `vignette(package="<Paketname>")` auflistet und mit `vignette("<Thema>", package="<Paketname>")` aufgerufen werden kann. Alle verfügbaren Themen können durch `vignette()` ohne Angabe von Argumenten angezeigt werden.

```
> help(package="colorspace")          # ...
> vignette(package="colorspace")      # ...
> vignette("hcl-colors", package="colorspace") # ...
```

Die Ausgabe von `loadedNamespaces()` zeigt, welche Pakete geladen sind. Über `detach(package:<Paketname>, unload=TRUE)` kann ein geladenes Paket auch wieder entfernt werden.

```
> loadedNamespaces()
[1] "colorspace" "compiler"   "graphics"   "tools"
[5] "utils"       "grDevices"  "stats"      "datasets"
[9] "methods"     "base"

> detach(package=colorspace)
```

Eine Übersicht darüber, welche Datensätze in einem bestimmten Zusatzpaket vorhanden sind, wird mit `data(package="<Paketname>")` geöffnet (Abschn. 2.11). 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. Datensätze aus Paketen sind mit einer kurzen Beschreibung ausgestattet, die `help(<Datensatz>)` ausgibt.

```
> data(package="colorspace")          # ...
> data()                            # ...
> help(max_chroma_table)            # ...
```

1.3.4 Hinweise zum Arbeiten mit Zusatzpaketen

Zusatzpakete sind eine wichtige Stärke von R. Ihr dynamisches Wachstum macht R sehr vielfältig und schnell im Umsetzen neuer statistischer Methoden oder anderer, für die Datenanalyse wichtiger Techniken. Manche Zusatzpakete versprechen konsistenter und einfacher anzuwendende Lösungen auch in Bereichen, die der Basisumfang von R zwar selbst abdeckt, dabei

²⁸Besitzen verschiedene geladene Pakete Funktionen desselben Namens, maskieren die aus später geladenen Paketen jene aus früher geladenen (Abschn. 1.4.1) – `library()` zeigt dies mit einer Warnmeldung an. Um explizit auf eine so maskierte Funktion zuzugreifen, ist dem Funktionsnamen der Paketname mit zwei Doppelpunkten voranzustellen, etwa `base::mean()`. Will man Maskierungen verhindern, um nicht versehentlich mit der falschen Funktion zu arbeiten, kann man die globale Option `options(conflicts.policy="strict")` setzen (Abschn. 1.2.2). Bei Maskierung bricht `library()` dann mit einer Fehlermeldung ab.

evtl. umständlicher ist. Dies gilt etwa für den Umgang mit Zeichenketten (Abschn. 2.14) und Datumsangaben (Abschn. 2.15), für die Aufbereitung von Datensätzen (Abschn. 3.4) sowie für Diagramme (Kap. 15).

Sich in der Datenauswertung auf Zusatzpakete zu verlassen, birgt Risiken: Über CRAN verteilte Pakete durchlaufen zwar viele Tests, die einen technischen Mindeststandard garantieren. Trotzdem ist die Qualität der Pakete sehr heterogen. Die Dynamik der Zusatzpakete ist gleichzeitig ein Nachteil: Während der Basisumfang von R so ausgereift ist, dass darauf aufbauende Auswertungen wahrscheinlich noch lange ohne Änderungen mit neuen R Versionen nutzbar bleiben, ist dies bei auf Paketen basierenden Lösungen nicht im selben Maße sicher. Ihre Syntax und Funktionalität kann häufig versionsabhängig wechseln, was ihre Verwendung weniger zukunftssicher und reproduzierbar macht. Mögliche Auswege stellt Abschn. 4.4 vor.

In diesem Buch finden sich viele Hinweise auf Erweiterungsmöglichkeiten durch Zusatzpakete. Die Darstellung konzentriert sich aber auf den ausgereiften und stabilen Basisumfang von R.

1.4 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 (Chambers, 2016; Wickham, 2019a).

Daten werden in R in benannten Objekten gespeichert. Jedes Objekt ist eine konkrete Verkörperung (*Instanz*) einer der genannten 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 Funktion `inherits(<Objekt>, "<Klasse>")` prüft, ob ein vorliegendes Objekt von einer gewissen Klasse ist.

```
> class(mtcars)                      # Datensatz aus Basisumfang von R
[1] "data.frame"

> inherits(euro.cross, "matrix")      # Matrix aus Basisumfang von R
[1] TRUE
```

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 `as.vector(<Objekt>)` zu benutzen. Mehr Informationen zu diesem Thema finden sich in späteren Abschnitten, die die einzelnen Klassen behandeln.

```
> as.vector(euro.cross)              # wandle Matrix in Vektor um
[1] 1.0000000 0.3411089 7.0355300 # gekürzte Ausgabe ...
```

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

```
> length(euro)                      # Vektor aus Basisumfang von R
[1] 11
```

Objekte besitzen darüber hinaus einen Datentyp (*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 (Abschn. 1.4.5). Ein Objekt der Klasse `matrix` könnte also z. B. mehrere Wahrheitswerte speichern und somit den Datentyp `logical` besitzen.

```
> mode(euro.cross)                  # Matrix aus Basisumfang von R
[1] "numeric"
```

Ein Objekt kann zudem *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 sowie über `attr()` auch geändert werden.²⁹ Meist versieht R von sich aus Objekte mit Attributen, so kann etwa die Klasse eines Objekts im Attribut `class` gespeichert sein. Man kann Attribute aber auch selbst i. S. einer freien Beschreibung nutzen, die man mit einem Objekt assoziieren möchte – hierfür eignet sich alternativ auch `comment()`.

```
> attributes(euro)                # Vektor aus Basisumfang von R
$names
[1] "ATS" "BEF" "DEM" "ESP" "FIM" "FRF" "IEP" "ITL" "LUF" "NLG" "PTE"

> attr(euro, which="names")
[1] "ATS" "BEF" "DEM" "ESP" "FIM" "FRF" "IEP" "ITL" "LUF" "NLG" "PTE"
```

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

```
> str(euro.cross)                  # Matrix aus Basisumfang von R
num [1:11, 1:11] 1 0.3411 7.0355 0.0827 2.3143 ...
- attr(*, "dimnames")=List of 2
..$ : chr [1:11] "ATS" "BEF" "DEM" "ESP" ...
..$ : chr [1:11] "ATS" "BEF" "DEM" "ESP" ...
```

1.4.1 Objekte benennen

Objekte tragen i. d. R. einen Namen (ihr *Symbol*) beliebiger Länge, über den sie in Befehlen identifiziert werden. Objektnamen sollten mit einem Buchstaben beginnen, können aber ab der zweiten Stelle neben Buchstaben auch Ziffern, Punkte und Unterstriche enthalten. Von der

²⁹Mit `structure()` lassen sich auch mehrere Attribute gleichzeitig setzen, ohne dass dabei schon vorhandene Attribute verändert würden.

Verwendung anderer Sonderzeichen wie auch von Umlauten ist abzuraten, selbst wenn dies bisweilen möglich ist.³⁰ 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 Hilfe-Seite `?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 *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 Umgebungen gibt, die Objekte desselben Namens beinhalten können, ohne dass diese sich wechselseitig überschreiben würden (Abschn. 1.2.3). Ob Namenskonflikte, also mehrfach vergebene Objektnamen, vorliegen, kann mit `conflicts(detail=TRUE)` geprüft werden. `exists("<Name>")` gibt an, ob `<Name>` schon als Symbol verwendet wird.

```
> conflicts(detail=TRUE) # doppelt vergebene Symbole
$`package:methods`
[1] "body<-"      "kronecker"

$`package:base`
[1] "body<-"      "kronecker"

> exists("mean")          # Objekt mean ist schon vorhanden
[1] TRUE
```

1.4.2 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 zwei gängige Arten geschehen:

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

Zur Vermeidung von Mehrdeutigkeiten bei komplizierteren Eingaben sollte die erste Methode mit `<-` bevorzugt werden. Das vielleicht vertrautere `=` sollte der Zuweisung von Funktionsargumenten vorbehalten bleiben (Abschn. 1.2.5), um die Richtung der Zuweisung eindeutig zu halten. Im Fall des `<Objekt> <- <Wert>` Operators erfolgt die Zuweisung von rechts nach links in Richtung des Pfeils.

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
```

³⁰Wenn ein Objektname dennoch nicht zulässige Zeichen enthält, kann man nichtsdestotrotz auf das Objekt zugreifen, indem man den Namen in rückwärts gerichtete Hochkommata setzt: ``<Objektname>``

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

1.4.3 Objekte ausgeben

Bei Zuweisungen zu Objekten gibt R den neuen Wert nicht aus, der letztlich im Zielobjekt gespeichert wurde. Um sich den Inhalt eines Objekts anzeigen zu lassen, gibt es folgende Möglichkeiten:

```
> print(x1)           # print(<Objektname>) Funktion
> get("x1")          # get("<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
```

Die Variante `get("<Objektname>")` eignet sich besonders für Situationen, in denen der Name des auszugebenden Objekts nur als Zeichenkette in einem anderen Objekt gespeichert ist und deshalb nicht von Hand eingetippt werden kann.³¹

```
> varName <- "x1"      # Variable, die Objektnamen enthält
> get(varName)         # gewünschtes Objekt ausgeben
[1] 4.5
```

Um mit Zwischenergebnissen weiterzurechnen, sollten diese nicht auf der Konsole ausgegeben, dort abgelesen und später von Hand als fester Wert in einer Rechnung eingesetzt werden. Dies würde zum einen dazu führen, dass die Genauigkeit beim Rechnen mit Dezimalzahlen unnötig auf die Anzahl ausgegebener Dezimalstellen begrenzt wird. Zum anderen verhindert ein solches Vorgehen, dass die erstellten Befehle auf neue Situationen übertragbar sind, in denen sich nicht exakt dasselbe Zwischenergebnis einstellt. Stattdessen sollten Zwischenergebnisse immer einem eigenen Objekt zugewiesen werden, das dann in späteren Rechnungen auftauchen kann.

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.

1.4.4 Objekte anzeigen lassen, umbenennen und entfernen

Um sich einen Überblick über alle im workspace vorhandenen Objekte zu verschaffen, dient `ls()` (*list*). Objekte, deren Name mit einem Punkt beginnt, sind dabei versteckt – sie werden erst mit `ls(all.names=TRUE)` angezeigt. Zusatzinformationen zu jedem Objekt zeigt `ls.str()`. In RStudio sind diese auch im *Environment* Tab sichtbar.

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

> ls.str()
x1 : num 4.5
```

³¹Um analog Objekte mit einem später festgelegten Namen zu erstellen, s. Abschn. 1.4.4.

```
x2 : num 10
x3 : num 10
```

Um Objekte umzubenennen gibt es zwei Möglichkeiten: Zum einen kann das alte Objekt wie beschrieben einem neuen, passend benannten Objekt zugewiesen werden. Ergibt sich der gewünschte Name dagegen aus dem Inhalt einer Variable und kann deshalb nicht von Hand eingetippt werden, ist es mit `assign("<Name>", value=<Objekt>")` möglich, einem neuen Objekt den Inhalt eines bereits bestehenden Objekts `value` zuzuweisen.

```
> varNew1      <- x1          # Variable unter neuem Namen speichern
> newNameVar <- "varNew2"    # Variable, die neuen Objektnamen enthält
> assign(newNameVar, x1)     # weise neuem Objekt Wert von x1 zu
> varNew2                  # Objekt mit neuem Namen
[1] 4.5
```

Vorhandene Objekte können mit `rm(<Objekt>)` (*remove*) gelöscht werden. Sollen alle bestehenden Objekte entfernt werden, kann dies mit dem Befehl `rm(list=ls(all.names=TRUE))` oder in RStudio über das Besen-icon *Clear* im *Environment* Tab geschehen.

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

1.4.5 Datentypen

Der Datentyp eines Objekts bezieht sich auf die Art der in ihm gespeicherten Informationen und lässt sich mit `mode(<Objekt>)` ausgeben. Neben den in Tab. 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
komplexe Zahlen	3.14 + 1i	complex
Buchstaben und Zeichenfolgen (immer in Anführungszeichen einzugeben) ³²	"Hello"	character

³²Dies 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 *Escape-Sequenz* mit vorangestelltem *backslash* zu schreiben sind ("a\"b", vgl. `?Quotes`).

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

Für Zahlen (`numeric`) existieren u. a. zwei Möglichkeiten, sie in einem Computer intern zu repräsentieren: Ganze Zahlen können mit einem L hinter der Zahl gekennzeichnet werden (*long integer*, z. B. `5L`), wodurch R sie dann auch als solche speichert (`integer`). Andernfalls werden alle Zahlen in R als Gleitkommazahlen mit doppelter Genauigkeit gespeichert (`double`). Die Speicherart 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()`). Weitere Angaben zur internen Implementierung von Zahlen und den daraus resultierenden Beschränkungen gibt `.Machine` aus, etwa die größtmögliche ganze Zahl `.Machine$integer.max` oder die kleinste positive Gleitkommazahl, die noch von 0 unterscheidbar ist `.Machine$double.eps`.

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

So wie Objekte einer bestimmten Klasse in Objekte einer anderen Klasse umgewandelt werden können, 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 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 der Umwandlung von Datentypen besteht eine Hierarchie entsprechend der in Tab. 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 speichern (1.23 ist gleich $1.23 + 0i$), jeder logische 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 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 (Abschn. 2.13). Analog führt `as.logical("abc")`

zum Ergebnis NA, während `as.logical("TRUE")` und `as.logical("FALSE")` in TRUE bzw. FALSE umwandelbar sind.

1.4.6 Logische Werte, Operatoren und Verknüpfungen

Das Ergebnis eines logischen Vergleichs mit den in Tab. 1.3 genannten Operatoren sind Wahrheitswerte, also WAHR (TRUE) oder FALSCH (FALSE).³³ Die Abkürzungen T und F sollten nicht verwendet werden, da sich auch normale Variablen mit diesem Namen erstellen lassen. Wahrheitswerte lassen sich auch in numerischen Rechnungen nutzen, dem Wert TRUE entspricht dann die 1, dem Wert FALSE die 0.

Tabelle 1.3: Logische Operatoren, Funktionen und Konstanten

Name	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)
TRUE, FALSE (T, F)	logische Wahrheitswerte: WAHR, FALSCH (abgekürzt)

```

> TRUE == TRUE      > TRUE == FALSE     > !TRUE          > !FALSE
[1] TRUE            [1] FALSE           [1] FALSE         [1] TRUE

> TRUE != TRUE      > TRUE != FALSE    > TRUE != TRUE   > TRUE != FALSE
[1] FALSE           [1] TRUE            [1] FALSE          [1] TRUE

> 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

> xor(TRUE, FALSE) > xor(TRUE, TRUE)  > 4 <= 8        > 7 < 3
[1] TRUE            [1] FALSE           [1] TRUE           [1] FALSE

```

Auch für Zeichenketten sind die Relationen `<` und `>` entsprechend der alphabetischen Reihenfolge definiert.³⁴

```

> "A" < "B"
[1] TRUE

```

³³Für Hilfe zu diesem Thema vgl. `?Logic` sowie `?Comparison`.

³⁴Die alphabetische Reihenfolge hängt dabei von den Ländereinstellungen ab, die sich mit `Sys.getlocale()` erfragen und mit `Sys.setlocale()` ändern lässt.

Statt des logischen Vergleichsoperators `==` kann zum Prüfen zweier Objekte `x` und `y` auf exakte Gleichheit auch `identical(x, y)` eingesetzt werden. Dabei ist zu beachten, dass `identical()` anders als `==` die Gleichheit nur dann bestätigt, wenn `x` und `y` in R auch intern auf dieselbe Weise repräsentiert sind, etwa denselben Datentyp besitzen.

```
# FALSE, da 4L ganzzahlig, 4 aber als Gleitkommazahl repräsentiert ist
> identical(4L, 4)
[1] FALSE

> 4L == 4                                # TRUE, da bezeichneter Wert identisch
[1] TRUE
```

Die Funktion `isTRUE(<Objekt>)` prüft komplexere Objekte darauf, ob sie dem Wert `TRUE` entsprechen. Ihr Ergebnis ist in jedem Fall ein einzelner Wert – entweder `TRUE` oder `FALSE`. Auch Objekte, die möglicherweise Werte mit undefiniertem Wahrheitsstatus enthalten, lassen sich so eindeutig prüfen, ob sie `TRUE` sind. Dazu zählen fehlende Werte (`NA`, Abschn. 2.13), `NaN`, `NULL`, sowie Vektoren der Länge 0. Dies ist insbesondere bei Fallunterscheidungen mit `if()` relevant (Abschn. 17.1.1).

```
> NaN == TRUE
[1] NA

> NULL == TRUE
logical(0)

> isTRUE(NaN)
[1] FALSE

> isTRUE(NULL)
[1] FALSE
```

Anders als `==` bestätigt die `all.equal()` Funktion die Gleichheit zweier Objekte auch dann, wenn sie sich minimal unterscheiden. Ihre Verwendung ist dann zu empfehlen, wenn entschieden werden soll, ob zwei Dezimalzahlen denselben Wert haben.

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

Für `target` und `current` sind die zu vergleichenden Objekte zu nennen. In der auf `TRUE` gesetzten Voreinstellung für `check.attributes` berücksichtigt die Prüfung auf Gleichheit auch die Attribute eines Objekts, zu denen insbesondere die Benennungen einzelner Werte zählen (Abschn. 1.4, 2.1.4). Die Objekte `target` und `current` gelten auch dann als gleich, wenn ihre Werte nur ungefähr, d. h. mit einer durch `tolerance` festgelegten Genauigkeit übereinstimmen. Aufgrund der Art, in der Computer Gleitkommazahlen intern speichern und verrechnen, sind kleine Abweichungen in Rechenergebnissen nämlich schon bei harmlos wirkenden Ausdrücken möglich. So ergibt der Vergleich `0.1 + 0.2 == 0.3` fälschlicherweise `FALSE`. `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 0.³⁵ Dies sind keine R-spezifischen Probleme, sie können nicht allgemein verhindert werden ([Cowlishaw, 2008](#); [Goldberg, 1991](#)).

`all.equal()` liefert im Fall der Ungleichheit nicht `FALSE` zurück, sondern ein Maß der relativen Abweichung. Wird ein einzelner Wahrheitswert als Ergebnis benötigt, muss `all.equal()` deshalb mit `isTRUE()` verschachtelt werden.

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

> 0.123400001 == 0.123400000
[1] FALSE

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

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

³⁵Tauchen sehr kleine Zahlen, die eigentlich 0 sein sollten, zusammen mit größeren Zahlen in einem Ergebnis auf, eignet sich `zapsmall()`, um sie i.S. einer besseren Übersichtlichkeit auch tatsächlich als 0 ausgeben zu lassen.

Kapitel 2

Elementare Dateneingabe und -verarbeitung

Die folgenden Abschnitte sollen gleichzeitig die grundlegenden Datenstrukturen in R sowie Möglichkeiten zur deskriptiven Datenauswertung erläutern. Die Reihenfolge der Themen ist dabei so gewählt, dass die abwechselnd vorgestellten Datenstrukturen und darauf aufbauenden deskriptiven Methoden nach und nach an Komplexität gewinnen.

2.1 Vektoren

R ist auf die Verarbeitung von in Vektoren angeordneten Daten ausgerichtet. 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 sequentiell anzuordnende Wertemenge betrachten lassen, sind Vektoren als Organisationsform gut für die Datenanalyse geeignet. Vektoren sind in R die einfachste Datenstruktur für Werte, d. h. auch jeder 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>)`.¹ Die Elemente des Vektors werden hierbei auf eine Voreinstellung gesetzt, die vom Datentyp abhängt – 0 für `numeric()`, "" für `character` und FALSE für `logical`.

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

> character(2)
[1] "" "
```

Als häufiger genutzte Alternative lassen sich Vektoren auch mit der Funktion `c(<Wert1>, <Wert2>, ...)` erstellen (*concatenate*), die die Angabe der zu speichernden Werte benötigt. Ein das Alter von sechs Personen speichernder Vektor könnte damit so erstellt werden:

¹Ein leerer Vektor entsteht analog, z. B. durch `numeric(0)`. Auf 32bit-Systemen kann ein Vektor höchstens `.Machine$integer.max` viele $(2^{31} - 1)$ Elemente enthalten, auf heutigen 64bit-Systemen jedoch gut 2^{50} . Mit `vector(mode=<Klasse>, length=<Länge>)` lassen sich beliebige Objekte der für `mode` genannten Klasse der Länge `length` erzeugen.

```
> (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 `c()` zu nutzen, wobei statt eines einzelnen Wertes auch der Name eines schon existierenden Vektors angegeben werden kann.

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

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

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

Auch Zeichenketten können die Elemente eines Vektors ausmachen. Dabei zählt die leere Zeichenkette "" ebenfalls als ein Element.

```
> (chars <- c("lorem", "ipsum", "dolor", ""))
[1] "lorem" "ipsum" "dolor" ""
```

```
> length(chars)
[1] 4
```

Mit `LETTERS` und `letters` sind zwei aus Zeichen bestehende Vektoren bereits vordefiniert, 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"
```

2.1.2 Elemente auswählen und verändern

Um ein einzelnes Element eines Vektors abzurufen, wird seine Position im Vektor (sein Index) in eckigen Klammern hinter dem Objektnamen angegeben. Dies ist der `[<Index>]` Operator². Indizes beginnen bei 1 für die erste Position³ und enden bei der Länge des Vektors. Werden größere Indizes verwendet, erfolgt als Ausgabe die für einen fehlenden Wert stehende Konstante `NA` (Abschn. 2.13).

²Für Hilfe zu diesem Thema s. `?Extract`. Auch der Index-Operator ist eine Funktion, kann also gleichermaßen in der Form `"["(<Vektor>, <Index>)` verwendet werden (Abschn. 1.2.5, Fußnote 20).

³Dies mag selbstverständlich erscheinen, in anderen Sprachen wird jedoch oft der Index 0 für die erste Position und allgemein der Index $i - 1$ für die i -te Position verwendet. Für einen Vektor `x` ist das Ergebnis von `x[0]` immer ein leerer Vektor mit demselben Datentyp wie jener von `x`.

```
> age[4]                                # 4. Element von age
[1] 24

> (ageLast <- age[length(age)])          # letztes Element von age
[1] 21

> age[length(age) + 1]                  # Index > Länge des Vektors
[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 man zunächst einen eigenen Vektor erstellen, 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)                      # Indexvektor separat
> age[idx]
[1] 18 20 24

> age[c(3, 5, 6)]                        # Indexvektor im Operator
[1] 30 23 21

# Elemente mehrfach auswählen
> 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 erreichen, indem der Index des nicht erwünschten Elements mit negativem Vorzeichen in die eckigen Klammern geschrieben wird.⁴ Sollen mehrere Elemente nicht ausgegeben werden, verläuft der Aufruf analog zum Aufruf gewünschter Elemente, wobei mehrere Variationen mit dem negativen Vorzeichen möglich sind.

⁴Als Indizes dürfen in diesem Fall keine fehlenden Werte (`NA`) oder Indizes mit positivem Vorzeichen vorkommen, ebenso darf der Indexvektor nicht leer sein.

```
> 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      # speichere 25 an Position 4
> 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 neue Werte zuweisen. Dabei müssen die zugewiesenen Werte ebenfalls durch einen Vektor repräsentiert sein. Fehlt bei Zuweisungen der Index `<Vektor>[]`, werden alle Elemente des Vektors ersetzt. Wenn der zugewiesene Vektor dabei weniger Elemente als der veränderte Vektor besitzt, wird er automatisch passend verlängert (Abschn. 2.5.4).

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

# alle Elemente gleichzeitig ersetzen mit zyklischer Verlängerung
> age[] <- c(1, 2)    # nicht verwechseln mit: age <- c(1, 2)
> age
[1] 1 2 1 2 1 2
```

Um Vektoren zu verlängern, also mit neuen Elementen zu ergänzen, kann zum einen der `[<Index>]` Operator benutzt werden, wobei als Index nicht belegte Positionen angegeben werden.⁵ Zum anderen kann auch hier `c(<Wert1>, <Wert2>, ...)` Verwendung finden. Als Alternative steht die `append(<Vektor>, values=<Vektor>)` Funktion zur Verfügung, die an einen Vektor die Werte eines unter `values` genannten Vektors anhängt und den erweiterten Vektor zurückgibt.

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

⁵Bei 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 und dem Datentyp angelegt werden, die sie später benötigen.

```
[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"
```

Über die Option `append(..., after=Index)` können neue Werte auch an beliebiger Stelle eingefügt werden, mit `after=0` am Beginn des Vektors.

```
> append(charVec1, c("A", "B", "C"), after=0)
[1] "A" "B" "C" "Z" "Y" "X" "W" "V" "U"
```

2.1.3 Datentypen in Vektoren

Vektoren können Werte unterschiedlicher Datentypen speichern, etwa `numeric`, wenn sie Zahlen beinhalten, oder `character` im Fall von Zeichenketten. Letztere müssen dabei immer in Anführungszeichen stehen. Jeder Vektor kann aber nur einen Datentyp besitzen, alle seine Elemente haben also denselben Datentyp. Fügt man einem numerischen Vektor eine Zeichenkette hinzu, so werden seine numerischen Elemente automatisch in Zeichenketten umgewandelt,⁶ 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"
```

2.1.4 Elemente benennen

Es ist möglich, die Elemente eines Vektors beim Erstellen zu benennen. Die Elemente können dann nicht nur über ihren Index, sondern auch über ihren in Anführungszeichen gesetzten Namen angesprochen werden.⁷ In der Ausgabe wird der Name eines Elements in der über ihm stehenden Zeile mit aufgeführt. `names(<Vektor>)` gibt die Namen der Elemente eines Vektors aus.

⁶Allgemein gesprochen werden alle Elemente in den umfassendsten Datentyp umgewandelt, der notwendig ist, um alle Werte ohne Informationsverlust zu speichern (Abschn. 1.4.5).

⁷Namen werden als Attribut gespeichert und sind mit `attributes(<Vektor>)` sichtbar (Abschn. 1.4). Elemente lassen sich über den Namen nur auswählen, nicht aber mittels `<Vektor>[-"<Name>"]` ausschließen, hierfür bedarf es des numerischen Index.

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

> namedVec1["elem1"]
elem1
"first"

> names(namedVec1)
[1] "elem1"   "elem2"
```

Auch im nachhinein lassen sich Elemente benennen, bzw. vorhandene Benennungen ändern, etwa mit der Funktion `setNames(<Vektor>, c("<Name1>", ...))`. Sie setzt die Namen der Elemente des übergebenen Vektors auf die Zeichenketten, die als zweites Argument in Form eines Vektors genannt sind. Die Funktion verändert den als erstes Argument übergebenen Vektor selbst nicht, sondern liefert einen modifizierten Vektor zurück.

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

> (names2 <- names(namedVec2))                      # alte Namen
[1] "val1" "val2" "val3"

> (namedVec3 <- setNames(namedVec2, toupper(names2)))  # neue Namen
VAL1  VAL2  VAL3
10    -12    33
```

Ganz entfernt werden die Namen der Elemente mit `unname(<Vektor>)`.

```
> unname(namedVec3)
[1] 10  -12  33
```

2.1.5 Elemente löschen

Elemente eines Vektors lassen sich nicht im eigentlichen Sinne löschen. Denselben Effekt kann man stattdessen über zwei mögliche Umwege erzielen. 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 `length()` 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
```

2.2 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 (Abschn. 2.2.2).

2.2.1 Vektoren mit logischen Operatoren vergleichen

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 Personen gespeichert. Zunächst sollen jene Personen identifiziert werden, die jünger als 24 Jahre sind. Dazu wird der `<` Operator verwendet, der als Ergebnis einen Vektor mit Wahrheitswerten liefert, der für jedes Element separat angibt, ob die Bedingung `< 24` zutrifft. Andere Vergleichsoperatoren, wie gleich (`==`) oder ungleich (`!=`) funktionieren analog.

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

Wenn 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, ist das Symbol `|` für das logische, d. h. einschließende, ODER zu verwenden. 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

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

UND und ODER dürfen bei zusammengesetzten Prüfungen nicht weggelassen werden: Während man mathematisch also eine Bedingung etwa als $0 \leq x \leq 10$ formulieren würde, müsste sie in R in Form von mit UND verbundenen Einzelprüfungen geschrieben werden, also wie oben als `(0 <= x) & (x <= 10)`.

Die elementweise Prüfung von Vektoren macht den häufigsten Fall der Anwendung logischer Kriterien aus. Vor allem zur Fallunterscheidung (Abschn. 17.1.1) sind aber auch Prüfungen notwendig, die in Form eines einzelnen Wahrheitswertes eine summarische Auskunft darüber liefern, ob Kriterien erfüllt sind. Diese Prüfungen lassen sich mit `&&` für das logische UND bzw. mit `||` für das logische ODER formulieren. Zu beiden Seiten des Operators darf dabei jedoch nur ein einzelner Wert stehen.

```
> TRUE && TRUE
[1] TRUE
```

```
> FALSE || FALSE
[1] FALSE
```

`identical()` prüft zwei übergebene Vektoren summarisch auf Gleichheit und gibt nur dann das Ergebnis `TRUE` aus, wenn diese auch bzgl. ihrer internen Repräsentation exakt identisch sind (Abschn. 1.4.6).

```
> c(1, 2) == c(1L, 2L)
[1] TRUE TRUE

> identical(c(1, 2), c(1L, 2L))
[1] FALSE
```

Sollen Werte nur auf ungefähre Übereinstimmung geprüft werden, kann dies mit `all.equal()` geschehen (Abschn. 1.4.6). Dabei ist im Fall von zu vergleichenden Vektoren zu beachten, dass die Funktion keinen Vektor der Ergebnisse der elementweisen Einzelvergleiche 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 `isTRUE()` 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.⁸

```
> 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 auf einen logischen Vektor die Funktion `sum(<Vektor>)` angewendet, die alle Werte des Vektors aufaddiert (Abschn. 2.7.1).

```
> res <- age < 24
> sum(res)                      # TRUE zählt als 1, FALSE als 0
[1] 3
```

Alternativ kann verschachtelt in `length()` die Funktion `which(<Vektor>)` genutzt werden, die die Indizes der Elemente mit dem Wert TRUE ausgibt (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 wie numerische Indexvektoren zur Indizierung anderer Vektoren benutzt werden. Diese Art zu indizieren kann z. B. zur Auswahl von Teilstichproben genutzt werden, die durch bestimmte Merkmale definiert sind. Hat ein Element des logischen

⁸Dabei erzeugt `all(numeric(0))` das Ergebnis TRUE, da die Aussage „alle Elemente des leeren Vektors sind WAHR“ logisch WAHR ist – schließlich lässt sich kein Gegenbeispiel in Form eines Elements finden, das FALSCH wäre. Dagegen erzeugt `any(numeric(0))` das Ergebnis FALSE, da in einem leeren Vektor nicht mindestens ein Element existiert, das WAHR ist.

Indexvektors den Wert **TRUE**, so wird 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 [$\langle \text{Index} \rangle$] Operators benutzt werden. Dabei ist jedoch abzuwegen, ob der Übersichtlichkeit und Nachvollziehbarkeit der Befehle mit einer separaten Erstellung von Indexvektoren besser gedient ist.

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

```
> age[idx]
[1] 17 30 30
```

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

Bei logischen Indexvektoren kann anders als bei numerischen Indexvektoren die von R automatisch vorgenommene zyklische Verlängerung greifen (Abschn. 2.5.4): Logische Indexvektoren mit weniger Elementen als jene des indizierten Vektors werden durch zyklische Wiederholung soweit verlängert, dass sie mindestens die Länge des indizierten Vektors erreichen.

```
> age[c(TRUE, FALSE)]           # logischer Indexvektor kürzer als age
[1] 18 30 23
```

```
# durch zyklische Verlängerung von c(TRUE, FALSE) äquivalent zu
> age[c(TRUE, FALSE, TRUE, FALSE, TRUE, FALSE)]
[1] 18 30 23
```

Logische Indexvektoren bergen den Nachteil, dass sie zu Problemen führen können, 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 (Abschn. 2.13.3, Fußnote 56).

```
> vecNA <- c(-3, 2, 0, NA, -7, 5)      # Vektor mit fehlendem Wert
> (ok    <- 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 (Abschn. 2.1.2). Dies führt dazu, dass sich der Indexvektor nicht mehr dazu eignet, ausschließlich die Werte auszugeben, die eine bestimmte Bedingung erfüllen. Fehlende Werte im logischen Indexvektor können mit **is.na()** (Abschn. 2.13.1) identifiziert und dann logisch ausgeschlossen werden.

```
> vecNA[ok]                                # Auswahl mit ok erzeugt NA
[1] 2 NA 5

> vecNA[ok & !is.na(ok)]                  # entferne NA im Indexvektor
[1] 2 5
```

Logische Indizes in numerische Indizes wandelt die Funktion `which(<Vektor>)` um. Sie gibt also die Positionen der TRUE Werte zurück.⁹

```
> (numIdx <- which(ok))                  # numerische Indizes der TRUE Werte
[1] 2 6
```

Numerische Indexvektoren sind dann zu vermeiden, wenn Werte explizit ausgeschlossen werden sollen. Mit dem Vektor `x` und einem logischen Indexvektor `idx` kann anders als `x[!idx]` die Variante `x[-which(idx)]` unerwünschte Ergebnisse haben: Sind in `idx` keine TRUE Werte vorhanden, werden nicht alle Werte von `x` ausgegeben, sondern gar keiner, weil dies dann gleichbedeutend zu `x[integer(0)]` ist.

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, so ist z. B. die Menge $\{1, 1, 2, 2\}$ gleich der Menge $\{2, 1\}$.¹⁰

2.3.1 Doppelt auftretende Werte finden

`duplicated(<Vektor>)` gibt für jedes Element eines Vektors an, ob der Wert bereits an einer früheren Stelle des Vektors aufgetaucht ist. Mit dem Argument `fromLast=TRUE` beginnt die Suche nach wiederholt auftretenden Werten am Ende des Vektors.

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

> duplicated(x, fromLast=TRUE)            # umgekehrte Suchrichtung
[1] TRUE FALSE FALSE FALSE FALSE FALSE
```

Durch die Kombination beider Suchrichtungen lassen sich alle Duplikate identifizieren, insbesondere also auch das erste Auftreten eines mehrfach vorhandenen Wertes – so umgesetzt in `AllDuplicated()` aus dem Paket `DescTools` ([Signorell, 2020](#)).

⁹Umgekehrt lassen sich auch die in `<Indexvektor>` gespeicherten numerischen Indizes für `<Vektor>` in logische verwandeln: `seq_along(<Vektor>) %in% <Indexvektor>` (Abschn. [2.3.2, 2.4.1](#)).

¹⁰Das Paket `sets` ([D. Meyer & Hornik, 2009](#)) stellt eine eigene Klasse zur Repräsentation von Mengen zur Verfügung und implementiert auch einige hier nicht behandelte Mengenoperationen – etwa das Bilden der Potenzmenge.

```
# markiere alle Stellen, an denen die 1 auftritt
> duplicated(x) | duplicated(x, fromLast=TRUE)
[1] TRUE FALSE FALSE TRUE FALSE FALSE
```

`unique(<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 einer Variable.

```
> 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

`union(x=<Vektor1>, y=<Vektor2>)` bildet die Vereinigungsmenge $x \cup y$. 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 es keine Rolle, in welcher Reihenfolge `x` und `y` genannt werden.

```
> 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 $x \cap y$ zweier Mengen erzeugt `intersect(x=<Vektor1>, y=<Vektor2>)`. 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
```

`setdiff(x=<Vektor1>, y=<Vektor2>)` liefert als Ergebnis all jene Elemente von `x`, die nicht Element von `y` sind. Im Unterschied zu den oben behandelten Mengenoperationen ist die Reihenfolge von `x` und `y` hier bedeutsam, auch wenn das Ergebnis als Menge betrachtet wird. Die symmetrische Differenz von `x` und `y` erhält man durch `union(setdiff(x, y), setdiff(y, x))`.

```
> 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 `is.element(el=<Vektor>, set=<Menge>)` genutzt werden. Unter `el` ist der Vektor mit den zu prüfenden Elementen einzutragen und unter `set` die durch einen Vektor definierte Menge. Als Ergebnis wird ein logischer Vektor ausgegeben, der für jedes Element von `el` angibt, ob es in `set` enthalten ist. Die Kurzform in Operator-Schreibweise lautet `<Vektor> %in% <Menge>`.

```
> 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(x %in% y)` lässt sich prüfen, ob `x` eine Teilmenge von `y` darstellt, ob also jedes Element von `x` auch Element von `y` ist. Dabei ist `x` eine echte Teilmenge von `y`, wenn sowohl `all(x %in% y)` gleich TRUE als auch `all(y %in% x)` 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
```

2.3.3 Kombinatorik

Aus dem Bereich der Kombinatorik sind bei der Datenauswertung drei Themen von Bedeutung: Zunächst ist dies die Bildung 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 *Kombination*. Werden alle Elemente einer Grundmenge ohne Zurücklegen unter Beachtung der Reihenfolge gezogen, handelt es sich um eine vollständige *Permutation*. Schließlich kann die Zusammenstellung von Elementen aus verschiedenen Grundmengen notwendig sein, wobei jeweils ein Element aus jeder Grundmenge beteiligt sein soll.

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. Es gibt $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ viele k -Kombinationen. Da eine k -Kombination die Anzahl der Möglichkeiten darstellt, aus einer Menge mit n Elementen k auszuwählen, spricht man im Englischen beim Binomialkoeffizienten $\binom{n}{k}$ von „ n choose k “. Daraus leitet sich der Name der `choose(n=<Zahl>, k=<Zahl>)` Funktion ab, die $\binom{n}{k}$ ermittelt. Die Fakultät einer Zahl wird mit `factorial(<Zahl>)` berechnet (Abschn. 2.7.1).

```
> myN <- 5
> myK <- 4
> choose(myN, myK)
[1] 5

> factorial(myN) / (factorial(myK)*factorial(myN-myK))      # Kontrolle
[1] 5
```

Möchte man alle k -Kombinationen einer gegebenen Grundmenge x auch explizit anzeigen lassen, kann dies mit `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 (Abschn. 2.10). Stattdessen wird hier eine Matrix ausgegeben, die in jeder Spalte eine der k -Kombinationen enthält (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"
```

`combn()` lässt sich darüber hinaus anwenden, um in einem Arbeitsschritt eine frei wählbare Funktion auf jede gebildete k -Kombination 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.

```
> combn(c(1, 2, 3, 4), 3)          # alle 3-Kombinationen
     [,1]  [,2]  [,3]  [,4]
[1,]    1    1    1    2
[2,]    2    2    3    3
[3,]    3    4    4    4

# jeweilige Summe jeder 3-Kombination
> combn(c(1, 2, 3, 4), 3, FUN=sum)
[1] 6 7 8 9
```

```
# gewichtetes Mittel jeder 3-Kombination mit Argument w für Gewichte
> combn(c(1, 2, 3, 4), 3, weighted.mean, w=c(0.5, 0.2, 0.3))
[1] 1.8 2.1 2.3 2.8
```

`Permn(<Vektor>)` aus dem Paket `DescTools` stellt alle $n!$ Permutationen des übergebenen Vektors der Länge n als Zeilen einer Matrix zusammen (Abschn. 2.8). Für eine einzelne zufällige Permutation s. Abschn. 2.4.3.

```
> library(DescTools)                                # für Permn()
> Permn(1:3)
 [,1] [,2] [,3]
[1,]    1    2    3
[2,]    1    3    2
[3,]    3    1    2
[4,]    3    2    1
[5,]    2    3    1
[6,]    2    1    3
```

`expand.grid(<Vektor1>, <Vektor2>, ...)` bildet das kartesische Produkt der jeweils in Form eines Vektors übergebenen Grundmengen. Dies sind alle Kombinationen der Elemente der Grundmengen, wobei jeweils ein Element aus jeder Grundmenge stammt und die Reihenfolge nicht berücksichtigt wird. Dies entspricht der Situation, dass aus den Stufen mehrerer Faktoren alle Kombinationen von Faktorstufen gebildet werden. Das Ergebnis von `expand.grid()` ist ein Datensatz (Abschn. 2.11), 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.

```
> var1 <- c("control", "treatment")
> var2 <- c("f", "m")
> var3 <- c(1, 2)
> expand.grid(IV1=var1, IV2=var2, IV3=var3, stringsAsFactors=FALSE)
   IV1 IV2 IV3
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
```

Damit übergebene Vektoren von Zeichenketten automatisch zu Gruppierungsfaktoren (Klasse `factor`, Abschn. 2.6) werden, ist das Argument `stringsAsFactors=TRUE` zu setzen. Sollen auch numerische Werte zu Faktorstufen im versuchsplanerischen Sinn werden, muss die zugehörige Variable vorher in ein Objekt der Klasse `factor` umgewandelt werden.

2.4 Systematische und zufällige Wertefolgen erzeugen

Ein häufig auftretender Arbeitsschritt in R ist die Erstellung von Zahlenfolgen nach vorgegebenen Regeln, wie etwa sequentielle Abfolgen von Zahlen oder Wiederholungen bestimmter Wertemuster.

Aber auch Zufallszahlen und zufällige Reihenfolgen sind ein unverzichtbares Hilfsmittel der Datenauswertung, wobei ihnen insbesondere in der Planung von Analysen anhand simulierter Daten eine große Bedeutung zukommt.¹¹ 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. Aber auch bei der zufälligen Auswahl von Teilstichproben eines Datensatzes oder beim Erstellen zufälliger Reihenfolgen zur Zuordnung von Beobachtungsobjekten auf experimentelle Bedingungen kommen Zufallszahlen zum Einsatz.

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. Bei Dezimalzahlen wird der Endwert nicht überschritten und die Zahlenfolge ggf. vorher beendet.

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

> 5.4:2.1          # Endwert 2.1 wird nicht erreicht
[1] 5.4 4.4 3.4 2.4
```

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
```

`seq_len(<Anzahl>)` erzeugt die häufig nützliche, bei 1 beginnende Zahlenfolge in Einerschritten mit einer definierten Anzahl von Werten. Die Funktion ist in Situationen hilfreich, in denen sich die Länge der Sequenz erst aus einer Variable ergibt und auch 0 sein kann. In diesem Fall würde die Alternative `1:<Anzahl>`, also `1:0` nicht das gewünschte Ergebnis liefern.

¹¹ Wenn hier und im Folgenden von Zufallszahlen die Rede ist, sind immer *Pseudozufallszahlen* gemeint. Diese kommen nicht im eigentlichen Sinn zufällig zustande, sind aber von tatsächlich zufälligen Zahlenfolgen fast nicht zu unterscheiden. Pseudozufallszahlen hängen deterministisch vom Zustand des die Zahlen produzierenden Generators ab. Wird sein Zustand über `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 (Abschn. 4.4). Nach welcher Methode Zufallszahlen generiert werden, ist konfigurierbar und kann auch von der R Version abhängen, s. `?Random`.

```
> seq_len(4)
[1] 1 2 3 4

> target_length <- 0          # erwünschte Länge der Sequenz
> seq_len(target_length)      # gemeintes Ergebnis: leerer Vektor
integer(0)

> 1:target_length           # hier unerwünschtes Ergebnis: Sequenz 1:0
[1] 1 0
```

Zahlenfolgen mit beliebiger Schrittweite lassen sich mit `seq()` erzeugen.

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

Dabei können Start- (`from`) und Endwert (`to`) des durch die Sequenz abzudeckenden Intervalls ebenso gewählt werden wie die gewünschte Schrittweite (`by`) bzw. stattdessen die gewünschte Anzahl der Elemente der Zahlenfolge (`length.out`). Die Sequenz endet vor `to`, wenn die Schrittweite kein ganzzahliges Vielfaches der Differenz von Start- und Endwert ist.

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

> seq(from=2, to=11, by=2)          # Endpunkt 11 wird nicht erreicht
[1] 2 4 6 8 10

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

Eine Möglichkeit zum Erstellen einer bei 1 beginnenden Sequenz in Einerschritten, die genauso lang ist wie ein bereits vorhandener Vektor, besteht mit `seq_along(<Vektor>)`. Dies ist die bevorzugte Art, um 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 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)
```

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 ein Vektor einzutragen, der auf zwei verschiedene Arten wiederholt werden kann.¹² Eine für das Argument `times` genannte natürliche Zahl bestimmt, wie oft `x` als Ganzes aneinander gehängt wird.

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

Wird für das Argument `times` ein Vektor angegeben, so muss dieser dieselbe Länge wie `x` besitzen – hier wird ein kürzerer Vektor durch R nicht selbstdäig zyklisch wiederholt (Abschn. 2.5.4). Jedes Element des Vektors `times` gibt an, wie häufig das an gleicher Position stehende Element von `x` wiederholt werden soll, ehe das nächste Element von `x` wiederholt und angehängt wird.

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

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

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

2.4.3 Zufällig aus einer Urne ziehen

Die Funktion `sample()` erstellt einen aus zufälligen Werten bestehenden Vektor, indem sie das Ziehen aus einer Urne simuliert.

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

Für `x` ist ein Vektor zu nennen, der die Elemente der Urne festlegt, aus der gezogen wird. Dies sind die Werte, aus denen sich die Zufallsfolge zusammensetzt. 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. Ohne Zurücklegen kann aus einem Vektor der Länge n nicht häufiger als n mal gezogen werden. Wenn `replace=FALSE` und dennoch `size` größer als `length(x)` ist, erzeugt R deswegen eine

¹²Auch komplexe Objekte lassen sich mit `rep()` duplizieren, wenn sie mit `list(<Objekt>)` als Komponente einer Liste übergeben werden (Abschn. 2.10). Ein Datensatz (Abschn. 2.11) könnte etwa mit `rep(list(data.frame(A=1:2, B=c("Z", "Q"))), 3)` dupliziert werden. Die drei Duplikate sind im Ergebnis Komponenten einer Liste.

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(1:6, size=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"), size=8, replace=TRUE)
[1] "grün" "blau" "grün" "rot" "rot" "blau" "grün" "blau"
```

Für `sample()` existieren zwei Kurzformen, auf die jedoch aufgrund der Gefahr von Verwechslungen besser verzichtet werden sollte: `sample(<Vektor>)` ist gleichbedeutend mit `sample(<Vektor>, size=length(<Vektor>), replace=FALSE)`, erstellt also eine zufällige Permutation der Elemente von `<Vektor>` (Abschn. 2.3.3). Darauf aufbauend steht bei einer natürlichen Zahl als einzigem Argument `sample(<Zahl>)` kurz für `sample(1:<Zahl>, size=<Zahl>, replace=FALSE)`.

Wenn für `sample(<Vektor>)` ein Objektname übergeben wird, steht oft vor der Ausführung nicht fest, wie viele Elemente er beinhalten wird. Enthält `<Vektor>` z. B. durch die Auswahl einer Teilmenge unvorhergesehen als einziges Element eine Zahl, wird die Urne durch Elemente definiert, die womöglich nicht im ursprünglichen Vektor vorhanden waren.

```
> x <- c(2, 4, 6, 8)
> sample(x[(x %% 4) == 0]) # äquivalent zu sample(c(4, 8))
[1] 8 4

# Urne mit Elementen, die nicht aus x stammen
> sample(x[(x %% 8) == 0]) # äquivalent zu sample(8), d.h. sample(1:8)
[1] 2 1 7 5 4 8 6 3
```

2.4.4 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 erstellt werden (Abschn. 5.3). Diese Möglichkeit ist insbesondere für die Simulation empirischer Daten nützlich.

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

Als erstes Argument `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 jeweils 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 ist. `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.¹³ Auch diese Argumente können Vektoren der Länge n 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

# ziehe aus N(0, sd=1), N(50, sd=5), N(100, sd=10)
> rnorm(3, mean=c(0, 50, 100), sd=c(1, 5, 10))
[1] -2.980886 45.288597 106.080166
```

2.5 Daten transformieren

Häufig sind für spätere Auswertungen neue Variablen auf Basis der erhobenen Daten zu bilden. Im Rahmen solcher Datentransformationen können etwa Werte sortiert, umskaliert, ersetzt oder ausgewählt werden. Genauso ist es möglich, verschiedene Variablen zu einer neuen zu verrechnen und kontinuierliche Variablen in Kategorien einzuteilen oder in Rangwerte umzuwandeln.

2.5.1 Werte sortieren

Um die Reihenfolge eines Vektors umzukehren, kann `rev(<Vektor>)` (*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
```

¹³Der die Breite (Dispersion) einer Normalverteilung charakterisierende Parameter ist hier die Streuung σ , in der Literatur dagegen häufig die Varianz σ^2 .

Die Elemente eines Vektors lassen sich auch entsprechend ihrer Reihenfolge sortieren, 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 (Abschn. 1.4.6, Fußnote 34). Zum Sortieren stehen die Funktionen `sort()` und `order()` zur Verfügung.

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

`sort()` gibt eine sortierte Version des Vektors aus, ohne den übergebenen Vektor selbst zu verändern. 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. Für einen Vektor `x` ist daher `sort(x)` äquivalent zu `x[order(x)]`.¹⁴ Der Vorteil von `order()` erweist sich beim Umgang mit Matrizen und Datensätzen (Abschn. 2.8.6, 3.3.6). Die Sortierreihenfolge wird über das Argument `decreasing` kontrolliert. In der Voreinstellung `FALSE` 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
```

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

```
> sort(c("D", "E", "10", "A", "F", "E", "D", "4", "E", "A"))
[1] "10" "4" "A" "A" "D" "D" "E" "E" "E" "F"
```

2.5.2 Werte in zufällige Reihenfolge bringen

Zufällige Reihenfolgen können mit Kombinationen von `rep()` und `sample()` erstellt werden und entsprechen der zufälligen Permutation einer Menge (Abschn. 2.3.3). Sie sind z.B. bei der randomisierten Zuteilung von Beobachtungsobjekten zu Gruppen, beim Randomisieren der Reihenfolge von Bedingungen oder beim Ziehen einer Zufallsstichprobe aus einer Datenmenge nützlich.

¹⁴Sofern keine fehlenden Werte `NA` im Vektor vorhanden sind (Abschn. 2.13.5). Mit `x_ord` als dem mit Hilfe der Indizes `idx` sortierten Vektor `x` gilt dann auch, dass `x_ord[order(idx)]` die ursprüngliche Reihenfolge wiederherstellt.

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

# teile 12 Personen auf 3 unterschiedlich große Gruppen auf
> P <- 3 # Anzahl Gruppen
> Nj <- c(4, 3, 5) # Gruppengrößen
> (IV <- rep(seq_len(P), times=Nj)) # Gruppenzugehörigkeiten
[1] 1 1 1 1 2 2 2 3 3 3 3 3

# zufällige Permutation
> (IVrand <- sample(IV, size=length(IV), replace=FALSE))
[1] 2 1 1 3 3 1 3 1 2 2 3 3
```

Um allgemein n Beobachtungsobjekte auf p möglichst ähnlich große Gruppen aufzuteilen, können zunächst mit `sample()` die Indizes $1, \dots, n$ permutiert werden, um dann mit `%%` den Rest der ganzzahligen Division jedes Index mit p zu bilden, der die p Werte $0, \dots, p - 1$ annehmen kann.

```
> P <- 3
> N <- 20
> (sample(seq_len(N), size=N, replace=FALSE) %% P) + 1
[1] 2 2 2 3 3 3 1 3 1 2 3 2 1 1 2 3 2 1 3 1
```

2.5.3 Teilmengen von Daten auswählen

Soll aus einer vorhandenen Datenmenge eine Teilstichprobe gezogen werden, hängt das Vorgehen von der beabsichtigten Art der Ziehung ab. Grundsätzlich können zur Auswahl von Werten logische wie numerische Indexvektoren zum Einsatz kommen, die sich systematisch oder zufällig erzeugen lassen.

Eine rein zufällige Unterauswahl eines bestimmten Umfangs ohne weitere Nebenbedingungen kann mit `sample()` erstellt werden.¹⁵ Dazu betrachtet man den Datenvektor als Urne, aus der ohne Zurücklegen die gewünschte Anzahl von Beobachtungen gezogen wird.

```
> vec <- rep(c("rot", "grün", "blau"), each=10)
> sample(vec, size=5, replace=FALSE)
[1] "blau" "grün" "blau" "grün" "rot"
```

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

```
> selIdx1 <- seq(1, length(vec), by=10) # Elemente 1, 11, 21
> vec[selIdx1]
[1] "rot" "grün" "blau"
```

¹⁵Das Paket `car` (Fox & Weisberg, 2020) bietet hierfür die Funktion `some()`, die sich auch für Matrizen (Abschn. 2.8) oder Datensätze (Abschn. 2.11) eignet.

Soll nicht genau, sondern nur im Mittel jedes zehnte Element ausgegeben werden, eignet sich `rbinom()` zum Erstellen eines geeigneten Indexvektors. Dazu kann der Vektor der Trefferanzahlen aus einer Serie von jeweils nur einmal durchgeführten Bernoulli-Experimenten mit Trefferwahrscheinlichkeit $\frac{1}{10}$ in einen logischen Indexvektor umgewandelt werden:

```
> selIdx2 <- rbinom(length(vec), size=1, prob=0.1) == 1
> vec[selIdx2]
[1] "blau" "grün" "blau" "grün" "grün"
```

2.5.4 Daten umrechnen

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, wodurch sich Variablen leicht umskalieren lassen. Die Berechnungen einer Funktion werden dafür 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
```

Auch in Situationen, in denen mehrere Vektoren in einer Rechnung auftauchen, können diese wie Einzelwerte verwendet werden. 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, -1, 3)
> vec1*vec2
[1] -6 8 -5 18

> vec3 <- c(10, 100, 1000, 10000)
> (vec1 + vec2) / vec3
[1] 1e-01 6e-02 4e-03 9e-04
```

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

Zyklische Verlängerung von Vektoren (recycling)

Die Verrechnung mehrerer Vektoren scheint aufgrund der elementweisen Zuordnung zunächst vorauszusetzen, dass die Vektoren dieselbe Länge besitzen. Tatsächlich ist dies nicht unbedingt notwendig, weil R in den meisten Fällen diesen Zustand ggf. selbstdäig herstellt. Dabei wird der kürzere Vektor intern von R zyklisch wiederholt (also sich selbst angefügt, *recycling*), bis er mindestens die Länge des längeren Vektors besitzt. 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 Länge ein Hinweis auf fehlerhafte Berechnungen sein kann.

```
> age <- c(18, 20, 30, 24, 23, 21)
> 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)) # Länge der 3 Vektoren
[1] 6 12 5

> vec1*age                                # age 2 mal dupliziert
[1] 36 80 180 192 230 252 252 320 540 480 506 504

> vec2*age                                # vec2 2 mal dupliziert
[1] 36 80 180 192 230 42

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

z-Transformation

Durch eine *z*-Transformation wird eine quantitative Variable X so normiert, dass sie den Mittelwert $\bar{x} = 0$ und die Standardabweichung $s = 1$ besitzt. Dies geschieht für jeden Einzelwert x_i durch $\frac{x_i - \bar{x}}{s}$. `mean(<Vektor>)` berechnet den Mittelwert (Abschn. 2.7.3), `sd(<Vektor>)` ermittelt die korrigierte Streuung (Abschn. 2.7.6).

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

Eine andere Möglichkeit bietet die Funktion `scale(x=<Vektor>, center=TRUE, scale=TRUE)`. Sie berechnet die *z*-Werte mit Hilfe der korrigierten Streuung, gibt sie jedoch nicht in Form eines Vektors, sondern als Matrix mit einer Spalte aus.¹⁶ Weiterhin werden Mittelwert und korrigierte Streuung von x in Form von Attributen mit angegeben. Standardisierung und Zentrierung können unabhängig voneinander ausgewählt werden: Für die zentrierten, nicht aber standardisierten Werte von x ist etwa `scale=FALSE` zu setzen und `center=TRUE` zu belassen.

¹⁶Für x kann auch eine Matrix übergeben werden, deren *z*-transformierte Spalten dann die Spalten der ausgetragenen Matrix ausmachen (Abschn. 2.8).

```
> (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
```

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 \bar{x}_{neu} und eine beliebige Streuung s_{neu} besitzen. Dies geschieht für eine z -transformierte Variable Z mit $Z \cdot s_{\text{neu}} + \bar{x}_{\text{neu}}$.

```
> 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)                      # Kontrolle: Mittelwert
[1] 100

> sd(newAge)                        # Kontrolle: Streuung
[1] 15
```

Rangtransformation

`rank(Vektor)` gibt für jedes Element eines Vektors seinen Rangplatz 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 in der Voreinstellung jedoch denselben Rang. Das Verhalten, mit dem bei solchen *Bindungen* Ränge ermittelt werden, kontrolliert das Argument `ties.method` – Voreinstellung sind mittlere Ränge.

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

2.5.5 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 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)           # Summenscore
[1] 2 1 1 2 1 3 1 1

> (avgVar <- (quest1 + quest2 + quest3) / 3)      # mittlerer Score
[1] 0.66667 0.33333 0.33333 0.66667 0.33333 1.00000 0.333333 0.33333
```

2.5.6 Werte ersetzen oder recodieren

Mitunter werden Variablen zunächst auf eine bestimmte Art codiert, die sich später für manche Auswertungen als nicht zweckmäßig erweist und deswegen geändert werden soll. Dann müssen bestimmte Werte gesucht und ersetzt werden, was sich auf verschiedenen Wegen erreichen lässt. Dabei sollte die Variable mit recodierten Werten stets als neues Objekt erstellt werden, statt die Werte des alten Objekts zu überschreiben.

Mit `replace()` können Werte eines Vektors systematisch durch andere Werte ersetzt werden, wobei ein Indexvektor zum Einsatz kommt, der sich aus der Prüfung von Bedingungen ergeben kann.

```
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` über numerische oder logische Indizes an. Der Vektor `values` definiert für jeden durch den Index in `list` ausgewählten alten Wert, welcher neue Wert an dieser Stelle einzufügen ist. Nicht ausgewählte Elemente von `x` bleiben erhalten. Da `replace()` den unter `x` angegebenen Vektor nicht verändert, muss das Ergebnis ggf. einem neuen Objekt zugewiesen werden.

In einem Vektor seien dazu die Lieblingsfarben von sieben englischsprachigen Personen erhoben worden. Später soll die Variable auf deutsche Farbnamen recodiert werden.

```
> myColors <- c("red", "purple", "blue", "blue",
+           "green", "red", "green")

# tausche schrittweise alle englischen Farben aus
> farben <- replace(myColors, myColors == "red",      "rot")
> farben <- replace(farben,   farben == "purple",    "lila")
> farben <- replace(farben,   farben == "blue",       "blau")
> farben <- replace(farben,   farben == "green",      "gruen")
> farben
[1] "rot"  "lila"  "blau"  "blau"  "gruen" "rot"  "gruen"
```

Für Vektoren aus Zeichenketten folgt eine knappere Lösung aus der Möglichkeit, benannte Vektoren über Elementnamen zu indizieren. Dazu ist es notwendig, zunächst einen benannten Vektor mit Elementen nach dem Muster `<Wert alt>=<Wert neu>` zu erstellen, der die Ersetzungsregeln definiert. Dieser Vektor ist dann mit dem zu transformierenden Vektor zu indizieren. Die Recodierung mit dieser Methode funktioniert nur, wenn alle ursprünglichen Werte auch als Name vergeben werden, andernfalls resultieren fehlende Werte NA.

```
> repl <- c(red="rot", purple="lila", blue="blau", green="gruen")
> repl[myColors]
  red  purple  blue  blue  orange  red  green
"rot"  "lila"  "blau"  "blau"  "orange"  "rot"  "gruen"

# "violet" ist nicht im benannten Vektor enthalten -> erzeugt NA
> repl[c(myColors, "violet")]
  red  purple  blue  blue  orange  red  orange  <NA>
"rot"  "lila"  "blau"  "blau"  "orange"  "rot"  "orange"  NA
```

`case_match()` aus dem Paket `dplyr` (Wickham, Francois, Henry & Müller, 2020) ermöglicht es auf bequeme Weise, gleichzeitig bestimmte Werte nach einem Muster zu ändern, ohne dabei selbst logische Indexvektoren bilden zu müssen.

```
case_match(<Vektor>,
           <Wert1 alt> ~ <Wert1 neu>, <Wert2 alt> ~ <Wert2 neu>, ...)
```

Der Vektor mit zu ändernden Werten ist als erstes Argument anzugeben. Die Recodierung erfolgt anhand von beliebig vielen durch Komma getrennten Wertepaaren `<alt> ~ <neu>`. Zeichenketten sind dabei in Anführungszeichen zu setzen. Mehreren alten Werten kann derselbe neue Wert zugewiesen werden, indem statt eines einfachen Werts `<alt>` auf der linken Seite der `~` ein Vektor zu ersetzender Werte `c(<alt1>, <alt2>, ...)` steht. Nicht aufgeführte alte Werte werden durch NA ersetzt. Alle nicht explizit genannten alten Werte lassen sich mit dem Argument `.default` auf denselben neuen Standardwert setzen.

```
> library(dplyr)                                     # für case_match()
> case_match(myColors,
+             "red"     ~ "rot",
```

```

+      "blue" ~ "blau",
+      "purple" ~ "lila",
+      "green" ~ "gruen")
[1] "rot" "lila" "blau" "blau" "gruen" "rot" "gruen"

# Einteilung der Farben in Basisfarben und andere
> case_match(myColors,
+             c("red", "blue") ~ "basic",
+             .default="complex")
[1] "basic" "complex" "basic" "basic" "complex" "basic" "complex"

```

Auch numerische Vektoren lassen sich mit `case_match()` umcodieren. Flexibler ist dies aber mit den in Abschn. 2.5.7 und 2.6.7 vorgestellten Methoden möglich.

```

> orgVec <- c(5, 9, 11, 8, 9, 3, 1, 13, 9, 12, 5, 12)
> case_match(orgVec,
+             0:4 ~ "V00-04",
+             5:10 ~ "V05-10",
+             11:20 ~ "V11-20")
[1] "V05-10" "V05-10" "V11-20" "V05-10" "V05-10" "V00-04"
[7] "V00-04" "V11-20" "V05-10" "V11-20" "V05-10" "V11-20"

```

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

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`. Gegenüber `replace()` eignet sich `ifelse()` dann eher, wenn alle Werte verändert werden sollen.

```

> cutoff <- 10    # ersetze Werte unter cutoff durch 0, sonst durch 1
> (reVec <- ifelse(orgVec < cutoff, 0, 1))
[1] 0 0 1 0 0 0 0 1 0 1 0 1

# codiere Werte aus Zielmenge als "yes", sonst "no"
> targetSet <- c("A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K")
> response <- c("Z", "E", "O", "W", "H", "C", "I", "G")
> (respRec <- ifelse(response %in% targetSet, "yes", "no"))
[1] "no"  "yes" "no"  "no"  "yes" "yes" "yes" "yes"

```

Das Argument `x` sollte ein einfacher Vektor sein, da `ifelse()` die Klasse des übergebenen Objekts nicht respektiert – insbesondere Faktoren (Abschn. 2.6) sind deshalb für `ifelse()` ungeeignet.

2.5.7 Kontinuierliche Variablen in Kategorien einteilen

Als Spezialfall des Recodierens können neue Variablen dadurch entstehen, dass der Wertebereich von ursprünglich kontinuierlichen Variablen in Klassen eingeteilt wird. Auf diese Weise lässt sich eine quantitative in eine kategoriale Variable umwandeln. Hier soll der IQ-Wert mehrerer Personen zu einer Klasseneinteilung genutzt werden. Da sich `replace()` dafür weniger eignet, beruht der manuelle Lösungsansatz direkt auf einer zunächst leeren, schrittweise über logische Indexvektoren mit Werten zu füllenden Variable.

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

Als bequeme Alternative kann aus dem Paket `dplyr` die Funktion `case_when()` zum Einsatz kommen, die Abschn. 3.4.5 näher vorstellt.

```
> case_when((IQ <= 100)           ~ 1,
+            (IQ > 100) & (IQ <= 115) ~ 2,
+            (IQ > 115)           ~ 3)
[1] 2 2 1 1 1 2 1 1 3 2
```

`ifelse()` eignet sich für den Spezialfall, den Wertebereich in zwei Gruppen einzuteilen.

```
> ifelse(IQ >= 100, "hi", "lo")      # Dichotomisierung
[1] "hi" "hi" "lo" "lo" "lo" "hi" "lo" "lo" "hi" "hi"
```

Allgemein besonders leicht lassen sich quantitative Variablen zudem mit `cut()` diskretisieren (Abschn. 2.6.7). Hierdurch werden sie zu Faktoren, dem Gegenstand des folgenden Abschnitts.

2.6 Gruppierungsfaktoren

Die Klasse `factor` existiert, um die Eigenschaften kategorialer Variablen mit einer begrenzten Menge möglicher Werte abzubilden. Ein Objekt dieser Klasse nimmt die jeweilige Gruppenzugehörigkeit von Beobachtungsobjekten auf und speichert zusätzlich, welche Stufen die Variable prinzipiell umfasst. Für den Gebrauch in inferenzstatistischen Analysefunktionen ist es wichtig, dass Gruppierungsvariablen auch tatsächlich die Klasse `factor` besitzen. Bei numerisch codierter Gruppenzugehörigkeit besteht sonst die Gefahr der Verwechslung mit echten quantitativen Variablen, was etwa bei Regressionsmodellen für falsche Ergebnisse sorgen kann.

2.6.1 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", "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)
```

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.

```
> (sexFac <- factor(sex))
[1] m f f m m 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` übergebenen Vektor umbenannt werden. Dies könnte etwa sinnvoll sein, wenn die Faktorstufen in einem Vektor numerisch codiert sind, im Faktor aber inhaltlich aussagekräftigere Namen erhalten sollen. `labels` muss ein Vektor derselben Länge wie `levels` sein. Indem verschiedenen Werten in `levels` dieselbe Bezeichnung in `labels` zugewiesen wird, können Faktorstufen zu Gruppen zusammengefasst werden.

```
> (sexNum <- rbinom(10, size=1, prob=0.5))      # 0=Mann, 1=Frau
[1] 0 1 1 0 1 0 0 0 1 1

> factor(sexNum, levels=0:1, labels=c("man", "woman"))
[1] man woman woman man woman man man woman woman
Levels: man woman

# fasse Ausprägungen 1, 2 zu Gruppe A und 3, 4 zu Gruppe B zusammen
> factor(c(1, 2, 3, 4), levels=1:4, labels=c("A", "A", "B", "B"))
[1] A A B B
Levels: A B
```

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

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

```
> summary(sexFac)
f   m
4   4
```

Die im Faktor gespeicherten Werte werden intern auf zwei Arten repräsentiert – zum einen mit den Namen der Faktorstufen in einem `character` Vektor im Attribut `levels`, 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>)`.¹⁷

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

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

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

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

2.6.2 Faktoren kombinieren

Faktoren lassen sich wie gewöhnliche Vektoren mit `c()` aneinanderhängen. Der neue Faktor besitzt als Stufen die Vereinigungsmenge der Stufen der übergebenen Faktoren.

```
# Faktoren 1 und 2
> (fac1 <- factor(sample(LETTERS[1:5], 4), levels=LETTERS[1:5]))
[1] D E A C
Levels: A B C D E

> (fac2 <- factor(sample(letters[1:5], 3), levels=letters[1:5]))
[1] d c e
Levels: a b c d e
```

¹⁷Trotz dieser Codierung können Faktoren keinen mathematischen Transformationen unterzogen werden. Wenn die Namen der Faktorstufen aus Zahlen gebildet werden, kann es zu Diskrepanzen zwischen den Stufen und der internen Codierung kommen: `unclass(factor(10:15))` ergibt `1 2 3 4 5 6`. Dies ist bei der üblichen Verwendung von Faktoren aber irrelevant.

```
> c(fac1, fac2)
[1] D E A C d c e
Levels: A B C D E a b c d e
```

Gilt es, lediglich einen bereits bestehenden Faktor zu vervielfachen, eignet sich wie bei Vektoren `rep()`.

```
> rep(fac1, times=2)                      # Faktor 1 wiederholen
[1] Q A P U J Q A P U J
Levels: A J P Q U
```

In Situationen, in denen sich experimentelle Bedingungen aus der Kombination von zwei oder mehr Faktoren ergeben, ist es bisweilen nützlich, die mehrfaktorielle in eine geeignete einfaktorielle Struktur zu überführen. Dabei werden alle Kombinationen von Faktorstufen als Stufen eines neuen einzelnen Faktors betrachtet – etwa im Kontext einer assoziierten einfaktoriellen Varianzanalyse bei einem eigentlich zweifaktoriellen Design (Abschn. 7.6). Dies ist mit `interaction()` möglich.

```
interaction(<Faktor1>, <Faktor2>, ..., drop=FALSE)
```

Als Argument sind zunächst mehrere Faktoren gleicher Länge zu übergeben, die die Gruppenzugehörigkeit derselben Beobachtungsobjekte bzgl. verschiedener Gruppierungsfaktoren codieren. Auf ihren Stufenkombinationen basieren die Ausprägungen des neuen Faktors. Dabei kann es vorkommen, dass nicht für jede Stufenkombination Beobachtungen vorhanden sind, da einige Zellen im Versuchsdesign leer sind, oder kein vollständig gekreuztes Design vorliegt. In der Voreinstellung `drop=FALSE` erhält der neue Faktor auch für leere Zellen eine passende Faktorstufe. Mit `drop=TRUE` werden zu leeren Zellen gehörende Stufen dagegen weggelassen.

```
> (IV1 <- factor(rep(c("lo", "hi"), each=6)))    # Faktor 1
[1] lo lo lo lo lo lo hi hi hi hi hi
Levels: hi lo

> (IV2 <- factor(rep(1:3, times=4)))              # Faktor 2
[1] 1 2 3 1 2 3 1 2 3 1 2 3
Levels: 1 2 3

> interaction(IV1, IV2)                            # assoziiertes einfaktorielles Design
[1] lo.1 lo.2 lo.3 lo.1 lo.2 lo.3 hi.1 hi.2 hi.3 hi.1 hi.2 hi.3
Levels: hi.1 lo.1 hi.2 lo.2 hi.3 lo.3
```

2.6.3 Faktorstufen nachträglich ändern

Nachträgliche Änderungen an bestehenden Faktoren lassen sich mit dem Basisumfang von R oder mit Funktionen des Pakets `forcats` (Wickham, 2020a) vornehmen, deren Namen dem Muster `fct_<Verb>(<Faktor>, ...)` folgen. Das Ergebnis dieser Funktionen ist immer ein Faktor mit den geänderten Eigenschaften. Der übergebene Faktor bleibt durch `fct_<Verb>()` unverändert, ggf. muss das Ergebnis also einem neuen Objekt zugewiesen werden.

Wenn die Stufenbezeichnungen eines Faktors im nachhinein ersetzt werden sollen, so kann dem von `levels(<Faktor>)` ausgegebenen Vektor ein Vektor mit passend vielen neuen Namen zugewiesen werden. Alternativ ist dies mit `fct_recode()` möglich.

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

> library(forcats)                         # für fct_recode()
> fct_recode(sexFac, "F"="female", "M"="male")
[1] M F F M M M F F
Levels: F M
```

Einem bestehenden Faktor können nicht beliebige Werte als Element hinzugefügt werden, sondern lediglich bereits vorhandene Faktorstufen. 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. Alternativ ist dies mit `fct_expand()` möglich.

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

# bisher nicht existierende Faktorstufe "mid"
> 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"                         # neue Beobachtung "mid"
> status
[1] hi lo hi mid
Levels: hi lo mid

> fct_expand(status, "new_level")            # Faktor mit neuer Stufe
[1] hi lo hi mid
Levels: hi lo mid new_level
```

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 (Abschn. 2.10). Alternativ ist dies mit `fct_collapse()` möglich.

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

> fct_collapse(status, notHi=c("mid", "lo"))
[1] hi notHi hi notHi
Levels: hi notHi
```

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 nach wie vor alle ursprünglichen Stufen, wie in der Ausgabe unter `Levels:` deutlich wird. Sollen nur die in der gewählten Teilmenge tatsächlich auftretenden Ausprägungen auch mögliche Stufen sein, kann dies mit `droplevels()` erreicht werden. Alternativ ist dies mit `fct_drop()` – selektiv auch für einzelne level – möglich.

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

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

> fct_drop(status[1:2], "mid")
[1] hi lo
Levels: hi lo
```

Fehlende Werte (Abschn. 2.13) im ursprünglichen Vektor bleiben in der Voreinstellung fehlende Werte im aus dem Vektor erstellten Faktor. Beim Erstellen eines Faktors lassen sich jedoch fehlende Werte mit dem Argument `exclude=NULL` als eigene Kategorie werten. Im Nachhinein lässt sich derselbe Effekt mit `addNA()` erzielen. Dies ist etwa in Häufigkeitstabellen nützlich, um fehlende Werte mitzuzählen.

```
# fehlende Werte bilden keine eigene Stufe
> (fac <- factor(c("A", "B", NA, "A", NA)))
[1] A B <NA> A <NA>
Levels: A B

# fehlende Werte bilden eigene Stufe
> (facNA <- factor(c("A", "B", NA, "A", NA), exclude=NULL))
[1] A B <NA> A <NA>
Levels: A B <NA>
```

```
> table(fac)          # fehlende Werte werden nicht gezählt
fac
A B
2 1

> table(addNA(fac))    # fehlende Werte werden gezählt
A   B   <NA>
2   1       2
```

Alternativ ist es mit `fct_na_value_to_level()` möglich, fehlende Werte zu einer eigenen Faktorstufe zu machen. Um umgekehrt alle Beobachtungen einer bestimmten Faktorstufe zu fehlenden Werten zu machen, eignet sich die Funktion `fct_na_level_to_value()`, die auch die benannte Faktorstufe entfernt.

```
# fehlende Werte als Stufe hinzufügen
> (facNA_lvl <- fct_na_value_to_level(fac, "(N/A")))
[1] A   B   (N/A)  A   (N/A)
Levels: A B (N/A)

# Werte einer Faktorstufe -> fehlende Werte, Faktorstufe löschen
> fct_na_level_to_value(facNA_lvl, "(N/A)")
[1] A   B   <NA>  A   <NA>
Levels: A B
```

2.6.4 Geordnete Faktoren

Besteht eine Reihenfolge in den Stufen eines Gruppierungsfaktors i.S. einer ordinalen Variable, so lässt sich dieser Umstand mit der Funktion `ordered(<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. Für die Elemente georderter Faktoren sind die üblichen Ordnungsrelationen $<$, \leq , $>$, \geq definiert.

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

> ordStat[1] > ordStat[2]                      # hi > lo?
[1] TRUE
```

Sollen die Elemente des Faktors in eine zufällige Reihenfolge gebracht werden, um die Zuordnung von Beobachtungsobjekten zu Gruppen zu randomisieren, kann dies wie in Abschn. 2.5.2 geschehen.

```
> sample(ordStat, size=length(ordStat), replace=FALSE)
[1] hi lo 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.

2.6.5 Reihenfolge von Faktorstufen kontrollieren

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 `levels` angegeben, bestimmt die Reihenfolge dieser Elemente die Reihenfolge der Stufen. Ohne Verwendung von `levels` ergibt sich die Reihenfolge aus den sortierten Elementen des Vektors, der zum Erstellen des Faktors verwendet wird.¹⁸

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

# ohne levels: Reihenfolge aus sortierten Elementen des Vektors
> factor(vec)
[1] 3 4 3 2 1 4 1 1
Levels: 1 2 3 4

# levels angegeben -> levels bestimmt Reihenfolge der Stufen
> factor(vec, levels=c(4, 3, 2, 1))
[1] 3 4 3 2 1 4 1 1
Levels: 4 3 2 1
```

Um die Reihenfolge der Stufen nachträglich zu ändern, kann ein Faktor in einen geordneten Faktor unter Verwendung des `levels` Arguments umgewandelt werden (s.o.). Als weitere Möglichkeit wird mit `relevel(<Faktor>, ref="<Referenzstufe>")` die für `ref` übergebene Faktorstufe zur ersten Stufe des Faktors, die in manchen statistischen Modellen als Vergleichsgruppe dient.

```
> (abcde <- factor(sample(LETTERS[1:5], 4), levels=LETTERS[1:5]))
[1] B E A C
Levels: A B C D E

> relevel(abcde, ref="E")      # mache E zur ersten Faktorstufe
[1] B E A C
Levels: E A B C D
```

`reorder()` ändert die Reihenfolge der Faktorstufen ebenfalls nachträglich. Die Stufen werden so geordnet, dass ihre Reihenfolge durch die gruppenweise gebildeten empirischen Kennwerte einer Variable bestimmt ist.

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

¹⁸Sind dessen Elemente Zeichenketten mit numerischer Bedeutung, so ist zu beachten, dass die Reihenfolge dennoch alphabetisch bestimmt wird – die Stufe "10" käme demnach vor der Stufe "4".

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 der Kennwerte geordnet, die sich aus der gruppenweisen Anwendung von `FUN` ergeben (s. Abschn. 2.7.10 für `tapply()`).

```
> fac1 <- factor(rep(LETTERS[1:3], each=5))
> vec  <- rnorm(30, rep(c(10, 5, 15), each=5), 3)
> reorder(fac1, vec, FUN=mean)
[1] A A A A A B B B B C C C C C
Levels: B < A < C

# Kontrolle: Mittelwerte pro Gruppe
> tapply(vec, fac1, FUN=mean)
      A          B          C 
10.18135  6.47932 13.50108
```

Alternativ ist es mit `fct_relevel(<Faktor>, "<Stufe>", after=<Position>)` möglich, die Position von Faktorstufen beliebig zu kontrollieren.

```
> fct_relevel(fac1, "A", after=1)    # verschiebe Stufe A an 2. Position
[1] A A A A A A A A A B B B B B B B C C C C C C C C C C
Levels: B A C

> fct_relevel(fac1, "A", after=Inf)   # verschiebe Stufe A ans Ende
[1] A A A A A A A A A B B B B B B B C C C C C C C C C C
Levels: B C A
```

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 das Sortieren eines Faktors zu einem anderen Ergebnis führen als das Sortieren eines Vektors, auch wenn dieser oberflächlich dieselben Elemente enthält.

```
> (fac2 <- 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(fac2)
[1] B B B B B A A A A

> sort(as.character(fac2))
[1] "A" "A" "A" "A" "B" "B" "B" "B" "B" "B"
```

2.6.6 Faktoren nach Muster erstellen

Wenn Faktoren bei der Einteilung von Beobachtungsobjekten in Gruppen nach festem Muster erstellt werden müssen, lassen sie sich als Alternative zur manuellen Anwendung von `rep()` und `factor()` mit `gl()` erzeugen.

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

Das Argument `n` gibt die Anzahl der Stufen an, die der Faktor 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` zu setzen.

```
> (fac1 <- factor(rep(c("A", "B"), times=c(5, 5)))) # manuell
[1] A A A A A B B B B B
Levels: A B
```

```
> (fac2 <- gl(2, 5, labels=c("less", "more"), ordered=TRUE))
[1] less less less less less more more more more more
Levels: less < more
```

Bei mehreren Faktoren mit vollständig gekreuzten Faktorstufen kann `expand.grid()` verwendet werden, um alle Stufenkombinationen zu erstellen (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 (Abschn. 2.11).

```
> 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
```

2.6.7 Quantitative in kategoriale Variablen umwandeln

Aus den in einem Vektor gespeicherten Werten einer quantitativen Variable lässt sich mit `cut()` ein Gruppierungsfaktor erstellen – die quantitative Variable wird so in eine kategoriale

umgewandelt (s. auch Abschn. 2.5.7). Dazu muss zunächst der Wertebereich des Vektors in disjunkte Intervalle eingeteilt werden. Die einzelnen Werte werden dann entsprechend ihrer Zugehörigkeit zu diesen Intervallen Kategorien zugeordnet und erhalten als Wert die zugehörige Faktorstufe.

```
cut(x=<Vektor>, breaks=<Intervalle>, labels=<Bezeichnungen>,
     right=TRUE, ordered=FALSE)
```

Die Intervalle werden über das Argument **breaks** festgelegt. Hier ist entweder die Anzahl der (dann gleich breiten) Intervalle anzugeben, oder die Intervallgrenzen selbst als Vektor. Die Intervalle werden in der Voreinstellung **right=TRUE** in der Form $(a, b]$ gebildet, sind also nach unten offen und nach oben geschlossen. Anders gesagt ist die untere Grenze nicht Teil des Intervalls, die obere schon. Dagegen erzeugt **right=FALSE** Intervalle der Form $[a, b)$. Die Unter- und Obergrenze des insgesamt möglichen Wertebereichs müssen bei der Angabe von **breaks** 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, können sie über das Argument **labels** explizit angegeben werden. Dabei ist darauf zu achten, dass die Reihenfolge der neuen Benennungen der Reihenfolge der gebildeten Intervalle entspricht. Soll es sich im Ergebnis um einen geordneten Faktor handeln, ist **ordered=TRUE** zu setzen.

```
> IQ      <- rnorm(100, mean=100, sd=15)
> IQfac <- cut(IQ, breaks=c(0, 85, 115, Inf),
+                 labels=c("lo", "mid", "hi"))

> IQfac[1:5]
[1] hi lo mid mid mid lo
Levels: lo mid hi
```

Um annähernd gleich große Gruppen zu erhalten, können für die Intervallgrenzen bestimmte Quantile der Daten gewählt werden, etwa der Median für den Median-Split (Abschn. 2.7.3).

```
> medSplit <- cut(IQ, breaks=c(-Inf, median(IQ), Inf))
> summary(medSplit)           # Kontrolle: Häufigkeiten der Kategorien
medSplit
(-Inf,97.6]  (97.6,Inf]
              50          50
```

Für mehr als zwei etwa gleich große Gruppen lässt sich die Ausgabe von **quantile()** (Abschn. 2.7.5) direkt an das Argument **breaks** übergeben. Dies ist möglich, da **quantile()** neben den Quantilen auch das Minimum und Maximum der Werte ausgibt. Damit das unterste Intervall auch das Minimum einschließt – und nicht wie alle übrigen Intervalle nach unten offen ist, muss das Argument **include.lowest=TRUE** gesetzt werden.

```
# 4 ähnliche große Gruppen, unterstes Intervall unten geschlossen
> IQdiscr <- cut(IQ, quantile(IQ), include.lowest=TRUE)
> summary(IQdiscr)           # Kontrolle: Häufigkeiten der Kategorien
IQdiscr
[62.1,87.1]  (87.1,97.6]  (97.6,107]  (107,154]
```

25

25

25

25

Anstelle von `cut()` können auch Funktionen aus dem Paket `santoku` ([Hugh-Jones & Possenriede, 2024](#)) verwendet werden, das für viele Aufgaben spezialisierte Lösungen anbietet.

2.7 Deskriptive Kennwerte numerischer Daten

Die deskriptive Beschreibung von Variablen ist ein wichtiger Teil der Analyse empirischer Daten, die gemeinsam mit der grafischen Darstellung (Abschn. [14.6](#)) hilft, ihre Struktur besser zu verstehen. Die hier umgesetzten statistischen Konzepte und Techniken werden als bekannt vorausgesetzt und finden sich in vielen Lehrbüchern der Statistik ([Eid, Gollwitzer & Schmitt, 2017](#)).

R stellt für die Berechnung aller gängigen Kennwerte separate Funktionen bereit, die meist erwarten, dass die Daten in Vektoren gespeichert sind.^{[19](#)} 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 `summary(<Vektor>)` können die wichtigsten deskriptiven Kennwerte einer Datenreihe abgerufen werden – dies sind Minimum, erstes Quartil, Median, Mittelwert, drittes Quartil und Maximum. Die Ausgabe ist ein Vektor mit benannten Elementen.

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

2.7.1 Summen, Differenzen und Produkte

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

```
> sum(age)
[1] 146

> cumsum(age)
[1] 17 47 77 102 125 146
```

Um die Differenzen aufeinander folgender Elemente eines Vektors (also eines Wertes zu einem Vorgänger) zu berechnen, kann die Funktion `diff(<Vektor>, lag=1)` verwendet werden. Mit ihrem 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 der Vektor Elemente besitzt.

¹⁹Viele von ihnen werden in `Desc()` aus dem Paket `DescTools` integriert.

```
> diff(age)
[1] 13 0 -5 -2 -2

> diff(age, lag=2)
[1] 13 -5 -7 -4

Das Produkt aller Elemente eines Vektors wird mit prod(<Vektor>) berechnet, das kumulierte Produkt mit cumprod(<Vektor>).20 factorial(<Zahl>) ermittelt die Fakultät  $n!$  einer Zahl  $n$ .21

> prod(age)
[1] 184747500

> cumprod(age)
[1] 17 510 15300 382500 8797500 184747500

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

2.7.2 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.

```
> max(age)                      # Maximum
[1] 30

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

Den Index des größten bzw. kleinsten Wertes liefern which.max(<Vektor>) bzw. which.min(<Vektor>). Kommt der größte bzw. kleinste Wert mehrfach vor, ist das Ergebnis seine früheste Position. which.min() lässt sich etwa nutzen, um herauszufinden, welches Element eines Vektors am nächsten an einem vorgegebenen Wert liegt.

> which.min(age)                # Position des Minimums
[1] 1

> vec <- c(-5, -8, -2, 10, 9) # Vektor
> which.max(vec > 0)           # Index erstes Element größer 0
[1] 4

# Index des ersten Elements, das nicht in Zielmenge ist
```

²⁰Bei Gleitkommazahlen kumulieren sich bei wiederholter Multiplikation Rundungsfehler, die durch die interne Darstellungsart solcher Zahlen unvermeidlich sind (Abschn. 1.4.6). Numerisch stabiler als `prod(<Vektor>)` ist deswegen u. U. die Rücktransformation der Summe der logarithmierten Werte mit `exp(sum(log(<Vektor>)))` als Umsetzung von $\exp\left(\sum_i \ln x_i\right)$ – vorausgesetzt x ist echt positiv. `prod(numeric(0))` ist gleich 1.

²¹Für natürliche Zahlen gilt $n! = \Gamma(n + 1)$, in R als `gamma(<Zahl> + 1)` berechenbar.

```
> which.min(vec %in% c(-5, 10))
[1] 2

> val <- 0                      # Referenzwert
> which.min(abs(vec-val))       # welches Element liegt am nächsten an 0?
[1] 3
```

Um die Spannweite (*range*) von Werten eines Vektors, also die Differenz vom kleinsten und größten Wert zu ermitteln, ist `diff()` nützlich.

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

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

```
> 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
```

2.7.3 Mittelwert, Median und Modalwert

Mit `mean(x=⟨Vektor⟩)` wird das arithmetische Mittel $\frac{1}{n} \sum_i x_i$ eines Vektors x der Länge n berechnet.²²

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

Für die Berechnung eines gewichteten Mittels, bei dem die Gewichte nicht wie bei `mean()` für alle Werte identisch sind ($\frac{1}{n}$), eignet sich 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 Wertes auf den Mittelwert ist gleich dem Verhältnis seines Gewichts zur Summe aller Gewichte.

```
> weights <- c(0.6, 0.6, 0.3, 0.2, 0.4, 0.6)
> weighted.mean(age, w=weights)
[1] 23.70370
```

²²Hier ist zu beachten, dass `x` tatsächlich ein etwa mit `c(...)` gebildeter Vektor ist: Der Aufruf `mean(1, 7, 3)` gibt 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.

Um beim geometrischen Mittel $\frac{1}{n} \prod_i x_i$ sich fortsetzende Rundungsfehler zu vermeiden, eignet sich für echt positive x_i `exp(mean(log(x)))` als Umsetzung von $e^{\frac{1}{n} \sum_i \ln x_i}$ (Fußnote 20). Das harmonische Mittel $n / \sum_i \frac{1}{x_i}$ berechnet sich als Kehrwert des Mittelwertes der Kehrwerte (für $x_i \neq 0$), also mit `1 / mean(1/x)`. Alternativ stellt das Paket `DescTools` die Funktionen `Gmean()` und `Hmean()` bereit.

`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` erstmalig mindestens den Wert 0.5 erreicht (Abschn. 2.12.6), der also $\geq 50\%$ (und $\leq 50\%$) der Werte ist. Im Fall einer geraden Anzahl von Elementen in `x` wird zwischen den beiden mittleren Werten von `sort(<Vektor>)` gemittelt, andernfalls das mittlere Element von `sort(<Vektor>)` ausgegeben.

```
> 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
```

Für die Berechnung des Modalwertes, also des in einem Vektor am häufigsten vorkommenden Wertes, stellt der Basisumfang von R keine separate Funktion bereit. Es kann aber auf die Funktion `Mode(<Vektor>)` aus dem Paket `DescTools` ausgewichen werden.

```
> vec <- c(11, 22, 22, 33, 33, 33, 33)      # häufigster Wert: 33
> library(DescTools)                          # für Mode()
> Mode(vec)
[1] 33
attr(,"freq")
[1] 4
```

Eine manuelle Alternative bietet `table()` zur Erstellung von Häufigkeitstabellen (Abschn. 2.12). Die folgende Methode gibt zunächst den Index des Maximums der Häufigkeitstabelle aus. Den Modalwert erhält man zusammen mit seiner Auftretenshäufigkeit durch Indizieren der mit `unique()` gebildeten und dann geordneten Einzelwerte des Vektors mit diesem Index.

```
> (tab <- table(vec))                      # Häufigkeitstabelle
vec
11 22 33
 1 2 4

> (modIdx <- which.max(tab))                # Index des Modalwerts
33
```

```
> sort(unique(vec))[modIdx] # Modalwert selbst
[1] 33
```

2.7.4 Robuste Maße der zentralen Tendenz

Wenn Daten Ausreißer aufweisen, kann dies den Mittelwert stark verzerrn, so dass er die Lage der zugrundeliegenden Variable nicht mehr gut repräsentiert. Aus diesem Grund existieren Maße der zentralen Tendenz, die weniger stark durch einzelne extreme Werte beeinflusst werden.

Der gestützte Mittelwert wird ohne einen bestimmten Anteil an Extremwerten berechnet. Mit `mean(<Vektor>, trim=<Anteil>)` wird der gewünschte Anteil an Extremwerten auf beiden Seiten der empirischen Verteilung aus der Berechnung des Mittelwerts ausgeschlossen. Um etwa den Mittelwert ohne die extremen 5% der Daten zu berechnen, ist `trim=0.025` zu setzen.

```
> age <- c(17, 30, 30, 25, 23, 21)
> mean(age) # Vergleich: Mittelwert
[1] 24.33333

> mean(age, trim=0.2) # gestützter Mittelwert
[1] 24.75
```

Bei der Winsorisierung von Daten mit `Winsorize()` aus dem Paket `DescTools` wird ein bestimmter Anteil an Extremwerten auf beiden Seiten der empirischen Verteilung durch jeweils den letzten Wert ersetzt, der noch nicht als Extremwert gilt. Dafür legt das Argument `probs` in Form eines Vektors die beiden Quantile als Grenzen fest. Der übliche Mittelwert dieser Daten ist dann der winsorisierte Mittelwert.

```
> library(DescTools) # für Winsorize()
> (ageWins <- Winsorize(age, probs=c(0.2, 0.8))) # winsorisierte Daten
[1] 21 30 30 25 23 21

> mean(ageWins) # winsor. Mittelwert
[1] 25
```

Der Hodges-Lehmann-Schätzer des Lageparameters einer Variable (*Pseudo-Median*) berechnet sich als Median der $n(n+1)/2$ vielen Walsh-Mittel – der Mittelwerte aller Wertepaare (x_i, x_j) der Daten mit sich selbst (mit $j \geq i$). Vergleiche hierfür `HodgesLehmann()` aus dem Paket `DescTools` sowie `wilcox.test()` in Abschn. 10.5.3.

```
> library(DescTools) # für HodgesLehmann()
> HodgesLehmann(age)
est
24

# manuelle Kontrolle -> Mittelwert aller Wertepaare
```

```
> pairM <- outer(age, age, FUN="+") / 2
> median(pairM[lower.tri(pairM, diag=TRUE)])      # deren Median
[1] 24
```

Für Huber M -Schätzer der Lage von Verteilungen existiert `huberM()` aus dem Paket `robustbase` (Rousseeuw et al., 2020). Als Schätzer für die Differenz der Lageparameter von zwei Variablen wird der Hodges-Lehmann-Schätzer als Median aller $n_1 \cdot n_2$ paarweisen Differenzen der Werte aus beiden Datenreihen gebildet (Abschn. 10.5.4).

2.7.5 Prozentrang, Quartile und Quantile

Angewendet auf Vektoren mit logischen Werten lassen sich mit `sum()` Elemente zählen, die eine bestimmte Bedingung erfüllen. So kann der Prozentrang eines Wertes als prozentualer Anteil derjenigen Werte ermittelt werden, die nicht größer als er sind (Abschn. 2.12.6). Die Funktion `PercentRank()` aus dem Paket `DescTools` gibt – multipliziert mit 100 – für jeden Wert eines Vektors den Prozentrang aus. Bei der dabei intern vorgenommenen Rangtransformation mit `rank()` (Abschn. 2.5.4) werden allerdings bei Bindungen mittlere Ränge berechnet.

```
> library(DescTools)                      # für PercentRank()
> age <- c(17, 30, 30, 25, 23, 21)
> PercentRank(age) * 100
[1] 16.66667 83.33333 83.33333 66.66667 50.00000 33.33333

# manuell
> sum(age <= 25)                         # wie viele Elemente <= 25?
[1] 4

> 100 * (sum(age <= 25) / length(age))   # Prozentrang von 25
[1] 66.66667

# Prozentrang von 30, abweichende Definition von PercentRank()
> 100 * (sum(age <= 30) / length(age))
[1] 100
```

Mit `quantile(<Vektor>, probs=seq(0, to=1, by=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 etwa $\geq 25\%$ (und $\leq 75\%$) der Daten. Das Ergebnis von `quantile()` ist ein Vektor mit benannten Elementen.

```
> (quant <- quantile(age))
  0%    25%    50%    75%   100%
17.00  21.50  24.00  28.75  30.00

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

Über das Argument `probs=<Vektor>` können statt der Quartile auch andere Anteile angegeben werden, deren Wertegrenzen gewünscht sind. Zur Berechnung der Werte, die einen bestimmten Anteil der Daten abschneiden, wird ggf. zwischen den in `x` tatsächlich vorkommenden Werten interpoliert.²³

```
> vec <- sample(seq(0, to=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
```

2.7.6 Varianz, Streuung, Schiefe und Wölbung

Mit `var(<Vektor>)` wird die korrigierte Varianz $s^2 = \frac{1}{n-1} \sum_i (x_i - \bar{x})^2$ zur erwartungstreuen Schätzung der Populationsvarianz auf Basis einer Variable X der Länge n mit Mittelwert \bar{x} ermittelt.²⁴

```
> age <- c(17, 30, 30, 25, 23, 21)          # Daten
> N    <- length(age)                      # Anzahl Beobachtungen
> M    <- mean(age)                        # Mittelwert

> var(age)                                # korrigierte Varianz
[1] 26.26667

> sum((age-M)^2) / (N-1)                   # manuelle Berechnung
[1] 26.26667
```

Die Umrechnungsformel zur Berechnung der unkorrigierten Varianz $S^2 = \frac{1}{n} \sum_i (x_i - \bar{x})^2$ aus der korrigierten lautet $S^2 = \frac{n-1}{n} s^2$.²⁵

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

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

Die korrigierte Streuung s kann durch Ziehen der Wurzel aus der korrigierten Varianz oder mit `sd(<Vektor>)` berechnet werden. Auch hier basiert das Ergebnis auf der bei der Varianz erläuterten Korrektur zur Schätzung der Populationsstreuung auf Basis einer empirischen Stichprobe.

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

²³Zur Berechnung von Quantilen stehen verschiedene Rechenwege zur Verfügung, vgl. `?quantile`. Insbesondere ist zu beachten, dass andere Statistik-Software aufgrund anderer Voreinstellungen andere Ergebnisse liefern kann.

²⁴Für ein Diversitätsmaß kategorialer Daten s. Abschn. 2.7.7 und für robuste Varianzschätzer Abschn. 2.7.9.

²⁵Als Alternative lässt sich `cov.wt()` verwenden (Abschn. 2.8.10).

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

Die Umrechnungsformel zur Berechnung der unkorrigierten Streuung $S = \sqrt{\frac{1}{n} \sum_i (x_i - \bar{x})^2}$ aus der korrigierten lautet $S = \sqrt{\frac{n-1}{n}} s$.

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

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

Schiefe und Wölbung (Kurtosis) als höhere zentrale Momente empirischer Verteilungen lassen sich mit `Skew()` und `Kurt()` aus dem Paket `DescTools` ermitteln.

2.7.7 Diversität kategorialer Daten

Der Diversitätsindex H ist ein Streuungsmaß für kategoriale Daten, der mit Werten im Intervall $[0, 1]$ angibt, wie sehr sich die Daten auf die Kategorien verteilen. Sind f_j die relativen Häufigkeiten der p Kategorien (Abschn. 2.12), ist $H = -\frac{1}{\ln p} \sum (f_j \cdot \ln f_j)$. Für $f_j = 0$ ist $f_j \cdot \ln f_j$ dabei als 0 definiert. H ist genau dann 0, wenn die Daten konstant sind, also nur eine von mehreren Kategorien auftritt. Für eine Gleichverteilung, in der alle Kategorien dieselbe Häufigkeit besitzen, die Daten also maximal verteilt sind, ist $H = 1$. R verfügt über keine spezialisierte Funktion für die Berechnung von H . Es lässt sich aber der Umstand nutzen, dass H bis auf den Faktor $\frac{1}{\ln p}$ mit dem Shannon-Index aus der Informationstheorie übereinstimmt, der sich mit der aus dem Paket `DescTools` stammenden Funktion `Entropy()` berechnen lässt. Dafür ist das Argument `base` auf die Eulersche Zahl $e = \exp(1)$ zu setzen.²⁶ Die Funktion erwartet als Argument den Vektor der absoluten oder relativen Häufigkeiten der Kategorien.

```
# Faktor inkl. einer nicht besetzten Stufe "Q"
> fac <- factor(c("C", "D", "A", "D", "E", "D", "C", "E", "E", "B", "E"),
+                  levels=c(LETTERS[1:5], "Q"))

> P    <- nlevels(fac)                      # Anzahl Kategorien
> (Fj <- proportions(table(fac)))           # relative Häufigkeiten
      A          B          C          D          E          Q
0.09090909 0.09090909 0.18181818 0.27272727 0.36363636 0.00000000

> library(DescTools)                        # für Entropy()
> shannonIdx <- Entropy(Fj, base=exp(1))   # Shannon Index
> (H <- (1/log(P)) * shannonIdx)           # Diversität
[1] 0.8193845

> keep <- Fj > 0                          # Indizes Häufigk. > 0
```

²⁶Zudem gilt folgende Beziehung zur diskreten Kullback-Leibler-Divergenz KL_{eq} der beobachteten Häufigkeiten zur Gleichverteilung: $H = -\frac{1}{\ln p} KL_{eq} + 1$.

```
> -(1/log(P)) * sum(Fj[keep] * log(Fj[keep])) # Kontrolle ...
```

2.7.8 Kovarianz und Korrelation

Mit `cov(x=<Vektor1>, y=<Vektor2>)` wird die korrigierte Kovarianz $\frac{1}{n-1} \sum_i ((x_i - \bar{x}) \cdot (y_i - \bar{y}))$ zweier Variablen X und Y derselben Länge n berechnet.

```
> x <- c(17, 30, 30, 25, 23, 21)          # Daten Variable 1
> y <- c(1, 12, 8, 10, 5, 3)            # Daten Variable 2
> cov(x, y)                            # korrigierte Kovarianz
[1] 19.2

> N  <- length(x)                      # Anzahl Objekte
> Mx <- mean(x)                        # Mittelwert Var 1
> My <- mean(y)                        # Mittelwert Var 2
> sum((x-Mx) * (y-My)) / (N-1) # korrigierte Kovarianz manuell
[1] 19.2
```

Die unkorrigierte Kovarianz muss nach der bereits für die Varianz genannten Umrechnungsformel ermittelt werden (Abschn. 2.7.6, Fußnote 25).

```
> ((N-1) / N) * cov(x, y)      # unkorrig. Kovarianz aus korrigierter
[1] 16

> sum((x-Mx) * (y-My)) / N    # unkorrigierte Kovarianz manuell
[1] 16
```

Neben der voreingestellten Berechnungsmethode für die Kovarianz nach Pearson kann auch die Rang-Kovarianz nach Spearman oder Kendall berechnet werden (Abschn. 10.3.1).

Analog zur Kovarianz kann mit `cor(x=<Vektor1>, y=<Vektor2>)` die herkömmliche Pearson Korrelation $r_{XY} = \frac{\text{Kov}_{XY}}{s_X \cdot s_Y}$ oder die Rangkorrelation berechnet werden.²⁷ Für die Korrelation gibt es keinen Unterschied beim Gebrauch von korrigierten und unkorrigierten Streuungen, so dass sich nur ein Kennwert ergibt.

```
> cor(x, y)                      # Korrelation
[1] 0.8854667

> cov(x, y) / (sd(x) * sd(y))   # manuelle Berechnung
[1] 0.8854667
```

Für die Berechnung der Partialkorrelation zweier Variablen X und Y ohne eine dritte Variable Z kann die Formel $r_{(XY).Z} = \frac{r_{XY} - (r_{XZ} \cdot r_{YZ})}{\sqrt{(1-r_{XZ}^2) \cdot (1-r_{YZ}^2)}}$ umgesetzt werden, da die Basisinstallation von

²⁷Das Paket `correlation` (Makowski et al., 2024) berechnet neben der polychorischen und polyserialen Korrelation zur Schätzung der latenten Korrelation von künstlich in Kategorien eingeteilten Variablen auch viele weitere Varianten. Für die multiple Korrelation i.S. der Wurzel aus dem Determinationskoeffizienten R^2 in der multiplen linearen Regression s. Abschn. 6.2.2. Die kanonische Korrelation zweier Gruppen von Variablen, die an denselben Beobachtungsobjekten erhoben wurden, ermittelt `cancor()`.

R hierfür keine eigene Funktion bereitstellt.²⁸ 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, s. Abschn. 6.7.

```
> NN <- 100
> zz <- runif(NN)
> xx <- zz + rnorm(NN, 0, 0.5)
> yy <- zz + rnorm(NN, 0, 0.5)
> (cor(xx, yy) - (cor(xx, zz)*cor(yy, zz))) /
+   sqrt((1-cor(xx, zz)^2) * (1-cor(yy, zz)^2))
[1] 0.0753442
```

Die Semipartialkorrelation einer Variable Y mit einer Variable X ohne eine dritte Variable Z unterscheidet sich von der Partialkorrelation dadurch, dass nur von X der i.S. der linearen Regression durch Z aufklärbare Varianzanteil auspartialisiert wird, nicht aber von Y . Die Semipartialkorrelation berechnet sich durch $r_{(X.Z)Y} = \frac{r_{XY} - (r_{XZ} \cdot r_{YZ})}{\sqrt{1 - r_{XZ}^2}}$, oder als Korrelation von Y mit den Residuen der Regression von X auf Z (Abschn. 6.7).

```
> (cor(xx, yy) - (cor(xx, zz) * cor(yy, zz))) / sqrt(1-cor(xx, zz)^2)
[1] 0.06275869
```

2.7.9 Robuste Streuungsmaße und Kovarianzschatzer

Ausreißer können gewöhnliche Streuungen und Kovarianzen in dem Sinne stark verzerren, dass sie nicht mehr gut die Eigenschaften der zugrundeliegenden Verteilung der beteiligten Variablen repräsentieren. Aus diesem Grund existieren Streuungsmaße und Kovarianzschatzer, die weniger stark durch einzelne extreme Werte beeinflusst werden.

Mit `IQR(<Vektor>)` wird der Interquartilabstand erfragt, also die Differenz vom dritten und ersten Quartil.

```
> age <- c(17, 30, 30, 25, 23, 21)
> sd(age)                      # Vergleich: gewöhnliche Streuung
[1] 5.125102

> quantile(age)                # Quartile
  0%    25%    50%    75%   100%
17.00  21.50  24.00  28.75  30.00

> IQR(age)                     # Interquartilabstand
[1] 7.25
```

²⁸Wohl aber das Paket `correlation`.

Die mittlere absolute Abweichung vom Mittelwert oder Median ist manuell oder über `MeanAD()` aus dem Paket `DescTools` zu berechnen. Dagegen steht für den Median der absoluten Abweichungen vom Median `mad(<Vektor>)` bereit.²⁹

```
> mean(abs(age - mean(age))) # mittlere abs. Abweichung vom Mittelwert
[1] 4

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

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

Bei der Winsorisierung von Daten mit `Winsorize()` aus dem Paket `DescTools` wird ein bestimmter Anteil an Extremwerten auf beiden Seiten der Verteilung durch jeweils den letzten Wert ersetzt, der noch nicht als Extremwert gilt. Dafür legt das Argument `probs` in Form eines Vektors die beiden Quantile als Grenzen fest. Die übliche Varianz dieser Daten ist dann die winsorisierte Varianz.

```
> library(DescTools) # für Winsorize()
> ageWins <- Winsorize(age, probs=c(0.2, 0.8)) # winsorisierte Daten
> var(ageWins) # winsorisierte Varianz
[1] 17.2
```

Weitere robuste Streuungsschätzer stellt das Paket `robustbase` mit `Sn()`, `Qn()` und `scaleTau2()` bereit. Für robuste Schätzer einer theoretischen Kovarianzmatrix vgl. `covOGK()` und `covMcd()` aus demselben Paket. Die mittlere absolute Differenz nach Gini kann mit `GiniMd()` aus dem Paket `rms` ([Harrell Jr, 2020b](#)) ermittelt werden.

2.7.10 Kennwerte getrennt nach Gruppen berechnen

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 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 `ave()` und `tapply()` herangezogen werden.³⁰

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

²⁹Die Funktion multipliziert den eigentlichen Median der absoluten Abweichungen mit dem Faktor 1.4826, der über das Argument `constant` auf einen anderen Wert gesetzt werden kann. Der Faktor ist so gewählt, dass der Kennwert bei normalverteilten Variablen asymptotisch mit der Streuung übereinstimmt, da für standardnormalverteilte Variablen X gilt: $E(MAD(X)) = 0.6745 = \frac{1}{1.4826}$.

³⁰Abschnitte [3.3.12](#) und [3.4.10](#) illustrieren das analoge Vorgehen in Situationen, wenn die Variablen Teil eines Datensatzes sind.

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, deren Komponenten die einzelnen Faktoren sind (Abschn. 2.10). 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 `mean()` angewendet.

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 (Abschn. 7.4.6).

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.

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

> IQ <- round(rnorm(Njk*P*Q, mean=100, sd=15))
> ave(IQ, sex, FUN=mean)          # Mittelwerte nach Geschlecht
[1] 96.83333 100.33333 96.83333 100.33333 96.83333 100.33333 96.83333
[8] 100.33333 96.83333 100.33333 96.83333 100.33333
```

Die Ausgabe von `tapply()` dient der Übersicht über die gruppenweise berechneten Kennwerte, wobei das Ergebnis ein Objekt der Klasse `array` ist (Abschn. 2.9). Bei einem einzelnen Gruppierungsfaktor ist dies einem benannten Vektor ähnlich und bei zweien einer zweidimensionalen Kreuztabelle (Abschn. 2.12).

```
> (tapRes <- tapply(IQ, group, FUN=mean))      # Mittelwert pro Gruppe
      CG      T      WL
94.25 99.75 101.75

# Mittelwert pro Bedingungskombination
> tapply(IQ, list(sex, group), FUN=mean)
      CG      T      WL
f  98.0  91.0 101.5
m  90.5 108.5 102.0
```

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 (Abschn. 2.10): 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 die Funktion `identity()` verwendet, die ihr Argument unverändert ausgibt (Abschn. 3.3.7 für `split()`).

```
> tapply(IQ, sex, FUN=identity)          # IQ-Werte pro Geschlecht
$f
[1] 100 82 88 115 86 110

$m
[1] 120 97 97 107 80 101

> split(IQ, sex)                      # Kontrolle ...
> IQ[sex == "f"]                      # Kontrolle ...
> IQ[sex == "m"]                      # Kontrolle ...
```

2.7.11 Funktionen auf geordnete Paare von Werten anwenden

Eine Verallgemeinerung der Anwendung einer Funktion auf jeden Wert eines Vektors stellt die Anwendung einer Funktion 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.³¹ Da Operatoren nur Funktionen mit besonderer Schreibweise sind, können sie hier ebenfalls eingesetzt werden, müssen dabei aber in Anführungszeichen stehen (Abschn. 1.2.5, Fußnote 20). 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 Matrix (Abschn. 2.8).

Als Beispiel sollen alle Produkte der Zahlen 1–5 als Teil des kleinen 1×1 ausgegeben werden.

```
> outer(1:5, 1:5, FUN="*")
     [,1]  [,2]  [,3]  [,4]  [,5]
[1,]    1     2     3     4     5
[2,]    2     4     6     8    10
[3,]    3     6     9    12    15
[4,]    4     8    12    16    20
[5,]    5    10    15    20    25
```

³¹Vor dem Aufruf von FUN verlängert `outer()` X und Y so, dass beide die Länge `length(X)*length(Y)` besitzen und sich aus der Kombination der Elemente mit gleichem Index alle geordneten Paare ergeben.

2.8 Matrizen

Wenn für jedes Beobachtungsobjekt Daten von mehreren Variablen vorliegen, könnten die Werte jeder Variable in einem separaten Vektor gespeichert werden. Eine andere Möglichkeit zur gemeinsamen Repräsentation aller Variablen bieten Objekte der Klasse `matrix` als Spezialfall von arrays (Abschn. 2.9).³²

```
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. Fehlt eine Angabe entweder von `nrow` oder `ncol`, wird das fehlende Argument passend zur Länge des Vektors und zum jeweils anderen Argument gesetzt. 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.³³ 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

³²Eine 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 Matrizen intern lediglich Vektoren mit einem Attribut (Abschn. 1.4), das Auskunft über die Dimensionierung der Matrix, also die Anzahl ihrer Zeilen und Spalten liefert. Eine Matrix kann deshalb maximal so viele Werte speichern, wie ein Vektor Elemente besitzen kann (Abschn. 2.1.1, Fußnote 1). Für Rechenoperationen mit Matrizen im Kontext der linearen Algebra s. Abschn. 12.1.

³³Indem einer Liste (Abschn. 2.10) eine Dimensionierung als Attribut hinzugefügt wird, lässt sich diese Einschränkung indirekt umgehen: `mat <- list(1, "X")`; `dim(mat) <- c(2, 1)` erstellt eine heterogene Matrix. Dies bietet jedoch in üblichen Anwendungen keine Vorteile.

umgewandelt, was an den hinzugekommenen Anführungszeichen zu erkennen ist.³⁴ Auf die ehemals numerischen Werte können dann keine Rechenoperationen mehr angewendet werden. Dieser Umstand macht Matrizen letztlich weniger geeignet für empirische Datensätze, für die stattdessen Objekte der Klasse `data.frame` bevorzugt werden sollten (Abschn. 2.11).³⁵

2.8.2 Dimensionierung, Zeilen und Spalten

Die Dimensionierung einer Matrix (die Anzahl ihrer Zeilen und Spalten) liefert die Funktion `dim(<Matrix>)`, die auch auf arrays (Abschn. 2.9) oder Datensätze (Abschn. 2.11) anwendbar ist. Sie gibt einen Vektor aus, der die Anzahl der Zeilen und Spalten in dieser Reihenfolge als Elemente besitzt. Über `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

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

Eine Matrix wird mit `t(<Matrix>)` transponiert, wodurch ihre Zeilen zu den Spalten der Transponierten und entsprechend ihre Spalten zu Zeilen der Transponierten werden.

```
> t(ageMat)
 [,1]  [,2]
[1,]    17    30
[2,]    30    25
[3,]    23    21
```

Wird ein Vektor über `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(1:3)
 [,1]
[1,]    1
[2,]    2
[3,]    3
```

³⁴Allgemein gesprochen werden alle Elemente in den umfassendsten Datentyp umgewandelt, der notwendig ist, um alle Werte ohne Informationsverlust zu speichern (Abschn. 1.4.5).

³⁵Da Matrizen numerisch effizienter als Objekte der Klasse `data.frame` verarbeitet werden können, sind sie dagegen bei der Analyse sehr großer Datenmengen vorzuziehen.

Um eine Matrix in einen Vektor umzuwandeln, sollte entweder `as.vector(<Matrix>)` oder einfach `c(<Matrix>)` verwendet werden.³⁶ Die Anordnung der Elemente entspricht dabei dem Aneinanderhängen der Spalten der Matrix.

```
> 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. Dieses Ziel lässt sich mit den `row(<Matrix>)` und `col(<Matrix>)` erreichen.

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

> (rowMat <- row(pqMat))                # Zeilenindizes
     [,1]   [,2]   [,3]
[1,]    1     1     1
[2,]    2     2     2

> (colMat <- col(pqMat))                # Spaltenindizes
     [,1]   [,2]   [,3]
[1,]    1     2     3
[2,]    1     2     3
```

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 (Abschn. 2.8.5).

```
> cbind(rowIdx=c(rowMat), colIdx=c(colMat), 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
```

Eine andere Anwendungsmöglichkeit besteht darin, logische untere bzw. obere Dreiecksmatrizen zu erstellen, die auch von `lower.tri()` und `upper.tri()` erzeugt werden können. Diese Matrizen können später zur Auswahl von Elementen dienen (Abschn. 2.8.4).

```
> mat <- matrix(sample(1:10, 16, replace=TRUE), nrow=4, ncol=4)
> col(mat) >= row(mat)                  # obere Dreiecksmatrix
```

³⁶`c()` entfernt die Attribute der übergebenen Argumente bis auf ihre Elementnamen. Matrizen verlieren damit ihre Dimensionierung `dim` und ihre Klasse `matrix`.

```
[,1] [,2] [,3] [,4]
[1,] TRUE  TRUE  TRUE  TRUE
[2,] FALSE TRUE  TRUE  TRUE
[3,] FALSE FALSE TRUE  TRUE
[4,] FALSE FALSE FALSE TRUE
```

2.8.3 Elemente auswählen und verändern

In einer Matrix ist es ähnlich wie bei einem Vektor möglich, sich einzelne Elemente mit dem [$\langle\text{Zeile}\rangle$, $\langle\text{Spalte}\rangle$] Operator anzeigen zu lassen. Der erste Index in der eckigen Klammer gibt dabei die Zeile des gewünschten Elements an, der zweite dessen Spalte.³⁷

```
> ageMat
[,1] [,2] [,3]
[1,] 17   30   23
[2,] 30   25   21

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

Analog zum Vorgehen bei Vektoren können auch bei Matrizen einzelne Elemente unter Angabe ihres Zeilen- und Spaltenindex durch die Zuweisung eines Wertes verändert werden. Fehlt bei Zuweisungen der Index $\langle\text{Matrix}\rangle[]$, werden alle Elemente der Matrix ersetzt. Wenn der zugewiesene Vektor dabei weniger Elemente als die Matrix besitzt, wird er automatisch passend verlängert (Abschn. 2.5.4).

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

> ageMatCopy <- ageMat           # zu verändernde Kopie der Matrix

# Veränderung mit zyklischer Verlängerung
> ageMatCopy[] <- c(1, 2, 3) # nicht verwechseln mit ageMatCopy <- ...
> ageMatCopy
[,1] [,2] [,3]
[1,]    1    3    2
[2,]    2    1    3
```

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. Dabei können auch beide Dimensionen weggelassen werden. Um einzelne Zeilen oder Spalten zu entfernen, ist ihr Index mit vorangestelltem Minus-Zeichen zu verwenden.

³⁷Für Hilfe zu diesem Thema vgl. `?Extract`.

```
> ageMat[2, ]                      # Werte 2. Zeile
[1] 30 24 21

> ageMat[, 1]                      # Werte 1. Spalte
[1] 17 30

> ageMat[, ]                        # gesamte Matrix
 [,1]  [,2]  [,3]
[1,] 17    30    23
[2,] 30    24    21

> ageMat[, -1]                     # Werte bis auf 1. Spalte
 [,1]  [,2]
[1,] 30    23
[2,] 25    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.³⁸

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

Aufgrund der Voreinstellung `drop=TRUE` beim Indizieren unterscheidet sich bei einer Matrix $\langle m \rangle$ mit nur einer Zeile bzw. nur einer Spalte die Ausgabe von m (gleichbedeutend: $\langle m \rangle []$) von $\langle m \rangle[,]$.

```
> rowMat <- matrix(1:3, nrow=1)      # Matrix mit 1 Zeile
> rowMat                                # Ausgabe: Matrix
 [,1]  [,2]  [,3]
[1,] 1    2    3

> rowMat[]                                # Ausgabe: Matrix
 [,1]  [,2]  [,3]
[1,] 1    2    3

> rowMat[, ]                            # Ausgabe ohne drop=FALSE: Vektor
[1] 1 2 3
```

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.

³⁸Dagegen bleibt `ageMat[FALSE, FALSE]` eine leere Matrix mit 0 Zeilen und 0 Spalten.

```

> ageMat[, 2:3]                                # 2. und 3. Spalte
   [,1]  [,2]
[1,] 30   23
[2,] 24   21

> ageMat[, c(1, 3)]                          # 1. und 3. Spalte
   [,1]  [,2]
[1,] 17   23
[2,] 30   21

> ageMatNew <- ageMat
> (replaceMat <- matrix(c(11, 21, 12, 22), nrow=2, ncol=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, Elemente auszuwählen und zu verändern

Auf Matrixelemente kann auch zugegriffen werden, wenn nur ein einzelner Index genannt wird. Die Matrix wird dabei implizit in den Vektor der untereinander gehängten Spalten umgewandelt.

```

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

> idxVec <- c(1, 3, 4)
> ageMat[idxVec]
[1] 17 30 24

```

Weiter können Matrizen auch durch eine logische Matrix derselben Dimensionierung indiziert werden, die für jedes Element bestimmt, ob es ausgegeben werden soll. Solche Matrizen können das Ergebnis eines logischen Vergleichs oder der Funktionen `lower.tri()` bzw. `upper.tri()` sein. Das Ergebnis ist ein Vektor der ausgewählten Elemente.

```

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

```

```
> ageMat[idxMatLog]
[1] 30 30
```

Schließlich ist es möglich, eine zweispaltige numerische 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. Eine solche Matrix entsteht etwa bei der Umwandlung einer logischen in eine numerische Indexmatrix mittels `which()` (Abschn. 2.2.2), wenn das Argument `arr.ind=TRUE` gesetzt ist. Auch hier ist das Ergebnis ein Vektor der ausgewählten Elemente.

```
> (idxMatNum <- which(idxMatLog, arr.ind=TRUE))
      row   col
[1,]    2     1
[2,]    1     2

> ageMat[idxMatNum]
[1] 30 30
```

`arrayInd(<Indexvektor>, dim(<Matrix>))` konvertiert einen numerischen Indexvektor, der eine Matrix im obigen Sinn als Vektor indiziert, in eine zweispaltige numerische Indexmatrix.

```
> (idxMat <- arrayInd(idxVec, dim(ageMat)))
      [,1] [,2]
[1,]    1    1
[2,]    1    2
[3,]    2    2
```

Die Diagonalelemente einer Matrix gibt `diag(<Matrix>)` in Form eines Vektors aus.³⁹ Abschnitt 12.1.1 zeigt den Einsatz von `diag()`, um Diagonalmatrizen zu erstellen.

```
> diag(ageMat)
[1] 17 24
```

2.8.5 Matrizen verbinden

`cbind(<Vektor1>, <Vektor2>, ...)` und `rbind(<Vektor1>, <Vektor2>, ...)` fügen Vektoren zu Matrizen zusammen. Das `c` bei `cbind()` steht für *columns* (Spalten), das `r` entsprechend für *rows* (Zeilen). In diesem Sinn werden die Vektoren mit `cbind()` spaltenweise nebeneinander, und mit `rbind()` zeilenweise untereinander angeordnet. R verlängert Vektoren ungleicher Länge dabei zyklisch und gibt eine Warnung aus. Zeilen und Spalten können beim Erstellen auch in der Form `<Name>=<Vektor>` benannt werden.

```
> vec1 <- c(19, 19, 19, 31, 24)
> vec2 <- c(95, 76, 76, 94, 76)
> vec3 <- c(197, 179, 186, 189, 173)
> rbind(vec1, vec2, vec3)
```

³⁹Bei nicht quadratischen $(p \times q)$ -Matrizen X sind dies die Elemente x_{11}, \dots, x_{pp} (für $p < q$) bzw. x_{11}, \dots, x_{qq} (für $p > q$).

```
[,1] [,2] [,3] [,4] [,5]
vec1   19    19    31    19    24
vec2   95    76    94    76    76
vec3  197   178   189   184   173

> (mat <- cbind(age=vec1, weight=vec2, height=vec3))
      age  weight  height
[1,]  19      95     197
[2,]  19      76     178
[3,]  31      94     189
[4,]  19      76     184
[5,]  24      76     173
```

2.8.6 Matrizen sortieren

Die Zeilen von Matrizen können mit Hilfe von `order()` entsprechend der Reihenfolge der Werte in einer oder mehrerer ihrer Spalten sortiert werden. Die Funktion `sort()` 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: In der 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 (Abschn. 2.5.1).⁴⁰

```
> (rowOrder1 <- order(mat[, "age"]))           # Kriterium: Alter
[1] 1 2 4 5 3
```

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, ]
      age  weight  height
[1,]  19      95     197
[2,]  19      76     178
[3,]  19      76     184
[4,]  24      76     173
[5,]  31      94     189
```

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

⁴⁰Die Funktion sortiert *stabil*: Zeilen mit gleich großen Werten des Sortierkriteriums behalten ihre Reihenfolge relativ zueinander bei, werden also beim Sortiervorgang nicht zufällig vertauscht.

innerhalb jeder Gruppe nach der Reihenfolge der Werte einer anderen Variable. Es können noch weitere Sortierkriterien durch Komma getrennt als Argumente vorhanden sein, es gibt also keine Beschränkung auf nur zwei solcher Kriterien.

```
# sortiere primär nach Alter und sekundär nach Gewicht
> rowOrder2 <- order(mat[, "age"], partial=mat[, "weight"])
> mat[rowOrder2, ]
    age   weight   height
[1,] 19       76      178
[2,] 19       76      184
[3,] 19       95      197
[4,] 24       76      173
[5,] 31       94      189
```

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 numerischen Kriterien ein – vorangestellt werden, was als Umkehrung der mit `decreasing` eingestellten Reihenfolge zu verstehen ist.

```
# sortiere aufsteigend nach Gewicht und absteigend nach Größe
> rowOrder3 <- order(mat[, "weight"], -mat[, "height"])
> mat[rowOrder3, ]
    age   weight   height
[1,] 19       76      184
[2,] 19       76      178
[3,] 24       76      173
[4,] 31       94      189
[5,] 19       95      197
```

2.8.7 Randkennwerte berechnen

Die Summe aller Elemente einer numerischen Matrix lässt sich mit `sum(<Matrix>)`, die separat über jede Zeile oder jede Spalte gebildeten Summen durch `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.

```
> sum(mat)                                # Summe aller Elemente
[1] 1450

> rowSums(mat)                            # Summen jeder Zeile
[1] 311 273 314 279 273

> mean(mat)                               # Mittelwert aller Elemente
[1] 96.66667

> colMeans(mat)                           # Mittelwerte jeder Spalte
```

```
age  weight  height
22.4    83.4   184.2
```

2.8.8 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 `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 Kennwerte der Zeilen (1) oder Spalten (2) 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 von `FUN`, die an diese Funktion weitergereicht werden.

```
> apply(mat, 2, sum)                      # Summen jeder Spalte
age  weight  height
112      417     921

> apply(mat, 1, max)                      # Maximum jeder Zeile
[1] 197 178 189 184 173

> apply(mat, 1, range)                    # Range jeder Zeile
 [,1]  [,2]  [,3]  [,4]  [,5]
[1,]    19     19     31     19     24
[2,]   197    178    189    184    173

> apply(mat, 2, mean, trim=0.1)    # gestutzter Mittelwert jeder Spalte
age  weight  height
22.4    83.4   184.2
```

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

2.8.9 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 jedem Wert 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 mit `sweep()` durchführen.

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

Das Argument `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. Bei Objekten mit benannten Dimensionen ist auch der zugehörige Name der gemeinten Dimension möglich – etwa bei mit `xtabs()` erstellten Kreuztabellen (Abschn. 2.12). 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, Operatoren müssen dabei in Anführungszeichen stehen. Voreinstellung ist die Subtraktion `"+"`. Die drei Punkte `...` stehen für optionale, ggf. durch Komma getrennte Argumente von `FUN`, die an diese Funktion weitergereicht werden.

Im Beispiel sollen die Daten einer Matrix erst spaltenweise, dann zeilenweise zentriert werden (s. Abschn. 2.11.3 für `dimnames()`).

```
> Mj <- rowMeans(mat)                      # Mittelwerte der Zeilen
> Mk <- colMeans(mat)                     # Mittelwerte der Spalten

# benenne Dimensionen
> dimnames(mat) <- list(obs=NULL, vars=c("age", "weight", "height"))
> sweep(mat, "vars", Mk, "-")             # spaltenweise zentrieren
   age   weight   height
[1,] -3.4    11.6   12.8
[2,] -3.4    -7.4   -6.2
[3,]  8.6    10.6    4.8
[4,] -3.4    -7.4   -0.2
[5,]  1.6    -7.4   -11.2

> scale(mat, center=TRUE, scale=FALSE)      # Kontrolle mit scale() ...
> sweep(mat, 1, Mj, "-")                  # zeilenweise zentrieren
   age   weight   height
[1,] -84.66667 -8.666667 93.33333
[2,] -72.00000 -15.000000 87.00000
[3,] -73.66667 -10.666667 84.33333
[4,] -74.00000 -17.000000 91.00000
[5,] -67.00000 -15.000000 82.00000

> t(scale(t(mat), center=TRUE, scale=FALSE)) # Kontrolle mit scale()
# ...
```

2.8.10 Kovarianz- und Korrelationsmatrizen

Zum Erstellen von Kovarianz- und Korrelationsmatrizen können die aus Abschn. 2.7.8 bekannten Funktionen `cov(<Matrix>)` und `cor(<Matrix>)` verwendet werden, wobei die Werte in Form einer Matrix mit Variablen in den Spalten übergeben werden müssen. `cov()` liefert die korrigierte Kovarianzmatrix.

```
> cov(mat)                                # Kovarianzmatrix
      age   weight   height
age    27.80   22.55    0.4
weight 22.55  102.80   82.4
height  0.40   82.40   87.7

> cor(mat)                                # Korrelationsmatrix
      age       weight       height
age    1.0000000000  0.4218204  0.008100984
weight 0.421820411  1.0000000  0.867822404
height 0.008100984  0.8678224  1.000000000
```

`cov2cor(<K>)` wandelt eine korrigierte Kovarianzmatrix \mathbf{K} in eine Korrelationsmatrix um. Analog erzeugt `cor2cov(<R>, <s>)` aus dem Paket MBESS ([Kelley, 2020](#)) aus einer Korrelationsmatrix \mathbf{R} und einem Vektor der Streuungen \mathbf{s} der Variablen die zugehörige Kovarianzmatrix. Weiterhin existiert mit `cov.wt()` eine Funktion, die direkt die unkorrigierte Kovarianzmatrix ermitteln kann.

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

Unter `x` ist die Matrix einzutragen, von deren Spalten paarweise die Kovarianz bestimmt werden soll. Um diese Funktion auf einen einzelnen Vektor anwenden zu können, muss dieser zunächst mit `as.matrix(<Vektor>)` in eine einspaltige Matrix konvertiert werden. Mit `method` lässt sich wählen, ob die korrigierten oder unkorrigierten Varianzen und Kovarianzen ausgerechnet werden – Voreinstellung ist "unbiased" für die korrigierten Kennwerte. Sind die unkorrigierten Kennwerte gewünscht, ist "ML" zu wählen, da sie bei normalverteilten Variablen die Maximum-Likelihood-Schätzung der theoretischen Parameter auf Basis einer Stichprobe darstellen.

Das Ergebnis der Funktion ist eine Liste (Abschn. [2.10](#)), 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(mat, method="ML")
$cov
      age   weight   height
age    22.24   18.04    0.32
weight 18.04   82.24   65.92
height  0.32   65.92   70.16

$center
      age   weight   height
22.4     83.4   184.2

$n.obs
[1] 5
```

Mit `diag(<Matrix>)` lassen sich aus einer Kovarianzmatrix die in der Diagonale stehenden Varianzen extrahieren (Abschn. [12.1.1](#)).

```
> diag(cov(mat))
  age   weight   height
27.8    102.8    87.7
```

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

```
> vec <- rnorm(nrow(mat))
> cor(mat, vec)
      [,1]
age     -0.1843847
weight  -0.6645798
height  -0.6503452
```

2.9 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 die in einem array gespeicherten Werte alle denselben Datentyp aufweisen, wie es auch bei Vektoren und Matrizen der Fall ist (Abschn. 2.8.1).⁴¹

```
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 festgelegt, also die Anzahl der Dimensionen und der Werte pro Dimension. 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 Dimensionen und die einzelnen Stufen in jeder Dimension gleichzeitig mit Namen zu versehen. Dies geschieht unter Verwendung einer Liste (Abschn. 2.10), die für jede Dimension eine Komponente in Form eines Vektors mit den gewünschten Bezeichnungen der einzelnen Stufen besitzt. Die Namen der Dimensionen selbst können über die Benennung der Komponenten der Liste festgelegt werden.

Als Beispiel soll die Kontingenztafel dreier kategorialer Variablen dienen: Geschlecht mit zwei Stufen und zwei weitere Variablen mit drei bzw. zwei Stufen. Ein dreidimensionales array wird durch separate zweidimensionale Matrizen für jede Stufe der dritten Dimension ausgegeben.

⁴¹So, wie sich Matrizen mit `cbind()` und `rbind()` aus Vektoren zusammenstellen lassen, ermöglicht `abind()` aus dem gleichnamigen Paket (Plate & Heiberger, 2016) das Verbinden von Matrizen zu einem array.

```
> (myArr1 <- array(1:12, dim=c(2, 3, 2), dimnames=list(row=c("f", "m"),
+                                         column=c("CG", "WL", "T"), layer=c("high", "low"))))
, , layer = high
  column
row CG  WL  T
f   1   3   5
m   2   4   6

, , layer = low
  column
row 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[ , , "high"]      # zeige nur 1. Schicht
  column
row 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(<array>, perm=<Vektor>)` auch bei arrays Dimensionen ausgetauscht werden. Als erstes Argument ist das zu transformierende p -dimensionale array anzugeben. `perm` legt in Form eines Vektors mit den Elementen 1 bis p fest, welche Dimensionen vertauscht werden sollen. Die Position eines Elements von `perm` bezieht sich auf die alte Dimension, das Element selbst 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. Im Fall benannter Dimensionen kann `perm` statt der Werte 1, ..., p auch die Namen der Dimensionen enthalten.

```
# vertausche Zeile und Spalte
> aperm(myArr1, perm=c("column", "row", "layer"))
, , layer = high
  row
```

```

column f m
CG 1 2
WL 3 4
T 5 6

, , layer = low
      row
column f m
CG 7 8
WL 9 10
T 11 12

```

2.10 Listen

Vektoren, Matrizen und arrays sind dahingehend eingeschränkt, dass sie gleichzeitig nur Werte desselben Datentyps aufnehmen können. Da in empirischen Erhebungen meist Daten unterschiedlichen Typs – etwa numerische Variablen, Faktoren und Zeichenketten – anfallen, sind sie nicht unmittelbar geeignet, vollständige Datensätze zu speichern. Objekte der Klasse `list` sind in dieser Hinsicht flexibler: Sie erlauben es, gleichzeitig Variablen unterschiedlichen Datentyps und auch unterschiedlicher Klasse als Komponenten zu besitzen.

Listen eignen sich zur Repräsentation heterogener Sammlungen von Daten und werden deshalb von vielen Funktionen genutzt, um ihr Ergebnis zurückzugeben. Listen sind darüber hinaus die allgemeine Grundform von Datensätzen (Klasse `data.frame`), der gewöhnlich am besten geeigneten Struktur für empirische Daten (Abschn. 2.11).

Listen werden mit dem Befehl `list(<Komponente1>, <Komponente2>, ...)` erzeugt, wobei für jede Komponente ein (ggf. bereits bestehendes) Objekt zu nennen ist. Alternativ lässt sich eine Liste mit `<Anzahl>` vielen leeren Komponenten über `vector("list", <Anzahl>)` erstellen. Komponenten einer Liste können Objekte jeglicher Klasse und jedes Datentyps, also auch selbst wieder Listen sein. Die erste Komponente könnte also z. B. ein numerischer Vektor, die zweite ein Vektor von Zeichenketten und die dritte eine Matrix aus Wahrheitswerten sein. Die von `length(<Liste>)` ausgegebene Länge einer Liste ist die Anzahl ihrer Komponenten auf oberster Ebene, `lengths(<Liste>)` nennt die jeweilige Länge der Listenkomponenten.

```

> myList1 <- list(c(1,3), c(12,8,29,5))    # Liste erstellen
> length(myList1)                           # Anzahl Komponenten
[1] 2

> lengths(myList1)                         # Länge der Komponenten
[1] 2 4

> vector("list", 2)                        # Liste: 2 leere Komponenten
[[1]]
NULL

```

```
[[2]]
NULL
```

2.10.1 Komponenten auswählen und verändern

Um auf eine Listen-Komponente zuzugreifen, kann der `[[<Index>]]` Operator benutzt werden, der als Argument die Position der zu extrahierenden Komponente in der Liste benötigt. Dabei kann der Index auch in einem Objekt gespeichert sein. `[[<Index>]]` gibt immer nur eine Komponente zurück, selbst wenn mehrere Indizes in Form eines Indexvektors übergeben werden.⁴² Die zweite Komponente einer Liste könnte also so ausgelesen werden:

```
> myList1[[2]]
[1] 12 8 29 5

> idx <- 2
> myList1[[idx]]
[1] 12 8 29 5
```

Einzelne Elemente 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
```

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(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
```

⁴²Für Hilfe zu diesem Thema vgl. `?Extract`.

Auch bei Listen kann der [$\langle\text{Index}\rangle$] Operator verwendet werden, der immer ein Objekt des selben Datentyps zurückgibt wie den des indizierten Objekts. Im Unterschied zu [[$\langle\text{Index}\rangle$]] liefert er deshalb nicht die Komponente selbst zurück, sondern eine Liste, die wiederum als einzige Komponente das gewünschte Objekt besitzt – also gewissermaßen eine Teilliste ist. Wickham (2019a) verwendet die Analogie eines Zuges, bei dem die Waggons den Komponenten einer Liste entsprechen: [] koppelt Waggons vom Zug ab, während [[]] den Inhalt eines Waggons extrahiert. Während myList2[[2]] also eine numerische Matrix ausgibt, ist das Ergebnis von myList2[2] eine Liste, deren einzige Komponente diese Matrix ist.

```
> myList2[[2]]      # Komponente: numerische Matrix -> Datentyp numeric
   [,1]  [,2]
[1,]    1    3
[2,]    2    4

> mode(myList2[[2]])
[1] "numeric"

> myList2[2]        # Teilliste -> Datentyp list
[[1]]
   [,1]  [,2]
[1,]    1    3
[2,]    2    4

> mode(myList2[2])
[1] "list"
```

Wie auch bei Vektoren können die Komponenten einer Liste benannt sein und mittels $\langle\text{Liste}\rangle[[\text{"Variablenname"}]]$ über ihren Namen ausgewählt werden.

```
> (myList3 <- list(numvec=1:5, word="dolor"))
$numvec
[1] 1 2 3 4 5

$word
[1] "dolor"

> myList3[["word"]]
[1] "dolor"
```

Welche Namen die Komponenten einer Liste tragen, erfährt man mit `names(<Liste>)`. Ob eine bestimmte Komponente in einer Liste enthalten ist, gibt `hasName(<Liste>, "<Name>")` aus. `setNames(<Liste>, c("<Name1>", ...))` ändert die Namen der Komponenten der übergebenen Liste auf die Elemente des als zweites Argument genannten Vektors von Zeichenketten. Die Funktion verändert die Liste selbst nicht, sondern liefert eine modifizierte Liste zurück.

```
> hasName(myList3, "numbers")
[1] FALSE

> (myList4 <- setNames(myList3, c("numbers", "chars")))
```

```
$numbers
[1] 1 2 3 4 5

$chars
[1] "dolor"
```

Wenn man auf eine benannte Komponente zugreifen will, kann dies mittels `[]` Operator und ihrem numerischen Index oder ihrem Namen geschehen, zusätzlich aber auch über den Operator `Liste$<Name>`. Dieser bietet den Vorteil, dass er ohne Klammern und numerische Indizes auskommt und damit recht übersichtlich ist. Nur wenn der Name Leerzeichen enthält, wie es bisweilen bei von R zurückgegebenen Objekten der Fall ist, muss er zudem in Anführungszeichen stehen. `$` benötigt immer den Variablenamen selbst, anders als mit `[]` kann kein Objekt verwendet werden, das den Namen speichert.

```
> myList3$numvec
[1] 1 2 3 4 5

> 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

> component <- "n.obs"                # Anzahl Beobachtungen
> retList[[component]]
[1] 10
```

In automatisiert verarbeiteten Befehlsskripten ist es sicherer, die Variante `[["<Name>"]]` zu verwenden. Dies liegt daran, dass bei `$<Name>` unvollständige Anfänge von Variablennamen akzeptiert und automatisch ergänzt werden, wenn dies eindeutig möglich ist. Anfänge von Variablennamen mit mehreren möglichen Ergänzungen resultieren dagegen im Ergebnis `NULL`. Dieses *partial matching* Verhalten kann zu unentdeckten Fehlern in der Variablenauswahl führen.

```
> retList$n                         # wird ergänzt zu n.obs
[1] 10

> retList$c                          # nicht eindeutig ergänzbar
NULL
```

2.10.2 Komponenten hinzufügen und entfernen

Auf dieselbe Weise, wie sich die Komponenten einer Liste anzeigen lassen, können auch weitere Komponenten zu einer bestehenden Liste hinzugefügt werden, also mit

- `<Liste>[[<Index>]]`
- `<Liste>[["<neue Komponente>"]]`
- `<Liste>$<neue Komponente>`

```
> myList1[[3]]           <- LETTERS[1:5]    # 1. neue Komponente
> myList1[["neuKomp2"]] <- letters[1:5]   # 2. neue Komponente
> myList1$neuKomp3      <- 100:105       # 3. neue Komponente
> myList1
[[1]]
[1] 1 3

[[2]]
[1] 12 8 29 5

[[3]]
[1] "A" "B" "C" "D" "E"

$neuKomp2
[1] "a" "b" "c" "d" "e"

$neuKomp3
[1] 100 101 102 103 104 105
```

Um die Komponenten mehrerer Listen zu einer Liste zu verbinden, eignet sich wie bei Vektoren `c(<Liste1>, <Liste2>, ...)`.

```
> myListJoin <- c(myList1, myList2)          # verbinde Listen
```

Die Komponente einer Liste wird gelöscht, indem ihr die leere Menge `NULL` zugewiesen wird – bei Verwendung des `[<Index>]` Operators ist dies auch für mehrere Komponenten gleichzeitig möglich.

```
> myList1$neuKomp3 <- NULL                  # lösche Komponente
> myListJoin[c("neuKomp2", "neuKomp3")] <- NULL  # lösche Komponenten
```

2.10.3 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. Um aus einem solchen Objekt einen bestimmten Wert zu erhalten, kann 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 Verästelungsebene befindet.

```
# 4 Komponenten in der zweiten Ebene
> 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)

# 2 Komponenten in der ersten Ebene
> myListA <- list(AA=myListAA, AB=myMatAB)
> myListB <- list(BA=myListBA, BB=myVecBB)

# Gesamtliste
> 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

# Struktur der 2. Komponente (B) der 1. Ebene
> str(myList4$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

# Struktur der 3. Komponente (BA) der 2. Ebene
> str(myList4$B$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"

# Element der 1. Komponente (BAA) der 3. Ebene
> myList4$B$BA$BAA[4, 2]
```

```
[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 (Abschn. 1.4.5).

```
> 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"
```

Das Argument `recursive=FALSE` führt dazu, dass nur die oberste Verästelungsebene entfernt wird.

```
> myList6 <- list(L1=list(L1a=1:3,           L1b=LETTERS[1:3]),
+                  L2=list(L2a=rnorm(2), L2b=letters[7:8]))

> str(myList6)
List of 2
 $ L1:List of 2
   ..$ L1a: int [1:3] 1 2 3
   ..$ L1b: chr [1:3] "A" "B" "C"
 $ L2:List of 2
   ..$ L2a: num [1:2] -2.17 1.49
   ..$ L2b: chr [1:2] "g" "h"

> unlist(myList6, recursive=FALSE)
$L1.L1a
[1] 1 2 3

$L1.L1b
[1] "A" "B" "C"

$L2.L2a
[1] -2.166634  1.491034

$L2.L2b
[1] "g" "h"
```

2.11 Datensätze

Ein Datensatz ist eine spezielle Liste und besitzt die Klasse `data.frame` sowie den von `mode()` ermittelten Datentyp `list`. Datensätze erben die Grundeigenschaften einer Liste, besitzen aber bestimmte einschränkende Merkmale – so müssen ihre 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 zuordnen, von dem sie stammen. Da Datensätze gewöhnliche Objekte sind, ist es im Gegensatz zu einigen anderen Statistikprogrammen möglich, mit mehreren von ihnen gleichzeitig zu arbeiten.

Der Basisumfang von R beinhaltet im automatisch geladenen Paket **datasets** viele vorbereitete Datensätze, an denen sich statistische Auswertungsverfahren erproben lassen, vgl. `data()` für eine Übersicht. Weitere Datensätze werden durch Zusatzpakete bereitgestellt und lassen sich mit `data(<Datensatz>, package="<Paketname>")` laden.⁴³ Nähere Erläuterungen zu einem dokumentierten Datensatz gibt `help(<Datensatz>)` aus. Abschnitt 3.1 zeigt, wie Datensätze aus externen Quellen eingelesen werden können. Die notwendigen Schritte, um sie anschließend aufzubereiten und zu aggregieren, werden in Abschn. 3.3 und 3.4 dargestellt.

Objekte der Klasse **data.frame** sind die bevorzugte Organisationsweise für empirische Datensätze.⁴⁴ Die Komponenten spielen dabei die Rolle von Variablen. Werden Datensätze in R ausgegeben, stehen die Variablen als Komponenten in den Spalten, während jede Zeile für eine Beobachtungseinheit steht. Datensätze werden mit `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.⁴⁵ Die Objektnamen bilden die Variablennamen im Datensatz, wenn sie nicht mit `data.frame(<Name>=<Objekt>)` geändert werden.

Als Beispiel seien 12 Personen betrachtet, die zufällig auf drei Untersuchungsgruppen (Kontrollgruppe CG, Wartelisten-Gruppe WL, Treatment-Gruppe T) verteilt werden. Als Variablen werden demografische Daten, Ratings und der IQ-Wert simuliert. Zudem soll die fortlaufende Nummer jeder Person gespeichert werden.⁴⁶

```
> N      <- 12
> sex    <- sample(c("f", "m"), N, replace=TRUE)
> group  <- sample(rep(c("CG", "WL", "T"), 4), N, replace=FALSE)
> age    <- sample(18:35, N, replace=TRUE)
> IQ     <- round(rnorm(N, mean=100, sd=15))
> rating <- round(runif(N, min=0, max=6))
> (myDf1 <- data.frame(id=1:N, 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
```

⁴³Hervorzuheben sind DAAG (Maindonald & Braun, 2020), carData (Fox, Weisberg & Price, 2020) und HSAUR3 (Hothorn & Everitt, 2020).

⁴⁴Außer bei sehr großen Datensätzen, die sich effizienter als Matrix verarbeiten lassen.

⁴⁵Gleiches 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>, ...)`. Datensätze mit Listen als Variablen werden hier zur Vereinfachung nicht weiter betrachtet.

⁴⁶Die automatisierte Simulation von Datensätzen nach vorgegebenen Kriterien ist mit dem Paket **simstudy** (Goldfeld & Wujciak-Jens, 2024) möglich.

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
11	11	m	CG	20	92	1
12	12	f	T	21	98	1

In der ersten Spalte der Ausgabe befinden sich die Zeilennamen, die in der Voreinstellung mit den Zeilennummern übereinstimmen. Eine spätere Teilauswahl der Zeilen (Abschn. 3.3.3, 3.4.3) hebt diese Korrespondenz jedoch häufig auf.

Die Anzahl von Beobachtungen (Zeilen) und Variablen (Spalten) kann wie bei Matrizen mit `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 über Art und Werte aller Variablen eines Datensatzes erhält man durch `summary(<Datensatz>)`.

```
> dim(myDf1)
[1] 12  6

> nrow(myDf1)
[1] 12

> ncol(myDf1)
[1] 6

> summary(myDf1)                                # gekürzte Ausgabe
   id      sex      ...      age      IQ      ...
Min.   : 1.00  Length:12      ...   Min.   :20.00  Min.   : 81.00 ...
1st Qu.: 3.75  Class :character ...  1st Qu.:22.75  1st Qu.: 89.50 ...
Median : 6.50  Mode  :character ...  Median :25.50  Median : 93.50 ...
Mean   : 6.50                           Mean   :26.42  Mean   : 97.25 ...
3rd Qu.: 9.25                           3rd Qu.:29.25  3rd Qu.:104.50 ...
Max.   :12.00                           Max.   :35.00  Max.   :122.00 ...
```

Will man sich einen Überblick über die in einem Datensatz gespeicherten Werte verschaffen, können die Funktionen `head(<Datensatz>, n=<Anzahl>)` und `tail(<Datensatz>, n=<Anzahl>)` verwendet werden, die seine ersten bzw. letzten n Zeilen anzeigen. Mit `View(<Datensatz>)` ist es zudem möglich, ein separates Fenster – in RStudio ein Tab – mit dem Inhalt eines Datensatzes zu öffnen. Dessen Werte sind dabei vor Veränderungen geschützt.

2.11.1 Datentypen in Datensätzen

Mit `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     : chr  "f" "m" "m" "m" ...
 $ group   : chr  "T" "CG" "CG" "T" ...
 $ 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 ...
```

Ist gewünscht, dass ein `character` Vektor als Objekt der Klasse `factor` im Datensatz enthalten ist, so ist er vor oder nach der Zusammenstellung des Datensatzes manuell mit `factor()` zu konvertieren.

```
> group_fac <- factor(group)
> myDf2      <- data.frame(group_fac, IQ)
> str(myDf2)
'data.frame': 12 obs. of 2 variables:
 $ group_fac: Factor w/ 3 levels "CG","T","WL": 2 1 1 2 3 1 2 3 3 3 ...
 $ IQ        : num  112 122 95 102 82 113 92 90 88 81 ...
```

Matrizen und Vektoren können mit `as.data.frame(<Objekt>)` in einen Datensatz umgewandelt werden. Dabei sind die in Matrizen und Vektoren notwendigerweise identischen Datentypen nachträglich zu konvertieren, wenn sie eigentlich unterschiedliche Variablentypen repräsentieren. Listen können in Datensätze umgewandelt werden, wenn ihre Komponenten alle dieselbe Länge besitzen.

Umgekehrt lassen sich Datensätze mit `data.matrix(<Datensatz>)` und auch mit `as.matrix(<Datensatz>)` zu Matrizen machen, wobei alle Werte in denselben Datentyp umgewandelt werden: Bei `data.matrix()` ist der Datentyp immer `numeric`, bei `as.matrix()` der umfassendste Datentyp, der notwendig ist, um alle Werte ohne Informationsverlust zu speichern (Abschn. 1.4.5). Bei der Umwandlung in eine Liste mit `as.list(<Datensatz>)` ist dagegen keine Umwandlung der Datentypen notwendig.

2.11.2 Elemente auswählen und verändern

Um einzelne Elemente anzeigen zu lassen und diese zu ändern, lassen sich dieselben Befehle wie bei Listen verwenden. Um Teilmengen von Beobachtungen oder Variablen auszuwählen, sind aber die in Abschn. 3.3.3 bzw. 3.4.3 vorgestellten Methoden oft übersichtlicher.

```
> 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. Person
[1] 34

> myDf1$IQ[10:12] <- c(99, 110, 89) # ändere IQ-Werte Person 10-12
```

Als Besonderheit können Datensätze analog zu Matrizen mit dem Operator

- [*Index Element der Komponente*], *Index der Komponente*]

indiziert werden. Bei dieser Variante bleibt die gewohnte Reihenfolge von Zeile (entspricht einem Beobachtungsobjekt) – Spalte (entspricht einer Variable) erhalten, ist also in vielen Fällen dem [[*Index*]] Operator vorzuziehen.

```
> myDf1[3, 4]                         # 3. Element der 4. Variable
[1] 25

> myDf1[4, "group"]                   # 4. Person, Variable group
[1] T
Levels: CG T WL
```

Wie bei Matrizen gilt das Weglassen eines Index unter Beibehaltung des Kommas als Anweisung, die Werte von allen Indizes der ausgelassenen Dimension anzuzeigen.⁴⁷ Auch hier ist das Argument `drop=FALSE` notwendig, wenn ein einspaltiges Ergebnis bei der Ausgabe weiterhin ein Datensatz sein soll. Bei der Arbeit mit Indexvektoren, um Spalten eines Datensatzes auszuwählen, ist häufig im voraus nicht absehbar, ob letztlich nur eine oder mehrere Spalten auszugeben sind. Um inkonsistentes Verhalten zu vermeiden, empfiehlt es sich in solchen Situationen, `drop=FALSE` in jedem Fall zu verwenden.

```
> myDf1[2, ]                           # alle Werte der 2. Person -> Ergebnis Datensatz
  id sex group age   IQ  rating
2  2   m    CG   30  122      3

> myDf1[ , "age"]                     # alle Elemente Variable age -> Ergebnis Vektor
[1] 26 30 25 34 22 24 28 35 23 29 20 21

> myDf1[1:5, 4, drop=FALSE]           # Variable als Datensatz
  age
1  26
2  30
3  25
4  34
5  22
```

⁴⁷Das Komma ist von Bedeutung: So gibt etwa `⟨Datensatz⟩[3]` wie in Listen nicht die dritte Variable von `⟨Datensatz⟩` zurück, sondern einen Datensatz, dessen einzige Komponente diese Variable ist.

2.11.3 Namen von Variablen und Beobachtungen

Die Funktion `dimnames(<Datensatz>)` dient dazu, die Namen eines Datensatzes auf beiden Dimensionen (Zeilen und Spalten, also meist Beobachtungsobjekte und Variablen) zu erfragen und auch zu ändern.⁴⁸ 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.

```
> names(myDf1)
[1] "id" "sex" "group" "age" "IQ" "rating"
```

Die Bezeichnungen der Zeilen können mit `rownames(<Datensatz>)` ausgegeben und auch geändert werden. Abschnitt 3.3.2 bzw. 3.4.2 stellen vor, wie Variablen umbenannt werden können.

```
> (rows <- paste("Z", 1:12, sep=""))
[1] "Z1" "Z2" "Z3" "Z4" "Z5" "Z6" "Z7" "Z8" "Z9" "Z10" "Z11" "Z12"

> rownames(myDf1) <- rows
> head(myDf1)
  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
Z6 6   f     CG  24 113    0

> rownames(myDf1) <- NULL          # entferne Zeilennamen
```

2.11.4 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 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 `with()`.

⁴⁸Namen werden als Attribut gespeichert und sind mit `attributes(<Datensatz>)` sichtbar (Abschn. 1.4).

```
with(data=<Datensatz>, expr=<R-Befehle>)
```

Innerhalb der unter `expr` angegebenen Befehle sind die Variablennamen des unter `data` genannten Datensatzes bekannt. Der Datensatz selbst kann innerhalb von `with()` nicht verändert, sondern nur gelesen werden.

```
> with(myDf1, tapply(IQ, group, FUN=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)  # xtabs() -> folgender Abschnitt
   group
sex  CG  T  WL
  f    1  2  0
  m    3  2  4
```

Mit `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 (Abschn. 1.4.1).

```
> 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()` Kopien aller Variablen des Datensatzes angelegt werden.⁴⁹ Greift man daraufhin auf eine Variable `var` ohne Nennung von `myDf$` zu, so verwendet man diese Kopie. Insbesondere schlagen sich spätere Änderungen am Datensatz selbst nicht in den früher angelegten Kopien nieder, genauso wirken sich Veränderungen an den Kopien nicht auf den eigentlichen Datensatz aus. Wegen dieser Gefahr, nicht synchronisierte Änderungen vorzunehmen, sollte auf `attach()` generell verzichtet werden.

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.

⁴⁹Bei sehr großen Datensätzen empfiehlt es sich daher aus Gründen der Speichernutzung, nur eine geeignete Teilmenge von Fällen mit `attach()` verfügbar zu machen (Abschn. 3.3.3, 3.4.3).

```
> IQ[3] <- 130                                # Änderung der Kopie
> IQ[3]
[1] 130

> myDf1$IQ[3]                                 # Original
[1] 95

> detach(myDf1)
> IQ
Fehler: Objekt "IQ" nicht gefunden
```

2.12 Häufigkeiten bestimmen

Bei der Analyse kategorialer Variablen besteht ein typischer 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.

2.12.1 Einfache Tabellen absoluter und relativer Häufigkeiten

Eine Tabelle der absoluten Häufigkeiten von Variablenwerten erstellt `table(<Faktor>)` und erwartet dafür als Argument ein eindimensionales Objekt, das sich als Faktor interpretieren lässt, z.B. einen Vektor. Das Ergebnis ist eine Übersicht über die Auftretenshäufigkeit jeder vorkommenden Ausprägung, wobei fehlende Werte ignoriert werden.⁵⁰

```
> (myLetters <- sample(LETTERS[1:5], size=12, replace=TRUE))
[1] "C" "D" "A" "D" "E" "D" "C" "E" "E" "B" "E" "E"

> table(myLetters)
myLetters
A B C D E
1 1 2 3 5
```

Häufigkeitstabellen können alternativ mit der Funktion `xtabs()` erstellt werden, die sich insbesondere dann eignet, wenn die Variablen aus Datensätzen stammen.

```
xtabs(formula=<Modellformel>, data=<Datensatz>, addNA=FALSE)
```

Im ersten Argument wird eine *Modellformel* erwartet (Abschn. 5.2). Hier ist dabei rechts der `~` der Faktor zu nennen, von dessen Stufen die Häufigkeiten gezählt werden sollen. Durch `addNA=TRUE` führt die Häufigkeitstabelle fehlende Werte als separate Gruppe auf. Stammen die in der Modellformel genannten Variablen aus einem Datensatz, ist dieser unter `data` zu nennen. Eine links der `~` genannte Variable wird als Vektor von Häufigkeiten interpretiert, die pro Stufe des rechts der `~` genannten Faktors zu addieren sind.

⁵⁰Fehlende Werte können mit in die Auszählung einbezogen werden, indem der Vektor in `addNA()` eingeschlossen wird.

```
> (tab <- xtabs(~ myLetters))
myLetters
A  B  C  D  E
1  1  2  3  5
```

In der oberen Zeile der Ausgabe sind die Ausprägungen der Variable, in der unteren Zeile die jeweils zugehörigen Auftretenshäufigkeiten aufgeführt. Eindimensionale Häufigkeitstabellen 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.

```
> names(tab)
[1] "A" "B" "C" "D" "E"

> tab["B"]
B
1
```

Relative Häufigkeiten ergeben sich durch Division der absoluten Häufigkeiten mit der Gesamtzahl der Beobachtungen. Für diese Rechnung existiert die Funktion `proportions(<Tabelle>)`, welche als Argument eine Tabelle der absoluten Häufigkeiten erwartet und die relativen Häufigkeiten ausgibt. Durch Anwendung von `cumsum()` auf das Ergebnis erhält man die kumulierten relativen Häufigkeiten (für eine andere Methode und die Berechnung von Prozenträngen s. Abschn. 2.12.6).

```
> (relFreq <- proportions(tab))      # relative Häufigkeiten
myLetters
      A          B          C          D          E
0.08333333 0.08333333 0.16666667 0.25000000 0.41666667

> cumsum(relFreq)                  # kumulierte relative Häufigkeiten
      A          B          C          D          E
0.08333333 0.16666667 0.33333333 0.58333333 1.00000000
```

Kommen mögliche Variablenwerte in einem Vektor nicht vor, so tauchen sie auch in einer Häufigkeitstabelle nicht als ausgezählte Kategorie auf. Um deutlich zu machen, dass Variablen außer den tatsächlich vorhandenen Ausprägungen potentiell auch weitere Werte annehmen, können die Daten vorab in einen Faktor umgewandelt werden. Dem Faktor lässt sich dann der nicht auftretende, aber prinzipiell mögliche Wert als weitere Stufe hinzufügen.

```
> letFac <- factor(myLetters, levels=c(LETTERS[1:5], "Q"))
> letFac
[1] C D A D E D C E E B E E
Levels: A B C D E Q

> xtabs(~ letFac)
letFac
      A      B      C      D      E      Q
1  1  1  2  3  5  0
```

2.12.2 Iterationen zählen

Eine *Iteration* innerhalb einer Abfolge von Symbolen ist ein Abschnitt, der aus der ein- oder mehrfachen Wiederholung desselben Symbols besteht (*run*). Iterationen werden durch Iterationen eines anderen Symbols begrenzt, 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, der die jeweilige Länge jeder Iteration als Elemente besitzt. Die zweite Komponente `values` ist ein Vektor mit den Symbolen, um die es sich bei den Iterationen handelt (Abschn. 2.10).

```
> (vec <- rep(rep(c("f", "m"), times=3), times=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 jeweiligen Länge und dem wiederholten Symbol jeder Iteration lässt sich die ursprüngliche Sequenz 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"
```

2.12.3 Absolute und (bedingte) relative Häufigkeiten in Kreuztabellen

Mit `xtabs(~ <Faktor1> + <Faktor2> + ...)` können mehrdimensionale Kontingenztafeln erstellt werden.⁵¹ 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>`, usw. Das Ergebnis ist eine Kreuztabelle mit den gemeinsamen absoluten Häufigkeiten der Merkmale, wobei die Ausprägungen des zuerst genannten Faktors in den Zeilen stehen. Links der `~` kann auch eine – ggf. mit `cbind()` aus Variablen zusammengefügte – Matrix stehen, deren Spaltenwerte dann jeweils pro Stufenkombination addiert werden.

Als Beispiel sollen Personen betrachtet werden, die nach ihrem Geschlecht und dem Ort ihres Arbeitsplatzes unterschieden werden. An diesen Personen sei weiterhin eine Gruppenzugehörigkeit sowie eine Variable erhoben worden, die die absolute Häufigkeit eines Ereignisses codiert.

⁵¹ Analog `table(<Faktor1>, <Faktor2>, ...)`.

```

> N <- 10
> (persons <- data.frame(
+   sex = factor(sample(c("home", "office"), size=N, replace=TRUE)),
+   work = factor(sample(c("f", "m"), size=N, replace=TRUE)),
+   group=factor(sample(c("A", "B"), size=N, replace=TRUE)),
+   counts= sample(0:5, size=N, replace=TRUE)))
  sex   work group counts
1   f    home     B      0
2   m office     B      3
3   f office     A      3
4   m    home     B      1
5   m    home     A      1
6   f office     B      1
7   m office     A      3
8   f office     B      5
9   m office     A      2
10  m office     A      4

# gemeinsame absolute Häufigkeiten von Geschlecht und Arbeitsplatz
> (absFreq <- xtabs(~ sex + work, data=persons))
  work
sex home office
  f     1     3
  m     2     4

# Summe von counts pro Zelle
> xtabs(counts ~ sex + work, data=persons)
  work
sex home office
  f     0     9
  m     2    12

```

Um relative Häufigkeiten auf Basis von Kreuztabellen absoluter Häufigkeiten zu ermitteln, eignet sich die `proportions()` Funktion. Für bedingte relative Häufigkeiten besitzt sie ein zweites Argument `margin`, an das etwa 1 zur Bestimmung der auf die Zeilen bezogenen bedingten relativen Häufigkeiten übergeben werden kann. Jede Zeile der Tabelle absoluter Häufigkeiten wird dafür durch die zugehörige Zeilensumme dividiert. Mit `margin=2` erhält man analog die auf die Spalten bezogenen bedingten relativen Häufigkeiten. Bei benannten Zeilen bzw. Spalten ist auch `margin="Name"` möglich.

```

> (relFreq <- proportions(absFreq))           # relative Häufigkeiten
  work
sex home office
  f   0.1   0.3
  m   0.2   0.4

> proportions(absFreq, 1)          # auf Zeilen bedingte rel. Häufigk.

```

```

      work
sex      home     office
f  0.2500000  0.7500000
m  0.3333333  0.6666667

> proportions(absFreq, "work") # auf Spalten bedingte rel. Häufigk.
      work
sex      home     office
f  0.3333333  0.4285714
m  0.6666667  0.5714286

# manuelle Kontrolle
> rSums <- rowSums(relFreq)      # Zeilensummen
> cSums <- colSums(relFreq)      # Spaltensummen
> sweep(relFreq, 1, rSums, "/") # auf Zeilen bedingte rel. Häufigk...
> sweep(relFreq, 2, cSums, "/") # auf Spalten bedingte rel. Häufigk...

```

Um Häufigkeitsauszählungen für mehr als zwei Variablen zu berechnen, können beim Aufruf von `xtabs()` in der Modellformel rechts der `~` einfach weitere Faktoren mit `+` hinzugefügt werden. Die Ausgabe verhält sich dann wie ein 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 `ftable()` (*flat table*) zu nutzen.

```
ftable(x, row.vars=NULL, col.vars=NULL)
```

Unter `x` kann entweder eine bereits mit `xtabs()` erzeugte Kreuztabelle eingetragen werden, oder aber eine durch Komma getrennte Reihe von Faktoren bzw. von Objekten, die sich als Faktor interpretieren lassen. 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.

```

> with(persons,
+       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

```

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 bzw. auf Gleichheit von Verteilungen berechnet (Abschn. 10.2.1, 10.2.2).

```

> summary(xtabs(~ sex + work, data=persons))
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
```

2.12.4 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.

```
> colMeans(xtabs(~ sex + work, 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 (Voreinstellung ist `sum` für Randsummen). Die Funktion operiert separat über jeder der mit dem Vektor `margin` bezeichneten Dimensionen – in der Voreinstellung über alle. 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(~ sex + work, data=persons))      # Randsummen
           work
sex   home   office   sum
  f     1       3     4
  m     2       4     6
  sum   3       7    10
```

2.12.5 Datensätze aus Häufigkeitstabellen erstellen

In manchen Situationen liegen Daten nur in Form von Häufigkeitstabellen vor – etwa wenn sie aus der Literatur übernommen werden. Um die Daten selbst auszuwerten, ist es dann notwendig, sie zunächst in ein Format umzuwandeln, das die Variablen als separate Spalten verwendet und pro Beobachtungsobjekt eine Zeile besitzt. Diese Aufgabe erledigt die Funktion `Untable()` aus dem Paket `DescTools`. Als Ausgabe liefert sie einen Datensatz.

```
> cTab <- xtabs(~ sex + work, data=persons)      # Kreuztabelle
> library(DescTools)                            # für Untable()
> Untable(cTab)
           sex   work
1     f   home
2     f office
3     f office
```

```

4   f office
5   m   home
6   m   home
7   m office
8   m office
9   m office
10  m office

```

Eine andere Darstellung liefert die explizite Umwandlung der Kreuztabelle in einen Datensatz mit `as.data.frame()`. Der erzeugte Datensatz besitzt für jede Zelle der Kreuztabelle eine Zeile und neben den Spalten für die ausgezählten Variablen eine weitere Spalte `Freq` mit der Häufigkeit der Merkmalskombination.

```

> as.data.frame(cTab)
  sex   work Freq
1  f   home    1
2  m   home    2
3  f office   3
4  m office   4

```

2.12.6 Kumulierte relative Häufigkeiten und Prozentrang

`ecdf(x=⟨Vektor⟩)` (*empirical cumulative distribution function*) ermittelt für ordinale Daten die kumulierten relativen Häufigkeiten. Diese geben für einen Wert x_i an, welcher Anteil der Daten nicht größer als x_i ist. Das Ergebnis ist analog zur Verteilungsfunktion quantitativer Zufallsvariablen.

Das Ergebnis von `ecdf()` ist eine Stufenfunktion mit so vielen Sprungstellen, wie es unterschiedliche Werte in `x` gibt. 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/length(x)` jedes Elements von `x`. Tauchen in `x` Werte mehrfach auf, unterscheiden sich die Sprunghöhen dagegen entsprechend den relativen Häufigkeiten.

Die Ausgabe von `ecdf()` ist ihrerseits eine Funktion, die zunächst einem eigenen Objekt zugewiesen werden muss, ehe sie die kumulierten relativen Häufigkeiten ermitteln kann. Ist `Fn()` diese Stufenfunktion, und möchte man die kumulierten relativen Häufigkeiten der in `x` gespeicherten Werte erhalten, ist `x` selbst als Argument für `Fn()` einzusetzen. Andere Werte als Argument von `Fn()` sind aber genauso möglich. Indem `Fn()` für beliebige Werte ausgewertet wird, lassen sich empirische und interpolierte Prozentränge ermitteln (Abschn. 2.7.5). Mit `ecdf()` erstellte Funktionen können über `plot()` in einem Diagramm gezeigt werden (Abb. 10.1, Abb. 14.22, Abschn. 14.6.6).

```

> (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

# kumulierte relative Häufigkeiten der vorhandenen Werte

```

```
> Fn <- ecdf(vec)
> Fn(vec)
[1] 0.1 1.0 0.2 0.7 0.4 0.3 0.5 0.9 0.6 0.8

> 100 * Fn(0.1)                                # Prozentrang von 0.1
[1] 50

> 100 * (sum(vec <= 0.1) / length(vec))        # Kontrolle
[1] 50
```

Soll die Ausgabe der kumulierten relativen Häufigkeiten in der richtigen Reihenfolge erfolgen, müssen die Werte mit `sort()` geordnet werden.

```
> Fn(sort(vec))
[1] 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

Die Sprungstellen einer von `ecdf()` erstellten Funktion lassen sich mit `knots()` extrahieren. Das Ergebnis sind gerade die in `x` enthaltenen unterschiedlichen sortierten Werte.

```
> knots(Fn)
[1] -1.57 -1.01 -0.61 -0.29 -0.17 0.17 0.21 0.55 1.90 2.21
```

2.13 Fehlende Werte behandeln

Empirische Datensätze besitzen häufig zunächst keine zufriedenstellende Qualität (Abschn. 4.3), wozu auch unvollständige Daten beitragen – wenn also nicht für alle Beobachtungsobjekte Werte von allen erhobenen Variablen vorliegen. So können etwa Messgeräte defekt sein, Personen die Auskunft bzgl. bestimmter Fragen verweigern oder zu bearbeitende Aufgaben übersehen.

Fehlende Werte bergen aus versuchsplanerischer Perspektive die Gefahr, dass sie womöglich nicht zufällig, sondern systematisch entstanden sind und so zu verzerrten Ergebnissen führen.⁵² Bei der statistischen Auswertung sind verschiedene Strategien des Umgangs mit fehlenden Werten denkbar, die nicht unbedingt zu gleichen Ergebnissen führen.

2.13.1 Fehlende Werte codieren und identifizieren

Wenn ein Datensatz eingegeben wird und fehlende Werte vorliegen, müssen sie explizit mit der Konstante `NA` (*not available*) codiert werden. Auch bei `character` Vektoren ist dabei `NA` nicht in Anführungszeichen zu setzen.⁵³

⁵²Die hier relevante Unterscheidung zwischen den Szenarien *missing completely at random* (MCAR), *missing at random* (MAR) und *not missing at random* (NMAR) erläutert van Buuren (2018).

⁵³Für jeden Datentyp existiert jeweils eine passende Konstante, nämlich `NA_real_`, `NA_integer_` und `NA_character_`. All diese Konstanten werden aber als `NA` angezeigt. Der mit `typeof(NA)` ausgegebene Basis-Datentyp ist `logical`. Damit ist die Ausgabe etwa von `c(1, 2, 3)[NA]` gleich `NA NA NA`, da logische Indexvektoren zyklisch verlängert werden (Abschn. 2.1.2, 2.2.2).

```
> (vec1 <- c(10, 20, NA, 40, 50, NA))
[1] 10 20 NA 40 50 NA

> length(vec1)      # fehlende Werte werden bei length() mitgezählt
[1] 6
```

In manchen Situationen werden Zeichenketten prinzipiell ohne Anführungszeichen ausgegeben, etwa als Faktorstufen oder Datensätzen. Zur leichteren Unterscheidung von gültigen Zeichenketten erscheinen fehlende Werte deshalb dann als `<NA>`.⁵⁴

```
# Ausgabe von Zeichenketten mit Anführungszeichen -> nur NA
> LETTERS[c(1, NA, 3)]
[1] "A" NA "C"
```

```
# Ausgabe von Zeichenketten ohne Anführungszeichen -> <NA>
> factor(LETTERS[c(1, NA, 3)])
[1] A <NA> C
Levels: A C
```

Ob in einem Vektor fehlende Werte vorhanden sind, wird mit der Funktion `is.na(<Vektor>)` ermittelt.⁵⁵ Sie gibt einen logischen Vektor aus, der für jede Position angibt, ob das Element ein fehlender Wert ist. Im Fall eines Datensatzes liefert `is.na()` eine Matrix aus Wahrheitswerten, die für jedes Element des Datensatzes angibt, ob es sich um einen fehlenden Wert handelt.

```
> is.na(vec1)
[1] FALSE FALSE TRUE FALSE FALSE TRUE

> vec2 <- c(NA, 7, 9, 10, 1, 8)
> (datNA <- data.frame(vec1, vec2))
  vec1 vec2
1    10   NA
2    20    7
3    NA    9
4    40   10
5    50    1
6    NA    8

> is.na(datNA)
  vec1 vec2
[1,] FALSE  TRUE
[2,] FALSE FALSE
```

⁵⁴In solchen Situationen ist also `NA` die Ausgabe eines gültigen Elements und von einem fehlenden Wert `<NA>` zu unterscheiden. So erzeugt `factor(c("A", "NA", "C"))[c(NA, 2, 3)]` die Ausgabe `<NA> NA C`. Einzig die gültige Faktorstufe "`<NA>`" lässt sich in der Ausgabe nicht von einem fehlenden Wert unterscheiden. In diesem Fall kann nur mit Hilfe von `is.na()` festgestellt werden, ob es sich tatsächlich um einen fehlenden Wert handelt.

⁵⁵Der `==` Operator eignet sich nicht zur Prüfung auf fehlende Werte, da das Ergebnis von `<Wert> == NA` selbst `NA` ist (Abschn. 2.13.3). Da die Unterscheidung zwischen *not a number* und *missing* nicht zuverlässig möglich ist, ergibt auch die Prüfung `is.na(NaN)` das Ergebnis `TRUE`.

```
[3,] TRUE FALSE
[4,] FALSE FALSE
[5,] FALSE FALSE
[6,] TRUE FALSE
```

Bei einem großen Datensatz ist es mühselig, die Ausgabe von `is.na()` manuell nach TRUE Werten zu durchsuchen. Daher bietet sich `anyNA()` an, um zu erfahren, ob überhaupt fehlende Werte vorliegen. `sum(is.na())` ermittelt deren Anzahl und `which(is.na())` die zugehörigen Positionen.

```
> anyNA(vec1)                                # gibt es fehlende Werte?
[1] TRUE

> sum(is.na(vec1))                           # wie viele?
[1] 2

> which(is.na(vec1))                         # wo im Vektor?
[1] 3 6

> which(is.na(datNA), arr.ind=TRUE)          # wo im Datensatz?
  row col
[1,]   3   1
[2,]   6   1
[3,]   1   2
```

2.13.2 Fehlende Werte ersetzen und 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 (Abschn. 3.1, insbesondere das Argument `na.strings` in Abschn. 3.1.1 und 3.1.4).

Die Identifikation der zu ersetzenden Werte kann über `<Vektor> %in% <Menge>` erfolgen, wobei `<Menge>` ein Vektor mit allen Werten ist, die als fehlend gelten sollen (Abschn. 2.3.2). Der damit erzeugte Indexvektor lässt sich direkt an das Ergebnis von `is.na()` zuweisen, wodurch die zugehörigen Elemente auf NA gesetzt werden.

```
# fehlende Werte sind zunächst mit -999 und 999 codiert
> vec <- c(30, 25, 23, 21, -999, 999)    # Vektor mit fehlenden Werten
> is.na(vec) <- vec %in% c(-999, 999)    # ersetze missings durch NA
> vec
[1] 30 25 23 21 NA NA
```

Abschnitte 3.3.4 und 3.4.4 zeigen, wie fehlende Werte in Datensätzen identifiziert und codiert werden können.

2.13.3 Behandlung fehlender Werte bei der Berechnung einfacher Kennwerte

Wenn in einem Vektor oder einem Datensatz 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 das Ergebnis der Funktion ist seinerseits NA.⁵⁶ Allerdings lassen sich NA Einträge zunächst manuell entfernen, ehe die Daten an eine Funktion übergeben werden. Zu diesem Zweck existiert die Funktion `na.omit()`, die den übergebenen Vektor um fehlende Werte bereinigt ausgibt.

```
> sd(na.omit(vec))          # um NA bereinigten Vektor übergeben
[1] 5.125102

# fehlende Werte manuell entfernen
> keep <- !is.na(vec)      # Indizes der nicht fehlenden Werte
> mean(vec[keep])         # um NA bereinigten Vektor übergeben
[1] 24.33333
```

Die Behandlung fehlender Werte lässt sich in vielen Funktionen auch direkt über das Argument `na.rm` steuern. 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(vec)
[1] NA

> sum(vec, na.rm=TRUE)
[1] 146
```

Auf die dargestellte Weise lassen sich fehlende Werte u. a. in `sum()`, `prod()`, `range()`, `mean()`, `median()`, `quantile()`, `var()` und `sd()` behandeln.

2.13.4 Behandlung fehlender Werte in Matrizen

Bei den Funktionen `cov()` und `cor()` stehen zur Behandlung fehlender Werte der *zeilenweise* und *paarweise* Fallausschluss zur Verfügung, die über das Argument `use="complete.obs"` bzw. `use="pairwise.complete.obs"` ausgewählt werden. Bei der Kovarianz bzw. Korrelation zweier Variablen führen sie zum selben Ergebnis, unterscheiden sich aber bei Kovarianz- bzw. Korrelationsmatrizen von mehr als zwei Variablen.

⁵⁶Allgemein 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.

Zeilenweiser (fallweiser) Fallausschluss

Beim zeilenweisen Fallausschluss werden die Matrixzeilen komplett entfernt, in denen NA Werte auftauchen, 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. `na.omit(<Matrix>)` bereinigt eine Matrix mit fehlenden Werten um Zeilen, in denen NA Einträge auftauchen. Die Indizes der dabei ausgeschlossenen Zeilen werden der ausgegebenen Matrix als Attribut hinzugefügt. Mit der so gebildeten Matrix fließen im Beispiel die Zeilen 1 und 2 nicht mit in Berechnungen ein.

```
> 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

> na.omit(matNA)                                # Zeilen mit NA entfernen
      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))                      # Berechnung ohne NAs
ageNA  DV1  DV2
24.5  1.0  4.5
```

Bei `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. Dessen 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.equal(cov(matNA, use="complete.obs"), cov(na.omit(matNA)))
```

[1] TRUE

Paarweiser Fallausschluss

Der paarweise Fallausschluss unterscheidet sich erst bei der Berechnung von Kovarianz- bzw. Korrelationsmatrizen für mehr als zwei Variablen vom fallweisen Ausschluss. Beim paarweisen Fallausschluss werden die Werte einer auch NA beinhaltenden Zeile soweit als möglich in Berechnungen berücksichtigt, die Zeile wird 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(matNA, 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 pro Zeile geprüft, ob 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.⁵⁷

Angewendet auf die Daten in `matNA` bedeutet das beispielsweise, dass beim fallweisen Ausschluss das von Beobachtungsobjekt 1 gelieferte Wertepaar nicht in die Berechnung der Kovarianz von `ageNA` und `DV2` einfließt, weil der Wert für `DV1` bei diesem Beobachtungsobjekt fehlt. Beim paarweisen Ausschluss werden diese Werte dagegen berücksichtigt. Lediglich bei der Berechnung der Kovarianz von `DV1` und `DV2` werden keine Daten des ersten Beobachtungsobjekts 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
```

2.13.5 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 so dafür, dass fehlende Werte entfernt werden. Bei `order()` ist die Voreinstellung TRUE, wodurch fehlende Werte ans

⁵⁷Eine mit diesem Verfahren ermittelte Matrix kann auch nicht positiv semidefinit sein, und stellt dann keine Kovarianzmatrix bzw. Korrelationsmatrix im engeren Sinne dar.

Ende plaziert werden. Auf FALSE gesetzt bewirkt `na.last` die Plazierung fehlender Werte am Anfang.

2.13.6 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 Funktionen `na.omit()` und `na.fail()`, die sich auch direkt auf Daten anwenden lassen (Abschn. 2.13.3, 3.3.4, 3.4.4). Die Voreinstellung `na.omit` bewirkt den fallweisen Ausschluss (Abschn. 2.13.4), mit `na.fail` wird die Auswertung bei fehlenden Werten abgebrochen und eine Fehlermeldung ausgegeben. Global kann dieses Verhalten mit `options(na.action = "<Wert>")` geändert werden (vgl. `?na.action`). Generell empfiehlt es sich, Daten außerhalb von Auswertungsfunktionen um fehlende Werte zu bereinigen, bzw. sie mit Techniken der multiplen Imputation zu vervollständigen. So lässt sich die Konsistenz bzgl. der in einzelne Auswertungen eingeflossenen Beobachtungsobjekte besser sichern.

2.13.7 Multiple Imputation

Die *multiple Imputation* eliminiert fehlende Werte, indem sie an ihrer Stelle plausible mögliche Werte einfügt. Sie ersetzt fehlende Werte eines aus mehreren Variablen bestehenden Datensatzes in mehreren Durchgängen jeweils durch solche Werte, die mit bestimmten Annahmen und unter Berücksichtigung der tatsächlich vorhandenen Daten künstlich generiert wurden. Die simulierten Daten sollen so die Verteilungseigenschaften der empirisch beobachteten Daten teilen. Für jeden einzelnen Durchgang der Imputation aller fehlenden Werte lassen sich dann z. B. Parameterschätzungen eines statistischen Regressionsmodells berechnen. Die sich ergebene Menge unterschiedlicher Schätzungen – eine Schätzung aller Koeffizienten pro Imputation – kann dann über Durchgänge hinweg zu einer Gesamtschätzung kombiniert werden.

Multiple Imputation wird in R u. a. von den Paketen `mice` ([van Buuren & Groothuis-Oudshoorn, 2011](#); [van Buuren, 2018](#)) und `Amelia` ([Honaker, King & Blackwell, 2011](#)) unterstützt. Für weitere vgl. den Abschnitt *Official Statistics & Survey Methodology* der CRAN Task Views ([Templ, 2020](#)).

2.14 Zeichenketten verarbeiten

Zeichenketten tauchen bei der Auswertung von Daten z. B. als Bezeichnungen für Variablen oder Gruppen auf. Vor allem bei der Aufbereitung eines Rohdatensatzes ist es hilfreich, sie flexibel erstellen und manipulieren zu können.⁵⁸

⁵⁸Das Paket `stringr` ([Wickham, 2019b](#)) stellt für viele der im Folgenden aufgeführten Funktionen Alternativen bereit, die den Umgang mit Zeichenketten erleichtern und konsistenter gestalten sollen.

2.14.1 Objekte in Zeichenketten umwandeln

Mit `toString(<Objekt>)` lassen sich Objekte in Zeichenketten umwandeln. Das Ergebnis ist eine einzelne Zeichenkette mit den Inhalten des Objekts, wobei einzelne Elemente durch Komma mit folgendem Leerzeichen getrennt werden. Komplexe Objekte (z. B. Matrizen) werden dabei wie Vektoren verarbeitet.

```
> randVals <- round(rnorm(5), digits=2)
> toString(randVals)
[1] "-0.03, 1.01, -0.52, -1.03, 0.18"
```

Analog wandelt `capture.output(<Ausdruck>)` die normalerweise auf der Konsole erscheinende Ausgabe eines Befehls in eine Zeichenkette um. Die Ausgabe mehrerer Bearbeitungsschritte lässt sich mit `sink()` in eine Zeichenkette oder mit `sink("<Dateiname>")` in eine Datei umleiten. Soll dies als Protokoll aller Vorgänge in Kopie geschehen, während die Ausgabe auf der Konsole weiter angezeigt wird, ist dabei das Argument `split=TRUE` zu setzen. Eine aktive Umleitung wird mit `sink()` wieder beendet.

`formatC()` ist auf die Umwandlung von Zahlen in Zeichenketten spezialisiert und bietet sich vor allem für die formatierte Ausgabe von Dezimalzahlen an.

```
formatC(x=<Zahl>, digits=<Dezimalstellen>, width=<Breite>,
        flag=<Modifikation>, format=<Zahlentyp>)"
```

Ist `x` eine Dezimalzahl, wird sie mit `digits` vielen Dezimalstellen 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. Sollen der Zahl stattdessen Nullen vorangestellt werden, ist `flag="0"` zu setzen. Linksbündig ausgerichtete Zeichenketten sind mit `flag="-"` zu erreichen. 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 letztgenannten 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(3, digits=5, format="d")
[1] "      3"

> formatC(c(1, 2.345), width=5, format="f")
[1] "1.0000" "2.3450"
```

2.14.2 Zeichenketten erstellen und ausgeben

Die einfachste Möglichkeit zum Erstellen eigener Zeichenketten ist ihre manuelle Eingabe auf der Konsole oder im Editor. Für Vektoren von Zeichenketten ist dabei zu beachten, dass `length()` jede Zeichenkette als ein Element betrachtet. Dagegen gibt `nchar("<Zeichenkette>")` die Wortlänge jedes Elements an, aus wie vielen einzelnen Zeichen jede Zeichenkette des Vektors also besteht.

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

> nchar("ABCDEF")
[1] 6

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

Fehlende Werte `NA` wandelt `nchar()` intern in die Zeichenkette "NA" um. Daher liefert `nchar(NA)` das Ergebnis 2.

Zeichenketten nach Muster erstellen

Die Methode, Zeichenketten manuell in Vektoren zu erstellen, stößt dort schnell an ihre Grenzen, wo sie von Berechnungen abhängen sollen oder viele Zeichenketten nach demselben Muster erzeugt werden müssen. `paste()` und `sprintf()` sind hier geeignete Alternativen.

Mit `paste()` 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=" ", collapse=NULL)
```

Die ersten Argumente von `paste()` sind Objekte, deren Elemente jeweils die Bestandteile der zu erstellenden Zeichenketten ausmachen und zu diesem Zweck aneinandergefügt werden. Das erste Element des ersten Objekts wird dazu mit den ersten Elementen der weiteren Objekte 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. In der Voreinstellung `collapse=NULL` ist das Ergebnis 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 (Abschn. 2.5.4). Wird stattdessen für `collapse` eine Zeichenfolge übergeben, ist das Ergebnis eine einzelne Zeichenkette, deren Bestandteile durch diese Zeichenfolge getrennt sind.

```
> 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: #DF536B" "3: #61D04F" "4: #13A5FD" "5: #33DBDF"

> paste(1:5, letters[1:5], sep=". ", collapse=" ")
[1] "1.a 2.b 3.c 4.d 5.e"
```

Fehlende Werte `NA` wandelt `paste()` in die Zeichenkette "NA" um. Treten innerhalb von `paste()` die leere Menge `NULL` oder leere Vektoren `character(0)` gemeinsam mit anderen

Zeichen auf, werden sie wie die leere Zeichenkette "" der Länge 1 behandelt, was an den eingefügten Trennzeichen `sep` deutlich wird.

```
> paste(1, NA, 2, NULL, 3, character(0), sep="_")
[1] "1_NA_2__3_"
```

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 *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 und `%s` auf eine Zeichenfolge. Das Prozentzeichen selbst wird durch `%%` ausgegeben, doppelte Anführungszeichen durch `\"`,⁵⁹ Tabulatoren durch `\t` und Zeilenumbrüche durch `\n` (vgl. `?Quotes`).

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 Reihenfolge hergestellt, der Wert des ersten Objekts wird also an der Stelle des ersten variablen Elements eingefügt, etc.

```
> N    <- 20
> grp <- "A"
> M    <- 14.2
> sprintf("For %d participants in group %s, mean was %f", N, grp, M)
[1] "For 20 participants in group A, 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 können. Für detailliertere Informationen vgl. `?sprintf`.

```
> sprintf("%.3f", 1.23456)      # begrenze Ausgabe auf 3 Dezimalstellen
[1] "1.234"

# ganze Zahlen mit fester Breite und führender 0
> sprintf("%.4d", c(1, 52, 712))
[1] "0001" "0052" "0712"
```

⁵⁹Alternativ kann `fmt` in einfache Anführungszeichen '`<format string>`' gesetzt werden, innerhalb derer sich dann auch doppelte Anführungszeichen ohne voranstehenden backslash \ befinden können (Abschn. 1.4.5, Fußnote 32).

In der Voreinstellung setzen die meisten Ausgabefunktionen von R Zeichenketten in Anführungszeichen. In `print()` lässt sich dies mit dem Argument `quote=FALSE` verhindern, allgemein hat `noquote("<Zeichenkette>")` denselben Effekt.

```
> cVar <- "A string"
> print(cVar, quote=FALSE)
[1] A string

> noquote(cVar)
[1] A string
```

Zeichenketten verbinden

Um mehrere Zeichenketten kombiniert als eine einzige Zeichenkette lediglich auf der R Konsole auszugeben und nicht in einem Vektor zu speichern, kann die Funktion `cat()` (*concatenate*) verwendet werden. Sie erlaubt auch eine gewisse Formatierung – etwa in Form von Zeilenumbrüchen durch die Escape-Sequenz `\n` oder `\t` für Tabulatoren.

```
cat("<Zeichenkette 1>", "<Zeichenkette 2>", ..., 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. Die Ausgabe von `cat()` unterscheidet sich in zwei Punkten von der üblichen Ausgabe einer Zeichenkette: Zum einen wird sie nicht in Anführungszeichen gesetzt. Zum anderen wird am Anfang jeder Zeile auf die Ausgabe der Position des zu Zeilenbeginn stehenden Wertes, etwa [1], verzichtet.

```
> cat(cVar, "with\n", 4, "\nwords\n", sep="+")
A string+with
+4+
words
```

2.14.3 Zeichenketten manipulieren

Die folgenden Funktionen akzeptieren als Argument neben einer einzelnen Zeichenkette `Z` auch Vektoren von Zeichenketten.

- `tolower("<Z>")` und `toupper("<Z>")` konvertieren die Buchstaben in `Z` in Klein- bzw. Großbuchstaben.
- `strtrim("<Z>", width=<Anzahl>)` schneidet `Z` hinter `width` vielen Buchstaben ab.
- Demgegenüber erstellt `abbreviate("<Z>", minlength=<Anzahl>")` eine Kurzform von `Z` mit `minlength` vielen Buchstaben, wobei ursprünglich unterschiedliche Zeichenketten desselben Vektors unterscheidbar bleiben. Dies kann etwa beim Erstellen von übersichtlichen Variablen-Bezeichnungen für Diagrammachsen oder Tabellen sinnvoll sein.

- Leerzeichen und anderen Leerraum zu Beginn oder Ende von Z entfernt `trimws("⟨Z⟩", which)` (*trim whitespace*). Mit dem Argument `which="left"` betrifft dies nur den Anfang, analog mit `"right"` nur das Ende und mit `"both"` sowohl Anfang als auch Ende von Z.
- Analog zur Wiederholung von Zahlen mit `rep()` kann `strrep("⟨Z⟩", times=⟨Anzahl⟩)` eine Zeichenkette x durch Wiederholung verlängern. Ist x ein Vektor von Zeichenketten, wird jedes Element verlängert. Dafür kann `times` auch ein Vektor derselben Länge wie x sein und bestimmt dann für jedes Element von x, wie oft es aneinandergehängt werden soll.
- Mit `StrRev("⟨Z⟩")` (*string reverse*) aus dem Paket `DescTools` wird die Reihenfolge der Zeichen in Z umgekehrt.

```
> tolower(c("A", "BC", "DEF"))
[1] "a" "bc" "def"

> strtrim("AfairlyLongString", width=6)
[1] "Afairl"

> abbreviate("AfairlyLongString", minlength=6)
AfairlyLongString
"AfrlLS"

> trimws(c(" Quattuor ", " quinque   "), which="both")
[1] "Quattuor" "quinque"

> strrep(c("A", "B", "C"), 1:3)
[1] "A"     "BB"    "CCC"

> library(DescTools)           # für StrRev()
> StrRev(c("Lorem", "ipsum", "dolor", "sit"))
[1] "meroL" "muspi" "rolod" "tis"
```

Mit `strsplit()` (*string split*) ist es möglich, eine einzelne Zeichenkette in mehrere Teile zu zerlegen.

```
strsplit(x="⟨Zeichenkette⟩", split="⟨Zeichenkette⟩", fixed=FALSE)
```

Die Elemente des für x übergebenen Vektors werden dafür nach Vorkommen der unter `split` genannten Zeichenkette durchsucht, die als Trennzeichen interpretiert wird. Die Zeichenfolgen links und rechts von `split` machen die Komponenten der Ausgabe aus, die aus einer Liste von Vektoren von Zeichenketten besteht – eine Komponente für jedes Element des Vektors von Zeichenketten x. In der Voreinstellung `split=NULL` werden die Elemente von x in einzelne Zeichen zerlegt.⁶⁰ `strsplit()` ist damit die Umkehrung von `paste()`. Das Argument `fixed` bestimmt, ob `split` i.S. eines *regulären Ausdrucks* interpretiert werden soll (Voreinstellung `FALSE`, Abschn. 2.14.4) oder als exakt die übergebene Zeichenfolge selbst (`TRUE`).

⁶⁰Die Ausgabe ist ggf. mit `unlist()` (Abschn. 2.10.3) in einen Vektor, oder mit `do.call(cbind, ⟨Liste⟩)` bzw. `do.call(rbind, ⟨Liste⟩)` in eine Matrix umzuwandeln, wenn die Listenkomponenten dieselbe Länge besitzen. Weitere Möglichkeiten der Weiterverarbeitung von Listen zeigt Abschn. 3.3.13.

```
> strsplit(c("abc_def_ghi", "jkl_mno"), split="_")
[[1]]
[1] "abc" "def" "ghi"

[[2]]
[1] "jkl" "mno"

> strsplit("Xylophon", split=NULL)
[[1]]
[1] "X" "y" "l" "o" "p" "h" "o" "n"
```

2.14.4 Zeichenfolgen finden

Die Suche nach bestimmten Zeichenfolgen innerhalb von Zeichenketten ist mit `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.⁶¹

```
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 Funktion `pmatch()` 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 Index das Ergebnis, an dem dieses Element zum ersten Mal vorkommt. 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
```

Speziell um zu prüfen, ob Zeichenketten in einem Vektor `Z` mit einem bestimmten Muster beginnen bzw. enden, existieren die Funktionen `startsWith("<Z>", prefix="<Muster>")` und `endsWith("<Z>", suffix="<Muster>")`. `prefix` bzw. `suffix` müssen Zeichenketten sein, reguläre Ausdrücke sind hier nicht möglich.

⁶¹`adist()` berechnet die Levenshtein-Distanz zwischen zwei Zeichenketten als Maß für die Anzahl notwendiger elementarer Editiervorgänge, um eine Zeichenkette in die andere zu ändern. Über die Levenshtein-Distanz können mit `agrep()` auch ungefähr passende Zeichenketten gefunden werden. Weitere Ansätze hierfür sind im Paket `stringdist` ([van der Loo, 2014](#)) vorhanden.

```
> startsWith(c("Train", "Station"), "T")
[1] TRUE FALSE

> endsWith(c("Train", "Station"), "n")
[1] TRUE TRUE
```

`grep()` ähnelt dem gleichlautenden POSIX-Befehl Unix-artiger Betriebssysteme und bietet stark erweiterte Suchmöglichkeiten.

```
grep(pattern="<Suchmuster>", x="<Zeichenkette>", value=FALSE,
      ignore.case=FALSE)
```

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` reguläre Ausdrücke akzeptiert. Ein regulärer Ausdruck definiert eine Menge möglicher Zeichenfolgen, die dasselbe Muster besitzen, etwa „ein A gefolgt von einem B oder C und einem Leerzeichen“: `"A[BC] [[:blank:]]"` (vgl. [?regex](#), sowie [Goyvaerts & Levithan, 2012](#)). Der zu durchsuchende Vektor von Zeichenketten wird unter `x` genannt. Die Option `ignore.case` bestimmt, ob Groß- / Kleinschreibung beachtet werden soll.

Die Ausgabe ist in der Voreinstellung `value=FALSE` ein Vektor von Indizes derjenigen Elemente von `x`, die das gesuchte Muster enthalten. Mit `value=TRUE` werden die zum Muster passenden Elemente von `x` selbst ausgegeben. Hierfür existiert die Kurzform `grepv()`. Weiter gibt die ansonsten genauso zu verwendende Funktion `grepl()` einen logischen Indexvektor aus, der für jedes Element von `x` angibt, ob es `pattern` enthält.

```
# Indizes der passenden Elemente
> grep("A[BC] [[:blank:]]", c("AB ", "AB", "AC ", "A "))
[1] 1 3

# passende Elemente selbst
> grep( "A[BC] [[:blank:]]", c("AB ", "AB", "AC ", "A "),
+       value=TRUE)
[1] "AB " "AC "

# passendes Element vorhanden?
> grepl("A[BC] [[:blank:]]", c("AB ", "AB", "AC ", "A "))
[1] TRUE FALSE TRUE FALSE
```

2.14.5 Zeichenfolgen extrahieren

Aus Zeichenketten lassen sich mit `substring()` konsekutive Teilstücke von Zeichen extrahieren.

```
substring(text="<Zeichenkette>", first=<Beginn>, last=<Ende>)
```

Aus den Elementen des für `text` angegebenen Vektors von Zeichenketten wird jeweils jene Zeichenfolge extrahiert, die beim Buchstaben an der Stelle `first` beginnt und mit dem Buchstaben

an der Stelle `last` endet. Umfasst eine Zeichenkette weniger als `first` oder `last` Buchstaben, werden nur so viele ausgegeben, wie tatsächlich vorhanden sind – ggf. eine leere Zeichenkette.

```
> substring(c("ABCDEF", "GHIJK", "LMNO", "PQR"), first=4, last=5)
[1] "DE" "JK" "O" "
```

Um mit Hilfe von regulären Ausdrücken definierte Zeichenfolgen aus Zeichenketten extrahieren zu können, ist neben der Information, *ob* eine Zeichenkette die gesuchte Zeichenfolge enthält, auch die Information notwendig, an welcher Stelle sie ggf. auftaucht. Dies lässt sich mit `regexpr()` ermitteln.

```
regexpr(pattern="/Suchmuster", text="/Zeichenkette")
```

Die Argumente `pattern` und `text` haben jeweils dieselbe Bedeutung wie `pattern` und `x` von `grep()`. Das Ergebnis ist ein numerischer Vektor mit so vielen Elementen wie jene von `text`. Enthält ein Element von `text` das Suchmuster nicht, ist das Ergebnis an dieser Stelle `-1`. Andernfalls ist das Ergebnis die erste Stelle des zugehörigen Elements von `text`, an der das gefundene Suchmuster dort beginnt. Der ausgegebene numerische Vektor besitzt weiterhin das Attribut `match.length`, das seinerseits ein numerischer Vektor ist und codiert, wie viele Zeichen die Zeichenfolge umfasst, auf die das Suchmuster zutrifft. Auch hier steht die `-1` für den Fall, dass sich das Suchmuster nicht in der Zeichenkette findet. Das Ergebnis eignet sich besonders, um mit der Funktion `regmatches()` weiterverarbeitet zu werden, die die Fundstellen extrahiert.

```
> pat      <- "[[:upper:]]+"          # suche nach Großbuchstaben
> txt      <- c("abcDEFG", "ABCdefg", "abcdefg")
> (start <- regexpr(pat, txt))        # Start + Länge der Fundstellen
[1] 4 1 -1

attr(", "match.length")
[1] 4 3 -1

> regmatches(txt, start)            # extrahiere Fundstellen
[1] "DEFG" "ABC" "
```

Im Unterschied zu `regexpr()` berücksichtigt die ansonsten gleich zu verwendende Funktion `gregexpr()` nicht nur das erste Auftreten von `pattern` in einem Element von `text`, sondern auch ggf. spätere. Die Ausgabe ist eine Liste mit so vielen Komponenten, wie `text` Elemente besitzt. Das Ergebnis kann wieder direkt an `regmatches()` übergeben werden, um die Fundstellen zu extrahieren.

```
> txt2    <- c("abcDEFghijkl", "ABCdefgHIJ", "abcdefg")
> start2 <- gregexpr(pat, txt2)       # Start + Länge der Fundstellen
> regmatches(txt2, start2)           # extrahiere Fundstellen
[[1]]
[1] "DEFG" "KL"

[[2]]
[1] "ABC" "HIJ"
```

```
[[3]]
character(0)
```

Eine Alternative zu `regmatches()` bieten die in Abschn. 2.14.6 vorgestellten Funktionen `sub()` bzw. `gsub()`.

Da die Syntax regulärer Ausdrücke recht komplex ist, sorgt u. U. schon die Suche nach einfachen Mustern für Schwierigkeiten. Eine Vereinfachung bietet die Funktion `glob2rx()`, mit der Muster von Zeichenfolgen mit Hilfe gebräuchlicherer Platzhalter (*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 einen Vektor, dessen Elemente Zeichenfolgen aus Buchstaben und Platzhaltern sind. Die Ausgabe besteht aus einem Vektor mit regulären Ausdrücken, wie sie z. B. in `grep()` angewendet werden können.

```
> glob2rx("asdf*.txt") # Namen, die mit asdf beginnen und .txt enden
[1] "^\w+asdf.*\.\w+txt$"
```

2.14.6 Zeichenfolgen ersetzen

Wenn in Zeichenketten nach bestimmten Zeichenfolgen gesucht wird, dann häufig, um sie durch andere zu ersetzen. Dies ist etwa möglich, indem dem Ergebnis von `substring()` ein passender Vektor von Zeichenketten zugewiesen wird – dessen Elemente ersetzen dann die durch `first` und `last` begrenzten Zeichenfolgen in den Elementen von `text`. 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"
```

Auch `sub()` (*substitute*) und `gsub()` dienen dem Zweck, durch ein Muster definierte Zeichenfolgen innerhalb von Zeichenketten auszutauschen.

```
sub(pattern = "⟨Suchmuster⟩", replacement = "⟨Ersatz⟩", x = "⟨Zeichenkette⟩")
```

Für `pattern` kann ein regulärer Ausdruck übergeben werden, dessen Vorkommen in den Elementen von `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 Lorem 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"
```

In `pattern` können runde Klammern (`\langle Muster \rangle`) zur Definition von Gruppen innerhalb des regulären Ausdrucks verwendet werden. Die Gruppen sind intern numeriert. Zeichenketten, die das Muster einer Gruppe aufweisen, sind dann innerhalb von `replacement` über die Angabe der Gruppen-Nummer in der Form "`\langle Nummer \rangle`" abrufbar (*back referencing*). Alternativ zu `regmatches()` (Abschn. 2.14.5) lassen sich auf diese Weise Zeichenfolgen extrahieren.

Im Beispiel definiert der reguläre Ausdruck eine Zeichenkette, die mit einem oder mehr Buchstaben beginnt, dann in – eingeschlossen eine Gruppe von einer oder mehr Ziffern aufweist und mit einem oder mehr Buchstaben endet. Die Zifferngruppe wird durch back referencing extrahiert.

```
> gsub("^[:alpha:]+-([[:digit:]]+)-[:alpha:]+$",
+       "\\\1", "abc-412-def")
[1] "412"
```

2.14.7 Zeichenketten als Befehl ausführen

Durch die Kombination von `parse()` und `eval()` lassen sich Zeichenketten als Befehle interpretieren und wie direkt eingegebene 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.⁶² 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 (Abschn. 3.1.1).

```
> 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(<expression>)`.

```
> eval(obj1)
[1] 7

> eval(obj2)
[1] 1 4 9
```

⁶²Solcherart erstellte Objekte können mit `deparse()` wieder in Zeichenketten umgewandelt werden.

2.15 Datum und Uhrzeit

Insbesondere bei der Analyse von Zeitreihen⁶³ 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 Hilfe besonderer Klassen.⁶⁴

2.15.1 Datumsangaben erstellen und formatieren

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 unter Beachtung der Zeitzone nennt `Sys.Date()` in Form eines `Date` Objekts. Um selbst ein Datum zu erstellen, ist `as.Date()` zu verwenden.

```
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 `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.⁶⁵ In diesem Format erfolgt auch die Ausgabe, die sich jedoch mit `format(<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"
```

Ihre numerische Repräsentation lässt sich direkt zum Erstellen von `Date` Objekten nutzen.

```
as.Date(x=<Vektor>, origin=<Stichtag>)
```

⁶³Für die Auswertung von Zeitreihen vgl. [Shumway und Stoffer \(2016\)](#); [Hyndman und Athanasopoulos \(2019\)](#) sowie den Abschnitt *Time Series Analysis* der CRAN Task Views ([Hyndman, 2020](#)).

⁶⁴Für eine einführende Behandlung der vielen für Zeitangaben existierenden Subtilitäten vgl. [Grothendieck und Petzoldt \(2004\)](#) sowie `?DateTimeClasses`. Der Umgang mit Zeit- und Datumsangaben wird durch Funktionen des Pakets `lubridate` ([Golemund & Wickham, 2011](#)) erleichtert, das [Wickham und Golemund \(2023, Kap. 17\)](#) eingehend vorstellen: <http://r4ds.hadley.nz/datetime.html>

⁶⁵Siehe Abschn. [2.14.2](#) für `format strings` allgemein sowie `?strptime` für weitere mögliche Elemente bei Datumsangaben. Wenn das Betriebssystem es unterstützt, kann man mit `Sys.setlocale("LC_TIME", "<Länderkennung>")` temporär die Ländereinstellung ändern, um mit Namen für Wochentage und Monate in unterschiedlichen Sprachen umgehen zu können. Die aktive Einstellung gibt `Sys.getlocale()` aus.

Für `x` ist ein numerischer Vektor mit der seit dem Stichtag `origin` verstrichenen Anzahl von Tagen zu nennen. Negative Zahlen stehen dabei für die Anzahl der Tage vor dem Stichtag. Der Stichtag selbst muss in Form eines `Date` Objekts angegeben werden.

```
# Datum, das 374 Tage vor dem 16.12.1910 liegt
> (negDate <- as.Date(-374, origin="1910-12-16"))
[1] "1909-12-07"

> as.numeric(negDate)           # Anzahl Tage vor Standard-Stichtag
[1] -21940
```

2.15.2 Uhrzeit

Objekte der Klasse `POSIXct` (*calendar time*) sind die gebräuchlichste Lösung, um neben dem Datum auch die Uhrzeit eines Zeitpunkts zu repräsentieren. Datum und Zeitpunkt sind als Anzahl der Sekunden codiert, die seit einem Stichtag (meist der 1. Januar 1970) verstrichen sind.⁶⁶ Negative Zahlen stehen dabei für die Anzahl der Sekunden vor dem Stichtag. `POSIXct` Objekte berücksichtigen die Zeitzone sowie die Unterscheidung von Sommer- und Winterzeit.

`Sys.time()` gibt das aktuelle Datum nebst Uhrzeit in Form eines `POSIXct` Objekts aus, allerdings für eine Standard-Zeitzone. Das Ergebnis muss deshalb mit den u. g. Funktionen in die aktuelle Zeitzone konvertiert werden. Alternativ gibt `date()` Datum und Uhrzeit mit englischen Abkürzungen für Wochentag und Monat als Zeichenkette aus, wobei die aktuelle Zeitzone berücksichtigt wird.

```
> Sys.time()
[1] "2009-02-07 09:23:02 CEST"

> date()
[1] "Sat Feb 7 09:23:02 2009"
```

Objekte der Klasse `POSIXlt` (*local time*) speichern dieselbe Information, allerdings nicht in Form der seit einem Stichtag verstrichenen Sekunden, sondern als Liste mit benannten Komponenten: Dies sind numerische Vektoren u. a. für die Sekunden (`sec`), Minuten (`min`) und Stunden (`hour`) der Uhrzeit sowie für Tag der Woche (`wday` beginnend mit 0 für Sonntag), Tag des Monats (`mday`), Tag des Jahres (`yday`, beginnend mit 0 für den 1. Januar), Monat (`mon`, 0 für Januar, 11 für Dezember) und Jahr (`year`, Jahre seit 1900) 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 (Fußnote

⁶⁶Durch Verwendung von Dezimalzahlen lassen sich Sekundenbruchteile, etwa Millisekunden, repräsentieren.

65). Als erstes Argument ist auch die Anzahl der Sekunden zulässig, die seit einem Stichtag verstrichen sind. Dieser muss dann für das Argument `origin` als Datum genannt werden.

```
> (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"

# Zeitpunkt als Anzahl Sekunden seit 1.1.1970 (mit 250 Millisekunden)
> as.POSIXct(678295.25, origin=as.Date("1970-01-01"))
[1] "1970-01-08 21:24:55 CET"

> lDates$mday                                # Tag isoliert
[1] 5 31

> lDates$hour                                # Stunde isoliert
[1] 3 12
```

`POSIXct` Objekte können besonders einfach mit der Funktion `ISOdate()` erstellt werden, die intern auf `strptime()` basiert, aber keinen `format` string benötigt.

```
ISOdate(year=<Jahr>, month=<Monat>, day=<Tag>, hour=<Stunde>,
       min=<Minute>, sec=<Sekunde>, tz=<Zeitzone>)"
```

Für die sich auf Datum und Uhrzeit beziehenden Argumente können Zahlen im 24h-Format angegeben werden, wobei 12:00:00h Voreinstellung für die Uhrzeit ist. Mit `tz` lässt sich die Zeitzone im Form der standardisierten Akronyme festlegen, Voreinstellung ist hier "GMT".

```
# Zeitzone: Central European Time
> ISOdate(2010, 6, 30, 17, 32, 10, tz="CET")
[1] "2010-06-30 17:32:10 CEST"
```

Auch Objekte der Klassen `POSIXct` und `POSIXlt` können mit `format()` in der gewünschten Formatierung ausgegeben werden.

```
> format(myTime, "%H:%M:%S")          # nur Stunden, Minuten, Sekunden
[1] "09:23:02"

> format(lDates, "%d.%m.%Y")         # nur Tag, Monat, Jahr
[1] "05.08.1972" "31.03.1981"
```

Aus Datumsangaben der Klasse `Date`, `POSIXct` und `POSIXlt` lassen sich bestimmte weitere Informationen in Form von Zeichenketten extrahieren, etwa der Wochentag mit `weekdays(<Datum>)`, der Monat mit `months(<Datum>)` oder das Quartal mit `quarters(<Datum>)`.

```
> weekdays(lDates)
[1] "Samstag" "Dienstag"
```

```
> months(lDates)
[1] "August" "März"
```

2.15.3 Mit Datum und Uhrzeit rechnen

Objekte der Klasse `Date`, `POSIXct` und `POSIXlt` verhalten sich in vielen arithmetischen Kontexten in natürlicher Weise, da die sinnvoll für Datumsangaben interpretierbaren Rechenfunktionen besondere Methoden für sie besitzen (Abschn. 17.3.7): So werden zu `Date` Objekten addierte Zahlen 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 im Zeitintervall vom zweiten zum ersten Datum ausgegeben. Hierbei ergeben sich negative Zahlen, wenn das erste Datum zeitlich vor dem zweiten liegt.⁶⁷ Ebenso wie Zahlen lassen sich auch `difftime` Objekte zu `Date` Objekten addieren.

```
> myDate + 365
[1] "1975-11-01"

> (diffDate <- as.Date("1976-06-19") - myDate) # Zeitintervall
Time difference of 596 days

> as.numeric(diffDate) # Intervall als Zahl
[1] 596

> myDate + diffDate
[1] "1976-06-19"
```

Zu Objekten der Klasse `POSIXlt` oder `POSIXct` addierte Zahlen werden als Sekunden interpretiert. Aus der Differenz zweier solcher Objekte entsteht ebenfalls ein Objekt der Klasse `difftime`. Die Addition von `difftime` und `POSIXlt` oder `POSIXct` Objekten ist ebenfalls definiert.

```
> lDates + c(60, 120) # 1, 2 Minuten später
[1] "1972-08-05 03:38:00 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"
```

Mit `round()` bzw. `trunc()` lassen sich Objekte der Klasse `POSIXlt` oder `POSIXct` auf ganze Sekunden, Minuten, Stunden, Tage, Monate oder Jahre runden bzw. tranchieren, indem das Argument `units` auf `"secs"`, `"mins"`, `"days"`, `"months"` oder `"years"` gesetzt wird.

⁶⁷Die 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()`, um welche Einheit es sich handeln soll.

```
> round(lDates, units="days")                      # rund auf nächsten Tag
[1] "1972-08-05 CET"   "1981-04-01 CEST"

> trunc(lDates, units="years")                     # tranchiere auf Jahr
[1] "1972-01-01 CET"   "1981-01-01 CET"
```

In `seq()` (Abschn. 2.4.1) ändert sich die Bedeutung des Arguments `by` hin zu Zeitangaben, wenn für `from` und `to` Datumsobjekte übergeben werden. Für die Schrittweite werden dann etwa die Werte "`⟨Anzahl⟩ years`" oder "`⟨Anzahl⟩ days`" akzeptiert (?`seq.POSIXt`). Dies gilt analog auch für das Argument `breaks` der `cut()` Funktion (Abschn. 2.6.7). Sie teilt kontinuierliche Datumsangaben in Kategorien ein, die etwa durch Stunden (`breaks="hour"`) oder Kalenderwochen (`breaks="week"`) definiert sind (?`cut.POSIXt`). Für weitere geeignete arithmetische Funktionen s. `methods(class="POSIXt")` und `methods(class="Date")`.

```
# jährliche Schritte vom 01.05.2010 bis zum 01.05.2013
> seq(ISOdate(2010, 5, 1), ISOdate(2015, 5, 1), by="years")
[1] "2010-05-01 12:00:00 GMT" "2011-05-01 12:00:00 GMT"
[3] "2012-05-01 12:00:00 GMT" "2013-05-01 12:00:00 GMT"

# 4 zweiwöchentliche Schritte vom 22.10.1997
> seq(ISOdate(1997, 10, 22), by="2 weeks", length.out=4)
[1] "1997-10-22 12:00:00 GMT" "1997-11-05 12:00:00 GMT"
[3] "1997-11-19 12:00:00 GMT" "1997-12-03 12:00:00 GMT"

# 100 zufällige Daten zwischen 13.06.1995 und 4 Wochen später
> secsPerDay <- 60 * 60 * 24                  # Sekunden pro Tag
> randDates  <- ISOdate(1995, 6, 13)
+           + sample(0:(28*secsPerDay), 100, replace=TRUE)

# runde Zufallsdaten Tag-genau
> head(round(randDates, units="days"), n=3)
[1] "1995-07-03 GMT" "1995-06-24 GMT" "1995-07-11 GMT"

# teile Daten in Kalenderwochen ein
> randWeeks <- cut(randDates, breaks="week")
> summary(randWeeks)                           # Häufigkeiten
1995-06-12  1995-06-19  1995-06-26  1995-07-03  1995-07-10
      15          26          20          37          2
```

Kapitel 3

Daten importieren, exportieren, aufbereiten und aggregieren

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. Typischerweise liegen Datensätze aber in Form von mit anderen Programmen erstellten Dateien oder Datenbanken vor (Abschn. 3.1). Die notwendigen Schritte, um importierte Datensätze aufzubereiten und zu aggregieren, werden in Abschn. 3.3 und 3.4 dargestellt. Aspekte der Datenqualität diskutiert Abschn. 4.3.

3.1 Daten importieren und exportieren

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 eines Referenz-Datensatzes verwendet werden, um diesen gegen unbeabsichtigte Veränderungen zu schützen. Der Datenaustausch mit anderen Programmen wird vertieft im Manual „R Data Import/Export“ ([R Core Team, 2020b](#)) behandelt. Siehe Abschn. 3.2 für die Form der Pfadangaben zu Dateien in den folgenden Abschnitten.

3.1.1 Datentabellen im Textformat

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 (Abschn. 1.4.1). Andernfalls werden Variablennamen beim Importieren automatisch mit der Funktion `make.names()` zu einer gültigen Zeichenkette umgewandelt und müssen ggf. manuell nachbearbeitet werden (Abschn. 2.14).

Mit `read.table()` werden in Textform vorliegende Daten geladen und in einem Objekt der Klasse `data.frame` ausgegeben. Wichtige Argumente von `read.table()` sind in Tab. 3.1 dargestellt. RStudio vereinfacht den Import von Textdaten über das Menü *Tools: Import Dataset*.

Tabelle 3.1: 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 ¹
<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 ist jeglicher zusammenhängender <i>whitespace</i> (Leerzeichen oder Tabulatoren), unabhängig davon, wie viele davon aufeinander folgen. Andere häufig verwendete Werte sind das Komma (",") oder das Tabulatorzeichen ("\"t") ²
<code>strip.white</code>	Bestimmt, ob whitespace vor bzw. nach einem Wert entfernt werden soll. In der Voreinstellung <code>FALSE</code> geschieht dies bei numerischen Werten, nicht aber bei Zeichenketten. Sollte auf <code>TRUE</code> gesetzt werden, wenn auch bei <code>sep</code> von der Voreinstellung abgewichen wird
<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. In der Voreinstellung <code>NA</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 automatisch in Gruppierungsfaktoren (<code>factor</code>) konvertieren oder als <code>character</code> Vektoren speichern (Voreinstellung <code>FALSE</code>)

Für das Argument `file` können nicht nur lokal gespeicherte Dateien angegeben werden: Die Funktion liest unter Windows mit `file="clipboard"` auch Werte aus der Zwischenablage, die

¹ Werden die einzulesenden Daten von R nicht gefunden, ist zunächst mit `dir()` zu prüfen, ob das von R durchsuchte Verzeichnis (ohne explizite Angabe eines Pfades ist es das mit `getwd()` angezeigte Arbeitsverzeichnis) auch jenes ist, das die Datei enthält.

² Sobald für das Argument `sep` ein selbst gewählter Wert wie "\t" vergeben wird, ändert sich die Bedeutung dieses Zeichens: Tauchen in der Datei dann etwa zwei Tabulatoren hintereinander auf, interpretiert R dies als eine leere Zelle der Datenmatrix, also als fehlenden Wert. Ebenso gelten zwei nur durch ein Leerzeichen getrennte Werte nicht mehr als zwei Zellen.

dort in einem anderen Programm etwa mit `Strg+c` hineinkopiert wurden.³ Ebenso liest sie Daten von der Konsole, wenn `file=stdin()` verwendet wird.

```
> (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
```

Alternativ kann ein Datensatz auch innerhalb des Befehlsskripts in Form einer Zeichenkette definiert werden, die `read.table()` dann mit der Option `text` einliest. Sind dabei Zeichenketten in doppelte Anführungszeichen gesetzt, müssen die äußeren Anführungszeichen einfache sein (Abschn. 1.4.5, Fußnote 32).

```
> txt <- 'X Y Z
123 "A B C" 34.8
99 "D E F" 3543.2'

> read.table(text=txt, header=TRUE)
  X     Y     Z
1 123 A B C   34.8
2 99 D E F 3543.2
```

Online verfügbare Dateien können mit `file=url("<URL>")` direkt von einem Webserver geladen werden.⁴ Anders als in Webbrowsersn muss dabei der Protokollteil der Adresse (etwa `http://` oder `https://`) explizit genannt werden, also z. B.

```
> read.table(file=url("http://www.server.de/datei.txt"), ...)
```

Der Abschnitt **Web Technologies and Services** der CRAN Task Views ([Chamberlain, Leeper, Mair, Ram & Gandrud, 2020](#)) stellt Pakete vor, um Daten direkt aus Webseiten zu extrahieren. Eine vertiefte Beschreibung findet sich in [Wickham und Grolemund \(2023, Kap. 24\)](#).⁵

Zum Speichern von Objekten in Textdateien dient `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()` (Tab. 3.1). Statt in eine Datei kann `write.table()` mit `file="clipboard"`

³Unter MacOS ist `file=pipe("pbpaste")` zu verwenden.

⁴Statt einer Datei akzeptieren die meisten Funktionen für das Argument `file` allgemein eine `connection`, bei der es sich etwa auch um die Verbindung zu einem Vektor von Zeichenketten handeln kann, vgl. `?textConnection`.

⁵<https://r4ds.hadley.nz/webscraping>

eine begrenzte Menge von 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. ein Datensatz **myDf** im aktuellen Arbeitsverzeichnis in Textform gespeichert und später wieder eingelesen werden soll, so lauten die Befehle:

```
> myDf <- data.frame(IV=factor(rep(c("A", "B"), 5)), DV=rnorm(10))
> write.table(myDf, file="data.txt", row.names=FALSE)
> myDf <- read.table("data.txt", header=TRUE, stringsAsFactors=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.

Für Daten in Textform aus sehr großen Dateien arbeiten **fread()** und **fwrite()** aus dem Paket **data.table** ([Dowle & Srinivasan, 2020](#)) deutlich schneller als **read.table()** und **write.table()**. Verbesserungen verspricht auch das Paket **readr** ([Wickham, Hester & Francois, 2020](#)).

3.1.2 R-Objekte

Eine andere Möglichkeit zur Verwaltung von Objekten in externen Dateien bieten **save(<Daten>, file=<Dateiname>)** zum Speichern und **load("<Dateiname>")** zum Öffnen. Unter **<Daten>** können dabei verschiedene Objekte durch Komma getrennt angegeben werden. Alternativ lässt sich das Argument **list** verwenden, das einen Vektor mit den Namen der zu speichernden Objekte akzeptiert. **save(list=ls(), ...)** würde also alle sichtbaren Objekte des workspace speichern.

Die Daten werden in einem R-spezifischen, aber plattformunabhängigen 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.

```
> save(myDf, file="data.RData") # speichere myDf im Arbeitsverzeichnis
> load("data.RData")           # lies myDf wieder ein
```

Dies kann auch eine unerwünschte Eigenschaft sein. Es besteht nämlich die Gefahr, dass schon bestehende Objekte im workspace unbeabsichtigt und unbemerkt überschrieben werden. Sicherer ist es deshalb, stattdessen die Funktion **saveRDS(<Daten>, file=<Dateiname>)** zum Speichern bzw. **readRDS(file=<Dateiname>)** zum Öffnen einzelner Objekte zu verwenden. Das Ergebnis von **readRDS()** muss explizit einem Objekt zugewiesen werden.

```
> saveRDS(myDf, file="data.rda") # speichere myDf im Arbeitsverz.  
> myDf <- readRDS("data.rda")    # myDf einlesen und Objekt zuweisen
```

Ähnlich wie `save()` Objekte in einem binären Format speichert, schreibt `dump("<Objekt>", file=<Dateiname>")` die Inhalte von Objekten in eine Textdatei mit R-Befehlen, die sich auch durch gewöhnliche Texteditoren bearbeiten lässt. Auf diese Weise erzeugte Dateien lassen sich mit `source(file=<Dateiname>")` einlesen. Die Namen der Objekte bleiben dabei erhalten.

```
> dump("myDf", file="dumpMyDf.txt")  
> source("dumpMyDf.txt")
```

3.1.3 Daten mit anderen Programmen austauschen

Der Austausch von Daten zwischen R und anderen Programmen ist oft in Form von reinen Textdateien möglich. Diese Methode ist auch recht sicher, da sich die Daten leicht mit einem Texteditor ansehen sowie prüfen lassen und der korrekte Transfer in allen Stufen kontrolliert werden kann. In diesem Fall kommen in R meist `read.table()` und `write.table()` zum Einsatz (Abschn. 3.1.1).

Beim Import- und Export von Daten in Dateiformaten kommerzieller Programme besteht dagegen oft die Schwierigkeit, dass die Formate 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 sicher zu ermitteln. Beim Austausch von Daten über proprietäre Formate ist aus diesem Grund Vorsicht geboten – bevorzugt sollten einfach strukturierte Datensätze verwendet werden.⁶

Soweit möglich sollten Variablennamen schon im Quelldatensatz so gewählt werden, dass sie den R-Konventionen entsprechen und beispielsweise kein #, %, ' oder Leerzeichen enthalten (Abschn. 1.4.1). Andernfalls werden Variablennamen beim Importieren meist automatisch mit der Funktion `make.names()` zu einer gültigen Zeichenkette umgewandelt und müssen ggf. manuell nachbearbeitet werden (Abschn. 2.14).

Programme zur Tabellenkalkulation

Wurde ein Programm zur Tabellenkalkulation (etwa Microsoft Excel oder OpenOffice Calc) zur Dateneingabe benutzt, empfiehlt es sich, die zu importierenden Daten dort als rechteckigen Bereich mit Beobachtungen in den Zeilen und Variablen in den Spalten anzugeben, wobei Variablennamen die erste Zeile des Datenbereichs ausmachen. Nach Möglichkeit sind die Variablennamen entsprechend den R-Konventionen zu wählen.

Ein einfacher Datentransfer ist dann möglich, wenn die Daten aus der Tabellenkalkulation in eine Textdatei exportiert werden, wobei als Spalten-Trennzeichen der Tabulator verwendet

⁶Für den Transfer zwischen vielen hier nicht erwähnten Programmen existieren Zusatzpakete, die sich auf CRAN finden lassen (Abschn. 1.3). Das Programm Stat/Transfer ([Circle Systems, 2019](#)) ist eine kommerzielle Lösung, um Daten zwischen R, Tabellenkalkulationen, SAS, Stata und einigen anderen Formaten auszutauschen.

wird. Der Transfer von Datumsangaben ist dabei jedoch fehlerträchtig. 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="<Datei>", 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 unter Windows also auch wie folgt möglich: Zunächst wird in der Tabellenkalkulation der gewünschte Datenbereich inkl. der Variablenamen 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 bei sehr kleinen Datensätzen 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)
```

Um Excel-Dateien im derzeit aktuellen `.xlsx` Format ohne Zwischenschritt direkt in R zu verwenden, eignet sich das Paket `readxl` ([Wickham & Bryan, 2019](#)) mit der Funktion `read_xlsx()`. Details erläutern [Wickham und Grolemund \(2023, Kap. 20\)](#).⁸ Verschiedene optionale Argumente erlauben es dabei auch solche Dateien einzulesen, deren Daten nicht bereits mit R-Konventionen harmonisiert wurden.

- **sheet**: Zeichenkette, die das einzulesende Tabellenblatt bezeichnet, in der Voreinstellung das erste.
- **skip**: Beginnt der Datenbereich im Arbeitsblatt nicht in der ersten Zeile, kann für dieses Argument die Anzahl zu überspringender Zeilen angegeben werden.
- **range**: Ist der Datenbereich frei auf dem Arbeitsblatt plaziert, kann der Bereich in Form einer Zeichenkette wie etwa "B3:E15" definiert werden. Die Angabe des Zellbereichs erfolgt in Excel-Konvention. Die Spalte einer Zelle ist zuerst in Form eines Buchstabens zu nennen, direkt gefolgt von einer Zahl für die Zeilennummer. Vor dem Doppelpunkt steht die linke obere Zelle, dahinter die rechte untere.
- **col_names**: Voreinstellung ist `TRUE` für den Fall, dass die Variablenamen in der ersten Zeile des Datenbereichs stehen. Auch möglich ist ein Vektor von Zeichenketten, der die Variablenamen definiert. Dies ist nützlich, wenn in Excel keine Spaltennamen vergeben wurden oder diese stark vom in R notwendigen Format abweichen. In letzterem Fall muss die entsprechende Zeile mit **skip** übersprungen werden.
- **na**: Ein Vektor von Zeichenketten, der die in Excel verwendeten Codes für fehlende Werte aufführt, etwa `c("", "NA")`. In der Voreinstellung gelten leere Zellen als fehlende Werte.

⁷Sofern dies nicht in den Ländereinstellungen des Betriebssystems geändert wurde.

⁸<https://r4ds.hadley.nz/spreadsheets>

- `col_types`: Ein Vektor von Zeichenketten, der den jeweiligen Datentyp der Variablen festlegt. Möglich sind u. a. "numeric", "text", "date" sowie "skip" zum Überspringen einer Spalte und "guess", um `read_xlsx()` selbst die Wahl zu überlassen. Letzteres ist die Voreinstellung.

Die Ausgabe hat die Klasse `tbl_df` (Abschn. 3.4.1), die sich in einen gewöhnlichen Datensatz umwandeln lässt.

```
> library(readxl)                      # für read_xlsx()
> d <- read_xlsx("data.xlsx", sheet="Datenblatt", skip=2,
+                 col_names=c("ID", "Gruppe", "Wert"),
+                 col_types=c("text", "text", "numeric"))

> as.data.frame(d)
  ID Gruppe Wert
1  1      A    4
2  2      A    6
3  3      B    7
4  4      B    8
5  5      A   11
```

Das Paket `openxlsx` (Schauberger & Walker, 2020) ist eine Alternative zu `readxl`. `openxlsx` verfügt dabei über den größeren Funktionsumfang, insbesondere mit vielen Optionen zum Schreiben besonders formatierter Excel-Dateien.

Bei uneinheitlich formatierten Datumsangaben in Excel-Tabellen kann es dazu kommen, dass Datumsangaben nur als ganzzahliger Wert importiert werden, der die Anzahl der Tage seit eines Excel-spezifischen Stichtags darstellt. Bei der Umwandlung in ein Datum helfen die Funktionen `convertToDate()` bzw. `convertDateTime()` aus dem Paket `openxlsx`.

Excel konvertiert automatisch alle Zeichenketten in Datumsangaben, wenn sie ein als Datum interpretierbares Muster aufweisen, etwa die Gen-Namen SEPT2 oder MARCH1. Eine Zeichenkette wie 2310009E13 wird automatisch als Dezimalzahl in verkürzter Exponentialschreibweise interpretiert. Bei Daten aus Programmen zur Tabellenkalkulation ist es deshalb wichtig, die Datenqualität eingehend zu prüfen (Abschn. 4.3).

SPSS, Stata und SAS

SPSS verfügt mit dem *Integration Plug-in for R* über eine nachträglich installierbare Erweiterung, mit der R-Befehle direkt in SPSS verwendet werden können. Auf diese Weise lassen sich dort nicht nur in R verwaltete Datensätze nutzen, sondern auch ganze Auswertungsschritte bis hin zur Erstellung von Diagrammen in R-Syntax durchführen. Genauso erlaubt es die Erweiterung, mit SPSS erstellte Datensätze im R-Format zu exportieren.⁹

⁹Für einen detaillierten Vergleich der Arbeit mit R, SAS und SPSS vgl. Muenchen (2011) sowie Kleinman und Horton (2014), die auch den Datenaustausch zwischen diesen Programmen behandeln.

SPSS-Datensätze können in R mit Funktionen gelesen und geschrieben werden, die das Paket `haven` (Wickham & Miller, 2020) bereitstellt.¹⁰ So liest die `read_sav("<Dateiname>")` Funktion `<Dateiname>.sav` Dateien.

```
> library(haven)           # für read_sav(), write_sav(), as_factor()
> myDf_sav <- read_sav("myDf.sav")    # lade SPSS Datei
> head(myDf_sav, n=4)
# A tibble: 4 x 2
  IV      DV
  <dbl+lbl> <dbl>
1 1 [A]   1.13
2 2 [B]   1.68
3 1 [A]  -1.18
4 2 [B]  -0.815
```

Obwohl das Ergebnis von `read_sav()` zunächst wie ein Datensatz aussieht, weist es als Objekt der Klasse `tbl_df` (*tibble*) neben vielen Gemeinsamkeiten auch subtile Unterschiede auf (Abschn. 3.4.1). Eine Besonderheit ist der Umgang mit *labels*. In SPSS können Variablen ein separates, dort als *Beschriftung* bezeichnetes *label* besitzen. Dies ist eine ausführlichere Bezeichnung einer Variable, als es der Variablenname zulassen würde. Beim Import bleiben Variablen-Labels als Zeichenkette im Attribut `"label"` der Variable im Datensatz erhalten.

Zusätzlich können in SPSS auch einzelne Werte einer Variable mit *labels* versehen sein (*Werte*), wobei dies nicht notwendigerweise nur für Faktoren möglich ist. Auch müssen nicht alle vorkommenden Werte ein *label* besitzen. Da das Konzept von Werte-Labels in SPSS nicht vollständig deckungsgleich mit den Stufen von Faktoren in R ist, speichert `read_sav()` Variablen mit Werte-Labels zunächst in einer Zwischenstufe als Objekte der Klasse `haven_labelled`. Dies sind numerische Variablen mit einem Attribut `labels`, das über einen benannten Vektor die Zuordnung von Werten und *labels* codiert.

```
> str(myDf_sav)
Classes 'tbl_df', 'tbl' and 'data.frame': 10 obs. of 2 variables:
 $ IV: 'haven_labelled' num 1 2 1 2 1 2 1 2 1 2
   ..- attr(*, "label")= chr "Gruppierungsfaktor"
   ..- attr(*, "format.spss")= chr "F8.0"
   ..- attr(*, "labels")= Named num 1 2
     ...- attr(*, "names")= chr "A" "B"
 $ DV: num 1.13 1.681 -1.178 -0.815 -0.443 ...
   ..- attr(*, "label")= chr "Messwerte"
   ..- attr(*, "format.spss")= chr "F8.2"
```

Objekte der Klasse `haven_labelled` lassen sich in einem anschließenden Schritt explizit mit `as_factor()` in Faktoren konvertieren, wobei die *labels* zu Faktorstufen werden.

```
# konvertiere Variable IV in normalen Faktor
> myDf_org <- transform(myDf, IV=as_factor(IV))
```

¹⁰Eine Alternative, auch für Dateien aus SAS und Stata, ist das im Basisumfang von R enthaltene Paket `foreign`. Zudem kann SPSS Textdateien einlesen, wie sie mit `write.table(..., row.names=FALSE, sep="t", dec=",")` erstellt werden.

```
> str(myDf_org)
'data.frame': 10 obs. of 2 variables:
 $ IV: Factor w/ 2 levels "A","B": 1 2 1 2 1 2 1 2 1 2
 ..- attr(*, "label")= chr "Gruppierungsfaktor"
 $ DV: num 1.13 1.681 -1.178 -0.815 -0.443 ...
 ..- attr(*, "label")= chr "Messwerte"
```

`write_sav(<Datensatz>, path=<Dateiname>)` aus dem Paket `haven` speichert in R bearbeitete Datensätze im SPSS-Format als Dateien mit der Endung `.sav`. Dabei wird das Messniveau einer Variable in SPSS automatisch entsprechend der Klasse des R-Vektors gewählt. Die Funktion wandelt auch das Attribut `label` einer Variable im R-Datensatz in die *Bezeichnung* in SPSS um. Wenn eine Variable die Klasse `haven_labelled` besitzt, wird eine im Attribut `labels` definierte Zuordnung von Bezeichnungen zu Werten automatisch in SPSS *Werte* übernommen.

```
# intVar hat Klasse integer wegen :
> myDf <- data.frame(IV=factor(sample(c("A", "B"), 10, replace=TRUE)),
+                      DV=rnorm(10),
+                      intVar=sample(1:3, 10, replace=TRUE))

# setze Variablen-Label
> attr(myDf$IV, "label") <- "Gruppierungsfaktor"

# setze Werte-Labels, Werte müssen auch Klasse integer haben
> class(myDf$intVar) <- "haven_labelled"
> attr(myDf$intVar, "labels") <- c("Val1"=1L, "Val2"=2L, "Val3"=3L)

# speichere als SPSS Datensatz
> write_sav(myDf, "myDf.sav")
```

Auch für Stata und SAS gibt es die Möglichkeit, Daten mit R auszutauschen, wofür ebenfalls Funktionen aus dem Paket `haven` dienen. Dabei erfolgt die Konvertierung von Variablen-Labels und Werte-Labels analog zum Import von SPSS-Dateien. `read_dta()` liest Dateien in Statas `dta` Format, `write_dta()` schreibt sie. Der Austausch mit SAS geschieht analog über `read_sas()` und `write_sas()` für Dateien im Format `sas7bdat`.

Datenbanken

In R lassen sich Daten aus Datenbanken vieler verschiedener Formate direkt lesen und schreiben.¹¹ Dies bietet sich etwa bei extrem großen Datensätzen an, die zuviel Arbeitsspeicher belegen würden, wenn man sie als Ganzes in R öffnen wollte (Abschn. 17.5). Zunächst muss dafür eine Verbindung zur Datenbank hergestellt werden. Daraufhin lassen sich SQL-Kommandos

¹¹Für eine detaillierte Beschreibung der Verwendung von Datenbanken vgl. Wickham und Grolemund (2023, Kap. 21): <https://r4ds.hadley.nz/databases>. Die Verwendung von Datenbanken in RStudio ist unter <https://solutions.posit.co/connections/db/> dokumentiert. Im Abschnitt *Databases* der CRAN Task Views (Tang & Balamuta, 2023) finden sich relevante Zusatzpakete.

wie SELECT oder CALL in der üblichen Syntax anwenden,¹² um Daten zwischen der Datenbank und R auszutauschen. Das sonst notwendige Semikolon am Ende eines SQL-Befehls ist dabei optional.

Für Verbindungen zu einer Datenbank definiert das Paket DBI ([R Special Interest Group on Databases, Wickham & Müller, 2019](#)) eine einheitliche Schnittstelle. Diese wird dann für unterschiedliche Datenbanktypen in weiteren Zusatzpaketen implementiert, entweder als *client* für das plattformunabhängige ODBC Protokoll (*Open DataBase Connectivity*) mit dem Paket odbc ([Hester & Wickham, 2020](#)) oder in einzelnen Paketen, die spezifisch etwa für MySQL-, PostgreSQL- oder SQLite-Datenbanken sind.

Dabei ist zu beachten, dass ODBC-Treiber für den gewünschten Datenbanktyp zusätzlich auf Ebene des Betriebssystems installiert werden müssen. Auch muss dort ein *Data Source Name* (DSN) als Name der Datenbankverbindung erstellt werden. Übernimmt dies ein Datenbank-Server nicht automatisch selbst, kann dafür unter Windows *ODBC-Datenquellen* aus der Gruppe *Verwaltung* der Systemsteuerung verwendet werden. Je nach eingesetzter R-Version muss die Verbindung für 32bit- oder 64bit-Clients ausgelegt sein. vignette("RODBC") erläutert weitere Besonderheiten, etwa für verschiedene Datenbank-Typen, und gibt Hinweise zur Installation von Treibern sowie zum Erstellen von DSN.¹³

Als Beispiel für eine Datenbank-Verbindung über die DBI-Schnittstelle soll mit dem Paket RSQLite ([Müller, Wickham, James & Falcon, 2020](#)) zunächst eine SQLite-Datenbank neu erstellt werden, um darin einen R-Datensatz als *table* abzuspeichern. SQLite-Datenbanken zeichnen sich dadurch aus, dass die Datenbank eine Datei ist und kein separater Datenbank-Server lokal oder im Netzwerk laufen muss. RSQLitebettet den notwendigen server für eine DB-Datei ein.

Nachdem mit dbDriver("<DB-Typ>") ein für SQLite passendes Treiber-Objekt erzeugt wurde, kann die Verbindung zur Datenbank mit dbConnect(<DB-Treiber>, "<DB-Name>") hergestellt werden. Existiert die zum übergebenen Datenbank-Namen gehörende Datei noch nicht, wird sie automatisch neu angelegt.

```
> library(RSQLite)           # für db<Funktion>()
> drv <- dbDriver("SQLite")   # SQLite-Treiber erzeugen
> con <- dbConnect(drv, "myDf.db") # erstelle neue DB in Datei myDf.db
```

dbWriteTable(<DB-Verbindung>, name="<table-Name>", value=<Datensatz>) speichert den übergebenen R-Datensatz in der verbundenen Datenbank unter dem angegebenen table-Namen. dbListTables(<DB-Verbindung>) gibt einen Vektor mit allen table-Namen der Datenbank zurück. dbListFields(<DB-Verbindung>, "<table-Name>") nennt die Spalten-Namen des gewünschten Datenbank-tables.

```
# Daten simulieren
> IQ      <- rnorm(2*50, mean=100, sd=15)
> rating <- sample(LETTERS[1:3], 2*50, replace=TRUE)
> sex     <- factor(rep(c("f", "m"), times=50))
> myDf   <- data.frame(sex, IQ, rating)
```

¹²<https://www.1keydata.com/sql/> – <https://sqlzoo.net/>

¹³Siehe auch <https://www.stata.com/support/faqs/data-management/configuring-odbc-win/>

```
# Datensatz myDf in table MyDataFrame der Datenbank speichern
> dbWriteTable(con, name="MyDataFrame", value=myDf, row.names=FALSE)
[1] TRUE
```

```
> dbListTables(con)                      # alle tables der Datenbank
[1] "MyDataFrame"

> dbListFields(con, "MyDataFrame")       # alle Spalten des table
[1] "sex"   "IQ"   "rating"
```

dbReadTable(<DB-Verbindung>, "<table-Name>") gibt das gewünschte Datenbank-table vollständig als R-Datensatz aus. dbGetQuery(<DB-Verbindung>, "<SQL-Befehl>") übermittelt einen SQL-Befehl und liefert das Ergebnis vollständig zurück.

```
> out <- dbReadTable(con, "MyDataFrame") # table vollständig speichern
> head(out, n=4)                      # Kontrolle
  sex      IQ rating
1 f 92.51217 A
2 m 89.28590 C
3 f 104.96750 A
4 m 102.98000 C
```

```
# SQL-Kommando: pro Gruppe berechnete Kennwerte (Mittelwert, Summe IQ)
> dbGetQuery(con, "SELECT sex, AVG(IQ) AS mIQ,
+              SUM(IQ) AS sIQ
+              FROM MyDataFrame GROUP BY sex")
  sex      mIQ      sIQ
1 f 101.23562 5061.781
2 m 99.34861 4967.430
```

dbSendQuery(<DB-Verbindung>, "<SQL-Befehl>") leitet den übergebenen SQL-Befehl an die Datenbank weiter und gibt ein Objekt zurück, aus dem sich mit dbFetch(<query>, n=<Anzahl>) die gewünschte Anzahl an zur Anfrage passenden Zeilen schrittweise lesen lassen. Ob alle Daten ausgelesen wurden, gibt dbHasCompleted(<query>) aus. dbClearResult(<query>) setzt die zum schrittweise Lesen offene query zurück. Dieses Vorgehen ist insbesondere bei sehr umfangreichen Ergebnissen sinnvoll. Siehe Abschn. 17.1.2 für die Verwendung von while(), um einen Befehl so oft zu wiederholen, wie eine bestimmte Nebenbedingung erfüllt ist.

```
# IQ und rating für Personen, die rating A vergeben haben
> res <- dbSendQuery(con, "SELECT IQ, rating
+                           FROM MyDataFrame WHERE rating = 'A'")

# rufe Daten schrittweise solange ab, wie noch nicht alle passenden
# Zeilen ausgegeben wurden
> while(!dbHasCompleted(res)) {
+   partial <- dbFetch(res, n=3)          # rufe 3 Zeilen ab
+   print(partial)                      # gib diese Zeilen aus
```

```
+ }
      IQ rating
1 102.80542      A
2 114.06305      A
3 91.76563       A

      IQ rating
4 120.3715       A
5 116.5732       A
6 108.8182       A
# ...

> dbClearResult(res)                      # query zurücksetzen
[1] TRUE
```

`dbRemoveTable(<DB-Verbindung>, "<table-Name>")` löscht eine Table in der verbundenen Datenbank, `dbDisconnect(<DB-Verbindung>)` beendet eine Datenbank-Verbindung. Der Rückgabewert beider Funktionen zeigt an, ob der jeweilige Befehl erfolgreich ausgeführt wurde.

```
> dbRemoveTable(con, "MyDataFrame")          # table löschen
[1] TRUE

> dbDisconnect(con)                        # DB-Verbindung trennen
[1] TRUE
```

3.1.4 Daten in der Konsole einlesen

Während es `c(<Wert1>, <Wert2>, ...)` zwar erlaubt, auf der Konsole Vektoren aus Werten zu bilden und auch ganze Datensätze einzugeben, ist dieses Vorgehen aufgrund der ebenfalls einzutippenden Kommata ineffizient. 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 unstrukturierte Textdateien eingelesen werden (Abschn. 3.1.5). 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 Befehlsaufruf `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
```

3.1.5 Unstrukturierte Textdateien nutzen

Nicht immer liegen einzulesende Daten bereits in Form einer Tabelle vor, für die sich `read.table()` eignet. Daten aus Textdateien, die noch nicht in rechteckiger Form organisiert sind, können mit der Funktion `scan()` eingelesen werden. Sie eignet sich für Textdateien, in denen jede Zeile eine Aneinanderreihung gleich aufgebauter Gruppen von Werten unterschiedlichen Datentyps ist. Eine Gruppe könnte etwa eine Zeichenkette gefolgt von einer Zahl sein.

Als erstes Argument von `scan()` ist der Dateiname zu nennen. Das zweite Argument `what` akzeptiert nicht nur wie in Abschn. 3.1.4 demonstriert einen einzelnen Wert, sondern auch eine Liste mit benannten Komponenten. In dieser Liste ist jedem Namen der zu erwartende Datentyp in der Form von `numeric()`, `character()` oder `logical()` zuzuweisen. Das Ergebnis ist eine Liste mit den im Argument `what` definierten Komponenten. Sie kann ggf. mit `as.data.frame()` in einen Datensatz umgewandelt werden.

```
# Aufbau Datei scan.txt:
# Lorem 123 ipsum 352
# dolor 394 sit 939
> x <- scan("scan.txt", what=list(A=character(), B=numeric()))
Read 4 records
> x
$A
[1] "Lorem" "ipsum" "dolor" "sit"

$B
[1] 123 352 394 939

> as.data.frame(x)
   A    B
1 Lorem 123
2 ipsum 352
3 dolor 394
4 sit 939
```

Neben einer Vielzahl weiterer Argumente von `scan()` legt etwa `nlines` die Anzahl einzulesender Zeilen fest (in der Voreinstellung die gesamte Datei), `skip` die Anzahl von Zeilen, die am Beginn der Datei übersprungen werden sollen und `na.strings`, die Codierung fehlender Werte. Ist dies eine benannte Liste analog zu `what`, kann für jeden Wertetyp ein eigener Code festgelegt werden.

Mit `readLines("<Pfad>", n=<Anzahl>)` können auch ganz unstrukturierte Daten im Textformat eingelesen werden. Das Ergebnis ist ein Vektor aus Zeichenketten, wobei jedes seiner Elemente jeweils eine Zeile der Datei speichert. Die einzelnen Zeilen können dann mit Methoden zur Manipulation von Zeichenketten weiterverarbeitet werden, die Abschn. 2.14 vorstellt.

Über das Argument `n` von `readLines()` lässt sich steuern, wie viele Zeilen der Datei gleichzeitig gelesen werden sollen – in der Voreinstellung die gesamte Datei. Setzt man etwa `n=5`, umfasst die Ausgabe des ersten Aufrufs die Zeilen 1–5 in Form von fünf Elementen eines Vektors aus Zeichenketten. Dabei merkt sich `readLines()` die Position, an der zuletzt gelesen wurde und setzt beim nächsten Aufruf an dieser Position fort. Ein erneuter Aufruf mit `n=4` würde also die Zeilen 6–9 zurückliefern.

Analog speichert `writeLines(<Vektor>, "<Datei>")` die Inhalte eines Vektors aus Zeichenketten in die angegebene Datei, wobei jedes Element des Vektors in eine separate Zeile geschrieben wird.

3.2 Dateien verwalten

R verfügt über eine Reihe von Funktionen, die Dateien finden und verändern können, die zum Import oder Export von Daten benötigt werden. Dabei spielen Pfadangaben zu Ordnern und Dateien eine zentrale Rolle.

Obwohl unter Windows üblicherweise der *backslash* \ in Pfadangaben als Verzeichnistrenner dient, sollte er in R nicht verwendet werden. Stattdessen ist bei allen Pfadangaben wie unter MacOS und Linux bevorzugt der *forward slash* / zu benutzen, etwa "c:/work/r/datei.txt". Alternativ ist weiterhin der doppelte backslash \\ möglich: "c:\\work\\r\\datei.txt". Pfade können entweder relativ zum aktuellen, von `getwd()` ausgegebenen Arbeitsverzeichnis sein ("path/file.txt"), oder absolut, d. h. wie beim vorherigen Beispiel beginnend mit dem Laufwerksbuchstaben (Windows) oder dem Stammverzeichnis "/" (MacOS, Linux).

3.2.1 Dateien auswählen

Die Pfadangabe einer Datei lässt sich entweder interaktiv oder über ein vorgegebenes Suchmuster bestimmen. Den Namen einer einzelnen Datei erhält man interaktiv über ein Dialogfeld zur Dateiauswahl, das durch `file.choose()` aufgerufen wird. Die Funktion gibt den Namen der ausgewählten Datei inkl. des vollständigen Pfades zurück. Eine Variante, die gleichzeitig mehrere Dateien auswählen lässt, existiert nur unter Windows. Dort gibt `choose.files()` einen Vektor von Pfadnamen zu den ausgewählten Dateien zurück.

Mit `list.files()` können alle Dateien in einem Ordner aufgelistet werden, deren Name zu einem Suchmuster passt.

```
list.files("<Ordner>", pattern="<Suchmuster>", all.files=FALSE,
          full.names=FALSE, recursive=FALSE, ignore.case=FALSE)
```

Als erstes Argument ist der Pfad zu einem Ordner anzugeben. `pattern` akzeptiert ein Suchmuster in Form eines regulären Ausdrucks (Abschn. 2.14.4), das sich für Dateinamen häufig einfacher mit `glob2rx()` über Platzhalter wie `*` formulieren lässt (Abschn. 2.14.5). Versteckte Dateien (etwa solche, deren Name mit einem Punkt beginnt), werden nur mit `all.files=TRUE` angezeigt. In der Voreinstellung erhält man nur die Namen der Dateien im angegebenen Ordner, die zum Suchmuster passen. Benötigt man den vollständigen Pfad zur Datei, ist `full.names=TRUE` zu setzen. Die Argumente `recursive` und `ignore.case` bestimmen jeweils, ob auch Unterordner durchsucht bzw. ob Groß- und Kleinschreibung im Dateinamen beim Abgleich mit dem Suchmuster ignoriert werden sollen.

```
# vollständige Pfadnamen für alle Dateien im Verzeichnis d:/files
# mit der Endung .txt
> paths <- list.files(path="d:/files", pattern="\.\.txt$",
+                      full.names=TRUE)

# lies diese Dateien in eine Liste aus Datensätzen ein
> DFList <- lapply(paths, function(f) {
+   read.table(f, header=TRUE) })

# verbinde alle Datensätze in der Liste zu einem gemeinsamen Datensatz
> DFall <- do.call(rbind, DFList)
```

Alternativ erhält man mit `Sys.glob("<Pfad>")` aus einem vorgegebenen Pfad einen Vektor von Dateinamen, die auf ein bestimmtes Muster passen. Die Pfadangabe gibt sowohl den Pfad als auch das Muster für die Dateinamen vor, wobei bestimmte Platzhalter erlaubt sind: `*` steht für eine beliebige Zeichenfolge, `?` für ein einzelnes beliebiges Zeichen, `.` für das aktuelle Verzeichnis, `..` für das Verzeichnis eine Ebene über dem aktuellen Verzeichnis und `~` für das Heimverzeichnis des Benutzers. Ob die zurückgegebenen Dateipfade absolut oder relativ zum aktuellen Arbeitsverzeichnis sind, orientiert sich am übergebenen Suchmuster.

```
# alle Dateien mit Endung .tex im Parallelordner gddmr_tex
> texFiles1 <- Sys.glob("../gddmr_tex/*.tex")           # relativer Pfad
> head(texFiles1, n=4)
[1] "../gddmr_tex/gddmr.tex"
[2] "../gddmr_tex/r1_00_preface.tex"
[3] "../gddmr_tex/r1_00_settings.tex"
[4] "../gddmr_tex/r1_01_first_steps.tex"

> texFiles2 <- Sys.glob("d:/work/gddmr_tex/*.tex")     # absoluter Pfad
> head(texFiles2, n=2)
[1] "d:/work/gddmr_tex/r_long_tex/gddmr.tex"
[2] "d:/work/gddmr_tex/r_long_tex/r1_00_preface.tex"
```

3.2.2 Dateipfade manipulieren

In der vollständigen Pfadangabe zu einer Datei steht einerseits die Information zum Ordner, der die Datei enthält, andererseits die Information zum Dateinamen sowie ggf. ihrer Endung. Diese Informationen lassen sich getrennt aus einem vollständigen Pfad extrahieren.

So gibt `basename("⟨Pfad⟩")` nur den Dateinamen aus, entfernt also die Angaben zum Ordner aus der übergebenen Pfadangabe. Als Komplement nennt `dirname("⟨Pfad⟩")` den Pfad zum Ordner, der eine Datei enthält, entfernt also den Dateinamen. Nur die Dateiendung (ohne Ordner und Dateinamen) erhält man mit `file_ext("⟨Pfad⟩")` aus dem im Basisumfang von R enthaltenen Paket `tools`. Als Komplement gibt `file_path_sans_ext("⟨Pfad⟩")` aus demselben Paket den vollständigen Pfad mit Dateinamen, aber ohne Endung zurück.

```
> basename("c:/path/to/file.txt")
[1] "file.txt"

> dirname("c:/path/to/file.txt")
[1] "c:/path/to"

> library(tools)      # für file_ext(), file_path_sans_ext()
> file_ext("c:/path/to/file.txt")
[1] "txt"

> file_path_sans_ext("c:/path/to/file.txt")
[1] "c:/path/to/file"
```

3.2.3 Dateien verändern

Die meisten Dateioperationen, die man interaktiv mit einem Dateimanager vornimmt, können mit den folgenden Funktionen automatisiert werden:

- `dir.exists("⟨Ordnerpfad⟩")` prüft, ob der Ordner mit dem angegebenen Pfad bereits existiert und tatsächlich ein Ordner ist.
- `file.exists("⟨Dateipfad⟩")` prüft, ob die Datei mit dem angegebenen Pfad bereits existiert.
- `file.remove("⟨Dateipfad⟩")` löscht die Datei mit dem angegebenen Pfad.
- `file.copy(from="⟨Dateipfad⟩", to="⟨Dateipfad⟩", overwrite=FALSE)` erstellt die Kopie einer Datei `from` an der Stelle `to`, wobei dort schon unter demselben Namen existierende Dateien nur mit `overwrite=TRUE` überschrieben werden.
- `file.rename(from="⟨Dateipfad⟩", to="⟨Dateipfad⟩")` verschiebt eine Datei `from` an die Stelle `to`.
- `file.create("⟨Dateipfad⟩")` erstellt eine leere Datei mit dem übergebenen Pfad.
- `dir.create("⟨Ordnerpfad⟩")` erstellt einen Ordner mit dem übergebenen Pfad.

Mit dem Rückgabewert TRUE oder FALSE melden die Funktionen, ob eine Dateioperation erfolgreich durchgeführt werden konnte.

```
# neues Verzeichnis newDir relativ zum aktuellen Arbeitsverzeichnis
> dir.create("newDir")
> file.create("newDir/newFile.txt")    # neue Datei in newDir
[1] TRUE

# Kopie der neu erstellten Datei
> file.copy("newDir/newFile.txt", to="newDir/fileA.txt")
[1] TRUE

# benenne Kopie um
> file.rename("newDir/fileA.txt", to="newDir/fileB.txt")
[1] TRUE

> file.remove("newDir/newFile.txt")    # entferne ursprüngliche Datei
[1] TRUE

> file.exists("newDir/newFile.txt")    # existiert ursprüngliche Datei?
[1] FALSE

> file.exists("newDir/fileB.txt")      # existiert umbenannte Kopie?
[1] TRUE
```

3.3 Datensätze aufbereiten und aggregieren

Bevor Datensätze analysiert werden können, müssen sie häufig in eine andere als ihre ursprüngliche Form gebracht werden – etwa um neue Variablen zu bilden, Beobachtungen bzw. Variablen auszuwählen sowie unterschiedliche Datensätze zusammenzuführen. Der folgende Abschnitt demonstriert diese Auswertungsschritte mit dem Basisumfang von R, während der weitgehend parallel aufgebaute Abschn. 3.4 dazu das beliebte Zusatzpaket `dplyr` verwendet. Als Datengrundlage dient der in Abschn. 2.11 erstellte Datensatz `myDf1`.

```
> head(myDf1)
  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
```

3.3.1 Verkettung von Befehlen mit der pipe

Daten aufzubereiten erfordert es gewöhnlich, mehrere Verarbeitungsschritte hintereinander durchzuführen. Für die dabei notwendige Verkettung von Funktionsaufrufen existieren im Wesentlichen zwei Möglichkeiten. Die bisher in diesem Buch verwendete Syntax ist die in R konventionelle. Sie besteht darin, den in einem Schritt veränderten Datensatz als Zwischenergebnis zu speichern, um diesen dann im nächsten Schritt zu verwenden.

```
# konventionelle Syntax mit temporären Objekten
> myDf2 <- setNames(myDf1, # benenne Variablen um
+                      c("ID", "SEX", "GROUP", "AGE", "IQ", "RATING"))

> myDf3 <- transform(myDf2, IQ_Z=scale(IQ), # fasse Gruppen zusammen
+                      group2=forcats::fct_collapse(GROUP, CG_WL=c("CG", "WL")))

> myDf4 <- subset(myDf3, group2 == "CG_WL") # Teilmenge Beobachtungen
> sort_by(myDf4, ~ RATING) # sortiere nach Rating
ID SEX GROUP AGE IQ RATING      IQ_Z group2
12 12   f     CG  30  88      1 -0.7389921  CG_WL
1   1   m     WL  25  93      2 -0.4884863  CG_WL
8   8   m     CG  20 141      2  1.9163693  CG_WL
2   2   f     WL  32  81      3 -1.0897002  CG_WL # Ausgabe gekürzt ...
```

Durch das *pipe* Symbol `|>` ist es alternativ möglich, Funktionen direkt miteinander zu verketteten. Die pipe bewirkt, dass das Ergebnis des vor ihr stehenden Befehls automatisch als erstes Argument des nach ihr stehenden Funktionsaufrufs eingefügt wird. `x |> f(y)` ist also gleichwertig zu `f(x, y)` und `x |> f() |> g()` zu `g(f(x))`. Über pipes „fließt“ der Datensatz von einem Verarbeitungsschritt zum nächsten analog zu einem Produkt auf einer Fertigungsstraße. Beim Lesen von Befehlen kann das `|>` Symbol gedanklich als „und dann“ mitgesprochen werden.

Typischerweise beginnt eine durch pipes verbundene Abfolge von Befehlen mit dem Datensatz. Jeder durch `|>` verbundene Arbeitsschritt folgt dann in einer neuen Zeile – notwendig ist dies aber nicht. In den durch `|>` verbundenen Funktionsaufrufen entfällt die explizite Nennung des zu verarbeitenden Datensatzes als eigentlich erstem Argument. Mit pipes verbundene Arbeitsschritte verändern den Datensatz, ohne diese Änderungen zu speichern. Dafür muss das Ergebnis explizit einem Objekt zugewiesen werden. Es ist üblich, dass diese Zuweisung am Beginn der Befehlskette steht. Eine Sequenz von miteinander verketteten Auswertungsschritten könnte etwa so aussehen:

```
# beginne mit Datensatz myDf1, transformiere ihn in Befehlskette
# und weise das letztlich erstellte Ergebnis einem neuen Objekt zu
> df_mod <- myDf1 |>
+   setNames(c("ID", "SEX", "GROUP", "AGE", "IQ", "RATING")) |>
+   transform(IQ_Z=scale(IQ), group2=forcats::fct_collapse(GROUP, CG_WL=c("CG", "WL"))) |>
+   subset(group2 == "CG_WL") |>
+   sort_by(~ RATING)
```

```
> head(df_mod, n=4)
  ID SEX GROUP AGE  IQ RATING      IQ_Z group2
12 12   f     CG  30  88       1 -0.7389921  CG_WL
1  1   m     WL  25  93       2 -0.4884863  CG_WL
8  8   m     CG  20 141       2  1.9163693  CG_WL
2  2   f     WL  32  81       3 -1.0897002  CG_WL
```

Bei Funktionen aus dem Basisumfang von R steht der Datensatz in der Liste der Argumente häufig nicht an erster Position. In ihrem Aufruf kann der implizit durch die pipe übergebene Datensatz deshalb mit dem Symbol `_` eingefügt werden.

```
> myDf1 |> lm(IQ ~ age + rating, data=_)
Coefficients:
(Intercept)      age      rating
80.2834    0.9912   -3.5676
```

Die folgenden Abschnitte mit Funktionen des Basisumfangs von R verwenden weiter die konventionelle Syntax ohne pipe. Die analoge Datenaufbereitung mit Funktionen des Pakets `dplyr` (Abschn. 3.4) erfolgt dagegen mit für das Paket kanonischen pipe-Syntax.

3.3.2 Variablen umbenennen

Ob eine bestimmte Variable in einem Datensatz enthalten ist, gibt `hasName(<Datensatz>, <'Name'>)` aus. Variablen lassen sich mit der Funktion `setNames(<Datensatz>, c('<Name1>', ...))` umbenennen. Sie setzt die Variablennamen des übergebenen Datensatzes auf die Elemente des als zweites Argument genannten Vektors von Zeichenketten. Die Funktion verändert den Datensatz selbst nicht, sondern liefert einen modifizierten Datensatz zurück.

```
> (names1 <- names(myDf1))                      # vorhandene Variablen
[1] "id" "sex" "group" "age" "IQ" "rating"

> hasName(myDf1, "ID")                          # Variable ID enthalten?
[1] FALSE

> names1[3] <- "fac"                            # Variable 3 umbenennen
> df_mod1 <- setNames(myDf1, names1)
> head(df_mod1, 3)
  id sex fac age  IQ rating
1  1   m  WL  25  93      2
2  2   f  WL  32  81      3
3  3   f   T  28 116      4

> names1[names1 == "fac"] <- "cond"            # fac nach cond umbenennen
> df_mod2 <- setNames(myDf1, names1)
> head(df_mod2, 3)
  id sex cond age  IQ rating
```

```

1 1 m WL 25 93      2
2 2 f WL 32 81      3
3 3 f T 28 116     4

# Variablennamen in Großbuchstaben ändern
> df_mod3 <- setNames(myDf1, toupper(names1))
> head(df_mod3, n=3)
  ID SEX COND AGE  IQ RATING
1  1   m   WL  25  93      2
2  2   f   WL  32  81      3
3  3   f     T  28 116     4

```

3.3.3 Teilmengen von Daten auswählen

Bei der Analyse eines Datensatzes möchte man häufig eine Auswahl der Daten treffen, etwa nur Personen aus einer bestimmten Gruppe untersuchen, nur Beobachtungsobjekte berücksichtigen, die einen bestimmten Testwert überschreiten, oder aber die Auswertung auf eine Teilgruppe von Variablen beschränken. Die Auswahl von Variablen auf der einen und Beobachtungsobjekten auf der anderen Seite unterscheidet sich dabei konzeptuell nicht voneinander.

Die Funktion `subset()` bietet eine Alternative zur Auswahl von Beobachtungen und Variablen mit den ebenfalls möglichen Methoden zur Indizierung eines Datensatzes (Abschn. 2.11.2), die aber weniger übersichtlich sind. `subset()` gibt einen Datensatz mit der gewünschten Teilmenge von Variablen und Beobachtungen zurück.

```

subset(x=<Datensatz>,
       subset=<Auswahl Zeilen>,
       select=<Auswahl Spalten>)

```

Variablen auswählen

Das Argument `select` dient dazu, eine Teilmenge der Spalten auszuwählen, wofür ein Vektor mit auszugebenden Spaltenindizes oder -namen benötigt wird. Dabei müssen Variablennamen ausnahmsweise nicht in Anführungszeichen stehen.¹⁴ Fehlt ein Vektor für `select`, werden alle Spalten ausgegeben.

```

> subset(myDf1, select=c(group, IQ))    # Variablen group und IQ
  group  IQ
1      T 112
2     CG 122
3     CG  95
4      T 102                                # Ausgabe gekürzt ...

```

¹⁴Dafür kommt die Technik des *non-standard evaluation* zum Einsatz, die Wickham (2019a, Kap. 19, 20) erläutert.

Um alle bis auf einige Variablen auszugeben, eignet sich ein negatives Vorzeichen vor dem numerischen Index oder vor dem Namen der wegzulassenden Variablen.

```
> subset(myDf1, select=c(-sex, -IQ))    # ohne sex und IQ
   id group age rating
1   1     T  26      1
2   2    CG  30      3
3   3    CG  25      5
                                # Ausgabe gekürzt ...
```

Alle Variablen, deren Namen einem bestimmten Muster entsprechen, können mit den in Abschn. 2.14.4 vorgestellten Funktionen zur Suche nach Zeichenfolgen identifiziert und dann über einen Indexvektor ausgewählt werden.

```
# Variablen, deren Name mit i / I beginnt und weitere Zeichen enthält
> (colIdx <- grep("^i.+", names(myDf1), ignore.case=TRUE))
[1] 1 5

> subset(myDf1, select=colIdx)
   id  IQ
1   1 112
2   2 122
3   3  95
                                # Ausgabe gekürzt ...
```

Analog ist vorzugehen, um Variablen eines bestimmten Datentyps auszuwählen, hier alle numerischen Variablen.

```
> (colIdx_num <- vapply(myDf1, is.numeric, logical(1)))
   id   sex  group   age   IQ  rating
TRUE FALSE FALSE TRUE  TRUE    TRUE

> subset(myDf1, select=colIdx_num)
   id age  IQ rating
1   1 21 111      5
2   2 18 115      3
3   3 33 104      3
4   4 31  92      3
                                # Ausgabe gekürzt ...
```

Beobachtungen auswählen

Das Argument `subset` ist ein Indexvektor zum Indizieren der Zeilen des Datensatzes `x`. Dieser Vektor ergibt sich häufig aus einem logischen Vergleich der Werte einer Variable mit einem Kriterium.¹⁵ Die Variablennamen des Datensatzes sind innerhalb von `subset()` bekannt, ihnen muss also im Ausdruck zur Bildung des Indexvektors nicht `<Datensatz>$` vorangestellt werden. Fehlt ein Vektor für `subset`, werden alle Zeilen ausgegeben.

¹⁵Fehlende Werte behandelt `subset()` als `FALSE`, sie müssen also nicht extra vom logischen Indexvektor ausgeschlossen werden. Um die Stufen der Faktoren auf die in der Auswahl noch tatsächlich vorhandenen Ausprägungen zu reduzieren, ist `droplevels(<Datensatz>)` zu verwenden (Abschn. 2.6.3).

```
> subset(myDf1, sex == "f")                      # Daten weibliche Personen
   id sex group age  IQ rating
1   1   f     T  26 112      1
6   6   f     CG  24 113      0
12 12  f     T  21  98      1

> subset(myDf1, id == rating)                   # Wert id = Wert rating
   id sex group age  IQ rating
1   1   f     T  26 112      1
```

Auch die Auswahl nach mehreren Kriterien gleichzeitig ist möglich, indem der Indexvektor durch entsprechend erweiterte logische Ausdrücke gebildet wird (Abschn. 2.2.1). Dabei 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 Personen mit einem Rating größer als 2
> subset(myDf1, (sex == "m") & (rating > 2))
   id sex group age  IQ rating
2   2   m     CG  30 122      3
3   3   m     CG  25  95      5
4   4   m     T   34 102      5          # Ausgabe gekürzt ...

# alle Personen mit einem eher hohen ODER eher niedrigen IQ-Wert
> subset(myDf1, (IQ < 90) | (IQ > 110))
   id sex group age  IQ rating
1   1   f     T  26 112      1
2   2   m     CG  30 122      3
5   5   m     WL  22  82      2          # Ausgabe gekürzt ...
```

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 Operator `⟨Menge1⟩ %in% ⟨Menge2⟩` als Kurzform von `is.element()` 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.

```
# Personen aus Wartelisten- ODER Kontrollgruppe
> subset(myDf1, group %in% c("CG", "WL"))
   id sex group age  IQ rating
2   2   m     CG  30 122      3
3   3   m     CG  25  95      5
5   5   m     WL  22  82      2
6   6   f     CG  24 113      0          # Ausgabe gekürzt ...
```

3.3.4 Doppelte und fehlende Werte ausschließen

Doppelte Werte können in Datensätzen etwa auftreten, nachdem sich teilweise überschneidende Daten aus unterschiedlichen Quellen in einem Datensatz integriert wurden. Alle später auftretenden Duplikate mehrfach vorhandener Zeilen werden durch `duplicated(<Datensatz>)` identifiziert und durch `unique(<Datensatz>)` ausgeschlossen (Abschn. 2.3.1). Lediglich die jeweils erste Ausfertigung bleibt so erhalten.

```
# Datensatz mit doppelten Werten herstellen
> myDfDouble <- rbind(myDf1, myDf1[sample(seq_len(nrow(myDf1)), 4), ])

# doppelte Zeilen identifizieren (alle Ausfertigungen)
> duplicated(myDfDouble) | duplicated(myDfDouble, fromLast=TRUE)
[1] FALSE FALSE FALSE TRUE FALSE TRUE FALSE TRUE FALSE
[12] FALSE TRUE TRUE TRUE TRUE

> myDfUnique <- unique(myDfDouble)      # doppelte Zeilen ausschließen
> any(duplicated(myDfUnique))           # Kontrolle: noch doppelte?
[1] FALSE
```

Fehlende Werte werden in Datensätzen weitgehend wie in Matrizen behandelt (Abschn. 2.13.4). Auch hier müssen also `is.na()` und `anyNA()` benutzt werden, um das Vorhandensein fehlender Werte zu prüfen.¹⁶

```
> myDfNA          <- myDf1          # Kopie
> myDfNA$IQ[2]    <- NA            # auf missing setzen
> myDfNA$rating[3] <- NA            # auf missing setzen
> is.na(myDfNA)[1:3, ]
   id  sex group age   IQ rating
1 FALSE FALSE FALSE FALSE FALSE FALSE
2 FALSE FALSE FALSE FALSE TRUE  FALSE
3 FALSE FALSE FALSE FALSE FALSE TRUE

# prüfe jede Variable, ob sie mindestens ein NA enthält
> apply(myDfNA, 2, anyNA)
   id  sex group age   IQ rating
FALSE FALSE FALSE FALSE TRUE  TRUE
```

Selbst definierte Funktionen (Abschn. 17.3) helfen in Kombination mit `apply()` dabei, pro Variable die Anzahl fehlender Werte bzw. die Anzahl nicht fehlender Beobachtungen zu zählen.

```
# Anzahl fehlender Werte je Variable
> apply(myDfNA, 2, function(x) { sum(is.na(x)) })
   id  sex group age   IQ rating
0     0     0     0     1     1
```

¹⁶Das Paket `naniar` (Tierney, Cook, McBain & Fay, 2020) hilft dabei, das Ausmaß fehlender Werte und ihr Verteilungsmuster in den Variablen eines Datensatzes durch verschiedene Diagramme zu analysieren.

```
# Anzahl nicht fehlender Werte je Variable
> apply(myDfNA, 2, function(x) { length(na.omit(x)) })
  id  sex group age   IQ rating
12  12    12   12    11    11
```

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. Die vollständigen Fälle, oder die Fälle mit fehlenden Werten können mit `subset()` ausgegeben werden. Für die Zeilen mit fehlenden Werten ist dabei der Indexvektor aus der logischen Negation des Ergebnisses von `complete.cases()` zu bilden.

```
> complete.cases(myDfNA)
[1] TRUE FALSE FALSE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

```
# zähle vollständige und unvollständige Zeilen
> table(complete.cases(myDfNA))
FALSE  TRUE
2     10
```

```
> subset(myDfNA, !complete.cases(myDfNA)) # Zeilen mit fehlendem Wert
  id sex group age   IQ rating
2  2   m    CG   30  NA    3
3  3   m    CG   25  95    NA
```

Die Variante `complete.cases(<Var1>, <Var2>, ...)` berücksichtigt für die Prüfung, ob für ein Beobachtungsobjekt alle Werte vorliegen, nur die durch Komma getrennt übergebenen Variablen.

```
# Index aller Beobachtungen mit vorliegendem Wert für IQ
> idx_cmpl_IQ <- with(myDfNA, complete.cases(IQ))
> subset(myDfNA, idx_cmpl_IQ)
  id sex group age   IQ rating
1  1   f      T  21  111     5
3  3   m      T  33  104    NA
4  4   m    CG  31   92     3           # Ausgabe gekürzt ...
```

Weiterhin können wie bei Matrizen alle Zeilen mit `na.omit()` gelöscht werden, in denen Werte fehlen.

```
> head(na.omit(myDfNA), n=4)                      # nur vollständige Zeilen
  id sex group age   IQ rating
1  1   f      T  26  112     1
4  4   m      T  34  102     5
5  5   m    WL  22   82     2
6  6   f    CG  24  113     0
```

3.3.5 Variablen entfernen, hinzufügen und transformieren

Variablen eines Datensatzes werden gelöscht, indem ihnen die leere Menge `NULL` zugewiesen wird – im Fall mehrerer Variablen gleichzeitig in Form einer Liste mit der Komponente `NULL`.

```
> dfTemp      <- myDf1          # Kopie erstellen
> dfTemp$group <- NULL         # eine Variable löschen
> head(dfTemp, n=3)
  id  sex  age   IQ rating
1  1    w   26  112      1
2  2    m   30  122      3
3  3    m   25   95      5

> dfTemp[c("sex", "IQ")] <- list(NULL) # mehrere Variablen löschen
> head(dfTemp, n=3)
  id  age rating
1  1    26      1
2  2    30      3
3  3    25      5
```

Wie bei Listen (Abschn. 2.10.1) können einem bestehenden Datensatz neue Variablen mit den Operatoren `<Datensatz>$<neue Variable>` und `<Datensatz>["<neue Variable>"]` hinzugefügt werden. Analog zum Vorgehen bei Matrizen kann an einen Datensatz auch mit `cbind(<Datensatz>, <Vektor>, ...)` eine weitere Variable passender Länge als Spalte angehängt werden.¹⁷

Im Beispiel soll der Beziehungsstatus der Personen dem Datensatz hinzugefügt werden.

```
> married <- sample(c(TRUE, FALSE), nrow(myDf1), replace=TRUE)
> myDf2  <- myDf1          # erstelle Kopie
> myDf2$married1  <- married        # Möglichkeit 1
> myDf2["married2"] <- married        # Möglichkeit 2
> myDf3 <- cbind(myDf1, married)      # Möglichkeit 3
> head(myDf3, n=3)
  id  sex group  age   IQ rating married
1  1    f     T   26  112      1    TRUE
2  2    m    CG   30  122      3    TRUE
3  3    m    CG   25   95      5   FALSE
```

Alternativ kann auch die Funktion `transform()` Verwendung finden. Sie überschreibt einen Datensatz nicht, sondern gibt einen veränderten Datensatz zurück, der einem neuen Objekt zugewiesen werden kann.

```
transform(<Datensatz>, <Variablenname1>=<Ausdruck1>, ...)
```

¹⁷Dagegen 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.

Unter **<Ausdruck>** ist anzugeben, wie sich die Werte der neuen Variable ergeben, die unter **<\-Variablenname>** gespeichert und an **<Datensatz>** angehängt wird. Es können mehrere solcher Zuweisungen durch Komma getrennt vorgenommen werden. Die Variablennamen des Datensatzes sind zur Verwendung innerhalb von **<Ausdruck>** bekannt. Trägt man links des **=** den Namen einer schon bestehenden Variable ein, wird diese überschrieben.

Im Beispiel soll das Quadrat des Ratings angefügt und zwei Gruppen anhand des IQ gebildet werden.

```
> myDf4 <- transform(myDf1,
+                      rSq=rating^2,
+                      IQgrp=cut(IQ, breaks=c(0, 100, Inf)))

> head(myDf4, n=3)
  id sex group age  IQ rating married rSq      IQgrp
1  1   f     T  26 112       1 FALSE    1 (100,Inf]
2  2   m    CG  30 122       3 FALSE    9 (100,Inf]
3  3   m    CG  25  95       5 FALSE   25  (0,100]
```

3.3.6 Datensätze sortieren

Datensätze können im Prinzip wie Matrizen mit **order()** über den Zwischenschritt eines erstellten Indexvektors sortiert werden (Abschn. 2.8.6). Einfacher ist dies jedoch mit **sort_by()**.

```
sort_by(<Datensatz>, <Modellformel>, na.last=TRUE, decreasing=FALSE)
```

Unter **<Datensatz>** ist der Datensatz anzugeben, der in eine bestimmte Reihenfolge zu bringen ist. Im typischen Fall, dass die Sortierkriterien Variablen desselben Datensatzes sind, lassen sie sich als Modellformel **~ <Kriterium1>** übergeben (Abschn. 5.2). **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 Stufen bestimmt, die nicht deren alphabetischer Reihenfolge entsprechen muss (Abschn. 2.6.5).

Soll nach zwei Kriterien sortiert werden, weil die Reihenfolge durch eine Variable noch nicht vollständig festgelegt ist, können weitere Variablen in der Modellformel mit **+** angeschlossen werden.

```
> sort_by(myDf1, ~ rating)           # sortiere nach rating
  id sex group age  IQ rating
11 11   m     T  26 115       1
12 12   f    CG  30  88       1
1  1   m    WL  25  93       2
8   8   m    CG  20 141       2           # Ausgabe gekürzt ...

> sort_by(myDf1, ~ group + rating)  # sortiere nach group und rating
  id sex group age  IQ rating
```

```
12 12   f    CG  30  88      1
8   8   m    CG  20 141      2
5   5   m    CG  28  85      3
10 10   f    CG  35  71      3      # Ausgabe gekürzt ...
```

3.3.7 Datensätze aufteilen

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. Sollen die Zeilen des Datensatzes in Gruppen aufgeteilt werden, die sich aus der Kombination der Stufen zweier Faktoren ergeben, sind diese in eine Liste einzuschließen. In diesem Fall sorgt das Argument `drop=TRUE` dafür, dass nur jene Stufen-Kombinationen berücksichtigt werden, für die auch Beobachtungen vorhanden sind.

Wenn `f` Teil des Datensatzes ist, bietet sich die alternative Syntax mittels Modellformel `split(<Datensatz>, ~ <Faktor>)` bzw. bei mehreren Faktoren `split(<Datensatz>, ~ <F1> + <F2> + ...)` an.

Das Ergebnis ist eine Liste, die für jede Faktorstufe eine Komponente in Form eines Datensatzes besitzt. Diese Datensätze können etwa dazu dienen, Auswertungen getrennt nach Gruppen vorzunehmen (Abschn. 3.3.12).

```
> split(myDf1, myDf1$group)          # oder: split(myDf1, ~ group)
$CG
  id sex group age IQ rating
2  2   m    CG  30 122     3
3  3   m    CG  25  95     5
6  6   f    CG  24 113     0
11 11  m    CG  20  92     1

$T
  id sex group age IQ rating
1  1   f     T  26 112     1
4  4   m     T  34 102     5
7  7   m     T  28  92     3
12 12  f     T  21  98     1

$WL
  id sex group age IQ rating
5  5   m    WL  22  82     2
8  8   m    WL  35  90     2
9  9   m    WL  23  88     3
10 10  m    WL  29  81     5
```

```
# teile Beobachtungen nach sex und group auf
> split(myDf1, ~ sex + group)      # Ausgabe gekürzt ...
```

`split()` akzeptiert auch einen Vektor `x`, dessen Elemente entsprechend der im Faktor `f` definierten Gruppenzugehörigkeit aufgeteilt werden. Das Ergebnis ist wieder eine Liste mit einer Komponente je Faktorstufe, wobei jede Komponente ein Vektor ist. Um eine Matrix `x` analog zu einem Datensatz bzgl. ihrer Zeilen aufzuteilen, muss man explizit die Methode `split.data.frame()` aufrufen (Abschn. 17.3.7). In der erzeugten Liste ist dann jede Komponente eine Matrix.

3.3.8 Datensätze zeilen- oder spaltenweise verbinden

Wenn zwei oder mehr Datensätze `df` vorliegen, die hinsichtlich ihrer Variablen identisch sind, so fügt die Funktion `rbind(<df1>, <df2>, ...)` die Datensätze analog zum Vorgehen bei Matrizen zusammen, indem sie sie untereinander anordnet. Auf diese Weise könnten z. B. die Daten mehrerer Teilstichproben kombiniert werden, an denen dieselben Variablen erhoben wurden.¹⁸ Die Reihenfolge der Variablen muss in den Datensätzen nicht übereinstimmen. Wenn die Datensätze Gruppierungsfaktoren enthalten, ist zunächst sicherzustellen, dass alle dieselben Faktorstufen in derselben Reihenfolge besitzen (Abschn. 2.6.3, 2.6.5).

```
> (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
11 11   m     CG  20  92      1
12 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 hinsichtlich der Beobachtungsobjekte gibt (Abschn. 3.3.4).

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)` so kombiniert werden, dass die Variablen nebeneinander angeordnet sind. Dabei ist sicherzustellen, dass die zu demselben Beobachtungsobjekt gehörenden Daten in derselben Zeile jedes Datensatzes stehen.

¹⁸`bind_rows()` aus dem Paket `dplyr` kann auch Datensätze miteinander verbinden, die sich bzgl. der Variablen unterscheiden (Abschn. 3.4.7).

3.3.9 Datensätze zusammenführen

`merge()` kann flexibel zwei Datensätze zusammenführen, die sich nur teilweise in den Variablen oder in den Beobachtungsobjekten entsprechen. Auf diese Weise lassen sich einander ergänzende Informationen aus verschiedenen Datenquellen integrieren.

```
merge(x=<Datensatz 1>, y=<Datensatz 2>,
      by.x, by.y, by, all.x, all.y, all)
```

Zunächst sei die Situation betrachtet, dass die unter `x` und `y` angegebenen Datensätze Daten derselben Beobachtungsobjekte beinhalten, die über eine eindeutige ID identifiziert sind.¹⁹ Dabei sollen einige Variablen bereits in beiden, andere Variablen hingegen nur in jeweils einem der beiden Datensätze vorhanden sein. Die in beiden Datensätzen gleichzeitig vorhandenen Variablen enthalten dann dieselbe Information, 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.²⁰

Beispiel sei ein Datensatz mit je zwei Messungen eines Merkmals an drei Personen mit eindeutiger ID. In einem zweiten Datensatz sind für jede ID die Informationen festgehalten, die konstant über die Messwiederholungen sind – hier das Geschlecht und eine Gruppenzugehörigkeit.²¹

```
# Datensatz mit 2 Messwerten pro ID
> (IDDV <- data.frame(ID=factor(rep(1:3, each=2)),
+                         DV=round(rnorm(6, mean=100, sd=15))))
```

ID	DV
1	93
2	105
3	112
4	101
5	109
6	101


```
# Datensatz mit Informationen, die pro ID konstant sind
> (IV <- data.frame(ID=factor(1:3),
+                         IV=factor(c("A", "B", "A")),
+                         sex=factor(c("f", "f", "m"))))
```

ID	IV	sex
1	A	f
2	B	f
3	A	m

¹⁹ Dabei zu beachtende Aspekte der Datenqualität bespricht Abschnitt 4.3. Hinweise für den Fall uneindeutiger IDs geben Abschn. 2.14, Fußnote 61 und Abschn. 4.3, Fußnote 11.

²⁰ Zur Identifizierung gleicher Variablen werden die Spaltennamen mittels `intersect(names(x), names(y))` herangezogen. Bei Gruppierungsfaktoren ist es wichtig, dass sie in beiden Datensätzen dieselben Stufen in derselben Reihenfolge besitzen.

²¹ Diese Datenstruktur entspricht einer *normalisierten* Datenbank mit mehreren *tables* zur Vermeidung von Redundanzen.

Als Ziel soll ein Datensatz erstellt werden, der die pro Person konstanten und veränderlichen Variablen integriert, wobei die Zuordnung von Informationen über die ID geschieht. Die pro Person über die Messwiederholungen konstanten Werte werden dabei von `merge()` automatisch passend oft wiederholt.

```
> merge(IDDV, IV)
   ID DV IV sex
1  1  93  A   f
2  1 105  A   f
3  2 112  B   f
4  2 101  B   f
5  3 109  A   m
6  3 101  A   m
```

Verfügen x und y im Prinzip über teilweise dieselben Variablen, jedoch mit abweichender Bezeichnung, kann über die Argumente `by.x` und `by.y` manuell festgelegt werden, welche ihrer Variablen trotz ungleichen Namens übereinstimmen und deshalb 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. Beide Argumente `by.x` und `by.y` müssen gleichzeitig verwendet werden und müssen dieselbe Anzahl von gleichen Spalten bezeichnen. Haben beide Datensätze dieselben Variablennamen, kann auch auf das Argument `by` zurückgegriffen werden, das sich dann auf die Spalten beider Datensätze gleichzeitig bezieht. Es empfiehlt sich, `by` in jedem Aufruf von `merge()` explizit zu setzen, um nicht zum Abgleich versehentlich gleich benannte Spalten zu verwenden, die eigentlich andere Informationen tragen.

Im folgenden Beispiel besitzt die in beiden Datensätzen eigentlich übereinstimmende ID-Variable unterschiedliche Namen. Die Spalte der Initialen wird künstlich als nicht übereinstimmende Variable gekennzeichnet, indem sie nicht an `by.x` und `by.y` übergeben wird.

```
> (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

# anderer Name für ID, initials gleich, zusätzlich: IV2, DV2
> (dfb <- data.frame(ID_mod=1:4, initials=c("AB", "CD", "EF", "GH"),
+                      IV2=c("A", "B", "A", "B"), DV2=c(91,89,92,79)))
   ID_mod initials IV2 DV2
1       1        AB    A  91
2       2        CD    B  89
3       3        EF    A  92
4       4        GH    B  79

# initials nicht an by.x bzw. by.y übergeben -> doppelt übernommen
> merge(dfa, dfb, by.x="ID", by.y="ID_mod")
```

	ID	initials.x	IV1	DV1	initials.y	IV2	DV2		
1	1		AB	-	10		AB	A	91
2	2		CD	-	10		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 Zeilen, bei denen die in beiden Datensätzen vorkommenden Variablen identische Werte aufweisen – im Kontext von Datenbanken wird dieses Verhalten als *inner join* bezeichnet.²² Dabei werden alle Zeilen entfernt, deren Werte für die gemeinsamen Variablen zwischen den Datensätzen abweichen. Werden mit `by` Spalten als übereinstimmend 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` bzw. `all.y` auf `TRUE` gesetzt werden. `all.x` bewirkt dann, dass alle Zeilen in `x`, die auf den übereinstimmenden Variablen andere Werte als in `y` haben, ins Ergebnis aufgenommen werden (im Kontext von Datenbanken ein *left outer join*). Die in `y` (aber nicht in `x`) enthaltenen Variablen werden für diese Zeilen auf `NA` gesetzt. Für das Argument `all.y` gilt dies analog (bei Datenbanken ein *right outer join*). Das Argument `all=TRUE` steht kurz für `all.x=TRUE` in Kombination mit `all.y=TRUE` (bei Datenbanken ein *full outer join*).

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
```

²²Details zu verschiedenen *join* Operationen inkl. anschaulicher Visualisierungen geben Wickham und Grolemund (2023, Kap. 19): <http://r4ds.hadley.nz/joins>

Analoges gilt für das Einschließen der ersten beiden Zeilen von `dfA`. 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
```

3.3.10 Organisationsform einfacher Datensätze ändern

Bisweilen weicht der Aufbau eines Datensatzes von dem beschriebenen ab, etwa wenn sich Daten derselben, in verschiedenen Bedingungen erhobenen Zielgrößen in unterschiedlichen Spalten befinden, wobei eine Spalte mit einer Bedingung korrespondiert. `stack()` ändert die Organisationsform der Daten so, dass der Datensatz eine Zeile pro Beobachtung, eine Spalte für die Werte der Zielgröße in allen Bedingungen und eine Spalte mit dem Faktor enthält, dessen Stufen die zu jedem Wert gehörende Bedingung definieren.

```
stack(x=<Datensatz / Liste>, select=<Spalten>)
```

Unter `x` ist ein Datensatz oder eine Liste mit benannten Komponenten anzugeben. Bei einem Datensatz können die zu reorganisierenden Variablen über das Argument `select` ausgewählt werden. Es erwartet dafür einen Vektor mit Spaltenindizes oder Variablennamen. Das Ergebnis ist ein Datensatz, in dessen erster Variable `values` sich alle Werte der Zielgröße befinden. Diese Variable entsteht, indem die ursprünglichen Variablen von `x` durch `c(<Variable1>, <Variable2>, ...)` aneinander gehängt werden. Die zweite Variable `ind` des erzeugten Datensatzes ist ein Faktor, dessen Stufen codieren, aus welcher Variable von `x` ein einzelner Wert der Variable `values` stammt. Hierzu werden die Variablennamen von `x` herangezogen.

```
> vec1 <- sample(1:10, 3, replace=TRUE)
> vec2 <- sample(1:10, 2, replace=TRUE)
> vec3 <- sample(1:10, 1, replace=TRUE)
> (lTmp <- list(cond1=vec1, cond2=vec2, cond3=vec3))
$cond1
[1] 7 9 1

$cond2
[1] 6 5

$cond3
[1] 2

> (res <- stack(lTmp))
```

```

values    ind
1      7 cond1
2      9 cond1
3      1 cond1
4      6 cond2
5      5 cond2
6      2 cond3

> str(res)
'data.frame':   6 obs. of  2 variables:
 $ values: int  7 9 1 6 5 2
 $ ind    : Factor w/ 3 levels "cond1","cond2",...: 1 1 1 2 2 3

```

Die Funktion `unstack(⟨Datensatz⟩, form=⟨Modellformel⟩)` kehrt das Ergebnis von `stack()` um. Sie transformiert also einen Datensatz, der aus einer Variable mit den Werten der Zielgröße 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 Zielgröße. Das Ergebnis von `unstack()` ist nur dann ein Datensatz, wenn alle Faktorstufen gleich häufig vorkommen, die resultierenden Variablen also dieselbe Länge aufweisen. Andernfalls ist das Ergebnis eine Liste.

```

> unstack(res)
$cond1
[1] 7 9 1

$cond2
[1] 6 5

$cond3
[1] 2

```

Kommen im Datensatz `x` mehrere Zielgrößen und Faktoren vor, kann in `unstack()` über eine an das Argument `form` zu übergebende *Modellformel* $\text{AV} \sim \text{UV}$ festgelegt werden, welche Zielgröße (`AV`) ausgewählt und nach welchem Faktor (`UV`) die Trennung der Werte der Zielgröße vorgenommen werden soll (Abschn. 5.2).

```

# füge zwei neue Variablen zum Datensatz res hinzu
> Nj      <- 3
> res$IVnew <- factor(sample(rep(c("A", "B"), Nj), 2*Nj,
+                         replace=FALSE))

> res$DVnew <- sample(100:200, 2*Nj)
> unstack(res, form=DVnew ~ IVnew)
     A     B
1 183 193
2 129 115
3 142 140

```

Die Organisationsformen eines Datensatzes, zwischen denen `stack()` und `unstack()` wechseln, werden im Kontext von Daten aus Messwiederholungen als *Long*-Format und *Wide*-Format bezeichnet. Häufig sind die Datensätze dann jedoch zu komplex, um noch mit diesen Funktionen bearbeitet werden zu können. Die im folgenden Abschnitt beschriebene Funktion `reshape()` ist dann besser geeignet.

3.3.11 Organisationsform komplexer Datensätze ändern

Wurden an denselben Beobachtungsobjekten zu verschiedenen Messzeitpunkten Daten derselben Zielgröße erhoben, können die Werte jeweils eines Messzeitpunkts als zu einer eigenen Variable gehörend betrachtet werden. In einem Datensatz findet sich jede dieser Variablen dann in einer separaten Spalte. Diese Organisationsform folgt dem Prinzip, dass pro Zeile die Daten jeweils eines Beobachtungsobjekts aus verschiedenen Variablen stehen. Eine solche Struktur wird als Wide-Format bezeichnet, weil der Datensatz durch mehr Messzeitpunkte mehr Spalten hinzugewinnt, also breiter wird. Das Wide-Format entspricht einer multivariaten Betrachtungsweise von Daten aus Messwiederholungen (Abschn. 7.5.3, 12.6.4). Es setzt voraus, dass die Objekte zu denselben Zeitpunkten beobachtet wurden und damit der Messzeitpunkt pro Spalte konstant ist.

Für die univariat formulierte Analyse von abhängigen Daten (Abschn. 7.5) ist jedoch oft die Organisation im Long-Format notwendig. Die zu den verschiedenen Messzeitpunkten gehörenden Werte eines Beobachtungsobjekts stehen hier in separaten Zeilen. Auf diese Weise beinhalten mehrere Zeilen Daten desselben Beobachtungsobjekts. 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 welchem Objekt 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. Das Long-Format eignet sich auch für Situationen, in denen mehrere Objekte zu verschiedenen Zeitpunkten unterschiedlich häufig beobachtet wurden.

Vorgehen bei einem Messwiederholungsfaktor

Im Beispiel sei an vier Personen eine Zielgröße zu drei Messzeitpunkten erhoben worden. Bei zwei der Personen sei dies in Bedingung *A*, bei den anderen beiden in Bedingung *B* einer Intervention geschehen. Damit liegen zwei Gruppierungsfaktoren vor, zum einen als Intra-Gruppen Faktor der Messzeitpunkt, zum anderen ein Zwischen-Gruppen Faktor (*Split-Plot* Design, Abschn. 7.8). Zunächst soll demonstriert werden, wie sich das Long-Format manuell aus gegebenen Vektoren herstellen lässt. Soll mit einem solchen Datensatz etwa eine univariante Varianzanalyse mit Messwiederholung gerechnet werden, muss sowohl die Personen- bzw. Blockzugehörigkeit eines Messwertes als auch der Messzeitpunkt jeweils in einem Faktor gespeichert sein.

```
> Nj      <- 2                                # Gruppengröße
> P       <- 2                                # Anzahl Gruppen
> Q       <- 3                                # Anzahl Messzeitpunkte
> id      <- 1:(P*Nj)                          # Blockzugehörigkeit
```

```

> DV_t1 <- round(rnorm(P*Nj, -1, 1), 2) # Zielgröße zu t1
> DV_t2 <- round(rnorm(P*Nj, 0, 1), 2) # Zielgröße zu t2
> DV_t3 <- round(rnorm(P*Nj, 1, 1), 2) # Zielgröße zu t3
> IVbtw <- factor(rep(c("A", "B"), Nj)) # Gruppe: between

# Datensatz im Wide-Format
> (dfW1 <- data.frame(id, IVbtw, DV_t1, DV_t2, DV_t3))
   id IVbtw 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
> idL    <- factor(rep(id, Q))           # Faktor ID-Code
> DVl    <- c(DV_t1, DV_t2, DV_t3)       # gemeinsamer Vektor Zielgröße
> IVwth  <- factor(rep(1:3, each=P*Nj))  # Zeitpunkt: within
> IVbtwL <- rep(IVbtw, times=Q)          # Gruppe: between

# Datensatz im Long-Format
> dfL1a <- data.frame(id=idL, IVbtw=IVbtwL, IVwth=IVwth, DV=DVl)
> sort_by(dfL1a, ~ id)                   # sortierte Ausgabe
   id IVbtw IVwth    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

```

`reshape()` bietet die Möglichkeit, einen Datensatz ohne manuelle Zwischenschritte zwischen Wide- und Long-Format zu transformieren.²³

```
reshape(data=<Datensatz>, varying, timevar="time",
        idvar="id", direction=c("wide", "long"), v.names="<Name>")
```

- Zunächst wird der Datensatz unter `data` eingefügt. Um ihn vom Wide- ins Long-Format zu transformieren, muss das Argument `direction="long"` gesetzt werden.

²³Bei sehr großen Datensätzen kann es notwendig sein, auf die Funktionen `melt()` und `cast()` aus dem Paket `data.table` auszuweichen.

- Daneben ist unter **varying** anzugeben, welche Variablen im Wide-Format dieselbe Zielgröße zu unterschiedlichen Messzeitpunkten repräsentieren. Die Variablen werden im Long-Format über unterschiedliche Ausprägungen der neu gebildeten 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 Zielgröße gehören.²⁴
- Besitzt der Datensatz eine Variable, die codiert, von welcher Person ein Wert stammt, kann sie im Argument **idvar** genannt werden. Andernfalls wird eine solche Variable auf Basis der Zeilenindizes gebildet und trägt den Namen **id**. Auch andere Variablen von **data**, die pro Messzeitpunkt zwischen den Personen 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.
- Der Name der Variable im Long-Format mit den Werten der Zielgröße kann über das Argument **v.names** bestimmt werden.

```
> dfL1b <- reshape(dfWide, varying=c("DV_t1", "DV_t2", "DV_t3"),
+                     direction="long", idvar=c("id", "IVbtw"),
+                     v.names="DV")

# erste Zeilen der sortierten Ausgabe, identisch zu dfL1a
> head(sort_by(dfL1b, ~ id))
  id IVbtw time    DV
1.A.1 1      A    1 -1.68
1.A.2 1      A    2  0.78
1.A.3 1      A    3  2.73
2.B.1 2      B    1 -2.52
2.B.2 2      B    2  0.69
2.B.3 2      B    3  0.80
```

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 umgewandelt werden mit:

- `<Datensatz>$<Variable> <- factor(<Datensatz>$<Variable>)`

Ist der Datensatz vom Long- ins Wide-Format zu transformieren, muss **direction="wide"** gesetzt werden. Für das Argument **v.names** wird jene Variable genannt, die die Werte der Zielgröße 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 Zielgröße eines Objekts zuordnen bzw. pro Messzeitpunkt über die Objekte variieren, etwa weil sie die Ausprägung von Zwischen-Gruppen Faktoren darstellen.

²⁴Im Fall zweier Zielgrößen, für die jeweils eine Gruppe von zwei Spalten im Wide-Format vorhanden ist, könnte das Argument also `varying=list(c("DV1_t1", "DV1_t2"), c("DV2_t1", "DV2_t2"))` lauten.

```
> reshape(dfL1b, v.names="DV", timevar="IVwth",
+         idvar=c("id", "IVbtw"), direction="wide")
   id IVbtw   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
```

Vorgehen bei mehreren Messwiederholungsfaktoren

Ist eine Zielgröße an denselben Objekten mehrfach in den kombinierten Bedingungen zweier Intra-Gruppen Faktoren erhoben worden (Abschn. 7.7), ist das einfachste Vorgehen zum Wechsel vom Wide- ins Long-Format, zunächst beide Faktoren mit `interaction()` in einen einzigen Faktor umzuwandeln, der alle möglichen Stufenkombinationen enthält. Dann lässt sich wie oben beschrieben fortfahren. Alternativ ist `reshape()` zweimal anzuwenden. 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 Personen in jeder Stufenkombination eines Faktors mit drei und eines Faktors mit zwei Messzeitpunkten Werte einer Zielgröße erhoben worden.

```
> N      <- 4                                     # Anzahl Personen
> id     <- 1:N                                  # ID-Code
> t_11   <- round(rnorm(N,  8,  2), 2)          # Zielgröße zu t1-1
> t_12   <- round(rnorm(N, 10,  2), 2)          # Zielgröße zu t1-2
> t_21   <- round(rnorm(N, 13,  2), 2)          # Zielgröße zu t2-1
> t_22   <- round(rnorm(N, 15,  2), 2)          # Zielgröße zu t2-2
> t_31   <- round(rnorm(N, 13,  2), 2)          # Zielgröße zu t3-1
> t_32   <- round(rnorm(N, 15,  2), 2)          # Zielgröße zu t3-2

# Datensatz im Wide-Format
> (dfW2 <- data.frame(id, t_11, t_12, t_21, t_22, t_31, t_32))
   id t_11  t_12  t_21  t_22  t_31  t_32
1  1  9.24 11.89 11.03 13.54 13.60 17.02
2  2  7.17  9.13 12.36 15.40 13.02 12.90
3  3  6.07  8.41  8.34 16.20 14.49 14.19
4  4  6.21  6.99 10.40 14.94 15.35 17.60

# Transformation ins Long-Format bzgl. IV1
> (dfL2_IV1 <- reshape(dfW2, 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")))
   id IV1 IV2-1 IV2-2
1.1  1    1  9.24 11.89
2.1  2    1  7.17  9.13
3.1  3    1  6.07  8.41
```

```

4.1 4 1 6.21 6.99
1.2 1 2 11.03 13.54
2.2 2 2 12.36 15.40
3.2 3 2 8.34 16.20
4.2 4 2 10.40 14.94
1.3 1 3 13.60 17.02
2.3 2 3 13.02 12.90
3.3 3 3 14.49 14.19
4.3 4 3 15.35 17.60

```

Da IV1 nun wie id pro Messzeitpunkt von IV2 über die Personen variiert, muss die Variable im zweiten Schritt ebenfalls unter `idvar` genannt werden.

```

> dfL2_IV1_IV2 <- reshape(dfL2_IV1, varying=c("IV2-1", "IV2-2"),
+                           direction="long", timevar="IV2",
+                           idvar=c("id", "IV1"), v.names="DV")

> head(dfL2_IV1_IV2)
  id IV1 IV2      DV
1.1.1 1   1   1  9.24
2.1.1 2   1   1  7.17
3.1.1 3   1   1  6.07
4.1.1 4   1   1  6.21
1.2.1 1   2   1 11.03
2.2.1 2   2   1 12.36

```

Auch die umgekehrte Transformation vom Long- ins Wide-Format benötigt zwei Schritte, wenn zwei Intra-Gruppen Faktoren vorhanden sind.

```

# Schritt 1: Stufen der IV2 in Spalten aufteilen
> dfW2_IV2 <- reshape(dfL2_IV1_IV2, v.names="DV", timevar="IV2",
+                       idvar=c("id", "IV1"), direction="wide")

# Schritt 2: Stufen der IV1 in Spalten aufteilen
> (dfW2_IV1_IV2 <- reshape(dfW2_IV2, v.names=c("DV.1", "DV.2"),
+                           timevar="IV1", idvar="id", direction="wide"))
  id DV.1.1 DV.2.1 DV.1.2 DV.2.2 DV.1.3 DV.2.3
1.1.1 1   9.24  11.89  11.03  13.54  13.60  17.02
2.1.1 2   7.17  9.13   12.36  15.40  13.02  12.90
3.1.1 3   6.07  8.41   8.34   16.20  14.49  14.19
4.1.1 4   6.21  6.99   10.40  14.94  15.35  17.60

```

3.3.12 Daten getrennt nach Gruppen auswerten und aggregieren

Nachdem Datensätze mit `split()` geteilt wurden (Abschn. 3.3.7), liegen die Daten aus den Bedingungen separat vor und können getrennt verarbeitet werden, etwa zur Berechnung von Kennwerten pro Gruppe. Bei numerischen Kennwerten eignet sich `sapply()`, um dieselbe

Funktion auf jede Komponente einer Liste anzuwenden. Da jede Komponente der Liste ein Datensatz ist, muss selbst eine geeignete Funktion erstellt werden, die als Argument einen Datensatz akzeptiert und daraus den gewünschten Kennwert berechnet (Abschn. 17.3).

```
# berechne Mittelwert von IQ getrennt nach Gruppen
> dat_spl <- split(myDf1, ~ group))      # teile Datensatz in Gruppen
> sapply(dat_spl, function(x) { c(M=mean(x$IQ)) })
   CG.M      T.M    WL.M
105.50 101.00 85.25
```

Mit diesem Ansatz können auch gleichzeitig mehrere numerische Kennwerte pro Gruppe berechnet werden. Die einzelnen Kennwerte stehen in der ausgegebenen Matrix dann in den Zeilen, die zugehörige Gruppe ist durch die Spaltennamen gekennzeichnet. Da Ergebnistabellen oft so aufgebaut sind, dass eine Zeile pro Gruppe und eine Spalte pro Kennwert steht, ist das Ergebnis noch zu transponieren.

```
# berechne Mittelwert und Streuung von IQ getrennt nach Gruppen
> (m_sd <- sapply(dat_spl, function(x) {
+   c(M=mean(x$IQ), SD=sd(x$IQ)) }))
   CG          T          WL
M 105.50000 101.000000 85.250000
SD 14.38749  8.406347  4.425306

> t(m_sd)                                # transponiert
      M          SD
CG 105.50 14.387495
T 101.00 8.406347
WL 85.25 4.425306
```

Sollen die Gruppen durch Kombination der Stufen aus mehreren Faktoren gebildet werden, ist der Aufruf von `split()` entsprechend zu erweitern.

```
# teile Datensatz nach zwei Faktoren auf
> dat_spl2 <- split(myDf1, list(myDf1$sex, myDf1$group), drop=TRUE)
> sapply(dat_spl2, function(x) { c(M=mean(x$IQ), SD=sd(x$IQ)) }) # ...
```

Wenn getrennt nach Gruppenzugehörigkeit mehrere Kennwerte berechnet werden sollen, die nicht alle numerisch sind, kann auf `lapply()` ausgewichen und die erzeugte Liste manuell mit `do.call(rbind, <Liste>)` zu einem Datensatz verbunden werden.

```
# Funktion: Datensatz Gruppenzugehörigkeiten & gruppenweise Kennwerte
> aggr_fun <- function(x) {
+   data.frame(Sex=unique(x$sex),
+             Group=unique(x$group),
+             M=mean(x$IQ),
+             SD=sd(x$IQ))
+ }

> m_sdL <- lapply(dat_spl2, aggr_fun)           # wende Funktion an
```

```
> do.call(rbind, m_sdL) # verbinde erzeugte Liste zu Datensatz
   Sex Group      M       SD
f.CG   f    CG 113.00      NA
m.CG   m    CG 103.00 16.522712
f.T    f     T 105.00  9.899495
m.T    m     T  97.00  7.071068
m.WL   m    WL  85.25  4.425306
```

Um für Variablen eines Datensatzes Kennwerte nicht nur über alle Beobachtungen hinweg, sondern getrennt nach Gruppen zu berechnen, ist auch die Funktion `aggregate()` geeignet.

```
aggregate(<Modellformel>, FUN=<Funktion>, data=<Datensatz>)
```

Sie berechnet mit der Funktion FUN einen Kennwert für eine Variable $\langle AV \rangle$ getrennt nach Gruppen, die durch einen Faktor $\langle UV \rangle$ definiert werden. Beide Variablen müssen in einer Modellformel $\langle AV \rangle \sim \langle UV \rangle$ aufgeführt sein (Abschn. 5.2). Stammen die Variablen aus einem Datensatz, ist dieser unter data zu nennen.

Die Modellformel erweitert sich zu $\langle AV \rangle \sim \langle UV_1 \rangle + \langle UV_2 \rangle + \dots$, wenn die Gruppen durch Kombination mehrerer Faktoren gebildet werden sollen. Um denselben Kennwert jeweils von mehreren Variablen gleichzeitig zu berechnen, ist die Modellformel multivariat zu formulieren, d. h. links der \sim mit `cbind(<Variable1>, <Variable2>, ...)`.

```
# Mittelwert jeweils von age und IQ getrennt nach sex und group
> aggregate(cbind(age, IQ) ~ sex + group, FUN=mean, data=myDf1)
  sex group    age     IQ
1   f    CG 24.00 113.00
2   m    CG 25.00 103.00
3   f     T 23.50 105.00
4   m     T 31.00  97.00
5   m    WL 27.25  85.25
```

Den Kennwert über alle Beobachtungen hinweg erhält man, wenn rechts der Tilde nur ~ 1 steht.

```
# Gesamtmittelwert jeweils von age und IQ
> aggregate(cbind(age, IQ) ~ 1, FUN=mean, data=myDf1)
      age     IQ
1 26.41667 97.25
```

Um gleichzeitig mehrere für jede Gruppe getrennt berechnete Kennwerte in einem Datensatz zusammenzustellen ist ein zweistufiges Vorgehen möglich. Zunächst werden verschiedene Kennwerte pro Gruppe wie gezeigt einzeln berechnet. Anschließend lassen sich die erzeugten Datensätze mit `merge()` zusammenfügen. Inhaltlich passende Spaltennamen sind dabei mit dem Argument `suffixes` festzulegen.

```
# separate Datensätze für Mittelwert und Streuung je Gruppe
> d_mean <- aggregate(cbind(age, IQ) ~ sex+group, FUN=mean, data=myDf1)
> d_sd   <- aggregate(cbind(age, IQ) ~ sex+group, FUN=sd,   data=myDf1)
```

```
# verbinde Datensätze und benenne dabei Spalten passend
> merge(d_mean, d_sd, by=c("sex", "group"), suffixes = c(".M",".SD"))
   sex group age.M    IQ.M   age.SD    IQ.SD
1   f     CG 24.00 113.00      NA      NA
2   f     T 23.50 105.00 3.535534 9.899495
3   m     CG 25.00 103.00 5.000000 16.522712
4   m     T 31.00  97.00 4.242641 7.071068
5   m     WL 27.25  85.25 6.020797 4.425306
```

3.3.13 Funktionen auf Variablen anwenden

Der folgende Abschnitt stellt dar, wie allgemein Funktionen auf einzelne Variablen aus Datensätzen oder auf Gruppen von solchen Variablen angewendet werden können.

`lapply(X=<Liste>, FUN=<Funktion>, ...)` (*list apply*) verallgemeinert die Funktionsweise von `apply()` (Abschn. 2.8.8) auf Listen und Datensätze. X kann ein Vektor, eine Liste oder ein Datensatz sein. Bei einem Vektor wird die an FUN übergebene Funktion auf jedes Element angewendet, bei Listen dagegen auf jede Komponente bzw. im Fall eines Datensatzes auf jede Variable. Das Ergebnis ist eine Liste mit ebenso vielen Komponenten wie X Elemente bzw. Komponenten enthält. Ist FUN nur sinnvoll auf numerische Variablen anwendbar, können diese aus einem Datensatz zunächst mit `subset(..., select=<Indizes>)` extrahiert werden.

```
# Mittelwerte der numerischen Variablen
> numDf <- subset(myDf1, select=c(age, IQ, rating))
> (myL <- lapply(numDf, mean))
$age
[1] 26.41667

$IQ
[1] 97.25

$rating
[1] 2.583333
```

`sapply(<Vektor>, FUN=<Funktion>)` (*simplified apply*) arbeitet wie `lapply()`, gibt aber nach Möglichkeit 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 Zeilen aus diesen Werten gebildet sind. Die Rückgabewerte sollten dann alle denselben Datentyp besitzen²⁵

```
# range der numerischen Variablen
> sapply(numDf, range)
  age   IQ  rating
[1,] 20   82      0
[2,] 35  122      5
```

²⁵Die verwandte Funktion `vapply()` unterscheidet sich nur dadurch, dass sie als letztes Argument zusätzlich den Prototypen eines Rückgabewerts von FUN erwartet, etwa `numeric(1)`. Dies macht `vapply()` im Kontext selbst geschriebener Funktionen (Abschn. 17.3) weniger fehleranfällig.

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. Das Ergebnis kann anschließend als Indexvektor für die Spalten verwendet werden, um eine Gruppe von Variablen mit einer bestimmten Eigenschaft auszuwählen.²⁶

```
> (numIdx <- sapply(myDf1, is.numeric))      # numerische Variable?
  id    sex   group   age    IQ  rating
TRUE FALSE FALSE TRUE  TRUE    TRUE

> dataNum <- subset(myDf1, select=numIdx)    # nur numerische Variablen
> head(dataNum, n=3)
  id   age   IQ  rating
1 1    26  112      1
2 2    30  122      3
3 3    25  95       5

> data.matrix(dataNum)  # wandle numerische Variablen in Matrix um ...
```

Mit `do.call()` ist eine etwas andere automatisierte Anwendung einer Funktion auf die Komponenten einer Liste bzw. auf die Variablen eines Datensatzes möglich. Während `lapply()` 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=<Funktion>, 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 Funktionsaufrufs aus dem Funktionsnamen einerseits und den 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=myL[[1]], age=myL[[2]], IQ=myL[[3]], rating=myL[[4]])
> do.call(c, myL)
  id      age      IQ  rating
6.500000 26.416667 97.250000 2.583333
```

²⁶`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. als Argument nur einen einzelnen Wert, nicht aber Vektoren akzeptiert. In diesem Fall betrachtet `sapply()` jedes Element des Vektors als eigene Variable, die nur einen Wert beinhaltet.

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 der Funktion `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))
> tab   <- table(work, hiLo, group)    # 3D-Kreuztabelle der Faktoren

# wandle 3D-Kreuztabelle mit ftable um -> lege fest, welche Faktoren
# in Zeilen (row.vars), welche in Spalten (col.vars) stehen sollen
> argLst <- list(tab, row.vars="work", col.vars=c("hiLo", "group"))
> do.call(ftable, argLst)
  hiLo     hi      lo
  group   A   B   A   B
work
home       1   3   1   3
office      2   3   2   5
```

3.3.14 Funktionen für mehrere Variablen anwenden

`lapply()` und `sapply()` wenden eine Funktion nacheinander auf jeweils eine Variable eines Datensatzes bzw. einer Liste an. `mapply()` 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>, <Datensatz 1>, <Datensatz 2>, ...,
       MoreArgs=<Liste mit Optionen für FUN>)
```

Die anzuwendende Funktion ist als erstes Argument `FUN` zu nennen. Es folgen so viele Datensätze oder Listen, 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 (Abschn. 7.3). 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.

```
> N      <- 100
> x1    <- rnorm(N, 10, 10)        # Variablen für ersten Datensatz
> y1    <- rnorm(N, 10, 10)
> x2    <- x1 + rnorm(N, 5, 4)    # Variablen für zweiten Datensatz
```

```
> y2      <- y1 + rnorm(N, 10, 4)
> myDf2 <- data.frame(x1, y1)
> myDf3 <- data.frame(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"      # Ausgabe gekürzt ...

y1
statistic -33.75330
parameter 198
p.value 2.291449e-84
alternative "less"
method     "Two Sample t-test"      # Ausgabe gekürzt ...
```

3.4 Datensätze aufbereiten und aggregieren mit dplyr

Das Zusatzpaket **dplyr** ist sehr populär, um die typischen Schritte in der Datenaufbereitung durchzuführen. Verglichen mit dem Basisumfang von R bietet **dplyr** im wesentlichen zusätzliche Funktionalität. Stattdessen ist es das Ziel, durch leicht miteinander kombinierbare, einheitlich zu verwendende und damit auch schnell zu lernende Funktionen alternative Lösungswege für die häufigsten Aufgaben anzubieten.²⁷ Als Kurzreferenz für **dplyr** ist ein *cheat sheet*²⁸ verfügbar, ausführlich wird das Paket in [Wickham und Gromelund \(2023\)](#) vorgestellt. RStudio unterstützt das Arbeiten mit **dplyr** etwa dadurch, dass in Funktionsaufrufen Variablennamen aus dem in dieser Funktion transformierten Datensatz automatisch vervollständigt werden.²⁹

Die Arbeit mit **dplyr** besitzt im Vergleich zum Basisumfang von R Vor- und Nachteile. Zu den Vorteilen zählt, dass viele Arbeitsschritte schneller zu formulieren sind und sich leichter so miteinander verbinden lassen, dass sie dem natürlichen Fluss der typischen Schritte einer Datenaufbereitung folgen. Aufgaben wie das Umbenennen von Variablen (Abschn. 3.4.2) oder die Berechnung von Kennwerten getrennt nach Gruppen (Abschn. 3.4.10) sind mit **dplyr** bequemer zu erledigen als mit dem Basisumfang von R. Zu den Nachteilen zählt die sich von anderen R Funktionen stark unterscheidende Syntax (Abschn. 3.4.1). Auch unterlag das Paket in den letzten Jahren – jedoch seit Version 1.0 eher leichten – Veränderungen, die es erforderlich gemacht haben, Auswertungsbefehle anzupassen (Abschn. 1.3.4, 4.4).

²⁷Dies gilt auch für das Paket **data.table**, das ähnliche Ziele wie **dplyr** verfolgt, dafür aber eine andere, ebenfalls idiomatische Syntax verwendet. Eine Stärke von **data.table** ist seine Leistungsfähigkeit beim Umgang mit großen Datenmengen (Abschn. 17.5).

²⁸<https://posit.co/resources/cheatsheets/>

²⁹Mittlerweile wurde eine Reihe von *data science* Paketen entwickelt, die sich an **dplyr** orientieren und das Ziel haben, gut miteinander interagieren zu können. Sie firmieren unter dem Begriff *tidyverse*: <https://www.tidyverse.org/>

Alle im weiteren Verlauf des Abschnitts vorgestellten Funktionen stammen aus dem Paket `dplyr`, sofern dies nicht anders vermerkt ist. Als Datengrundlage dient der in Abschn. 2.11 erstellte Datensatz `myDf1`.

```
> head(myDf1)
#> #>   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
```

3.4.1 Besonderheiten

Verglichen mit dem Basisumfang von R weist die Syntax teils Unterschiede und Eigenheiten auf. Bei Auswertungen muss aufmerksam unterschieden werden, welche Schritte innerhalb der Funktionsfamilie aus `dplyr` ablaufen und welche außerhalb, da für beide Situationen teils unterschiedliche Regeln gelten.

Eine zentrale Rolle spielt in `dplyr` die Vorgehensweise, mehrere Verarbeitungsschritte mit der pipe-Syntax `|>` miteinander zu verketten, anstatt das Ergebnis jedes einzelnen Schritts einem Objekt zuzuweisen und dies dann wiederzuverwenden (Abschn. 3.3.1). Zwar lassen sich `dplyr` Funktionen auch mit konventioneller Syntax verwenden, sowohl in der offiziellen als auch in der von Dritten erstellten Dokumentation dominiert aber die pipe-Syntax. Alle `dplyr` Hauptfunktionen arbeiten nach demselben Muster, das an die pipe-Syntax angepasst ist. Ihr erstes Argument ist immer der aufzubereitende Datensatz, die weiteren Argumente bestimmen dessen Modifikation. Die Ausgabe einer Hauptfunktion ist immer der geänderte Datensatz.

Innerhalb von `dplyr` Funktionen sind Variablennamen generell ohne Anführungszeichen zu verwenden. Die technische Grundlage dieser mittels *non-standard evaluation* umgesetzten Eigenschaft ist komplex. Die dafür verwendeten internen Prozeduren weichen stark vom Vorgehen in Funktionen des Basisumfangs von R ab. Bei der Verwendung in selbst erstellten Funktionen (Abschn. 17.3) sind deshalb viele Besonderheiten zu beachten, auf die `vignette("programming", package="dplyr")` sowie Wickham (2019a, Kap. 19, 20) eingehen.

Zwar ist das Ergebnis von `dplyr` Hauptfunktionen ein Datensatz, allerdings besitzt dieser etwa im Fall von `group_by()` (Abschn. 3.4.10) die Klasse `tbl_df (tibble)`. Grundsätzlich verhalten sich tibbles wie Objekte der Klasse `data.frame`, besitzen aber teils subtil andere Eigenschaften. So gibt die Auswahl einer einzelnen Variable mit `<tibble>[, "<Variable>"]` wieder einen Datensatz der Klasse `tbl_df` anstatt eines Vektors zurück, ohne dass dafür das Argument `drop=FALSE` zu setzen wäre. Einzelne Variablen lassen sich stattdessen mit `<tibble>[["<Variable>"]]` oder `<tibble>$<Variable>` aus dem Datensatz herauslösen. Die Ausgabe von tibbles auf der Konsole unterscheidet sich etwa dadurch, dass Variablen nicht dargestellt werden, wenn es die verfügbare Breite der Konsole nicht erlaubt.

3.4.2 Variablen umbenennen

Variablen lassen sich mit `rename(<Datensatz>, <Name neu>=<Name alt>, ...)` umbenennen. Durch Komma getrennt können in einem Aufruf beliebig viele solcher Umbenennungen erfolgen.

```
> myDf1 |> rename(score=rating,
+                      fac=group)
#> #>   id sex fac age  IQ score
#> #>   1   f   T  26 112     1
#> #>   2   m   CG 30 122     3
#> #>   3   m   CG 25  95     5
#> #>   # Ausgabe gekürzt ...
```

Abschnitt 3.4.11 zeigt, wie sich Gruppen von Variablen gleichzeitig nach demselben Muster umbenennen lassen.

3.4.3 Teilmengen von Daten auswählen

Variablen auswählen

Teilmengen von Variablen können mit `select(<Datensatz>, <Variable1>, ...)` ausgewählt werden. Dafür lassen sich einzelne Variablen durch Komma getrennt aufführen. Die dabei verwendete Reihenfolge ist auch jene im erstellten Datensatz.³⁰

```
> myDf1 |> select(group, IQ)      # nur group, IQ in dieser Reihenfolge
#> #>   group  IQ
#> #>   1       T 112
#> #>   2       CG 122
#> #>   3       CG  95
#> #>   # Ausgabe gekürzt ...
```

Für den Fall, dass die Namen der auszuwählenden Variablen einem Muster folgen, kann das Namensmuster in `select(..., <Hilfsfunktion>)` mit verschiedenen Hilfsfunktionen spezifiziert werden:

- `starts_with("<Zeichenkette>")` wählt alle Variablen aus, deren Name mit einer festen Zeichenkette beginnt. Mit einem Vektor von Zeichenketten können mehrere alternative Anfänge bezeichnet werden. Analog arbeitet `ends_with()` für das Namensende.
- `contains("<Zeichenkette>")` wählt alle Variablen aus, die eine feste Zeichenkette irgendwo im Namen tragen. Ein Vektor von Zeichenketten lässt dafür mehrere Alternativen zu. Analog arbeitet `matches()`, um Zeichenketten über reguläre Ausdrücke zu definieren.
- `all_of(c("<Name1>", "<Name2>", ...))` sowie `any_of(c("<Name1>", "<Name2>", ...))` spezifizieren die auszuwählenden Variablen als (Vektoren von) Zeichenketten. Dies ist nützlich, wenn die Variablenauswahl beim Schreiben der Auswertung noch nicht feststeht, sondern sich erst zur Laufzeit aus vorherigen Verarbeitungsschritten ergibt. Alle

³⁰Da auch das Paket MASS eine Funktion `select()` besitzt, kommt es zu Fehlern, wenn MASS nach `dplyr` geladen wurde (Abschn. 1.3.3, Fußnote 28). In diesem Fall sollte der Aufruf in der Form `dplyr::select()` erfolgen.

mit `all_of()` bezeichneten Variablen müssen auch im Datensatz vorhanden sein, während dies bei `any_of()` nicht der Fall sein muss.

- `<Variable Beginn>:<Variable Ende>` wählt die Variablen aus, die links und rechts vom `:` stehen sowie alle im Datensatz zwischen ihnen liegenden Spalten. Statt der Variablennamen können auch ihre numerischen Spalten-Indizes genannt werden.

Die Voreinstellung `ignore.case=TRUE` in den Hilfsfunktionen `starts_with()`, `ends_with()`, `contains()` und `matches()` sorgt dafür, dass sie Groß- und Kleinschreibung nicht berücksichtigen, wenn sie prüfen, ob ein Namensmuster zutrifft.

```
# Variablen, deren Name mit i / I beginnt und weitere Zeichen enthält
> myDf1 |> select(matches("^i.+"))
  id  IQ
1  1 112
2  2 122
3  3  95                                # Ausgabe gekürzt ...
```

Analog kann `select` auch Variablen anhand einer logischen Bedingung auswählen, etwa abhängig von ihrem Datentyp. Dafür ist statt der genannten Hilfsfunktionen eingeschlossen in `where()` eine Funktion zu übergeben, die für jede Variable einen logischen Wert zurückgibt. Mit Hilfe der üblichen logischen Operatoren `!`, `&`, `|` lassen sich auch zusammengesetzte Bedingungen formulieren.

```
> myDf1 |> select(where(is.numeric))      # numerische Variablen
  id age  IQ rating
1  1  26 112     1
2  2  30 122     3
3  3  25  95     5                                # Ausgabe gekürzt ...

# nicht numerische Variablen, die mit g / G beginnen
> myDf1 |> select(!where(is.numeric) & starts_with("g"))
  group
1      T
2     CG
3     CG                                # Ausgabe gekürzt ...
```

Variablen lassen sich mit `select(<Datensatz>, -<Name>, ...)` aus einem Datensatz entfernen. Auch hier lassen sich die Hilfsfunktionen anwenden, um Namensmuster zu definieren.

```
# entferne sex & Variablen, die a als nicht letztes Zeichen enthalten
> myDf1 |> select(-sex, -matches("a.+"))
  id group  IQ
1   1      T 112
2   2     CG 122
3   3     CG  95                                # Ausgabe gekürzt ...

# entferne Variablen von group bis IQ
> myDf1 |> select(-(group:IQ))
  id sex rating
```

```

1   1   f     1
2   2   m     3
3   3   m     5
# Ausgabe gekürzt ...

```

Beobachtungen auswählen

`filter(<Datensatz>, <Kriterium1>, ...)` wählt eine Teilmenge von Beobachtungen aus, die bestimmte Kriterien erfüllt.³¹ Hinter dem Datensatz als erstem Argument lassen sich durch Komma getrennt beliebig viele Kriterien definieren, die die einzuschließenden Beobachtungen erfüllen müssen. Alle Einzelkriterien werden also zur Gesamtauswahl implizit durch ein logisches UND verbunden. Um Kriterien durch logisches ODER zu verbinden, muss ein einzelner entsprechender Ausdruck explizit mit | konstruiert werden.³²

```

# Personen in Gruppe CG / WL mit numerischer id > rating UND IQ > 90
> myDf1 |> filter(group %in% c("CG", "WL"),
+                     id > rating,
+                     IQ > 90)
  id sex group age  IQ rating
1  6   f    CG  24 113      0
2 11   m    CG  20  92      1

# alle Personen mit einem eher hohen ODER eher niedrigen IQ-Wert
> myDf1 |> filter((IQ < 90) | (IQ > 110))
  id sex group age  IQ rating
1  1   f    T  26 112      1
2  2   m    CG  30 122      3
3  5   m    WL  22  82      2
# Ausgabe gekürzt ...

```

Abschnitt 3.4.11 demonstriert, wie sich die Auswahlbedingung für Beobachtungen gleichzeitig auf eine definierte Gruppe von Variablen beziehen kann.

Demgegenüber lassen sich mit `slice(<Datensatz>, <Indizes>)` Beobachtungen entsprechend ihres Zeilen-Index im Datensatz auswählen. Durch ein vorangestelltes - werden Beobachtungen mit dem bezeichneten Zeilen-Index entfernt. `slice_head(..., n=<Anzahl>)` und `slice_tail(..., -n=<Anzahl>)` dienen dazu, die ersten bzw. letzten n Beobachtungen eines Datensatzes auszugeben.

```

> myDf1 |> slice(5:7)          # Beobachtungen 5, 6, 7
  id sex group age  IQ rating
1  5   m    WL  22  82      2
2  6   f    CG  24 113      0
3  7   m    T  28  92      3

```

³¹Das zum Standardumfang von R gehörende Paket `stats` enthält eine Funktion gleichen Namens und erzeugt deshalb immer einen Namenskonflikt (Fußnote 30). Bei Bedarf ist deshalb der Aufruf mit `stats::filter()` notwendig. Analog gilt dies für die Funktion `lag()`.

³²Wie in `subset()` gelende fehlende Werte als `FALSE` (Abschn. 3.3.3, Fußnote 15).

`slice_sample(<Datensatz>, n=<Anzahl>, prop=<Anteil>)` wählt aus einem Datensatz eine zufällige Teilmenge von Beobachtungen einer bestimmten Anzahl (Argument `n`) oder eines bestimmten Anteils (Argument `prop`) aus.

Im Gegensatz zur Auswahl von Beobachtungen über den Zeilen-Index mit `<tibble>[<Index>, <-]` sowie mit `head()` und `tail()` aus dem Basisumfang von R berücksichtigen die genannten Varianten von `slice()` eine implizite Gruppeneinteilung des Datensatzes (Abschn. 3.4.10).

Im Gegensatz zu anderen `dplyr` Funktionen dient `pull(<Datensatz>, <Name>)` dazu, eine Variable aus einem Datensatz herauszulösen. Dies ist äquivalent zu `<Datensatz>[["<Name>"]]`, aber leichter in eine pipe zu integrieren.

3.4.4 Doppelte und fehlende Werte ausschließen

Die Funktion `distinct()` entfernt duplizierte Beobachtungen aus einem Datensatz. Abgesehen von einer evtl. besseren Effizienz bei großen Datensätzen ist sie äquivalent zu `unique()` aus dem Basisumfang von R. Dies gilt auch für die Funktion `n_distinct()`, die analog zu `length(unique(<Variable>))` die Anzahl vorkommender Werte liefert.

Wenn man Variablennamen an `distinct()` übergibt, werden zunächst alle außer die bezeichneten Variablen aus dem Datensatz entfernt, ehe die vorhandenen Duplikate gelöscht werden.

```
> myDf1 |> select(sex, group) |> # wähle Variablen sex, group aus
+   distinct()                      # entferne Duplikate
  sex group
1   f     T
2   m    CG
3   m     T
4   m    WL
5   f    CG

> myDf1 |> distinct(sex, group)    # äquivalent ...
```

`na_if(<Vektor>, <Wert>)` dient innerhalb von `mutate()` dazu, bestimmte Werte als fehlend (`NA`) zu kennzeichnen und alle anderen Werte unverändert zu lassen. Ein häufiger Fall ist die leere Zeichenkette `" "` für `character` Variablen.

```
> myDf9999          <- myDf1      # Kopie erstellen
> myDf9999$IQ[2]    <- 9999       # Code für fehlenden Wert
> myDf9999$rating[3] <- 9999       # Code für fehlenden Wert
> (myDfNA <- myDf9999 |> mutate(IQ=na_if(IQ, 9999),
+                                     rating=na_if(rating, 9999)))
  id sex group age  IQ rating
1   1   f     T   26 112      1
2   2   m    CG   30   NA      3
3   3   m    CG   25   95     NA
4   4   m     T   34  102      5  # Ausgabe gekürzt...
```

Um Zeilen mit fehlenden Werten komplett aus einem Datensatz zu entfernen, kann `na.omit()` an eine pipe angeschlossen werden.

3.4.5 Variablen entfernen, hinzufügen und transformieren

Mit `dplyr` Funktionen können Variablen eines Datensatzes nicht direkt gelöscht werden. Dazu ist es notwendig, den Datensatz durch eine Auswahl von Variablen zu überschreiben, die z. B. mit `select()` erzeugt wurde (Abschn. 3.4.3).

`mutate()` dient dazu, Variablen eines Datensatzes zu verändern. Durch einen Aufruf der Form `mutate(<Datensatz>, <Name neu>=<Ausdruck>)` wird eine neue Variable hinzugefügt. Dabei kann `<Ausdruck>` der Name einer außerhalb des Datensatzes erstellten Variable passender Länge sein oder ein Befehl, der solch eine Variable erzeugt. Durch Komma getrennt können in einem Aufruf auf diese Weise viele neue Variablen erstellt werden. Die in diesem Kontext nützliche Hilfsfunktion `n()` steht für die Anzahl an Beobachtungen.

```
# neue simulierte Variable Beziehungsstatus
> myDf1 |> mutate(married=sample(c(TRUE, FALSE), n(), replace=TRUE))

  id sex group age  IQ rating married
1  1   f      T  26 112       1    TRUE
2  2   m     CG  30 122       3    TRUE
3  3   m     CG  25  95       5   FALSE  # Ausgabe gekürzt ...
```

Steht im Aufruf von `mutate()` links der Zuweisung durch `=` der Name einer schon bestehenden Variable, wird diese überschrieben. Alle nicht aufgeführten Variablen bleiben dagegen unverändert. Im Unterschied zur sonst weitgehend analogen Funktion `transform()` aus dem Basisumfang von R können dabei spätere Schritte auf Variablen zugreifen, die an einer früheren Position im selben Aufruf von `mutate()` erstellt wurden. Variablen können so in einem Aufruf schrittweise erstellt und modifiziert werden. Das im folgenden Beispiel verwendete Paket `forcats` wird in Abschn. 2.6 vorgestellt.

```

> library(forcats)                                # für fct_collapse()
> myDf1 |>          # kombiniere Faktorstufen CG und WL zu CG_WL
+   mutate(group=fct_collapse(group, CG_WL=c("CG", "WL")),
+         ratingSq=rating^2,                      # quadriertes rating
+         ratingSqZ=scale(ratingSq))    # z-transformiertes rating^2
  id sex group age  IQ rating ratingSq   ratingSqZ
  1  1    f      T  26 112      1       1 -0.84593920
  2  2    m  CG_WL  30 122      3       9 -0.04187818
  3  3    m  CG_WL  25  95      5      25  1.56624386 # Ausgabe gekürzt...

```

Innerhalb von `mutate()` kommen oft Hilfsfunktionen zum Einsatz. Dazu zählt als Ersatz für `ifelse()` aus dem Basisumfang von R die weitgehend analoge Funktion `if_else()`, die jedoch implizite Typumwandlungen vermeidet.

`if_else(condition, true=⟨Vektor⟩, false=⟨Vektor⟩, missing=⟨Wert⟩)`

`if_else()` achtet strikt darauf, dass die Prüfung der unter `condition` genannten vektorisierten Bedingung logische Werte zurückgibt und die unter `true` sowie `false` definierten Vektoren denselben Datentyp besitzen. Ferner lässt sich über das Argument `missing` explizit festlegen, welches Ergebnis für fehlende Werte einzusetzen ist.

Erweiterte Fallunterscheidungen durch komplexe Vergleiche mit mehr als zwei möglichen Fällen sind mit `case_when(`*<Bedingung1>* ~ *<Ergebnis1>*, ...) möglich – analog zu `switch()` (Abschn. 17.1.1). Jeder durch Komma getrennte Ausdruck definiert dabei eine logische Bedingung. Für alle Beobachtungen, die diese Bedingung erfüllen, wird das bezeichnete Ergebnis eingesetzt. Alle definierten Ergebnisse müssen dafür denselben Datentyp besitzen. Sind mehrere Bedingungen erfüllt, wird als Ergebnis jenes der ersten erfüllten Bedingung verwendet. Die Bedingungen sollten deshalb vom Spezifischen zum Allgemeinen geordnet sein. Zum Schluss kann mit `.default=<Ergebnis>` ein *Default*-Ergebnis an alle Beobachtungen zugewiesen werden, die vorher keine Bedingung erfüllt haben.

```
# age_even prüft, ob das Alter eine gerade Zahl ist
> myDf1 |> mutate(age_even=if_else((age %% 2) == 0, TRUE, FALSE),
+                     sex_IQ=case_when(
+                         ((sex == "f") & (IQ < 100)) ~ "female_lo",
+                         ((sex == "f") & (IQ >= 100)) ~ "female_hi",
+                         ((sex == "m") & (IQ < 100)) ~ "male_lo",
+                         ((sex == "m") & (IQ >= 100)) ~ "male_hi",
+                         .default                      = "other")) # catch all

  id sex group age  IQ rating age_even     sex_IQ
1  1   f      T  26 112       1    TRUE female_hi
2  2   m     CG  30 122       3    TRUE  male_hi
3  3   m     CG  25  95       5   FALSE male_lo  # Ausgabe gekürzt...
```

Die Funktion `coalesce(<Vektor1>, <Vektor2>, ...)` dient dazu, fehlende Werte zu ersetzen. Dafür prüft sie für jede Beobachtung nacheinander alle übergebenen Vektoren daraufhin, ob in ihnen ein Wert für diese Beobachtung vorhanden ist, also an derselben Position im Vektor steht. Als Ergebnis erstellt `coalesce()` einen Vektor, bei dem für jedes Element der erste nicht fehlende Wert der übergebenden Vektoren an derselben Position eingesetzt wird.

Dieselbe Datentransformation lässt sich auch gleichzeitig auf eine Gruppe von Variablen anwenden (Abschn. 3.4.11).

3.4.6 Datensätze sortieren

`arrange(⟨Datensatz⟩, ⟨Kriterium1⟩, ...)` sortiert die Beobachtungen eines Datensatzes. Hinter dem zu sortierenden Datensatz als erstem Argument definieren alle weiteren Argumente die Sortierkriterien, die in der Voreinstellung zu einer aufsteigenden Reihenfolge führen. Soll

bzgl. eines Kriteriums in absteigender Reihenfolge sortiert werden, ist es in die Hilfsfunktion `desc()` einzuschließen.

```
> myDf1 |> arrange(rating)           # sortiere aufsteigend nach rating
   id sex group age  IQ rating
1   6   f     CG  24 113      0
2   1   f      T  26 112      1
3  11   m     CG  20  92      1      # Ausgabe gekürzt ...
                                          
# sortiere aufsteigend nach group und absteigend nach IQ
> myDf1 |> arrange(group, desc(IQ))
   id sex group age  IQ rating
1   2   m     CG  30 122      3
2   6   f     CG  24 113      0
3   3   m     CG  25  95      5      # Ausgabe gekürzt ...
```

Die Reihenfolge der Variablen innerhalb eines Datensatzes lässt sich über deren Reihenfolge im Aufruf von `select(<Variable1>, <Variable2>, ...)` zur Variablenauswahl kontrollieren (Abschn. 3.4.3). Dabei dient die Hilfsfunktion `everything()` dazu, auch alle verbleibenden, noch nicht explizit genannten Variablen in der ursprünglichen Reihenfolge einzuschließen.

```
# ziehe Variablen group und age an den Anfang
> myDf1 |> select(group, age, everything())
   group age id sex  IQ rating
1     T  26  1   f 112      1
2    CG  30  2   m 122      3
3    CG  25  3   m  95      5      # Ausgabe gekürzt ...
```

Mit `relocate(<Datensatz>, .after=<Variable>, .before=<Variable>)` lässt sich eine Variable auch direkt an eine definierte Position im Datensatz schieben.

```
# schiebe Variable group ans Ende
> myDf1 |> relocate(group, .after=rating)
   id sex age  IQ rating group
1   1   f  26 112      1     T
2   2   m  30 122      3    CG
3   3   m  25  95      5    CG      # Ausgabe gekürzt ...
```

3.4.7 Datensätze zeilen- oder spaltenweise verbinden

Die Funktion `bind_rows()` verbindet über mehrere Datensätze hinweg verteilte Beobachtungen zu einem gemeinsamen Datensatz, indem sie diese untereinander anhängt. Dafür akzeptiert `bind_rows()` entweder durch Komma getrennt einzelne Datensätze oder aber eine Liste von Datensätzen – analog zu `do.call(rbind, <Liste>)` aus dem Basisumfang von R.

Anders als bei `rbind()` müssen die Datensätze aber nicht denselben Aufbau besitzen. Variablen, die nur in manchen Datensätzen vorhanden sind, werden zunächst in allen Datensätzen, wo sie fehlen, hinzugefügt und dort für alle Beobachtungen auf NA gesetzt.

```
# neuer, verglichen mit myDf1 unvollst. Datensatz: ohne group, IQ
> myDf2 <- data.frame(id=21,
+                         sex=factor("f", levels=c("f", "m")),
+                         age=48,
+                         rating=3)

# verbundener und um NA ergänzter Datensatz
> bind_rows(myDf1, myDf2) |> tail(n=3)
  id sex group age IQ rating
1 11   m    CG  20  92      1
2 12   f     T  21  98      1
3 21   f    <NA>  48   NA      3
```

Analog zu `cbind()` fügt `bind_cols()` Datensätze mit unterschiedlichen Variablen für dieselben Beobachtungen spaltenweise zu einem gemeinsamen Datensatz zusammen.

3.4.8 Datensätze zusammenführen

Die Funktionen `left_join()` und `inner_join()` führen flexibel Datensätze zusammen, die sich nur teilweise in den Variablen oder in den Beobachtungsobjekten entsprechen. Details zu diesen Funktionen erläutert [vignette\("two-table", package="dplyr"\)](#). In Abschn. 3.3.9 (s. insbesondere Fußnote 22) finden sich Erklärungen zu möglichen Szenarien hinsichtlich des Grads der Überlappung beider Datensätze und zum Gebrauch der analogen Funktion `merge()` aus dem Basisumfang von R.

`left_join(<Datensatz1>, <Datensatz2>, by)` entspricht als am häufigsten eingesetzte Variante `merge(..., all.x=TRUE)` und sorgt dafür, dass alle Zeilen des ersten übergebenen Datensatzes erhalten bleiben, auch wenn Beobachtungsobjekte nicht im zweiten Datensatz auftauchen. Zusätzliche Variablen aus dem zweiten Datensatz für dort ebenfalls vorkommende Beobachtungsobjekte werden dem Ergebnis hinzugefügt und ggf. mit NA ergänzt. Mit einer an `by` übergebenen Spezifikation mittels `join_by()` können die Variablen festgelegt werden, die der Identifikation derselben Beobachtungsobjekte dienen. Bei identischen Variablennamen hat der Aufruf die Form `join_by(<Name>)`, wobei durch Komma getrennt weitere Variablen genannt werden können. Tragen die einander entsprechenden Variablen in beiden Datensätzen nicht denselben Namen, können die unterschiedlichen Namen mit `join_by(<Name1> == <Name2>, ...)` einander zugeordnet werden. Es empfiehlt sich, die Option `by` explizit zu setzen. Lässt man sie weg, werden automatisch alle Variablen mit in beiden Datensätzen identischem Namen herangezogen.

```
> (IDDV <- data.frame(ID=factor(rep(1:3, each=2)),
+                         DV=round(rnorm(6, 100, 15))))
  ID  DV
1  1  76
2  1 103
3  2  98
4  2  91
5  3 131
```

6 3 78

```
> (IV <- data.frame(ID=factor(1:3),
+                     IV=factor(c("A", "B", "A")),
+                     sex=factor(c("f", "f", "m"))))
   ID IV sex
1  1  A  f
2  2  B  f
3  3  A  m

# äquivalent: merge(IDDV, IV, all.x=TRUE)
> IDDV |> left_join(IV, by=join_by(ID))
   ID DV IV sex
1  1  76  A  f
2  1 103  A  f
3  2  98  B  f
4  2  91  B  f
5  3 131  A  m
6  3  78  A  m
```

`right_join()` verhält sich genauso, jedoch mit vertauschten Rollen des ersten und zweiten Datensatzes, ist also äquivalent zu `merge(..., all.y=TRUE)`. `full_join()` sorgt analog zu `merge(..., all=TRUE)` dafür, dass alle Zeilen aus beiden übergebenen Datensätzen erhalten bleiben und ergänzt ggf. fehlende Informationen durch `NA`. Die drei genannten *join* Operationen werden zusammen als *outer join* bezeichnet. Ihre Unterschiede werden sichtbar, wenn sich zwei Datensätze nur teilweise hinsichtlich ihrer Beobachtungsobjekte und Variablen überlappen.

```
> (dfa <- data.frame(ID=1:4,
+                      initials=c("AB", "CD", "EF", "GH"),
+                      IV1=c("-", "-", "+", "+"),
+                      DV1=c(10, 10, 11, 14)))
   ID initials IV1 DV1
1  1        AB  -  10
2  2        CD  -  10
3  3        EF  +  11
4  4        GH  +  14

# Überlappung: Beobachtungen mit ID = EF, GH, anderer Name für ID
> (dfB <- data.frame(ID_mod=3:6,
+                      initials=c("EF", "GH", "IJ", "KL"),
+                      IV2=c("A", "B", "A", "B"),
+                      DV2=c(92, 79, 101, 81)))
   ID_mod initials IV2 DV2
1      3        EF  A  92
2      4        GH  B  79
3      5        IJ  A 101
4      6        KL  B  81
```

```
# äquivalent
# merge(dfA, dfB, all.x=TRUE,
#       by.x=c("ID", "initials"), by.y=c("ID_mod", "initials"))
> dfA |> left_join(dfB, by=join_by(ID == ID_mod, initials))
  ID initials IV1 DV1  IV2 DV2
1  1        AB   -  10 <NA>  NA
2  2        CD   -  10 <NA>  NA
3  3        EF   +  11    A  92
4  4        GH   +  14    B  79

# äquivalent
# merge(dfA, dfB, all.y=TRUE,
#       by.x=c("ID", "initials"), by.y=c("ID_mod", "initials"))
> dfA |> right_join(dfB, by=join_by(ID == ID_mod, initials))
  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

# äquivalent
# merge(dfA, dfB, all.x=TRUE, all.y=TRUE,
#       by.x=c("ID", "initials"), by.y=c("ID_mod", "initials"))
> dfA |> full_join(dfB, by=join_by(ID == ID_mod, initials))
  ID initials  IV1 DV1  IV2 DV2
1  1        AB   -  10 <NA>  NA
2  2        CD   -  10 <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
```

Demgegenüber erhält `inner_join()` nur jene Beobachtungen, die in beiden Datensätzen mit identischen Informationen vorhanden sind und entspricht damit `merge()`.

```
# äquivalent
# merge(dfA, dfB,
#       by.x=c("ID", "initials"), by.y=c("ID_mod", "initials"))
> dfA |> inner_join(dfB, by=join_by(ID == ID_mod, initials))
  ID initials  IV1 DV1  IV2 DV2
1  3        EF   +  11    A  92
2  4        GH   +  14    B  79
```

3.4.9 Organisationsform komplexer Datensätze ändern

Das Paket `tidyverse` (Wickham & Henry, 2020) stellt mit `pivot_longer()` und `pivot_wider()` Funktionen bereit, die analog zu `reshape()` Datensätze zwischen Wide-Format und Long-Format transformieren (Abschn. 3.3.11, insbesondere auch Fußnote 23). Ihr Ergebnis ist jeweils ein Datensatz der Klasse `tbl_df` (Abschn. 3.4.1). Details und viele Beispiele erläutert `vignette("pivot")`. Zur Transformation ins Wide-Format dient:

```
pivot_longer(<Datensatz Wide>, cols, names_to, values_to, names_prefix)
```

Für `cols` ist anzugeben, welche Variablen im Wide-Format dieselbe Variable zu unterschiedlichen Messzeitpunkten repräsentieren. Sie können auch wie bei `select()` (Abschn. 3.4.5) mit Hilfsfunktionen über ihr Namensmuster oder ihre Spalten-Position definiert werden. Die Variablen werden im Long-Format über unterschiedliche Ausprägungen einer neu gebildeten Variable identifiziert, deren Name eine für `names_to` angegebene Zeichenkette spezifiziert. In der Voreinstellung werden die ursprünglichen Spaltennamen als Ausprägungen dieses Messwiederholungsfaktors verwendet. Um dabei einen Namensbestandteil zu entfernen, kann dieser als regulärer Ausdruck (2.14.4) an `names_prefix` übergeben werden. Den Namen der Variable im Long-Format mit den Messwerten legt das Argument `values_to` fest. Alle nicht unter `cols` aufgeführten Variablen werden als solche betrachtet, die pro Beobachtungsobjekt nicht über Messzeitpunkte variieren. Die in den folgenden Beispielen verwendeten Datensätze sind jene aus Abschn. 3.3.11.

```
# Datensatz Wide-Format
> (datW <- data.frame(id, IVbtw, DV_t1, DV_t2, DV_t3))
  id IVbtw DV_t1 DV_t2 DV_t3
1  1      A -0.16 -0.67  0.92
2  2      B -0.86  0.16  1.00
3  3      A -1.52  1.72  0.93
4  4      B -1.52  0.53  0.90

# ins Long-Format umwandeln, dabei "DV_" aus Ausprägungen entfernen
> datL <- datW |> pivot_longer(cols=starts_with("DV_"),
+                               names_to="time", values_to="DV",
+                               names_prefix="DV_")

> datL
# A tibble: 12 x 4
  id IVbtw time     DV
  <int> <fct> <chr> <dbl>
1  1   A     t1    -1.61
2  1   A     t2    -0.42
3  1   A     t3     1.53
4  2   B     t1     0.08
5  2   B     t2    -0.31
6  2   B     t3    -0.12
7  3   A     t1     0.07
8  3   A     t2     0.61
```

```

9      3 A     t3    -0.42
10     4 B     t1    -1.36
11     4 B     t2    -0.04
12     4 B     t3     0.88

```

Analog dient zur Transformation ins Wide-Format:

```
pivot_wider(<Datensatz Long-Format>,
            id_cols, names_from, values_from, names_prefix)
```

`id_cols` definiert die Variablen, die pro Beobachtungsobjekt über die Messzeitpunkte hinweg konstant sind. `names_from` bezeichnet die Variable, die im Long-Format die Ausprägung des Messwiederholungsfaktors definiert. Ihre Ausprägungen werden zu den Spaltennamen des Datensatzes im Wide-Format, wobei eine für `names_prefix` genannte Zeichenkette jeweils vorangestellt wird. Für `values_from` ist die Variable mit den Messwerten zu nennen. Gegebenenfalls können dies auch mehrere Variablen sein.

```

# ins Wide-Format umwandeln
> datL |> pivot_wider(id_cols=c(id, IVbtw),
+                         names_from=time, values_from=DV,
+                         names_prefix="DV_")
# A tibble: 4 x 5
  id IVbtw DV_t1 DV_t2 DV_t3
  <int> <fct> <dbl> <dbl> <dbl>
1 1 A     -0.16 -0.67  0.92
2 2 B     -0.86  0.16   1
3 3 A     -1.52  1.72  0.93
4 4 B     -1.52  0.53   0.9

# füge zweite Variable mit Messwerten hinzu
> datL |> mutate(DVsq=DV^2) |>
+   pivot_wider(id_cols=c(id, IVbtw),
+               names_from=time, values_from=c(DV, DVsq))
# A tibble: 4 x 8
  id IVbtw DV_t1 DV_t2 DV_t3 DVsq_t1 DVsq_t2 DVsq_t3
  <int> <fct> <dbl> <dbl> <dbl>   <dbl>   <dbl>   <dbl>
1 1 A     -1.61 -0.42  1.53   2.59    0.176   2.34
2 2 B     0.08  -0.31 -0.12  0.0064  0.0961  0.0144
3 3 A     0.07  0.61 -0.42  0.0049  0.372   0.176
4 4 B     -1.36 -0.04  0.88   1.85    0.0016  0.774   # ...

```

3.4.10 Datensätze getrennt nach Gruppen auswerten und aggregieren

Mit `group_by(<Datensatz>, <Faktor1>, ...)` bereitet man einen Datensatz darauf vor, entsprechend des übergebenen Faktors separat nach Gruppen ausgewertet zu werden.³³ Über gibt

³³Details erläutert `vignette("grouping", package="dplyr")`

man mehrere Faktoren durch Komma getrennt und behält die Voreinstellung `.drop=TRUE` bei, werden dafür alle tatsächlich vorkommenden Kombinationen ihrer Stufen gebildet. Mit `.drop=FALSE` werden auch die prinzipiell möglichen, nicht aber tatsächlich beobachteten Stufenkombinationen gebildet. Die Ausprägungen der eingesetzten Gruppierungsvariablen werden versteckt in den erzeugten Datensatz aufgenommen.

Anschließend auf den implizit gruppierten Datensatz angewendete `dplyr` Funktionen erkennen die Aufteilung automatisch und arbeiten getrennt nach Gruppen. Einen gruppierten Datensatz erkennt man in der Ausgabe durch die zusätzliche, mit `Groups:` beginnende Zeile, die vor den eigentlichen Daten steht. Die interne Gruppierung wird durch `ungroup(<Datensatz>)` wieder aufgehoben. Dieser Schritt sollte abschließend immer erfolgen, damit folgende Datentransformationen nicht unbeabsichtigt getrennt nach Gruppen ablaufen können.

```
> myDf1_grp <- myDf1 |>
+   group_by(sex) |>                                # trenne Auswertung nach sex
+   mutate(IQ_rank=rank(IQ)) |>      # IQ-Rang pro Gruppe
+   arrange(sex, IQ)                         # ordne nach Gruppe und IQ-Rang

> myDf1_grp # gruppiertes Datensatz -> zusätzliche Zeile "Groups: ..."
# A tibble: 12 x 7
# Groups:   sex [2]
  id sex  group  age    IQ rating IQ_rank
  <dbl> <chr> <chr> <dbl> <dbl> <dbl>    <dbl>
1 12 f     T      21    98     1      1
2 1 f     T      26   112     1      2
3 6 f     CG     24   113     0      3
4 10 m    WL     29    81     5      1  # Ausgabe gekürzt ...

> myDf1_grp |> ungroup()                      # entferne Gruppierung
# A tibble: 12 x 7
  id sex  group  age    IQ rating IQ_rank
  <dbl> <chr> <chr> <dbl> <dbl> <dbl>    <dbl>
1 12 f     T      21    98     1      1
2 1 f     T      26   112     1      2  # Ausgabe gekürzt ...
```

Funktionen aus dem Basisumfang von R erkennen die implizite Gruppeneinteilung dagegen nicht. Sie verarbeiten daher wie gewöhnlich alle Beobachtungen einer Variable gemeinsam.

```
# slice_head() erkennt Gruppeneinteilung -> Zeile 1:2 je Gruppe
> myDf1_grp |> slice_head(n=2)
# A tibble: 4 x 6
# Groups:   sex [2]
  id sex  group  age    IQ rating IQ_rank
  <dbl> <chr> <chr> <dbl> <dbl> <dbl>    <dbl>
1 12 f     T      21    98     1      1
2 1 f     T      26   112     1      2
3 10 m    WL     29    81     5      1
4 5 m     WL     22    82     2      2
```

```
# head() erkennt interne Gruppenaufteilung nicht -> Zeile 1:2 overall
> myDf1_grp |> head(n=2)
# A tibble: 2 x 7
# Groups:   sex [1]
  id sex   group   age     IQ rating IQ_rank
  <dbl> <chr> <chr> <dbl> <dbl>   <dbl>    <dbl>
1 12 f     T       21     98     1       1
2  1 f     T       26    112     1       2
```

`summarise(<Datensatz>, <Kennwert1>=<Ausdruck1>, ...)` dient dazu, einen Datensatz durch Kennwerte zu aggregieren, die über durch Komma getrennte Zuweisungen definiert werden. Die zur Berechnung eines Kennwerts eingesetzten Funktionen dürfen dafür nur einen einzelnen Wert zurückgeben. Angewendet auf einen nicht implizit gruppierten Datensatz werden die Kennwerte über alle Beobachtungen hinweg gebildet. Wird stattdessen ein vorher durch `group_by()` implizit gruppierter Datensatz übergeben, erfolgt die Berechnung der Kennwerte pro Gruppe. Die Hilfsfunktion `n()` ermittelt hier die Anzahl von Beobachtungen je Gruppe.

```
> myDf1 |> group_by(group) |>      # trenne Auswertung nach group
+   summarise(age_M=mean(age),      # gruppenweiser Mittelwert von age
+             age_SD=sd(age),        # gruppenweise Streuung von age
+             IQ_M=mean(IQ),         # gruppenweiser Mittelwert von IQ
+             IQ_SD=sd(IQ),         # gruppenweise Streuung von IQ
+             n=n())                # Gruppengröße
# A tibble: 3 x 5
  group age_M age_SD  IQ_M IQ_SD      n
  <chr> <dbl>  <dbl> <dbl>  <dbl> <int>
1 CG     24.8   4.11 106.  14.4      4
2 T      27.2   5.38 101   8.41      4
3 WL    27.2   6.02  85.2  4.43      4
```

Jeder Aufruf von `summarise()` entfernt die Gruppierungsinformation aufgrund der letzten vorher in `group_by()` verwendeten Gruppierungsvariable.

```
> df_aggr <- myDf1 |>          # speichere Ergebnis in df_aggr
+   group_by(sex, group) |>      # gruppiere nach sex und group
+   summarise(rating_M=mean(rating)) # lösche Gruppierung nach group

> df_aggr                         # weiterhin nach sex gruppiert
# A tibble: 5 x 3
# Groups:   sex [2]
  sex   group rating_M
  <chr> <chr>    <dbl>
1 f     CG        0
2 f     T         1
3 m     CG        3
4 m     T         4
5 m     WL        3
```

```
# Gruppengrößen in aggregierten Daten getrennt nach sex -> 2, 3
> df_aggr |> summarise(n=n())
# A tibble: 2 x 2
  sex     n
  <chr> <int>
1 f         2
2 m         3

# NICHT äquivalent: Gruppengrößen in Originaldaten -> 3, 9
> myDf1 |> group_by(sex) |>
+   summarise(n=n())
# A tibble: 2 x 2
  sex     n
  <chr> <int>
1 f         3
2 m         9
```

`reframe(<Datensatz>, <Kennwert1>=<Ausdruck1>, ...)` verallgemeinert `summarise()` dahingehend, dass auch Funktionen zur Berechnung eines Kennwerts zugelassen sind, die mehrere Werte zurückgeben.

```
> myDf1 |>
+   reframe(Qs_age=quantile(age, probs=c(0.25, 0.5, 0.75)),
+             Qs_IQ =quantile(IQ, probs=c(0.25, 0.5, 0.75))) |>
+   mutate(Q=sprintf("Q%.2f", c(0.25, 0.5, 0.75)))
  Qs_age Qs_IQ    Q
1 22.75  89.5 Q0.25
2 25.50  93.5 Q0.50
3 29.25 104.5 Q0.75
```

Es können auch mehrere Variablen gleichzeitig jeweils mit derselben Funktion aggregiert werden (Abschn. 3.4.11).

3.4.11 Funktionen auf Gruppen von Variablen anwenden

Die Funktionen `rename()`, `filter()`, `mutate()`, `summarise()` und `group_by()` können jeweils dieselbe Transformation auch gleichzeitig auf Gruppen von Variablen anwenden. Details erläutert `vignette("colwise", package="dplyr")`. Die Auswahl von Variablen erfolgt dabei mit denselben Methoden wie bei `select()` (Abschn. 3.4.3) anhand von Namensmustern, Eigenschaften oder über ihre Position im Datensatz.

Variablen umbenennen

Mit `rename_with()` werden Gruppen von Variablen umbenannt. Als zweites Argument hinter dem Datensatz ist dafür die auf die Variablennamen anzuwendende Funktion zu nennen. Das

letzte Argument definiert die ausgewählte Gruppe von Variablen.

Für das zweite Argument kann eine schon bestehende Funktion wie bei `lapply()` (Abschn. 3.3.13) direkt übergeben werden. Benötigt die Funktion aber weitere Argumente, ist der Funktionsaufruf mit Hilfe einer besonderen Syntax mit vorangestellter `~` zu formulieren. Auf diese Weise wird eine *lambda-Funktion* definiert, ohne dass dafür ein Funktionkopf oder den Funktionsrumpf umschließende `{}` notwendig wären (Abschn. 17.3).³⁴ In der angewendeten Funktion steht statt eines konkreten Variablenamens der `.` als Platzhalter für eine der mehreren zu verarbeitenden Variablennamen – und nicht wie innerhalb der Hauptfunktionen für den gesamten Datensatz.

```
# Kleinbuchstaben für Variablen, die mit i / I beginnen
> myDf1 |> rename_with(tolower, matches("^i.+"))
  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 # Ausgabe gekürzt ...

# stelle allen character Vektoren Präfix fac_ voran
> myDf1 |> rename_with(~paste0("fac_", .), where(is.character))
  id fac_sex fac_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 # Ausgabe gekürzt ...
```

Fehlt die Auswahl bestimmter Variablen, werden alle Variablen umbenannt.

```
# konvertiere alle Variablennamen in Großbuchstaben
> myDf1 |> rename_with(toupper)
  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 # Ausgabe gekürzt ...
```

Variablen auswählen

Die Auswahl von Variablengruppen anhand einfacher oder zusammengesetzter Kriterien mit `select()` zeigt Abschn. 3.4.3.

Beobachtungen auswählen

Für das folgende Beispiel zur Auswahl von Beobachtungen mit `filter()` sollen zunächst im verwendeten Datensatz Beobachtungen zufällig auf NA gesetzt werden. In der dazu definierten Funktion (Abschn. 17.3) wird sichergestellt, dass der Datentyp des eingefügten NA Werts gleich dem Datentyp des übergebenen Vektors ist (Abschn. 2.13.1, Fußnote 53).

³⁴Die Verwendung der Tilde zur Definition einer lambda-Funktion ist spezifisch für `dplyr` und andere `tidyverse` Pakete. Sonst markiert sie eine Modellformel (Abschn. 5.2).

```
> getNA <- function(x, p=c(0.7, 0.3)) {
+   NAvail <- x[length(x) + 1] # missing Wert des zu x passenden Typs
+   if_else(sample(c(TRUE, FALSE), length(x), replace=TRUE, prob=p),
+         x, NAvail)
+ }

# füge in Variablen zufällig fehlende Werte passenden Typs ein
> myDf1NA <- myDf1 |> mutate(group=getNA(group),
+                               age=getNA(age),
+                               IQ=getNA(IQ),
+                               rating=getNA(rating))

> myDf1NA
  id sex group age  IQ rating
1  1   f    <NA> NA  NA     1
2  2   m     CG  30 122     3
3  3   m     CG  25  95     5
4  4   m      T  NA 102     5           # Ausgabe gekürzt ...
```

Mit den Hilfsfunktionen `if_any()` sowie `if_all()` lassen sich im Aufruf von `filter()` die zu berücksichtigenden Variablen auswählen und das Auswahlkriterium definieren. Das erste Argument beider Hilfsfunktionen definiert dazu mit den für `select()` (Abschn. 3.4.3) vorgestellten Methoden eine Gruppe von Variablen. Ihr zweites Argument ist eine gewöhnliche Funktion oder eine mit `~` eingeleitete lambda-Funktion, die eine zu prüfende Bedingung definiert. Diese Funktion muss einen Vektor akzeptieren und einen ebenso langen Vektor von Wahrheitswerten zurückgeben. In einer lambda-Funktion steht statt eines konkreten Variablenamens der `.` als Platzhalter für eine der mehreren zu verarbeitenden Variablen, die über das erste Argument definiert sind.

Mit `if_any()` wird eine Beobachtung ausgewählt, wenn die im zweiten Argument übergebene Funktion für mindestens eine der gewählten Variablen den Wert `TRUE` liefert (logisches ODER). Mit `if_all()` muss die definierte Bedingung dagegen auf alle ausgewählte Variablen gleichzeitig zutreffen, damit eine Beobachtung im Ergebnis auftaucht (logisches UND).

```
# nur Beobachtungen mit NA auf >= 1 Variable group bis IQ
> myDf1NA |> filter(if_any(group:IQ, is.na))
  id sex group age  IQ rating
1  1   f    <NA> NA 112     1
2  3   m    <NA> 25  NA    NA
3  4   m    <NA> 34 102     5
4  5   m    <NA> 22  82     2
5  6   f     CG  NA 113    NA
6  7   m      T  28  NA    NA
7  9   m    <NA> 23  88     3
8 11   m     CG  20  NA    NA
9 12   f      T  21  NA     1

# nur Beobachtungen mit NA auf allen Variablen group bis IQ
```

```
> myDf1NA |> filter(if_all(group:IQ, is.na))
[1] id sex group age IQ rating
<0 Zeilen> (oder row.names mit Länge 0)

# nur Beobachtungen, für die alle numerischen Variablen ungerade sind
myDf1NA |> filter(if_all(where(is.numeric), ~(. %% 2) == 1L))
  id sex group age IQ rating
1  3   m    CG  25  95      5
```

Variablen transformieren

Im Aufruf `mutate(across(...))` definiert das erste Argument mit den für `select()` (Abschn. 3.4.3) vorgestellten Methoden eine Gruppe von Variablen. Das zweite Argument ist dann eine gewöhnliche Funktion oder eine mit `~` eingeleitete lambda-Funktion, die die jeweils auf jede Variable anzuwendende Transformation bestimmt. Dabei wird jede ausgewählte Variable mit der transformierten Version überschrieben.

```
# numerische Variablen zentrieren
> myDf1 |> mutate(across(where(is.numeric),
+                           ~scale(., center=TRUE, scale=FALSE)))
  id sex group      age      IQ      rating
1 -5.5   f     T -0.4166667 14.75 -1.5833333
2 -4.5   m     CG  3.5833333 24.75  0.4166667
3 -3.5   m     CG -1.4166667 -2.25  2.4166667 # Ausgabe gekürzt ...
```

Es gibt alternativ die Möglichkeit, Transformationen als Komponenten einer benannten Liste zu definieren. Der Name einer Listenkomponente wird dann mit Unterstrich an den ursprünglichen Namen angehängt und die transformierte Variable unter diesem Namen dem Datensatz hinzugefügt. Auch lambda-Funktionen können dabei Listenkomponenten definieren.

```
> myDf1 |>
+   mutate(across(age:IQ,
+                 list(ctr=~scale(., center=TRUE, scale=FALSE),
+                      scl=~scale(., center=FALSE, scale=TRUE))))
  id sex group age  rating   age_ctr  IQ_ctr   age_scl  IQ_scl
1   1   f     T  26  112      1 -0.4166667 14.75  0.9278723 1.0940173
2   2   m     CG  30  122      3  3.5833333 24.75  1.0706219 1.1916974
3   3   m     CG  25   95      5 -1.4166667 -2.25  0.8921849 0.9279611
# Ausgabe gekürzt ...

# wende Funktion getNA() auf alle Variablen an -> fügt zufällig NA ein
> myDf1 |> mutate(across(everything(), getNA))
  id  sex group age  IQ rating
1   1   f     T  26  112      1
2   2   m    <NA> 30  122      3
3  NA <NA> <NA>  NA   95     NA # Ausgabe gekürzt ...
```

Daten getrennt nach Gruppen auswerten

`group_by(across(...))` erlaubt es, die zum Gruppieren eines Datensatzes verwendeten Variablen wie mit `select()` (Abschn. 3.4.10) auf Basis eines Namensmusters, der Position im Datensatz oder ihrer Eigenschaften auszuwählen.

```
# gruppiere auf Basis aller mit s / S beginnenden Variablen
> myDf1 |> group_by(across(starts_with("s"))) |>
+   summarise(M_IQ=mean(IQ))
# A tibble: 2 x 2
  sex     M_IQ
  <chr> <dbl>
1 f       108.
2 m       93.8

# gruppiere auf Basis aller character Vektoren
> myDf1 |> group_by(across(where(is.character))) |>
+   summarise(M_IQ=mean(IQ))
# A tibble: 5 x 3
# Groups:   sex [2]
  sex   group  M_IQ
  <chr> <chr> <dbl>
1 f     CG      113
2 f     T       105
3 m     CG      103
4 m     T       97 
5 m     WL      85.2
```

Daten aggregieren

Bei der Auswahl zu aggregierender Variablen (Abschn. 3.4.10) mit `summarise(across(...))` ist darauf zu achten, dass sich die definierten Kennwerte sinnvoll auf die Variablen-Typen anwenden lassen.

```
# Anzahl eindeutiger Werte by sex und rating
> vars_sel <- c("sex", "rating")
> myDf1 |> group_by(group) |>
+   summarise(across(all_of(vars_sel), n_distinct))
# A tibble: 3 x 3
  group   sex rating
  <chr> <int> <int>
1 CG      2     4
2 T       2     3
3 WL     1     3
```

Wie bei `mutate()` eignet sich eine benannte Liste, um die erstellten Variablen geeignet zu benennen. Der Name einer Listenkomponente wird an den ursprünglichen Variablennamen mit Unterstrich angehängt.

```
> myDf1NA |> group_by(sex) |>
+   summarise(across(where(is.numeric),
+                     list(median=~median(., na.rm=TRUE))))
```

A tibble: 2 x 5

	sex	id_median	age_median	IQ_median	rating_median
	<chr>	<dbl>	<dbl>	<dbl>	<dbl>
1	f	6	22.5	106.	1
2	m	7	28	93.5	3

3.4.12 Häufigkeiten bestimmen

Häufigkeitsauszählungen mit `xtabs()` und `proportions()` (Abschn. 2.12.3) fügen sich nicht gut in pipe-basierte Arbeitsschritte mit `dplyr` ein. Eine Alternative ist die pipe-kompatible Funktion `count(<Faktor1>, <Faktor2>, ...)` analog zu `as.data.frame(xtabs(...))` (Abschn. 2.12.5). Die Stufenkombinationen der durch Komma getrennt übergebenen Faktoren bestimmen die Definition der Gruppen, deren absolute Häufigkeit gezählt wird.

```
> library(dplyr)          # für count(), add_count()
> myDf1 |> count(sex, group) # Häufigkeit Kombinationen von sex, group
# A tibble: 5 x 3
  sex   group     n
  <chr> <chr> <int>
1 f      CG       1
2 f      T        2
3 m      CG       3
4 m      T        2
5 m      WL       4

> as.data.frame(xtabs(~ sex + group, data=myDf1))      # Vergleich
  sex group Freq
1  f    CG     1
2  m    CG     3
3  f    T      2
4  m    T      2
5  f    WL     0
6  m    WL     4
```

In der Voreinstellung `.drop=TRUE` werden nur die tatsächlich vorkommenden Stufenkombinationen gezählt. Mit `.drop=FALSE` tauchen auch die prinzipiell möglichen, tatsächlich aber nicht beobachteten Stufenkombinationen mit der Häufigkeit 0 im Ergebnis auf.

```
> myDf1 |> count(sex, group, .drop=FALSE)      # f & WL -> Häufigkeit 0
# A tibble: 6 x 3
```

```
sex   group   n
<chr> <chr> <int>
1 f     CG      1
2 f     T       2
3 f     WL      0
4 m     CG      3
5 m     T       2
6 m     WL      4
```

`add_count(<Faktor1>, ..., name="n")` führt dieselbe Zählung durch, fügt jedoch jeder Beobachtung im ursprünglichen Datensatz die absolute Häufigkeit der zugehörigen Gruppe in der neuen Variable `n` hinzu. Dieser Variablenname lässt sich über die Option `name` frei wählen.

```
> myDf1 |> add_count(sex, group)
# A tibble: 12 x 7
  id sex   group   age    IQ rating     n
  <dbl> <chr> <chr> <dbl> <dbl> <dbl> <int>
1 1   f     T      26   112     1     2
2 2   m     CG     30   122     3     3
3 3   m     CG     25   95      5     3 # ...
```

Durch anschließende Berechnungen innerhalb von `mutate()` erhält man auch relative Häufigkeiten analog zu `proportions()`.

```
> myDf1 |> count(sex, group) |>
+   mutate(freq_rel=n / sum(n))           # relative Häufigkeiten
# A tibble: 5 x 4
  sex   group   n freq_rel
  <chr> <chr> <int>    <dbl>
1 f     CG      1 0.0833
2 f     T       2 0.167
3 m     CG      3 0.25
4 m     T       2 0.167
5 m     WL      4 0.333

> myDf1 |> add_count(sex, group) |>          # relative Häufigkeiten
+   mutate(freq_rel=n / n())
# A tibble: 12 x 8
  id sex   group   age    IQ rating     n freq_rel
  <dbl> <chr> <chr> <dbl> <dbl> <dbl> <int>    <dbl>
1 1   f     T      26   112     1     2     0.167
2 2   m     CG     30   122     3     3     0.25
3 3   m     CG     25   95      5     3     0.25 # ...
```

Bedingte relative Häufigkeiten müssen mit zusätzlichen Zwischenschritten manuell berechnet werden.

```
> myDf1 |> count(sex, group, name="n_sex_group") |>
+   group_by(sex) |>
```

```

+   mutate(n_sex=sum(n_sex_group),
+         freq_cond_rel=n_sex_group / n_sex) |>
+   ungroup()
# A tibble: 5 x 5
  sex   group n_sex_group n_sex freq_cond_rel
  <chr> <chr>      <int> <int>        <dbl>
1 f     CG          1     3       0.333
2 f     T           2     3       0.667
3 m     CG          3     9       0.333
4 m     T           2     9       0.222
5 m     WL          4     9       0.444

# Kontrolle
> proportions(xtabs(~ sex + group, data=myDf1), margin="sex")
  group
sex      CG          T          WL
f 0.3333333 0.6666667 0.0000000
m 0.3333333 0.2222222 0.4444444

> myDf1 |> add_count(sex, name="n_sex") |>                      # Häufigkeiten sex
+   add_count(sex, group, name="n_sex_group") |> # " sex + group
+   mutate(freq_cond_rel=n_sex_group / n_sex) |> # bed. rel. Hfgk.
+   select(id, sex, group, n_sex, n_sex_group, freq_cond_rel)
# A tibble: 12 x 6
  id sex   group n_sex n_sex_group freq_cond_rel
  <dbl> <chr> <chr> <int>      <int>        <dbl>
1 1   f     T       3        2       0.667
2 2   m     CG      9        3       0.333
3 3   m     CG      9        3       0.333
4 4   m     T       9        2       0.222
# ...

```

Kapitel 4

Zuverlässige und reproduzierbare Datenauswertung

R bietet viele Möglichkeiten, die dabei helfen, Datenauswertungen zuverlässig und reproduzierbar zu machen. Dazu gehört es, mit automatisierbaren Befehlsskripten zu arbeiten (Abschn. 4.1). In den vergangenen Jahren ist zunehmend auch die Integration von Datenanalyse und schriftlicher Dokumentation in einem gemeinsamen Dokument in den Fokus gerückt (Abschn. 4.2). Besondere Lösungen sind notwendig, um ein hohes Niveau an exakter Reproduzierbarkeit der Ergebnisse herstellen zu können (Abschn. 4.4).

4.1 Befehlssequenzen im Editor bearbeiten

Für Datenanalysen, die über wenige Teilschritte hinausgehen, ist eine interaktive Arbeitsweise direkt auf der Konsole 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.

Um Befehlsskripte zu erstellen, bietet RStudio (Abschn. 1.1.4) einen Texteditor, der sich über den Menüeintrag *File* → *New File* → *R Script* öffnen lässt und daraufhin bereit für die Eingabe von Befehlszeilen ist. Der Texteditor erstellt automatisch zu jeder öffnenden Klammer eine passende schließende und hebt R-Befehle farblich hervor (*syntax-highlighting*). Mit der Tastenkombination **Strg+Return** wird der Befehl in derjenigen Zeile von R ausgeführt, in der sich die Eingabemarke gerade befindet (icon *Run*). Um in einem Schritt mehrere Befehlszeilen auswerten zu lassen, markiert man diese und führt sie ebenfalls mit **Strg+Return** aus. Um das komplette Skript ausführen zu lassen, kann das icon *Source* verwendet werden. Verursacht einer der auszuführenden Befehle eine Fehlermeldung, unterbricht dies die Verarbeitung nicht. Folgende Befehle werden dann also trotzdem ausgewertet. Einfache Warnungen werden gesammelt am Schluss aller Ausgaben genannt. Befehlsskripte lassen sich über den Menüeintrag *File* → *Save As...* speichern und über *File* → *Open File...* laden.

In externen Dateien gespeicherte Skripte lassen sich in der Konsole mit `source("<Dateiname>")` in einem Schritt komplett einlesen, wobei R die enthaltenen Befehle ausführt.¹ 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

¹ Es ist nicht notwendig, R im interaktiven Modus zu starten, um ein Befehlsskript ausführen zu lassen, dafür existiert auch ein *Batch-Modus*, vgl. `?Rscript`.

² Wird das einzulesende Skript nicht gefunden, ist zunächst mit `dir()` zu prüfen, ob das von R durchsuchte Verzeichnis (ohne explizite Angabe eines Pfades ist es das mit `getwd()` angezeigte Arbeitsverzeichnis) auch tatsächlich das Skript enthält.

dabei das Argument `echo=TRUE` gesetzt, werden die im Skript enthaltenen Befehle selbst mit auf der Konsole ausgegeben, andernfalls erscheint nur die Ausgabe dieser Befehle. Gegenüber der interaktiven Arbeitsweise bieten Skripte u. a. folgende Vorteile:

- Der Überblick über alle auszuführenden Befehle wird erleichtert, zudem können die Auswertungsschritte vorgeplant bzw. später gedanklich nachvollzogen werden.
- Teilschritte komplexer Auswertungen können einzeln nacheinander oder in größeren Blöcken ausgeführt werden, um Zwischenergebnisse auf ihre Richtigkeit zu prüfen.
- Ein für die Auswertung eines Datensatzes erstelltes Skript lässt sich häufig mit nur geringen Änderungen für die Analyse anderer Datensätze anpassen und erweitern. Diese Wiederverwendbarkeit einmal geleisteter Arbeit ist bei rein grafisch zu bedienenden Programmen nicht gegeben und spart auf längere Sicht Zeit. Zudem vermeidet eine solche Vorgehensweise Fehler, wenn bewährte Befehlssequenzen eingesetzt werden.
- Ein erstelltes Skript dokumentiert gleichzeitig die Auswertungsschritte und unterstützt damit die Reproduzierbarkeit der gewonnenen Ergebnisse statistischer Analysen (Abschn. 4.4).
- Skripte lassen sich zusammen mit dem Datensatz, für dessen Auswertung sie gedacht sind, an Dritte weitergeben. Neben dem Aspekt der so möglichen Arbeitsteilung kann die Auswertung von anderen Personen nachvollzogen und kontrolliert werden. Dies ist i. S. einer größeren Auswertungsobjektivität sinnvoll.³
- Es 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.

In RStudio können Skripte mit Hilfe besonders formatierter Kommentare in Abschnitte strukturiert werden. Jeder Kommentar, der mit mindestens 4 Bindestrichen oder `#` Symbolen endet, markiert dabei den Beginn eines neuen Abschnitts. Über die Gliederungsansicht von RStudio können die Abschnitte dann zur schnellen Navigation in langen Skripten verwendet werden (Abb. 4.1).

4.2 Dokumente erstellen

4.2.1 Grundprinzip

Mit den Paketen `knitr` (Xie, 2020a) und `rmarkdown` (J. J. Allaire, Xie et al., 2020) lassen sich R-Auswertungen in Dokumente einbetten, die beschreibenden Text, R-Befehle und die zugehörigen Ergebnisse dieser Befehle integrieren. Unterstützte Dateiformate sind PDF, Word

³Da R mit `shell()` auch auf Funktionen des Betriebssystems zugreifen kann, sollten aus Sicherheitsgründen nur geprüfte Skripte aus vertrauenswürdiger Quelle ausgeführt werden.

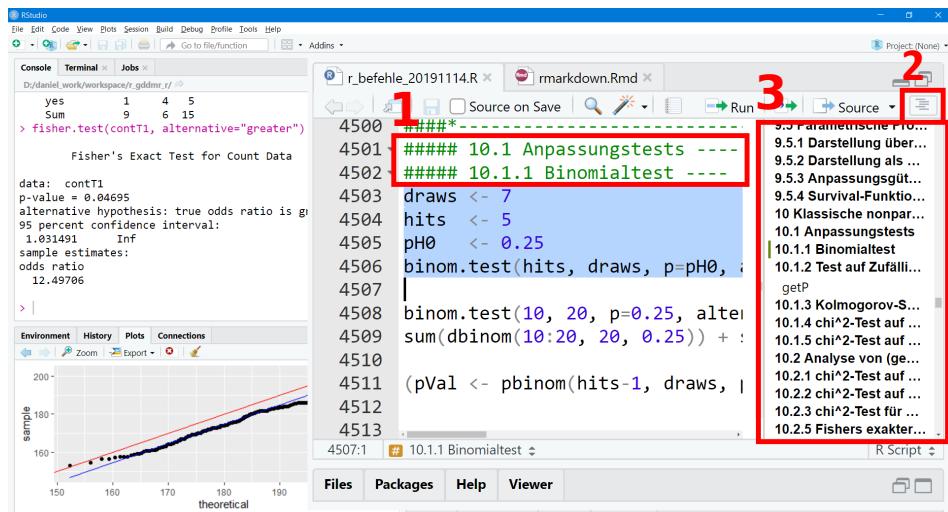


Abbildung 4.1: RStudio Gliederungsansicht für lange Befehlsskripte (vergrößert und rot umrandet). 1: Kommentar mit ---- am Schluss als Abschnittsnamen kennzeichnen. 2: Gliederungsansicht aktivieren. 3: Gliederungsansicht

und HTML. Auch Tabellen und Diagramme können Teil des Dokuments sein. Ziel solcher integrierten Dokumente ist es, die Datenauswertung und Berichterstellung in einem Arbeitsschritt zu vereinen.⁴ Die Datenanalyse kann so effizienter, besser nachvollziehbar, leichter replizierbar und weniger fehleranfällig werden (Abschn. 4.4). Umfassende Dokumentation zu `knitr` findet man auf der Homepage des Autors, für `rmarkdown` sind das RStudio *cheat sheet* und der *reference guide* beim Einstieg hilfreich.⁵ Xie (2015) und Xie et al. (2018) geben eine ausführliche Beschreibung, viele Beispiele liefert Xie et al. (2020).

Der Grundtext von R-Dokumenten wird wie R-Skripte im Textformat geschrieben, wobei Textauszeichnungen, etwa als Überschrift oder Aufzählung, durch bestimmte Schlüsselsymbole vorzunehmen sind. Die Formatierung kann dabei sehr detailliert in L^AT_EX-Syntax erfolgen (Datei mit Endung `.Rnw`), oder beschränkt auf einfache Merkmale im *Markdown*-Format (Datei mit Endung `.Rmd`). In beiden Fällen erhält der Autor nicht unmittelbar Zugriff auf jedes Formatierungsdetail wie etwa die Schriftgröße oder die Weite der Einrückung bei Aufzählungen. Die Aufgabe des Autors ist es stattdessen, die inhaltliche Bedeutung besonderer Textelemente festzulegen. Die Formatierungsdetails werden dann aus vom Dokumenttyp abhängigen Formatvorlagen übernommen. Das Ziel ist es, den Autor von weniger wichtigen Aspekten zu entlasten, um einen Fokus auf die Inhalte zu unterstützen. Ein besonderes Ziel des *Markdown*-Formats ist es, dass die wenigen Textauszeichnungen schnell erlernbar sind und sie die Lesbarkeit des Grundtexts nicht wesentlich beeinträchtigen.

Aus demselben Grundtext kann man flexibel ebenso ein PDF- oder Word-Dokument wie auch eine HTML-Seite erzeugen. Dafür ist zusätzlich weitere Software auf dem Computer erforderlich: Für PDF-Dokumente ist eine separate L^AT_EX-Installation notwendig, auch wenn der Text

⁴Ein vergleichbarer Ansatz sind Jupyter Notebooks (Kluyver et al., 2016), die ebenfalls auf Dokumenten in *Markdown*-Syntax basieren und sich etwa mit der JupyterLab Entwicklungsumgebung erstellen lassen (Abschn. 1.1.4).

⁵<https://yihui.org/knitr/> – <https://posit.co/resources/cheatsheets/>

im Markdown-Format geschrieben ist.⁶ Für Word-Dokumente wird das freie Programm pandoc ([McFarlane, 2024](#)) benötigt, das RStudio bereits mitliefert. Eine detaillierte Darstellung der Vielzahl erzeugbarer Dokumenttypen enthält [Xie et al. \(2018\)](#). Die folgende Auswahl von Zusatzpaketen deutet die Breite des Spektrums verfügbarer Formatvorlagen an:

- Brief: `linl` (<http://dirk.eddelbuettel.com/code/linl.html>) und `komalemma` (<https://github.com/rnuske/komalemma>)
- Zeitschriftenartikel: `rticles` (<https://github.com/rstudio/rticles>) und `pinp` (<http://dirk.eddelbuettel.com/code/pinp.html>)
- Präsentation: `revealjs` (<https://bookdown.org/yihui/rmarkdown/revealjs.html>) und `ioslides` (<https://bookdown.org/yihui/rmarkdown/ioslides-presentation.html>)
- Buch: `bookdown` (<https://bookdown.org/yihui/bookdown/>)
- Lebenslauf: `vitae` (<https://pkg.mitchelloharawild.com/vitae/>)

4.2.2 Arbeitsschritte

Ein Quelldokument mit R-Befehlen und Fließtext wird durch drei Arbeitsschritte zum fertigen Zieldokument. Die Funktion `render()` aus dem Paket `rmarkdown` kombiniert diese Arbeitsschritte in einem Aufruf:

1. Die Funktion `knit()` aus dem Paket `knitr` generiert zunächst zu jedem R-Befehl die zugehörige Ausgabe, die normalerweise auf der Konsole oder als Diagramm angezeigt wird.
2. Als Zwischenschritt fügt `knit()` anschließend jede von R erzeugte Ausgabe an der passenden Stelle zusammen mit den übrigen Textelementen in ein Dokument im reinen Markdown-Format (Dateiendung `.md`) oder im reinen L^AT_EX-Format (Dateiendung `.tex`) ein.
3. Aus der Markdown- oder L^AT_EX-Datei lässt sich Schließlich als Zieldokument eine HTML-Datei oder (mit einer externen L^AT_EX-Installation) ein PDF erzeugen. Für Word-Dokumente übernimmt `pandoc` diese Konvertierung.

Die Funktionalität von `knitr` und `rmarkdown` lässt sich mit den genannten R-Funktionen nutzen, die Quelldokumente verarbeiten und das erzeugte Zieldokument im gewünschten Format speichern. Einfacher ist es jedoch, dafür RStudio zu verwenden, wo alle Arbeitsschritte in der Oberfläche integriert sind (Abb. 4.2). Das Menü *Help → Markdown Quick Reference* bietet eine Übersicht der wichtigsten Formatierungsmöglichkeiten in Markdown-Syntax. Über den Menüeintrag *File → New File → R Markdown...* lässt sich eine neue Datei im Markdown-Format erstellen, die bereits ein typisches Grundgerüst enthält.

Das icon *Knit* (Abb. 4.2, Bereich 1) erzeugt das Zieldokument, wofür in Schritt 1 alle im Quelldokument vorhandenen R-Befehle in einer separaten R-session ausgeführt werden, um die Ausgabe zu extrahieren. Diese neue session hat keinen Zugriff auf die Objekte, die sich

⁶Eine Möglichkeit dafür bietet das Paket `tinytex` ([Xie, 2019, 2020b](#)) mit `install_tinytex()`, allgemein <https://www.latex-project.org/get/>

durch vorherige interaktive Auswertungen im workspace angesammelt haben. Alle benötigten Daten müssen durch Befehle im Quelldokument explizit geladen bzw. erzeugt werden. Dieses Vorgehen von RStudio dient dazu, die Ergebnisse im erzeugten Dokument unabhängig vom Zustand des workspace und damit besser reproduzierbar zu machen.

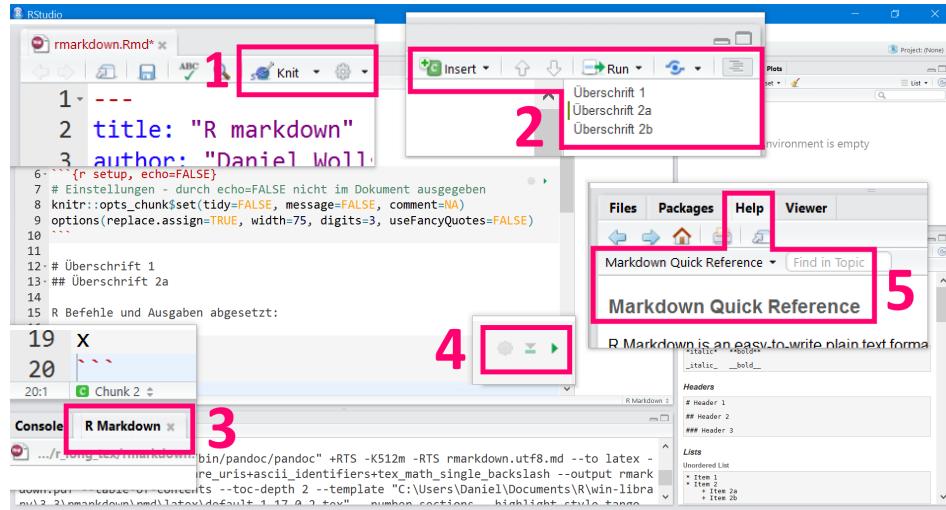


Abbildung 4.2: Werkzeuge für R-Dokumente in der Entwicklungsumgebung RStudio (vergrößert und rot umrandet). 1: Zieldokument erzeugen mit icon *Knit* inkl. Zugang zu Optionen. 2: Gliederungsansicht. 3: Reiter mit Ausgabe beim Erzeugen des Zieldokuments. 4: Werkzeuge zum Ausführen einzelner R-Befehlsblöcke. 5: Markdown Hilfe

4.2.3 Aufbau eines Quelldokuments

Ein R-Markdown Dokument beginnt mit dem *front matter* in *YAML-Syntax*.⁷ Dies sind oberhalb und unterhalb von --- eingeschlossene Zeilen, die Metadaten des Dokuments wie Angaben zum Autor, zum Titel oder zum Datum definieren. Auch lassen sich Optionen für Inhalte des Zieldokuments setzen, etwa die Darstellung eines Inhaltsverzeichnisses. Zugang zu diesen Optionen bietet RStudio auch im Zahnrad-icon (Abb. 4.2, Bereich 1). Das YAML-Format besteht aus <Variable>: <Wert> Zuweisungen, wobei die Einrückung durch Leerzeichen wichtig ist. Word-Dokumente lassen sich etwa mit `reference_docx: <Pfad/Datei>` auf einer Vorlage basieren, deren Formate für Fließtext und Überschriften ebenso übernommen werden wie etwa Seitenzahlen in der Fußzeile. Die Werte einer im Front Matter definierten Variable `params` sind im weiteren Dokument als Liste dieses Namens verfügbar. Auf diese Weise können Parameter zu Beginn des Dokuments gesetzt werden, die spätere Auswertungen steuern.

```
---
title: "R markdown"
author: "Daniel Wollschläger"
output:
  word_document:
```

⁷<https://yaml.org/>

```
toc: TRUE
```

Fließtext kann neben einfachen Beschreibungen auch Aufzählungen enthalten, Bilddateien einbinden oder Tabellen definieren. Das unten stehende Beispiel demonstriert die dafür notwendige Syntax.

R-Befehle können entweder in abgesetzten Befehlsblöcken (*chunks*) stehen, die mit einer Zeile ````{r}` beginnen und mit einer Zeile ````` enden. Oder sie werden im Text eingebettet (*inline*), wobei die sie in ``r <Befehle>`` eingeschlossen sein müssen.⁸

Für jeden Befehlsblock lässt sich über durch Komma getrennte Optionen separat steuern, wie er durch `knit()` behandelt wird.⁹ So können R-Befehle etwa in der Ausgabe wiederholt oder aber ausgeblendet werden, so dass nur ihr Ergebnis sichtbar ist (Tab. 4.1). Ein Block kann auch ausgeführt werden, dabei im Zieldokument aber samt Ausgabe unsichtbar bleiben – etwa um zu Beginn Pakete und Daten zu laden oder vorbereitende Berechnungen durchzuführen. Global für alle Befehlsblöcke eines Dokuments lassen sich diese Optionen zu Beginn mit `knitr::opts_chunk$set()` festlegen.

Tabelle 4.1: Auswahl von `knitr chunk options` für *Markdown*-Dokumente, um die Ausgabe von Befehlen und output zu steuern

Chunk Option	führt Befehle aus	zeigt Befehle	zeigt output	zeigt Diagramme
<code>results="hide"</code>	ja	ja	nein	ja
<code>include=FALSE</code>	ja	nein	nein	nein
<code>echo=FALSE</code>	ja	nein	ja	ja
<code>fig.show="hide"</code>	ja	ja	ja	nein
<code>eval=FALSE</code>	nein	ja	nein	nein

4.2.4 Beispiel

Das folgende Beispieldokument in Markdown-Syntax soll die wichtigsten Gestaltungsmöglichkeiten demonstrieren.

```
---
```

```
title: "R markdown"
subtitle: "Beschreibung"
author: "Daniel Wollschläger"
date: "`r format(Sys.time(), '%d.%m.%Y')`"
output:
  word_document:
    toc: TRUE
```

⁸In L^AT_EX-Dokumenten beginnen Befehlsblöcke mit `<>>=` und enden mit `@`, im Text eingebettete R-Befehle sind in `\Sexpr{}` einzuschließen. Inline R-Befehle sind auch im YAML front matter möglich, wenn sie in Anführungszeichen stehen. Weitere Minimalbeispiele zeigt <https://yihui.org/knitr/demo/minimal/>

⁹<https://yihui.org/knitr/options/>

```
reference_docx: "template.docx"
params:
  par_a: 193
  par_b: "parameter value"
---

```{r setup, include=FALSE}
Einstellungen - durch include=FALSE nicht im Dokument ausgegeben
knitr::opts_chunk$set(tidy=FALSE,
 message=FALSE,
 comment=NA,
 fig.width=6,
 fig.asp=0.618,
 fig.align="center")

options(replace.assign=TRUE,
 width=75,
 digits=3,
 useFancyQuotes=FALSE)
```

# Überschrift 1
## Überschrift 1a
```

R-Befehle und Ausgaben als abgesetzter Block (chunk):

```
```{r}
x <- 5
x
```
```

Im Text integrierte Ausgabe eines R-Befehls:
Wert des gerade erstellten Objekts `x` ist `r x`

Überschrift 1b

Diagramm - mit Angabe zur eingenommenen Breite, Bildauflösung
und Beschriftung

```
```{r out.width="70%", dpi=300, fig.cap='Bildunterschrift'}
plot(mpg ~ hp, data=mtcars, pch=19)
```
```

Textformatierungen: *kursiv*, **fett**, hoch^gestellt^, tief~gestellt~,
`Schreibmaschine`, Zeilenumbruch durch 2 Leerzeichen am Zeilenende

[Hyperlink] (<https://www.r-project.org/>)

- * Liste - Leerzeile vorher und nachher notwendig
 - a. Eine Ebene tiefer - a, b, c automatisch
 - i. Noch eine Ebene tiefer - i, ii, iii automatisch
 - ii. ABC
 - a. DEF
- * Listenelement

1. Geordnete Liste - Numerierung wird automatisch angepasst
 1. Eine Ebene tiefer
 1. ABC
1. Listenelement

> Blockquote
 > abgesetzt
 > formatiert

Tabelle - Leerzeile vorher und nachher notwendig

| rechts | links | normal | zentriert |
|--------|-------|--------|-----------|
| 47.5 | FRA | 0.23 | SFO |
| 39.2 | BGO | 0.07 | LAX |

```
# Überschrift 2
## Verwendung von Parametern aus *YAML front matter*

Param "par_a" ist: `r params$par_a`, "par_b" ist: `r params$par_b`

## Bilder einbinden

! [Alternativtext] (regrMult.png)
```

Mathematische Formeln in L^AT_EX-Syntax müssen im Fließtext in $\langle Formel \rangle$ und abgesetzt in $\langle Formel \rangle$ eingeschlossen sein. In HTML-Dokumenten werden diese Formeln automatisch mit Hilfe der JavaScript-Bibliothek MathJax dargestellt, in Word-Dokumenten durch den Formel-Editor. In Word-Dateien kann es dabei zu Problemen mit komplexen Formeln kommen.

Formel im Fließtext: $\mu = \beta_0 + \beta_1 X_1$

Abgesetzte Formel:

```
$$
\text{OR} = \frac{a \cdot d}{b \cdot c}
$$
```

Datensätze können mit `kable(<Datensatz>)` aus dem Paket `knitr` so ausgegeben werden, dass sie im erstellten Dokument als echte Tabelle formatiert sind. Dafür ist es einerseits notwendig, die chunk Option `results="asis"` zu verwenden, da `kable()` bereits Text im Markdown-Format ausgibt, der durch `knit()` nicht weiter konvertiert werden soll. Andererseits muss die Quelldatei `knitr` explizit mit `library()` laden bzw. den Aufruf in der Form `knitr::kable()` durchführen. Die so erstellten Tabellen sind schmucklos. Weit ansprechender lassen sie sich mit Funktionen der Pakete `flextable` (Gohel, 2020) und `gt` (Iannone et al., 2024) formatieren.

```
```{r results="asis"}
knitr::kable(head(mtcars[, c("mpg", "cyl", "hp")], n=3))
````
```

4.2.5 Quarto

Quarto¹⁰ ist eine Software um Dokumente zu erstellen, die das Konzept von R-Markdown erweitert. Das Grundprinzip bleibt dabei weitgehend dasselbe: Ausgangspunkt ist wieder ein Textdokument mit YAML front matter, mit Markdown formatiertem Text und eingebetteten Befehlen. Über dieselben Zwischenschritte wird daraus ein PDF, Word- oder HTML-Dokument erstellt, das die Ergebnisse der Berechnungen mit Diagrammen und Tabellen einbettet.

Als wesentlicher Unterschied zu R-Markdown ist Quarto eine eigenständige Software, die von R unabhängig ist. Sie lässt sich also auch etwa ausschließlich mit den Programmiersprachen Python oder Julia verwenden. Das bietet den Vorteil, dass an Quarto-Dokumenten Personen gemeinsam arbeiten können, die unterschiedliche Programmiersprachen nutzen. Weiterhin integriert Quarto Formatvorlagen für eine Reihe von Dokumenttypen wie Bücher, Präsentationen oder Blogs, weshalb es nicht notwendig ist, Zusatzpakete wie `bookdown`, `blogdown` oder `revealjs` zu installieren.

RStudio integriert Quarto sehr eng, so entfällt etwa die sonst notwendige Installation der eigenständigen Quarto-Software. Auch muss das R-Zusatzpaket `quarto` (J. Allaire & Dervieux, 2024) nicht gesondert installiert werden, das die Kommunikation an der Schnittstelle zwischen R und der Quarto-Software übernimmt. Als Besonderheit lassen sich Quarto-Dokumente in RStudio im Markdown Quelltext, aber auch über einen visuellen Editor erstellen, der Formatierungen über icons erlaubt und den Text entsprechend darstellt. Der Editor verfügt weiterhin über Menü-geführte Dialoge, um Markdown-Elemente wie Bilder, Tabellen oder Formeln einzufügen.

Weitere kleine Unterschiede von Quarto zu R-Markdown sind diese:

- Die Dateiendung von Quarto-Dokumenten ist `.qmd`
- In RStudio trägt das icon zum Erstellen des finalen Dokuments die Bezeichnung *Render*.
- Globale chunk Optionen für die Ausführung von R Befehlen lassen sich über das YAML front matter im Punkt `execute`: definieren, müssen also nicht über Funktionen aus dem Paket `knitr` gesetzt werden.

¹⁰<https://quarto.org/> mit Informationen zur Installation unter <https://quarto.org/docs/get-started/>.

- Die Syntax für die Definition von chunk spezifischen Optionen weicht von R-Markdown ab: Sie werden innerhalb des chunks in separaten, mit #| eingeleitet Zeilen aufgeführt.
- Die Unterstützung für Querverweise mit @, etwa um Abbildungen referenzieren zu können, ist bereits ohne weiteres Zusatzpaket vorhanden.

```
---
```

```
title: "Quarto Dokument"
author: "Daniel Wollschläger"
format: docx
fig-width: 6
fig-asp: 0.618
fig-align: center
out-width: "70%"
execute:
  warning: false
  message: false
---
```

```
```{r setup}
#| include: false
options(replace.assign=TRUE, width=75, digits=3, useFancyQuotes=FALSE)
```

# Überschrift 1
```

Abbildung mit Quarto-spezifischer Syntax für chunk Optionen und label, das später referenziert werden kann.

```
```{r}
#| label: fig-histogram
#| fig-cap: Bildunterschrift
x <- rnorm(100)
hist(x, breaks="FD")
```
```

Abbildung @fig-histogram zeigt ein Histogramm normalverteilter Daten.

Für Nutzer, die ausschließlich mit R und RStudio arbeiten, ergeben sich keine wesentlichen Konsequenzen aus der Wahl, ob Dokumente auf Basis von R-Markdown oder Quarto verfasst werden. Wer dagegen an Dokumenten gemeinsam mit Personen arbeitet, die mit Python entwickeln, kann von Quarto profitieren. In jedem Fall ist ein Wechsel zwischen R-Markdown und Quarto-spezifischer Markdown-Syntax unaufwendig.

4.3 Datenqualität prüfen

Nachdem Daten in R importiert wurden, sollten Sie auf ihre Qualität geprüft werden. Wichtige Kriterien einer hohen Datenqualität sind folgende:

- Einheitlichkeit der Codierung, insbesondere bei
 - Angaben zu Datum oder Uhrzeit (Abschn. 2.15): Aufgrund der Vielzahl national wie international unterschiedlicher Formatierungsmöglichkeiten sind diese Variablen beim Datenaustausch besonders fehlerträchtig.
 - Eigennamen: Vor allem Umlaute, Bindestriche bei Doppelnamen, mehrere Vornamen, Namenszusätze wie „de“, „von“ sowie Groß- und Kleinschreibung können dafür sorgen, dass Probleme beim Datenimport oder beim Zusammenführen mehrerer Datensätze mit Beobachtungen derselben Personen auftreten. Sie lassen sich mit Methoden zur Verarbeitung von Zeichenketten lösen, die in Abschn. 2.14 beschrieben sind.¹¹
 - Physikalischen Variablen: Hier ist bei der Integration mehrerer Datensätze darauf zu achten, dass dieselben physikalischen Einheiten verwendet werden.
- Vollständigkeit: Werden etwa Datensätze mit `merge()` zusammengeführt (Abschn. 3.3.9), besteht die Gefahr, dass aufgrund uneinheitlicher Codierung Einträge fälschlicherweise nicht als übereinstimmend gewertet und deshalb Personen vollständig gelöscht werden. Deshalb sollte nach Möglichkeit anhand der Menge der eindeutigen Personen-IDs sichergestellt werden, dass keine Personen bei der Datenaufbereitung verloren gehen – z. B. mit `setdiff(<IDs vor>, <IDs nach>)`.
- Richtigkeit:
 - Falsche Werte können etwa aus einer fehlerhaften Messung oder aus Tippfehlern bei der Eingabe herrühren. Deshalb sollte die Verteilung aller Variablen mit deskriptiven Kennwerten ebenso wie mit Diagrammen auf ihre Plausibilität geprüft werden. Relevant sind dabei z. B. die Verteilungsform, Werte außerhalb des möglichen Messbereichs und Ausreißer. Siehe Abschn. 2.7 für die deskriptive Beschreibung kontinuierlicher Größen sowie Abschn. 14.6 und 14.7 für Möglichkeiten, ihre (gemeinsame) Verteilung in Diagrammen zu veranschaulichen. Kategoriale Variablen lassen sich durch ihre (gemeinsamen) Häufigkeitsverteilungen mit den in Abschn. 2.12 erläuterten Mitteln charakterisieren. Die zugehörigen Säulendiagramme sind in Abschn. 14.4 beschrieben. Abschnitt 6.5.1 zeigt, wie Extremwerte und Ausreißer im Rahmen der Regressionsdiagnostik identifiziert werden können.
 - Fehlende Werte werden in verschiedenen Programmen uneinheitlich codiert, etwa mit besonderen Zahlen wie 999. Fehlende Werte müssen in R auf `NA` umcodiert werden, damit sie nicht fälschlicherweise in die Auswertung einbezogen werden (Abschn. 2.13).

¹¹Für die Identifizierung fast genau passender Zeichenketten vgl. die dortige Fußnote 61 sowie allgemein für Ansätze zum *record linkage* das Paket `fastLink` (Enamorado, Fifield & Imai, 2018).

- Eindeutigkeit: Ob beim Zusammenführen von Daten aus mehreren Quellen doppelte Fälle auftreten, kann wie in Abschn. 3.3.4 gezeigt untersucht und ggf. behoben werden.

4.4 Reproduzierbare Auswertungen sicherstellen

Ergebnisse statistischer Auswertungen sollten unabhängig von willkürlichen Randbedingungen sein und sich durch andere Personen auf anderen Computern unverändert reproduzieren lassen. Die technische Reproduzierbarkeit statistischer Auswertungen ist ein wichtiger Baustein, um allgemein die Reproduzierbarkeit wissenschaftlicher Untersuchungen zu sichern.¹² Die befehlsgesteuerte Arbeitsweise von R liefert eine zentrale Grundlage, um dieses Ziel zu erreichen (Gandrud, 2020). Denn sie erlaubt es, die Auswertungsschritte exakt zu dokumentieren und an Dritte zu kommunizieren. Diese Eigenschaft reicht jedoch nicht aus – wenn Datenanalysen wiederholt werden, können trotz gleicher Daten und Befehle durch eine Reihe von Einflüssen abweichende Ergebnisse resultieren oder Fehler auftreten.

Als Voraussetzung, um diesen potentiellen Problemen begegnen zu können, ist es zunächst wichtig, die gesamte Ausführungsumgebung zu dokumentieren. Dazu gehört das Betriebssystem, die R-Version, die in der R-session geladenen und durch die Befehle aktiv verwendeten Zusatzpakete mit ihren Versionsnummern sowie die Version des Befehlsskripts. Details zur Systemumgebung liefern die Befehle `Sys.info()` und `.Platform`. Informationen zur installierten R-Version gibt `R.Version()` aus. Schließlich erfährt man mit `sessionInfo()`, welche Zusatzpakete mit welcher Version geladen sind. Das Ziel ist es dann, eine gegebene Auswertungsumgebung wiederholbar herstellen zu können.

4.4.1 Potentielle Probleme und Maßnahmen

Die Reproduzierbarkeit statistischer Analysen ist eine Eigenschaft, die mehr oder weniger stark ausgeprägt sein kann. Je nach angestrebtem Grad der Reproduzierbarkeit können unterschiedlich aufwendige technische Maßnahmen getroffen werden, um Einflüsse auf die Auswertungsergebnisse zu kontrollieren:

Problem: Bei numerischen Methoden kann es in seltenen Fällen bedeutsam sein, wie Gleitkommazahlen auf Ebene der Computer-Hardware verarbeitet werden (Abschn. 1.4.6, Fußnote 35). Insbesondere zwischen den ARM-basierten Rechnerarchitekturen und solchen auf Basis von Intel bzw. AMD kann es leichte Unterschiede geben. Da Apple Computer Rechnerarchitekturen von ARM verwenden, während gegenwärtig die meisten Windows und Linux Computer auf Intel oder AMD basieren, ist der Einfluss der Hardware oft konfundiert mit jenem des Betriebssystems.

Maßnahme: Um sicherzugehen, dass wichtige Ergebnisse nicht Hardware-abhängig sind, können sie auf unterschiedlichen Rechnerarchitekturen repliziert werden. Dabei können Anbieter von Cloud-Computing helfen.

¹²<https://ropensci-archive.github.io/reproducibility-guide/> liefert eine Einführung. Auch das *German Reproducibility Network* verweist unter <https://reproducibilitynetwork.de/> auf weitere Ressourcen. Der Abschnitt **Reproducible Research** der CRAN Task Views (Blischak & Hill, 2020) stellt Pakete vor, die eine reproduzierbare Arbeitsweise unterstützen.

Problem: R verwendet intern grundlegende Software-Komponenten des Betriebssystems (*Bibliotheken*). Dies kann im Prinzip dazu führen, dass sich Ergebnisse zwischen verschiedenen Betriebssystemen oder verschiedenen Versionen desselben Betriebssystems auch dann unterscheiden, wenn die R-spezifische Auswertungsumgebung identisch ist. Solche Abweichungen sind allerdings selten, und dann üblicherweise so klein, dass sie nicht ins Gewicht fallen.

Maßnahme: *virtuelle Maschinen* oder *Container* können eine standardisierte und wiederherstellbare Betriebssystemumgebung erzeugen. So stehen etwa verschiedene Docker-Container für R (Boettiger & Eddelbuettel, 2017) zur Verfügung, die eine stabile Systemumgebung gewährleisten.¹³

Problem: R selbst ist versionsabhängigen Veränderungen unterworfen. Auch wenn diese im Normalfall keine relevanten Änderungen bei Ergebnissen bewirken, kann etwa die Verwendung anderer Algorithmen insbesondere bei numerischen Methoden Auswirkungen auf Berechnungen haben.

Maßnahme: Frühere Versionen von R sind über das CRAN Archiv weiter verfügbar. Auch virtuelle Maschinen und Container können die Verwendung einer bestimmten Version von R sicherstellen.

Problem: Zusatzpakete für R verändern sich meist weit schneller als R selbst und können deshalb häufiger zu versionsabhängig anderen Ergebnissen oder Fehlern führen (Abschn. 1.3.4).

Maßnahme: Das Paket `renv` (Ushey, 2020) liefert einen Ansatz, um die verwendeten Pakete für ein Auswertungsprojekt auf einen definierten projektspezifischen Versionsstand fixieren zu können. Datums-spezifische *snapshots* von CRAN Paketen bietet der *package manager* Service von Posit.¹⁴ Als alternative Herangehensweise bieten sich virtuelle Maschinen und Container an. Einen ganzheitlichen Ansatz, um eine Umgebung zu definieren, in der R selbst wie auch die Zusatzpakete auf einen bestimmten Versionsstand festgelegt sind, bietet das auf Nix¹⁵ basierende Paket `rix` (Rodrigues & Baumann, 2024).

Problem: Da R es zulässt, dass Pakete Funktionen mit demselben Namen enthalten, kann es zu Namenskonflikten kommen (Abschn. 1.3.3). In diesem Fall kann es passieren, dass sich derselbe Befehlsname auf verschiedene Funktionen aus unterschiedlichen Paketen bezieht – je nachdem, welche Pakete in einer R-session in welcher Reihenfolge geladen wurden.

Maßnahme: Bei Namenskonflikten zwischen Paketen können Funktionsaufrufe eindeutig gemacht werden, indem sie über die Variante `<Paketname>::<Funktion>()` erfolgen. Beim Laden von Paketen mit `library()` dient das Argument `warn.conflicts` wie auch die globale Option "conflicts.policy" dazu, den Umgang mit Namenskonflikten genau vorzugeben (Abschn. 1.2.2). Wenn gewünscht führt `library()` bei Maskierung dann zu einer Fehlermeldung.¹⁶

```
> options(conflicts.policy="strict") # Maskierungen generell verhindern
```

Problem: Manche R-Funktionen verhalten sich unterschiedlich in Abhängigkeit davon, wie bestimmte globale Optionen gewählt sind, die mit `options()` kontrolliert werden können. Unterscheiden sich diese globalen Einstellungen zwischen Computern, kann dies zu anderen Ergebnissen führen. Beispiele hierfür sind der Umgang mit fehlenden Werten ("na.action") oder das Erstellen von Kontrasten bei Regressionsmodellen mit kategorialen Prädiktoren ("contrasts").

¹³<https://www.rocker-project.org/>

¹⁴<https://p3m.dev/>

¹⁵<https://nixos.org/>

¹⁶<https://developer.r-project.org/Blog/public/2019/03/19/managing-search-path-conflicts/>

Analog hängt das Ergebnis von Funktionen, die auf Pseudozufallszahlen basieren, vom gewählten Generator und seinem Zustand ab.

Maßnahme: Um die Reproduzierbarkeit von Funktionsaufrufen zu erhöhen, sollten möglichst viele Funktionsargumente explizit verwendet werden. Der Rückgriff auf globale Optionen geschieht normalerweise nur, wenn Argumente beim Aufruf unspezifiziert bleiben. Mit `RNGkind("<Generator>")` kann der Zufallszahlen-Generator gewählt und mit `set.seed(<Zahl>)` sein Zustand so fixiert werden, dass die anschließend erzeugte Sequenz von Zufallszahlen immer dieselbe ist (Abschn. 2.4, Fußnote 11).

Problem: Wenn die Auswertungsskripte mit den R-Befehlssequenzen geändert werden, lässt sich nicht mehr rekonstruieren, welche Befehle zu einem früheren Zeitpunkt ausgeführt wurden, um bestimmte Ergebnisse zu erhalten.

Maßnahme: Auswertungsskripte und R-Dokumente (Abschn. 4.2) können mit aus der Softwareentwicklung stammenden *Versionskontrollsysteinen* versioniert werden. Besonders gut ist die Software git in RStudio integriert.¹⁷ Mit git ist es möglich, den Zustand von Auswertungsskripten oder R-Markdown Dokumenten zu bestimmten Zeiten „einzufrieren“ und auch nach späteren Änderungen wieder auf diesen Zustand zurückzusetzen.

4.4.2 Allgemeine Empfehlungen

Zusätzlich sind generelle Strategien bei der Umsetzung von Auswertungsprojekten empfehlenswert, die deren Reproduzierbarkeit unterstützen:

- Man sollte seine R-Umgebung so wenig wie möglich über die Konfigurationsdatei `.Rprofile` modifizieren (Abschn. 1.2.2).
- Man sollte beim Beenden einer Sitzung auf die Sicherung des workspace in einer `.RData` Datei verzichten, die in der nächsten Sitzung automatisch geladen wird (Abschn. 1.2.3).
- Alle in einem Projekt verwendeten Dateien sollten in einem gemeinsamen Verzeichnis (ggf. mit Unterverzeichnissen) liegen.
- Originaldaten sind von abgeleiteten Daten zu trennen, die durch Datenaufbereitung zu stande kommen. Insbesondere sollten manuelle Korrekturen durch Auswertungsskripte erfolgen und nicht in Datensätzen selbst vorgenommen werden.
- Daten sollten wie Skripte versioniert sein und durch Code-Bücher dokumentiert werden, die die Bedeutung und Codierung von Variablen beschreiben.
- Auswertungsbefehle sollten durch Kommentare erklärt werden.
- In Datensätzen sollten Variablen immer über ihren Namen, nicht über ihren – potentiell wechselnden – Spalten-Index ausgewählt werden.
- Idealerweise sollten R-Dokumente oder Notebooks (Abschn. 4.2) verwendet werden, um Ergebnisse direkt in den Fließtext von Berichten einzubauen und so Fehler durch *copy & paste* zu vermeiden.

¹⁷<https://git-scm.com/> – Tutorials finden sich unter <https://happygitwithr.com/> und bei Wickham (2023, Kap. 20): <https://r-pkgs.org/software-development-practices.html>

- Auch bei Auswertungsbefehlen sollte soweit wie möglich auf *copy & paste* verzichtet werden. Stattdessen sollten häufig eingesetzte Befehlsbausteine in wiederverwendbare Funktionen gekapselt werden (Abschn. 17.3). Selbst erstellte Pakete können häufig verwendete Funktionen bündeln und sind auf diese Weise gut an Dritte weiterzugeben ([Wickham, 2023](#)).

Kapitel 5

Hilfsmittel für die Inferenzstatistik

Bevor in den kommenden Kapiteln Funktionen zur inferenzstatistischen Datenanalyse besprochen werden, ist es notwendig Hilfsmittel vorzustellen, auf die viele dieser Funktionen zurückgreifen.¹ Dazu gehören die Syntax zur Formulierung linearer Modelle sowie einige Familien statistischer Verteilungen von Zufallsvariablen, die bereits bei der Erstellung zufälliger Werte aufgetaucht sind (Abschn. 2.4.4). Zunächst ist die Bedeutung inhaltlicher Begriffe zu klären, die im Kontext inferenzstatistischer Tests wichtig sind.

5.1 Wichtige Begriffe inferenzstatistischer Tests

Die Terminologie bei der Darstellung inferenzstatistischer Tests ist in der Literatur nicht einheitlich, deswegen soll der folgende Abschnitt präzisieren, mit welcher Bedeutung zentrale Begriffe im weiteren Verlauf des Buches verwendet werden. Die inhaltlichen Zusammenhänge der Logik schließender Statistik seien dabei als bekannt vorausgesetzt (Eid et al., 2017).

Die *Nullhypothese*, deren Konsistenz mit beobachteten Daten ein Test prüft, soll im Folgenden mit H_0 abgekürzt werden, die *Alternativhypothese* entsprechend mit H_1 . Die *Teststatistik* ist eine Zufallsvariable, deren Verteilung unter Gültigkeit der H_0 mit spezifischen Zusatzannahmen bekannt ist. Die Wahrscheinlichkeit dafür, dass sie Werte in beliebigen Intervallen annimmt, lässt sich dann über ihre Verteilungsfunktion berechnen. Theoretische Parameter einer Verteilung sollen i. d. R. mit griechischen Buchstaben (etwa μ für den Erwartungswert), empirische Kennwerte mit lateinischen Buchstaben benannt werden (etwa M für den Mittelwert). Besitzen empirische Schätzer keinen eigenen lateinischen Buchstaben, wird für sie der griechische Buchstabe des zu schätzenden Parameters mit einem Circumflex versehen (etwa $\hat{\epsilon}$). Im Interesse einfacherer Formulierungen wird im Folgenden nicht streng zwischen einer empirischen Variable und der zugehörigen Zufallsvariable im statistischen Sinn unterschieden.

Die Wahrscheinlichkeit unter Gültigkeit der H_0 , dass die Teststatistik Werte annimmt, die i. S. der H_1 mindestens so extrem wie der beobachtete sind, heißt *p-Wert*.² Der *kritische Wert* eines Tests soll den Wert bezeichnen, den die Teststatistik überschreiten muss, damit die Entscheidung für die H_1 ausfällt. Abweichend davon wird in der Literatur auch der mindestens zu erreichende Wert als der kritische bezeichnet, also der erste Wert des Ablehnungsbereichs

¹Dieses Buch behandelt nur die Umsetzung frequentistischer Verfahren. Für Bayes Datenanalyse (Kruschke, 2015; McElreath, 2020) vgl. den Abschnitt *Bayesian Inference* der CRAN Task Views (Park, 2020) – insbesondere die Pakete `rstanarm` (Gabry & Goodrich, 2020) und `brms` (Bürkner, 2020).

²Handelt es sich um eine zusammengesetzte H_0 – etwa bei gerichteter H_1 , ist hier immer die H_0 gemeint, die am dichtesten an der H_1 liegt.

der H_0 . Dieser Unterschied in der Bezeichnungskonvention ist nur für diskrete Verteilungen relevant.

Der Fehler, sich bei tatsächlicher Gültigkeit der H_0 für die H_1 zu entscheiden, ist der *Fehler erster Art* oder auch α -*Fehler*. Der Fehler, die H_0 bei tatsächlicher Gültigkeit der H_1 nicht zu verwerfen, wird als β -*Fehler* bezeichnet. Wenn zur sprachlichen Vereinfachung von der Größe oder Höhe eines Fehlers die Rede ist, soll immer die Wahrscheinlichkeit für das Eingehen eines solchen Fehlers gemeint sein. Als α -*Niveau* (auch einfach: α) wird die maximale Wahrscheinlichkeit eines Fehlers erster Art bezeichnet, die man bei Konstruktion einer Entscheidungsregel für den Test (Wahl des kritischen Wertes) einzugehen bereit ist. Ein statistischer Test fällt dann signifikant aus, wenn der p -Wert kleiner als das gesetzte α -Niveau ist. Die *power* oder *Teststärke* eines Tests ist die Wahrscheinlichkeit unter Gültigkeit der H_1 , dass die Teststatistik Werte größer als der kritische Wert annimmt.³

Mit dem *Vertrauensintervall* (VI) bzw. *Konfidenzintervall* zur Wahrscheinlichkeit $1 - \alpha$ für einen theoretischen Parameter θ ist ein offenes Intervall (a, b) gemeint: Die durch eine konkrete Formel zu spezifizierende Konstruktionsmethode liefert mit einer Wahrscheinlichkeit von $1 - \alpha$ Grenzen a und b , so dass θ im zugehörigen Intervall liegt ($a < \theta < b$). Ein statistischer Test zum Niveau α ist genau dann signifikant, wenn das zugehörige $1 - \alpha$ Vertrauensintervall den Wert des theoretischen Parameters unter H_0 nicht enthält.

Wenn sich ein Test sowohl gerichtet als auch ungerichtet durchführen lässt, bietet R beim Aufruf der Auswertungsfunktion eine Option zur Angabe, ob es sich um eine zweiseitige bzw. um welche einseitige H_1 (links- oder rechtsseitig) es sich handelt. Der ausgegebene p -Wert berücksichtigt die Art der Fragestellung und ist deshalb immer direkt mit dem gewählten α -Niveau zu vergleichen. Genauso werden Konfidenzintervalle für geschätzte Parameter entsprechend der Richtung der Fragestellung berechnet: Im Fall einer zweiseitigen Fragestellung wird das zweiseitige, im Fall einer einseitigen Fragestellung das passende einseitige Konfidenzintervall gebildet.

5.2 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. Eine solche Modellformel besitzt die Klasse **formula** und beschreibt, wie der systematische Anteil von Werten einer Zielgröße (abhängige Variable, AV) aus Werten einer oder mehrerer Prädiktoren (unabhängige Variablen, UVn) theoretisch hervorgeht.⁴ Die Annahme der prinzipiellen Gültigkeit eines linearen Modells über das Zustandekommen von Variablenwerten steht hinter

³Diese Darstellung vermischt die unterschiedlichen Ansätze von Fisher sowie von Neyman und Pearson. Während für Fisher der p -Wert entsprechend des Likelihood-Prinzips ein Maß für die lokale Evidenz vorliegender Daten gegen die H_0 ist, begrenzen Neyman und Pearson die langfristige Fehlerrate erster Art der häufig angewendeten Entscheidungsregel für die Wahl zwischen H_0 und H_1 auf α .

⁴Modellformeln verwenden die *Wilkinson-Rogers-Notation* (G. N. Wilkinson & Rogers, 1973; für Details vgl. **?formula**). Im Sinne des allgemeinen linearen Modells beschreibt die rechte Seite einer Modellformel die spaltenweise Zusammensetzung der Designmatrix, die `model.matrix(<Modellformel>)` für ein konkretes Modell ausgibt (Abschn. 12.9; Venables & Ripley, 2002, p. 144 ff.). Bei einem multivariaten Modell können auch mehrere Zielgrößen vorhanden sein.

vielen statistischen Verfahren, etwa der linearen Regression, Varianz- oder Kovarianzanalyse. Ein Formelobjekt wird wie folgt erstellt:

`<modellierte Variable> ~ <lineares Modell>`

Links der Tilde `~` steht die Variable, deren systematischer Anteil sich laut Modellvorstellung aus anderen Variablen ergeben soll. Die modellierenden Variablen werden in Form einzelner Terme rechts der `~` aufgeführt. Im konkreten Fall werden für alle Terme die Namen von Datenvektoren oder Faktoren derselben Länge eingesetzt.

Im Modell der einfachen linearen Regression (Kap. 6) sollen sich etwa die Werte des Kriteriums aus Werten des quantitativen Prädiktors ergeben, hier hat die Modellformel also die Form `<Kriterium> ~ <Prädiktor>`. In der Varianzanalyse (Abschn. 7.4) hat die AV die Rolle der modellierten und die kategorialen UVn die Rolle der modellierenden Variablen. Hier hat die Modellformel die Form `<AV> ~ <UV>`. Um R die Möglichkeit zu geben, beide Fälle zu unterscheiden, müssen im Fall der Regression die Prädiktoren numerische Vektoren sein, die Gruppierungsvariablen in der Varianzanalyse dagegen Objekte der Klasse `factor`.

Es können mehrere, in der Modellformel durch `+` getrennte Vorhersageterme in ein statistisches Modell eingehen. Ein einzelner Vorhersageterm kann dabei entweder aus einer Variable oder aber aus der Kombination von Variablen i.S. ihrer statistischen Interaktion bestehen. Die Beziehung zwischen den zu berücksichtigenden Variablen wird durch Symbole ausgedrückt, die sonst numerische Operatoren darstellen, rechts der `~` in einer Modellformel aber eine andere Bedeutung tragen. An Möglichkeiten, Variablen in einem Modell zu berücksichtigen, gibt es u.a. die in Tab. 5.1 aufgeführten.

Tabelle 5.1: Notation für die Modellformel linearer Modelle

| Operator | übliche Bedeutung | Bedeutung in einer Modellformel |
|----------------------|-------------------|---|
| <code>+</code> | Addition | den folgenden Vorhersageterm hinzufügen |
| <code>-</code> | Subtraktion | den folgenden Vorhersageterm ausschließen |
| <code>(A):(B)</code> | Sequenz | Interaktion $A \times B$ als Vorhersageterm |
| <code>(A)*(B)</code> | Multiplikation | Kurzform für $A + B + A:B$ (alle additiven und Interaktionseffekte) |
| <code>^</code> | potenzieren | Begrenzung des Grads zu berücksichtigender Interaktionen |
| <code>(A)/(B)</code> | Division | bei Verschachtelung von A in B (genestetes Design): Kurzform für $A + A:B$ |
| <code>1</code> | <code>1</code> | Schätzung des absoluten Terms (Gesamterwartungswert): Implizit vorhanden, wenn nicht durch <code>-1</code> ausgeschlossen |
| <code>-1</code> | <code>-1</code> | erzwingt Gesamterwartungswert von 0 – also die Vorhersage 0, wenn alle Kovariaten 0 sind ⁵ |

⁵Einer Modellformel mit `~ ... - 1` kann man ein `~ ... - 1 + offset(<0>)` hinzufügen, wobei dann die Variable `<0>` den Gesamterwartungswert festlegt – also die Vorhersage, wenn alle Prädiktoren 0 sind. `<0>` muss dabei ein Datenvektor sein.

Tabelle 5.1: (Forts.)

| | |
|---|--|
| . | bei Verwendung eines Datensatzes: alle vorhandenen Variablen, bis auf die bereits explizit verwendeten |
| . | bei Veränderung eines Modells: alle bisherigen Terme |

Als Beispiel gebe es eine kontinuierliche Zielgröße Y , drei quantitative Prädiktoren X_1, X_2, X_3 sowie zwei Faktoren F_1, F_2 . Neben den additiven Effekten der Terme können auch ihre Interaktionseffekte berücksichtigt werden. Zudem beinhaltet das Modell i. d. R. einen absoluten Term (den y -Achsenabschnitt der Vorhersagegerade im Fall der einfachen linearen Regression), der aber auch unterdrückt werden kann. Mit der Kombination von quantitativen Prädiktoren und Faktoren können etwa folgende lineare Modelle spezifiziert werden.

```

Y ~ X1                      # einfache lineare Regression Y auf X1
Y ~ X1 + X2 - 1              # multiple lineare Regression Y auf X1 und X2
                             # ohne absoluten Term (y-Achsenabschnitt)
Y ~ F1                      # einfaktorielle Varianzanalyse
Y ~ F1 + F2 + F1:F2          # zweifaktorielle Varianzanalyse
                             # mit beiden Haupteffekten und der Interaktion
Y ~ X1 + F1                  # Kovarianzanalyse mit Kovariate X1 und Faktor F1
Y ~ X1*X2                    # multiple lineare Regression Y auf X1 und X2
                             # sowie auf den Interaktionsterm von X1 und X2
Y ~ (X1 + X2 + X3)^2         # Regression von Y auf alle additiven Effekte
                             # sowie alle Interaktionseffekte bis zum 1. Grad
                             # Y ~ X1 + X2 + X3 + X1:X2 + X1:X3 + X2:X3

```

Die sich ergebenden Vorhersageterme einer Modellformel sowie weitere Informationen zum Modell können mit der Funktion `terms(<Modellformel>)` erfragt werden, deren Ausgabe hier gekürzt ist.

```

> 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 Modellformel 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 Modellformel `log(Y) ~ abs(X)`. Sollen hierbei innerhalb einer Modellformel Operatoren in ihrer arithmetischen Bedeutung zur Transformation von Variablen verwendet werden, muss der entsprechende Term in `I(<Transformation>)` eingeschlossen werden. Um etwa das Doppelte von X als Prädiktor für Y zu verwenden, lautet die Modellformel damit `Y ~ I(2*X)`.⁶

⁶Orthogonale Polynome eines numerischen Prädiktors erzeugt `poly()`. Für splines s. das im Basisumfang von R enthaltene Paket `splines` mit den Funktionen `ns()` und `bs()`.

5.3 Funktionen von Zufallsvariablen

Mit R lassen sich die Werte von häufig benötigten Dichte- bzw. Wahrscheinlichkeitsfunktionen, Verteilungsfunktionen und deren Umkehrfunktionen an beliebigen Stellen bestimmen.⁷ Tabelle 5.2 gibt Auskunft über einige der hierfür verfügbaren Funktionsfamilien sowie über ihre Argumente und deren Voreinstellungen. Für ihre Verwendung zur Erzeugung von Zufallszahlen s. Abschn. 2.4.4.

Tabelle 5.2: Vordefinierte Funktionen von Zufallsvariablen

| Familienname | Funktion | Argumente & Voreinstellung |
|-----------------------|------------------------------------|--|
| <code>binom</code> | Binomialverteilung | <code>size, prob</code> |
| <code>chisq</code> | χ^2 -Verteilung | <code>df, ncp=0</code> |
| <code>exp</code> | Exponentialverteilung | <code>rate=1</code> |
| <code>f</code> | F -Verteilung | <code>df1, df2, ncp=0</code> |
| <code>gamma</code> | Γ -Funktion | <code>shape, rate=1,</code>
<code>scale=1/rate</code> |
| <code>hyper</code> | Hypergeometrische Verteilung | <code>m, n, k</code> |
| <code>logis</code> | Logistische Verteilung | <code>location=0, scale=1</code> |
| <code>multinom</code> | Multinomialverteilung | <code>size, prob</code> |
| <code>norm</code> | Normalverteilung ⁸ | <code>mean=0, sd=1</code> |
| <code>pois</code> | Poisson-Verteilung | <code>lambda</code> |
| <code>signrank</code> | Wilcoxon-Vorzeichen-Rangverteilung | <code>x, n</code> |
| <code>t</code> | t -Verteilung | <code>df, ncp=0</code> |
| <code>unif</code> | Gleichverteilung | <code>min=0, max=1</code> |
| <code>weibull</code> | Weibull-Verteilung | <code>shape, scale=1</code> |
| <code>wilcox</code> | Wilcoxon-Rangsummenverteilung | <code>m, n</code> |

5.3.1 Dichtefunktion

Mit Funktionen, deren Namen nach dem Muster `d{Funktionsfamilie}` aufgebaut sind, lassen sich die Werte der Dichtefunktionen⁹ der in Tab. 5.2 genannten Funktionsfamilien bestimmen.

⁷Für weitere Verteilungen vgl. `?Distributions` sowie den Abschnitt *Probability Distributions* der CRAN Task Views ([Dutang & Kiener, 2020](#)).

⁸Für multivariate t - und Normalverteilungen vgl. das Paket `mvtnorm` ([Genz, Bretz, Miwa, Mi & Hothorn, 2020](#); [Genz & Bretz, 2009](#)).

⁹Im Fall diskreter (z. B. binomialverteilter) Variablen die Wahrscheinlichkeitsfunktion. 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` ist dies die Wahrscheinlichkeit, dass zwei Personen am selben Tag Geburtstag haben.

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 (Abschn. 2.4.4).

```
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, size=10, prob=0.5)
[1] 0.1171875

> choose(10, 7) * 0.5^7 * (1-0.5)^(10-7)      # manuelle Kontrolle
[1] 0.1171875
```

5.3.2 Verteilungsfunktion

Die Werte der zu einer Dichte- bzw. Wahrscheinlichkeitsfunktion gehörenden Verteilungsfunktion lassen sich mit Funktionen berechnen, deren Namen nach dem Muster `p(Funktionsfamilie)` aufgebaut sind. 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. Die Gegenwahrscheinlichkeit (Werte $> q$) wird mit dem Argument `lower.tail=FALSE` berechnet.¹⁰

```
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)          # F-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
```

¹⁰Bei der Verwendung von Verteilungsfunktionen diskreter (z. B. binomialverteilter) Variablen ist 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, ist als erstes Argument deshalb $q - 1$ zu übergeben, andernfalls würde nur die Wahrscheinlichkeit für Werte $> q$ bestimmt.

```
[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: Dazu ist der Wert der unteren Intervallgrenze von jenem der oberen zu subtrahieren.

```
# Standardnormalverteilung: Wkt. für Werte im Intervall mu +- sd
> m <- 100                                # Erwartungswert
> s <- 15                                   # Standardabweichung
> diff(pnorm(c(m-s, m+s), mean=m, sd=s))
[1] 0.6826895
```

Nützlich ist die Verteilungsfunktion insbesondere für die manuelle Berechnung des p -Wertes in inferenzstatistischen Tests: Ist q der Wert einer stetigen Teststatistik, liefert $1-p(\text{Familie})(q, \dots)$ ebenso den zugehörigen p -Wert des rechtsseitigen Tests wie $p(\text{Familie})(q, \dots, \text{lower.tail}=FALSE)$ (Fußnote 10).

5.3.3 Quantilfunktion

Die Werte der zu einer Dichte- bzw. Wahrscheinlichkeitsfunktion gehörenden Quantilfunktion lassen sich mit Funktionen berechnen, deren Namen nach dem Muster $q(\text{Funktionsfamilie})$ 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 $\text{lower.tail}=TRUE$) bzw. rechts ($\text{lower.tail}=FALSE$) abschneidet. Anders formuliert ist das Ergebnis der Wert, für den die zugehörige Verteilungsfunktion den Wert p annimmt.¹¹ 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.

¹¹Bei diskreten Verteilungen (z. B. Binomialverteilung) ist das Ergebnis bei $\text{lower.tail}=TRUE$ der kleinste Wert, der in der zugehörigen Wahrscheinlichkeitsfunktion mindestens p links abschneidet. Bei $\text{lower.tail}=FALSE$ ist das Ergebnis entsprechend der größte Wert, der mindestens p rechts abschneidet.

Kapitel 5 Hilfsmittel für die Inferenzstatistik

```
> 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

# kritischer Wert zweiseitiger z-Test, alpha=0.05
> qnorm(0.05/2, 0, 1, lower.tail=FALSE)
[1] 1.959964

# kritischer Wert einseitiger t-Test, alpha=0.01, df=18
> qt(0.01, 18, 0, lower.tail=FALSE)
[1] 2.552380
```

Kapitel 6

Lineare Regression

Die Korrelation zweier quantitativer Variablen ist ein Maß ihres linearen Zusammenhangs. Auch die lineare Regression analysiert den linearen Zusammenhang von Variablen, um die Werte einer Zielgröße (*Kriterium*) durch die Werte anderer Variablen (*Prädiktoren, Kovariaten, Kovariablen*) vorherzusagen. Für die allgemeinen statistischen Grundlagen dieser Themen vgl. Eid et al. (2017), eine vertiefte Behandlung von Regressionsanalysen in R liefern Fox und Weisberg (2019) und Faraway (2014).

6.1 Test des Korrelationskoeffizienten

Die empirische Korrelation zweier normalverteilter Variablen lässt sich daraufhin testen, ob sie mit der H_0 verträglich ist, dass die theoretische Korrelation gleich 0 ist.¹

```
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. Alternativ kann auch eine Modellformel `~ <Vektor1> + <Vektor2>` ohne Variable links der `~` angegeben werden. Stammen die in der Modellformel verwendeten Variablen aus einem Datensatz, ist dieser unter `data` zu nennen. Ob die H_1 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 (Abschn. 2.13.4).

```
> N    <- 20                                # Anzahl Beobachtungsobjekte
> DV1 <- rnorm(N, 100, 15)                  # Daten Variable 1
> DV2 <- DV1 + rnorm(N, 0, 50)              # Daten Variable 2
> cor.test(~ DV1 + DV2)                    # oder: cor.test(DV1, DV2)
Pearson's product-moment correlation
data: DV1 and DV2
t = 4.2167, df = 18, p-value = 0.0005186
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
0.3813889 0.8746200
sample estimates:
cor
```

¹Für Tests auf Zusammenhang von ordinalen Variablen s. Abschn. 10.3.1.

0.7049359

Die Ausgabe beinhaltet den empirischen t -Wert der Teststatistik (`t`) samt Freiheitsgraden (`df`) und zugehörigem p -Wert (`p-value`) sowie das je nach H_1 ein- oder zweiseitige Vertrauensintervall für die Korrelation, deren empirischer Wert ebenfalls aufgeführt ist. Das Ergebnis lässt sich manuell prüfen:²

```
> r      <- cor(DV1, DV2)                      # empirische Korrelation
> (tVal <- sqrt(N-2) * r / sqrt(1-r^2))    # Teststatistik t-Wert
[1] 4.216709

> (pVal <- 2*pt(tVal, N-2, lower.tail=FALSE))      # p-Wert
[1] 0.00051861

# 95%-Vertrauensintervall für die wahre Korrelation
> fishZ <- 0.5 * log((1+r) / (1-r))        # Fisher Z-Transformation
> fishV <- 1 / (N-3)                         # Varianz der Transformierten
> Zcrit <- qnorm(0.05/2, 0, 1, lower.tail=FALSE) # 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
> (ciLoUp <- tanh(c(Zlo, Zup)))
[1] 0.3813889  0.87462
```

Mit `r.test()` aus dem Paket `psych` (Revelle, 2020) lassen sich auch Hypothesen darüber testen, ob zwei theoretische Korrelationskoeffizienten aus unabhängigen oder abhängigen Stichproben identisch sind. `rcorr()` aus dem Paket `Hmisc` (Harrell Jr, 2020a) berechnet für mehrere Variablen die Korrelationsmatrix nach Pearson oder Spearman und testet die resultierenden Korrelationen gleichzeitig auf Signifikanz.

6.2 Einfache lineare Regression

Bei der einfachen linearen Regression werden anhand der paarweise vorhandenen Daten zweier Variablen X und Y die Parameter a und b der Vorhersagegleichung $\hat{Y} = bX + a$ so bestimmt, dass die Werte von Y (dem Kriterium) bestmöglich mit der Vorhersage \hat{Y} aus den Werten von X (dem Prädiktor) übereinstimmen. Dafür muss Y eine quantitative Variable sein, für X sind quantitative und dichotome Variablen möglich. Als Maß für die Güte der Vorhersage wird die Summe der quadrierten Residuen $E = Y - \hat{Y}$, also der Abweichungen von vorhergesagten und beobachteten Werten des Kriteriums herangezogen.³

²Für Fishers Z-Transformation vgl. `FisherZ()`, für die Rücktransformation `FisherZInv()` aus dem Paket `DescTools`.

³Für Maximum-Likelihood-Schätzungen der Parameter vgl. die `glm()` Funktion, deren Anwendung Kap. 8 demonstriert. Eine formalere Behandlung des allgemeinen linearen Modells findet sich in Abschn. 12.9. Für Methoden zur Einschätzung des Vorhersagefehlers in externen Stichproben s. Kap. 13.

6.2.1 Deskriptive Modellanpassung

Lineare Regressionsmodelle lassen sich mit der `lm()` Funktion anpassen und so die Parameter a und b schätzen.

```
lm(formula=<Modellformel>, data=<Datensatz>, subset=<Indexvektor>,
  na.action=<Behandlung fehlender Werte>)
```

Unter `formula` ist eine Modellformel der Form $\langle\text{Kriterium}\rangle \sim \langle\text{Prädiktor}\rangle$ als Spezifikation des Regressionsmodells anzugeben (Abschn. 5.2). Soll das Modell ohne den Parameter a gebildet werden, ist ihr `-1` anzuhängen. Stammen die angegebenen Variablen aus einem Datensatz, muss dieser unter `data` übergeben werden. Das Argument `subset` erlaubt es, nur eine Teilmenge der Fälle in die Berechnung einfließen zu lassen. Es erwartet einen numerischen oder logischen Indexvektor, der sich auf die Zeilen des Datensatzes bezieht. Mit dem Argument `na.action` kann bestimmt werden, wie mit fehlenden Werten umzugehen ist (Abschn. 2.13.6).⁴

Als Beispiel soll das Körpergewicht als Kriterium mit der Körpergröße als Prädiktor vorhergesagt werden. Das Körpergewicht wird hier entsprechend einem (sicher unrealistischen) linearen Modell aus der Körpergröße und einem zufälligen Fehler simuliert.

```
> N      <- 100                      # Anzahl Personen
> height <- rnorm(N, mean=175, sd=7)    # Prädiktor
> weight <- 0.4*height + 10 + rnorm(N, 0, 3) # Kriterium
> (fit   <- lm(weight ~ height))        # Regression
Coefficients:
(Intercept)  height
27.3351    0.3022
```

Im Ergebnis von `lm()` wird unter der Überschrift `Coefficients` in der Spalte (`Intercept`) die Schätzung für den Schnittpunkt a mit der y -Achse und unter dem Namen des Prädiktors (hier: `height`) die geschätzte Steigung b 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

> Yhat <- b*height + a                        # vorhergesagte Werte
```

⁴In der Voreinstellung `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 und lässt sich vermeiden, indem das Argument auf `na.exclude` gesetzt wird.

Soll statt des b -Gewichts das standardisierte b^z -Gewicht berechnet werden, sind die Variablen in der Modellformel zu z -standardisieren.⁵ Der y -Achsenabschnitt sollte in diesem Fall 0 ergeben, was gerundet der Fall ist. Das b^z -Gewicht ist in der einfachen linearen Regression 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)                      # Kontrolle
[1] 0.5599206

> cor(height, weight)                             # Kontrolle
[1] 0.5599206
```

Ein von `lm()` zurückgegebenes Objekt stellt ein deskriptives Modell der Daten dar, das in anderen Funktionen weiter verwendet werden kann. Es speichert die zur Modellanpassung berechneten Größen als Komponenten einer Liste. Zum Extrahieren der gespeicherten Kennwerte dienen Funktionen wie `residuals()` zum Anzeigen der Residuen, `deviance()` zur Berechnung der Quadratsumme der Residuen, `coef()` zur Ausgabe der Modellparameter, `fitted()` für die vorhergesagten Werte, weiterhin `vcov()` für die Kovarianzmatrix der geschätzten Parameter. Zur Extraktion der Designmatrix i. S. des allgemeinen linearen Modells dient `model.matrix()` (Abschn. 12.9). Analog liefert `model.frame()` den zur Modellanpassung verwendeten Datensatz zurück. Alle genannten Funktionen erwarten als Argument ein von `lm()` erzeugtes Objekt.

Für eine grafische Veranschaulichung der Regression können die Daten zunächst als Streudiagramm angezeigt werden (Abb. 6.1, Abschn. 14.2). Die aus der Modellanpassung hervorgehende Schätzung der Regressionsparameter wird dieser Grafik durch `abline(<lm-Modell>)` in Form eines Geradenabschnitts hinzugefügt (Abschn. 14.5). Ebenfalls abgebildet wird hier die Gerade der zur Simulation verwendeten fehlerbereinigten Modellgleichung `weight = 0.4*height + 10` und das Zentroid der Daten.

```
# Daten als Streudiagramm darstellen
> plot(weight ~ height, xlab="height [cm]", ylab="weight [kg]", pch=20,
+       main="Daten mit Zentroid, Regressionsgerade und Modellgerade")

> abline(fit, col="blue", lwd=2)                      # Regressionsgerade
> abline(a=10, b=0.4, col="gray", lwd=2)            # Modellgerade
> points(mean(weight) ~ mean(height), col="red", pch=4, cex=1.5, lwd=3)
> legend(x="topleft", legend=c("Daten", "Zentroid", "Regressionsgerade",
+       "Modellgerade"), col=c("black", "red", "blue", "gray"),
```

⁵Bei fehlenden Werten ist darauf zu achten, dass die z -Standardisierung bei beiden Variablen auf denselben Beobachtungsobjekten beruht. Gegebenenfalls sollten fehlende Werte der beteiligten Variablen aus dem Datensatz vorher manuell ausgeschlossen werden (Abschn. 2.13.6).

```
+      lwd=c(1, 3, 2, 2), pch=c(20, 4, NA, NA), lty=c(NA, NA, 1, 1))
```

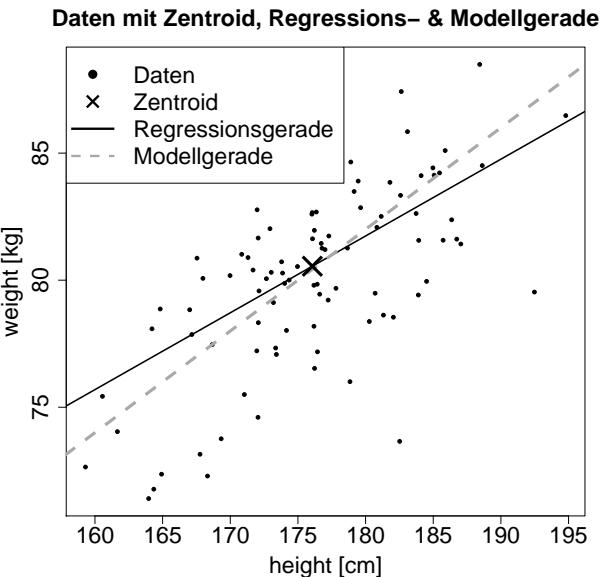


Abbildung 6.1: Lineare Regression: Darstellung der Daten mit Modell- und Regressionsgerade

6.2.2 Regressionsanalyse

Um weitere Informationen und insbesondere inferenzstatistische Kennwerte eines von `lm()` erstellten Modells i.S. einer Regressionsanalyse zu erhalten, wird `summary(<lm-Modell>)` verwendet.⁶

```
> (sumRes <- summary(fit))                                # gekürzte Ausgabe ...
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
```

⁶Für eine Mediationsanalyse mit dem Sobel-Test vgl. `sobel()` aus dem `multilevel` Paket (Bliese, 2016).
Mediationsanalysen sind mit dem Paket `mediation` (Tingley et al., 2019) möglich.

Die Ausgabe enthält unter der Überschrift **Residuals** eine Zusammenfassung der beobachtungsweisen Residuen. Diese sollten symmetrisch um 0 streuen. Ein deutlich von 0 abweichender Median sowie vom Betrag deutlich unterschiedliche Quartile (1Q, 3Q) deuten auf eine schiefe Verteilung hin.

Unter **Coefficients** werden die Koeffizienten (Spalte **Estimate**), ihr Standardfehler (**Std. Error**), *t*-Wert (**t value**) und der zugehörige *p*-Wert für den zweiseitigen *t*-Test (**Pr(>|t|)**) ausgegeben. Dieser Test wird mit der H_0 durchgeführt, dass das theoretische β -Gewicht gleich 0 ist (Abschn. 12.9.5). Die Größe des *p*-Wertes wird mit Sternchen hinter den Werten codiert, deren Bedeutung **Signif. codes** beschreibt.⁷ Die Tabelle der Koeffizienten lässt sich mit **coef()** extrahieren.

```
# geschätzte Koeffizienten, Standardfehler, t-Werte und p-Werte
> coef(sumRes)
            Estimate Std. Error   t value   Pr(>|t|)
(Intercept) 27.3350625 7.96061681 3.433787 8.737358e-04
height       0.3022421 0.04517855 6.689947 1.390238e-09

# Streuung der Schätzungen: Wurzel aus Diagonale der Kovarianzmatrix
> (sdCoef <- sqrt(diag(vcov(fit))))
(Intercept)      height
7.96061681    0.04517855

> (tVals <- coef(fit) / sdCoef)                                # t-Werte
(Intercept)      height
3.433787     6.689947
```

Residual standard error gibt den Standardschätzfehler als Maß für die Diskrepanz zwischen empirischen Werten des Kriteriums und der Modellvorhersage aus. Er ist gleich der Wurzel aus der mittleren Quadratsumme der Residuen als Schätzung der Fehlervarianz, also aus dem Quotienten der Quadratsumme der Residuen und ihrer Freiheitsgrade.

```
# Freiheitsgrade der Residual-Quadratsumme, vgl. fit$df.residual
> P      <- 1                                         # Anzahl Prädiktoren
> (dfSSE <- N - (P+1))                            # Freiheitsgrade
[1] 98

> (SSE <- sum(residuals(fit)^2))                  # QS Residuen
[1] 742.2144

> deviance(fit)                                     # Kontrolle ...
> MSE <- SSE / dfSSE                               # mittlere QS Residuen
> sqrt(MSE)                                       # Standardschätzfehler
[1] 3.143246
```

Ein weiteres Maß für die Güte der Schätzung ist der Determinationskoeffizient R^2 , in Modellen mit absolutem Term gleich der quadrierten Korrelation zwischen Vorhersage und Krite-

⁷Im Folgenden wird dieser Teil der Ausgabe mit `options(show.signif.stars=FALSE)` unterdrückt.

rium (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 gleich 1 ist. Im Gegensatz zum empirischen kann das korrigierte R^2 auch negativ werden.

```
# unkorrigiertes R^2: quadrierte Korrelation Vorhersage mit Kriterium
> Yhat <- fitted(fit)                                # Vorhersage
> (rSq <- cor(Yhat, weight)^2)                      # unkorrigiertes R^2
[1] 0.3135111

> 1 - ((N-1) / (N-P-1)) * (1-rSq)                  # korrigiertes R^2
[1] 0.3065061
```

Schließlich wird mit einem F -Test für das gesamte Modell die H_0 geprüft, dass (bei einer multiplen Regression, s. Abschn. 6.3.1) alle theoretischen β_j -Gewichte gleich 0 sind (Abschn. 12.9.6, 12.9.7). Die Einzelheiten des Tests sind der empirische Wert des F -Bruchs (F-statistic) gefolgt von den Freiheitsgraden (DF) der Vorhersage und der Residuen sowie dem p -Wert (p-value).

```
> MSpred <- sum((Yhat - mean(Yhat))^2) / P          # mittl. QS Vorhersage
> (Fval <- MSpred / MSE)                             # Teststatistik F-Wert
[1] 44.75539

(pVal <- pf(Fval, P, dfSSE, lower.tail=FALSE))      # p-Wert
[1] 1.390238e-09
```

Für die geschätzten Parameter errechnet `confint(<lm-Modell>, level=0.95)` das Konfidenzintervall mit der für das Argument `level` angegebenen Breite.

```
> confint(fit)
              2.5 %    97.5 %
(Intercept) 11.5374776 43.1326475
height       0.2125868  0.3918975
```

Die Informationskriterien nach Akaike (AIC) und Bayes (BIC) berücksichtigen einerseits die Güte der Modellpassung i. S. ihrer maximierten logarithmierten likelihood und bestrafen andererseits die Komplexität des Modells, gemessen an der Anzahl zu schätzender Parameter.⁸ Kleinere Werte stehen für eine höhere Informativität. Bei einer linearen Regression ergibt sich der AIC-Wert direkt aus der Quadratsumme der Residuen sowie der Anzahl zu schätzender Parameter: Bei $p + 1$ Koeffizienten für p Prädiktoren und den y -Achsenabschnitt sind dies $p + 1 + 1$.⁹ Besitzen zwei Modelle dieselbe Anpassungsgüte, erhält das Modell mit einer geringeren Anzahl von Parametern den kleineren AIC- bzw. BIC-Wert. Beide Werte werden durch `extractAIC(<lm-Modell>)` für ein Modell berechnet, wobei für BIC das Argument `k=log(<Stichprobengröße>)` zu setzen ist.¹⁰ `AIC(<lm-Modell>)` berechnet den AIC-Wert mit einer anders gewählten Konstante, was die für Modellvergleiche wesentliche Differenz zweier

⁸ AIC und BIC besitzen einen engen Bezug zu bestimmten Methoden der Kreuzvalidierung (Abschn. 13.1).

⁹ Zusätzlich zu β_0 und den β_j ist auch die Fehlerstreuung σ zu schätzen.

¹⁰ Der korrigierte AICc Wert für kleine Stichproben ist mit `aictab()` aus dem Paket `AICmodavg` (Mazerolle, 2020) berechenbar.

AIC-Werte aber nicht beeinflusst (Abschn. 6.3.3). Die Ausgabe führt als erstes Element die Anzahl zu schätzender Parameter auf.

```
> extractAIC(fit)                      # AIC-Wert
[1] 2.0000 231.0309

> extractAIC(fit, k=log(N))          # BIC-Wert
[1] 2.0000 236.2413

> N * log(SSE / N) + 2*(1+1)        # Kontrolle: AIC ...
> AIC(fit)                          # AIC-Berechnung: andere Konstante
[1] 787.1703

> N * (log(2*pi) + log(SSE / N) + 1) + 2*(1+1+1)    # Kontrolle ...
```

6.3 Multiple lineare Regression

Bei der multiplen linearen Regression dienen mehrere quantitative oder dichotome Variablen X_j als Prädiktoren zur Vorhersage des quantitativen Kriteriums Y .¹¹ Die Vorhersagegleichung hat hier die Form $\hat{Y} = b_0 + b_1X_1 + \dots + b_jX_j + \dots + b_pX_p$, wobei die Parameter b_0 und b_j auf Basis der empirischen Daten zu schätzen sind. Dies geschieht wie im Fall der einfachen linearen Regression mit `lm(<Modellformel>)` durch Minimierung der Summe der quadrierten Abweichungen von Vorhersage und Kriterium. Die Modellformel hat nun die Form `<Kriterium> ~ <Prädiktor 1> + ... + <Prädiktor p>`, d. h. alle p Prädiktoren X_j werden mit `+` verbunden auf die Rechte Seite der `~` geschrieben.

6.3.1 Deskriptive Modellanpassung und Regressionsanalyse

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.

```
> N      <- 100                                # Anzahl Personen
> height <- rnorm(N, mean=175, sd=7)           # Prädiktor 1
> age    <- rnorm(N, mean=30, sd=8)             # Prädiktor 2
> sport   <- abs(rnorm(N, mean=60, sd=30))     # Prädiktor 3

# Simulation des Kriteriums im Modell der multiplen linearen Regression
> weight <- 0.5*height - 0.3*age - 0.4*sport + 10 + rnorm(N, 0, 3)
> (fitHA <- lm(weight ~ height + age))         # gekürzte Ausgabe ...
Coefficients:
(Intercept)  height      age
```

¹¹Für die multivariate multiple Regression mit mehreren Kriteriumsvariablen Y_k s. Abschn. 12.5. Eine formalere Behandlung des allgemeinen linearen Modells findet sich in Abschn. 12.9.

```
-72.5404  0.8602 -0.4429
```

Die Schätzungen b_0 und b_j der theoretischen Parameter β_0 und β_j 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. Auch hier müssen die Variablen in der Modellformel z -standardisiert werden, um die standardisierten b_j^z -Gewichte zu berechnen (s. Abschn. 6.2.1, Fußnote 5 für das Vorgehen bei fehlenden Werten). Wie im Fall einer einfachen linearen Regression werden die Parameter mit `summary(⟨lm-Modell⟩)` auf Signifikanz getestet (Abschn. 6.2.2).

```
# für standardisierte b-Gewichte, vorausgesetzt keine fehlenden Werte
> lm(scale(weight) ~ scale(height) + scale(age)) # ...
> summary(fitHA)                                # Regressionsanalyse ...
```

Ist \mathbf{X} die mit `model.matrix(⟨lm-Modell⟩)` erzeugte Designmatrix i. S. des allgemeinen linearen Modells und \mathbf{y} der Vektor der Kriteriumswerte, lassen sich die Schätzungen b_0 und b_j als $\mathbf{X}^+ \mathbf{y}$ berechnen (mit der Pseudoinversen \mathbf{X}^+ von \mathbf{X}).¹² Der Vektor der vorhergesagten Werte berechnet sich als $\hat{\mathbf{y}} = \mathbf{H} \mathbf{y}$ (mit der *Hat-Matrix* $\mathbf{H} = \mathbf{X} \mathbf{X}^+$, s. Abschn. 12.1.7, 12.9.1, 12.9.4).

```
# Koeffizienten aus Produkt der Pseudoinversen X+ mit Kriterium
> X      <- model.matrix(fitHA)                      # Designmatrix
> Xplus <- solve(t(X) %*% X) %*% t(X)            # Pseudoinverse X+
> (b      <- Xplus %*% weight)                     # Parameterschätzung
(Intercept) -72.5404499
height       0.8602345
age         -0.4428933

# Vorhersage aus Produkt der Hat-Matrix mit Kriterium
> H      <- X %*% Xplus                           # Hat-Matrix
> Yhat <- H %*% weight                            # Vorhersage
> all.equal(fitted(fitHA), c(Yhat), check.attributes=FALSE)
[1] TRUE

# bz-Gewichte bei standardisierten Variablen
> S <- cov(cbind(height, age))                  # Kovarianzmatrix Prädiktoren
> (1/sd(weight)) * diag(sqrt(diag(S))) %*% b[-1] # Kontrolle ...
```

Die grafische Veranschaulichung mit `scatter3d()` aus dem Paket `car` zeigt die simulierten Daten zusammen mit der Vorhersageebene sowie die Residuen als vertikale Abstände zwischen ihr und den Daten (Abb. 6.2). Als Argument ist dieselbe Modellformel wie für `lm()` zu verwenden. Weitere Argumente kontrollieren das Aussehen der Diagrammelemente. Die Beobachterperspektive dieser Grafik lässt sich interaktiv durch Klicken und Ziehen mit der Maus ändern (Abschn. 14.7.2).

¹²Es sei vorausgesetzt, dass \mathbf{X} vollen Spaltenrang hat, also keine linearen Abhängigkeiten zwischen den Prädiktoren vorliegen. Dann gilt $\mathbf{X}^+ = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top$. Der hier gewählte Rechenweg ist numerisch nicht stabil und weicht von in R-Funktionen implementierten Rechnungen ab (Bates, 2004).

```
> library(car) # für scatter3d()
> scatter3d(weight ~ height + age, fill=FALSE)
```

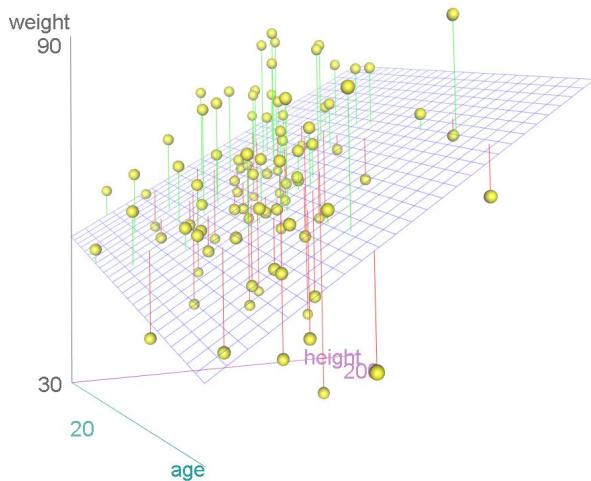


Abbildung 6.2: Daten, Vorhersageebene und Residuen einer multiplen linearen Regression

6.3.2 Modell verändern

Möchte man ein bereits berechnetes Modell nachträglich verändern, ihm etwa einen weiteren Prädiktor hinzufügen, kann dies mit `update()` geschehen. Bei sehr umfassenden Modellen kann dies numerisch effizienter sein, als ein Modell vollständig neu berechnen zu lassen.

```
update(<lm-Modell>, . ~ . + <weiterer Prädiktor>)
```

Als erstes Argument wird das zu erweiternde Modell eingetragen. In der folgenden Modellformel steht der `.` vor der `~` für die bisher verwendeten Kriterien, jener nach der `~` für alle bisher verwendeten Prädiktoren. Die Formel `. ~ .` signalisiert also, dass alle bisherigen Kriterien und Prädiktoren beizubehalten sind. Jeder weitere Prädiktor wird dann mit einem `+` angefügt. Das Ergebnis von `update()` ist das aktualisierte Modell.

```
# füge sport als zusätzlichen Prädiktor hinzu
> (fitHAS <- update(fitHA, . ~ . + sport))      # gekürzte Ausgabe ...
Coefficients:
(Intercept)  height      age      sport
     8.1012   0.5149  -0.2866  -0.4160
```

Auf die gleiche Weise wie Prädiktoren hinzugefügt werden können, lassen sie sich auch entfernen, wobei statt des `+` ein `-` zu verwenden ist.

```
# entferne Prädiktor height
> (fitAS <- update(fitHAS, . ~ . - height))      # gekürzte Ausgabe ...
Coefficients:
(Intercept)      age      sport
    98.5145  -0.2459  -0.4452
```

```
> (fitH <- update(fitHA, . ~ . - age))           # entferne age ...
```

6.3.3 Modelle vergleichen und auswählen

Existieren bei einer multiplen Regression viele potentielle Prädiktoren, stellt sich die Frage, welche letztlich im Modell berücksichtigt werden sollen. In einem hierarchischen Vorgehen lässt sich dafür testen, inwieweit die Hinzunahme von Prädiktoren zu einer bedeutsamen Verbesserung der Modellpassung führt. Dies ist in Form von *F*-Tests über Modellvergleiche möglich, die die Veränderung der Residualvarianz in Abhängigkeit vom verwendeten Prädiktorensatz testen. Dabei lassen sich sinnvoll nur *nested* Modelle mit demselben Kriterium vergleichen, bei denen der Prädiktorensatz eines eingeschränkten Modells vollständig im Prädiktorensatz des umfassenderen Modells enthalten ist, das zusätzlich noch weitere Prädiktoren berücksichtigt (Abschn. 12.9.6).

Um verschiedene Regressionsmodelle miteinander hinsichtlich der Quadratsumme der Residuen sowie Akaikes Informationskriterium AIC vergleichen zu können, lassen sich folgende Funktionen verwenden:

```
step(object=<lm-Modell>, scope=~ . + <Prädiktoren>, direction="both")
add1(object=<lm-Modell>, scope=~ . + <Prädiktoren>, test="none")
drop1(object=<lm-Modell>, scope=~ . - <Prädiktoren>, test="none")
```

`step()` berechnet im *Stepwise*-Verfahren die Auswirkung einer schrittweisen Modellveränderung durch Hinzunahme oder Weglassen eines Prädiktors auf den AIC-Wert als Vergleichskriterium für die Güte der Modellpassung. Als Ergebnis wird ein Modell zurückgegeben, das durch diese Schritte nicht substantiell verbessert werden kann.¹³

Dazu prüft `step()` sequentiell Teilmengen von denjenigen Prädiktoren der unter `object` angegebenen Regression, die in `scope` als Modellformel definiert werden. Diese Modellformel beginnt mit einer `~` und enthält danach alle Prädiktoren aus `object` (Voreinstellung `~ .`) sowie ggf. weitere mit `+` hinzuzufügende. Mit `step="backward"` beginnt die Exploration der Modelle beim umfassendsten Modell und entfernt dann Prädiktoren (Rückwärts-Elimination), analog mit `step="forward"` beim einfachsten Modell und fügt Prädiktoren hinzu (Vorwärts-Selektion). In der Voreinstellung `direction="both"` können von Schritt zu Schritt Prädiktoren sowohl hinzugefügt als auch entfernt werden.

```
# Vorwärts-Selektion
> step(fitH, scope=~ height + age + sport, direction="forward")
Start: AIC=501.38
weight ~ height
      Df Sum of Sq    RSS    AIC
+ sport  1    12962.8  1495.1 276.48
+ age    1    1246.3 13211.6 494.37
```

¹³Das Paket `lmSubsets` (Hofmann, Gatu, Kontoghiorghes, Colubi & Zeileis, 2020) ermöglicht die automatisierte Auswahl aller Teilmengen von Prädiktoren. Beide Verfahren sind mit vielen inhaltlichen Problemen verbunden, für eine Diskussion und verschiedene Strategien zur Auswahl von Prädiktoren vgl. Miller (2002). Für penalisierte Regressionsverfahren, die auch eine Auswahl von Prädiktoren vornehmen, s. Abschn. 6.6.1.

```

<none>           14457.9 501.38

Step: AIC=276.48
weight ~ height + sport
  Df Sum of Sq    RSS    AIC
+ age     1      515.5  979.64 236.20
<none>           1495.14 276.48

Step: AIC=236.2
weight ~ height + sport + age
Coefficients:
(Intercept)   height       sport       age
8.1012        0.5149      -0.4160     -0.2866

```

`add1()` berechnet separat den Effekt der Hinzunahme jeweils eines unter `scope` genannten Prädiktors zu einer unter `object` angegebenen Regression. Die Änderung in der Quadratsumme der Residuen bei jedem Schritt 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.

```

# Vergleich: nur height als Prädiktor, sowie auch age bzw. sport
> add1(fitH, . ~ . + age + sport, test="F")
Single term additions
Model:
weight ~ height
  Df Sum of Sq    RSS    AIC F value    Pr(>F)
<none>           14457.9 501.38
age     1      1246.3 13211.6 494.37     9.15  0.003183 ***
sport    1      12962.8 1495.1  276.48    840.98 < 2.2e-16 ***

```

In der Zeile `<none>` der Ausgabe finden sich die Kennwerte des eingeschränkten `object` Modells, dessen Modellformel unter `Model:` wiederholt wird. Die folgenden Zeilen nennen die Kennwerte der umfassenderen Modelle, bei denen zusätzlich jeweils einer der unter `scope` genannten Prädiktoren berücksichtigt wird. Die Spalte `Sum of Sq` führt die sequentielle Quadratsumme vom Typ I des zusätzlichen Prädiktors beim Wechsel hin zu einem solchen Modell auf. Sie ist gleich der Differenz der in der Spalte `RSS` genannten Quadratsummen der Residuen (*residual sum of squares*) und damit ein Maß für die Reduktion an Fehlervarianz beim Modellwechsel. Die inferenzstatistischen Kennwerte finden sich in den Spalten `Df` für die Differenz der Fehler-Freiheitsgrade vom eingeschränkten und umfassenderen Modell, `F value` für den *F*-Wert und `Pr(>F)` für den *p*-Wert.¹⁴

Der *F*-Wert ergibt sich als Quotient der angegebenen Quadratsumme geteilt durch die Differenz der Fehler-Freiheitsgrade und der Residual-Quadratsumme des umfassenderen Modells geteilt durch ihre Freiheitsgrade (Abschn. 12.9.7). Die Fehler-Freiheitsgrade eines Modells berechnen sich als Differenz zwischen der Stichprobengröße und der Anzahl zu schätzender Parameter.

¹⁴Auf ähnliche Weise kann mit `drop1()` die Quadratsumme vom Typ III eines Vorhersageterms als Effekt des Weglassens jeweils eines Prädiktors berechnet werden (Abschn. 7.6.2).

Kapitel 6 Lineare Regression

Hier sind dies im eingeschränkten Modell mit einem Regressionsgewicht und dem absoluten Term zwei, in jedem Modell mit einem zusätzlichen Prädiktor entsprechend drei.

```
# RSS Zeile <none>: Prädiktor height
> (rssH <- sum(residuals(lm(weight ~ height))^2))
[1] 14457.9

# RSS Zeile age: Prädiktoren height und age
> (rssHA <- sum(residuals(lm(weight ~ height + age))^2))
[1] 13211.64

# RSS Zeile sport: Prädiktoren height und sport
> (rssHS <- sum(residuals(lm(weight ~ height + sport))^2))
[1] 1495.140

# Fehler-Freiheitsgrade der drei Modelle
> dfEH  <- N - (1+1)                      # eingeschränktes Modell
> dfEHA <- dfEHS <- N - (2+1)            # beide umfassenderen Modelle

# mittlere partielle Effekt-QS für Hinzunahme von age
> MSha <- (rssH - rssHA) / (dfEH - dfEHA)

# mittlere Residual-QS umfassenderes Modell: Prädiktoren height und age
> MSEha <- rssHA / dfEHA

# F-Wert für Effekt der Hinzunahme von age
> (Fha <- MSha / MSEha)
[1] 9.150025

# analog für Hinzunahme von sport
> MShs  <- (rssH - rssHS) / (dfEH - dfEHS)
> MSEhs <- rssHS / dfEHS
> (Fhs  <- MShs / MSEhs)
[1] 840.9835
```

Die Spalte AIC listet den Wert von Akaike's Informationskriterium für beide Modelle auf.

Um nested Modelle gegeneinander zu testen, die sich um mehr als einen Prädiktor unterscheiden, können mit `lm()` erstellte Regressionen an die `anova()` Funktion übergeben werden, die sich auch für Varianzanalysen eignet (Abschn. 7.4.3). Die Modelle müssen sich auf dieselben Beobachtungen beziehen – bei fehlenden Werten ist sicherzustellen, dass in beide Modelle dieselben Beobachtungen einfließen.

```
anova(<eingeschränktes lm-Modell>, <umfassenderes lm-Modell>)
```

Im Beispiel soll das Kriterium `weight` entweder nur durch `height`, oder aber durch alle drei Prädiktoren vorhergesagt werden.

```
> anova(fitH, fitHAS)      # F-Test für Hinzunahme von age und sport
```

```
Analysis of Variance Table
Model 1: weight ~ height
Model 2: weight ~ height + age + sport
  Res.Df   RSS Df Sum of Sq    F    Pr(>F)
1     98 14457.9
2     96  979.6  2      13478 660.4 < 2.2e-16 ***

```

In der Ausgabe wird in der Spalte `RSS` die Residual-Quadratsumme für jedes Modell aufgeführt, die zugehörigen Freiheitsgrade in der Spalte `Res.Df`. In der Spalte `Df` steht deren Differenz, um wie viele zu schätzende Parameter sich die Modelle also unterscheiden. Die Reduktion der Residual-Quadratsumme beim Wechsel zum umfassenderen Modell findet sich in der Spalte `Sum of Sq`, daneben der zugehörige *F*-Wert (`F`) und *p*-Wert (`Pr(>F)`).

6.3.4 Moderierte Regression

Bei der multiplen Regression kann die Hypothese bestehen, dass die theoretischen Regressionsparameter eines Prädiktors von der Ausprägung eines anderen Prädiktors abhängen (Aiken & West, 1991). Im Rahmen eines einfachen kausalen Einflussmodells mit einem Prädiktor X_1 , einem Kriterium Y und einer Drittvariable X_2 ließe sich etwa vermuten, dass der Einfluss von X_1 auf Y davon abhängt, wie X_2 ausgeprägt ist.¹⁵ Bei quantitativen Variablen spricht man dann von einer *Moderation*, während man im Rahmen einer Varianzanalyse mit mehreren Faktoren den Begriff der *Interaktion* verwendet (Abschn. 7.6). Die Vorhersagegleichung der Regression mit zwei Prädiktoren erweitert sich durch den Interaktionsterm zu $\hat{Y} = b_0 + b_1X_1 + b_2X_2 + b_3(X_1X_2)$. Es wird also ein weiterer Vorhersageterm hinzugefügt, der gleich dem Produkt beider Prädiktoren ist. Im Aufruf von `lm()` geschieht dies durch Aufnahme des Terms `X1:X2` in die Modellformel.

Moderator sei hier das Alter bei der Regression des Körpergewichts auf die Körpergröße.

```
# Modell mit zentrierten Prädiktoren und Interaktionsterm
> heightC <- c(scale(height, center=TRUE, scale=FALSE))
> ageC     <- c(scale(age,     center=TRUE, scale=FALSE))
> fitHAI  <- lm(weight ~ heightC + ageC + heightC:ageC)
> coef(summary(fitHAI))
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 63.95331   1.17188 54.573 < 2e-16 ***
heightC      0.85591   0.16413  5.215 1.06e-06 ***
ageC        -0.44947   0.14696 -3.058  0.00288 **
heightC:ageC  0.01509   0.01945  0.776  0.43986
```

Im umgeformten Modell $\hat{Y} = (b_0 + b_2X_2) + (b_1 + b_3X_2)X_1$ wird ersichtlich, dass es sich wie jenes der einfachen linearen Regression des Prädiktors X_1 schreiben lässt, wobei y -Achsenabschnitt und Steigungsparameter nun lineare Funktionen des als Moderator betrachteten Prädiktors X_2 sind. Der Term $b_{0s} = b_0 + b_2X_2$ wird als *simple intercept*, $b_{1s} = b_1 + b_3X_2$ als *simple slope* oder auch als *marginal effect* bezeichnet. Für einen festen Wert des Moderators X_2 ergeben sich für

¹⁵Für Hinweise zur Analyse komplexerer Kausalmodelle s. Abschn. 12.3, Fußnote 33.

b_{0s} und b_{1s} konkrete Werte. Um einen Eindruck von der Bandbreite der Parameterschätzungen zu gewinnen, bieten sich für die Wahl fester Werte von X_2 dessen Mittelwert sowie die Werte \pm eine Standardabweichung um den Mittelwert an. Eine andere Möglichkeit sind der Median sowie das erste und dritte Quartil von X_2 .

Die Funktion `emtrends()` aus dem Paket `emmeans` (Lenth, 2020) berechnet die simple slopes für vorgegebene Werte des Moderators – auch für Modelle mit noch weiteren Prädiktoren, die nicht Teil einer Interaktion sind.

```
emtrends(<lm-Modell>, ~ <Moderator>, var="<Prädiktor>",
         at=list(<Moderator>=<Werte>))
```

Als erstes Argument ist ein mit `lm()` erstelltes Regressionsmodell zu übergeben. Das zweite Argument erwartet eine Modellformel, in der X_2 als Moderator auf der rechten Seite der `~` steht und die linke Seite leer ist. Für `var` ist der Name des moderierten Prädiktors X_1 zu nennen. Für welche Werte des Moderators X_2 die simple slopes b_{1s} berechnet werden sollen, kontrolliert `at`: Dies ist eine Liste mit dem Vektor der gewünschten Werte als Komponente, die den Namen des Moderators trägt.

Das Ergebnis ist in der Spalte `<Prädiktor>.trend` eine Übersicht über die geschätzten simple slopes b_{1s} mit jeweils zugehörigem Standardfehler und Konfidenzintervall. Für den zweiseitigen t -Test ist das zurückgegebene Objekt zu speichern und an `test()` zu übergeben.

```
> library(emmeans)                      # für emtrends(), test()
> ageC_vals <- c(-sd(ageC), 0, sd(ageC))
> (et <- emtrends(fitHAI, ~ageC, var="heightC",
+                   at=list(ageC=ageC_vals)))
ageC heightC.trend    SE df lower.CL upper.CL
-8.03      0.735 0.230 96    0.278    1.19
 0.00      0.856 0.164 96    0.530    1.18
 8.03      0.977 0.223 96    0.535    1.42

> test(et)
ageC heightC.trend    SE df t.ratio p.value
-8.03      0.735 0.230 96    3.190  0.0019
 0.00      0.856 0.164 96    5.215  <.0001
 8.03      0.977 0.223 96    4.388  <.0001
```

Das von `emtrends()` zurückgegebene Objekt wird von `plot()` akzeptiert, um die simple slopes b_{1s} und ihre jeweiligen Konfidenzintervalle in Abhängigkeit von der Ausprägung des Moderators in einem Diagramm darzustellen (Abb. 6.3). Das Ergebnis ist ein mit dem Paket `ggplot2` erstelltes Diagramm, das sich mit den in Kap. 15 vorgestellten Methoden anpassen lässt.

```
# Werte des Moderators, für die simple slopes des Prädiktors
# gezeigt werden sollen
> ageC_vals <- seq(-15, 20, by=5)
> et_plot <- emtrends(fitHAI, ~ageC, var="heightC",
+                      at=list(ageC=ageC_vals))

> plot(et_plot) + coord_flip() + theme_bw()
```

Zur Darstellung der vorhergesagten Werte des Regressionsmodells dient die Funktion `emmpip()`. Sie erlaubt es auf einfache Weise, verschiedene Vorhersagegeraden für den Zusammenhang von Zielgröße und Prädiktor getrennt für einzelne Werte des Moderators darzustellen.

```
emmpip(<lm-Modell>, <Moderator> ~ <Prädiktor>,
       at=list(<Moderator>=<Werte>, <Prädiktor>=<Werte>),
       CIs=TRUE, plotit=TRUE)
```

Als erstes Argument ist ein mit `lm()` erstelltes Regressionsmodell zu übergeben. Das zweite Argument erwartet eine Modellformel mit dem Moderator X_2 auf der linken Seite der `~` steht und dem Prädiktor X_1 auf der rechten Seite. Sollen Konfidenzintervalle für die Vorhersage gezeigt werden, ist `CIs=TRUE` zu setzen. Für welche Werte des Moderators sowie des Prädiktors die Vorhersagegeraden gezeigt werden sollen, kontrolliert `at`: Dies ist eine Liste mit jeweils einem Vektor der gewünschten Werte als Komponente für den Moderator und für den Prädiktor. Die Werte des Prädiktors werden auf der x -Achse abgetragen, die Werte des Moderators werden zur Definition separater Linien verwendet. Mit `plotit=FALSE` erzeugt der Aufruf kein Diagramm, sondern liefert nur die berechneten Daten zurück, die dann für ein selbst erstelltes Diagramm verwendet werden können (Abb. 6.3).

```
> r_hC <- range(heightC)
> d_ip <- emmpip(fitHAI, ageC ~ heightC,
+                   at=list(heightC=seq(r_hC[1], r_hC[2], length=20),
+                         ageC=c(-sd(ageC), 0, sd(ageC))),
+                   CIs=TRUE, plotit=FALSE) |>
+   transform(ageC=sprintf("%.2f", ageC))

> library(ggplot2)                      # für ggplot()
> ggplot(d_ip, aes(x=heightC, y=yvar, group=ageC,
+                   color=ageC, fill=ageC,
+                   ymin=LCL, ymax=UCL)) +
+   geom_ribbon(alpha=0.4, linetype="blank") +
+   geom_line() +
+   ylab("Prediction") +
+   theme_bw()
```

Es folgt die manuelle Kontrolle.

```
> coeffs <- coef(fitHAI)               # extrahiere Koeffizienten
> b0 <- coeffs[1]                     # y-Achsenabschnitt
> b1 <- coeffs[2]
> b2 <- coeffs[3]
> b3 <- coeffs[4]

# bedingte y-Achsenabschnitte
> b0 + b2*(mean(ageC) - sd(ageC))    # für M - 1sd ...
> b0 + b2* mean(ageC)                 # für M ...
> b0 + b2*(mean(ageC) + sd(ageC))    # für M + 1sd ...

# bedingte Steigungen
```

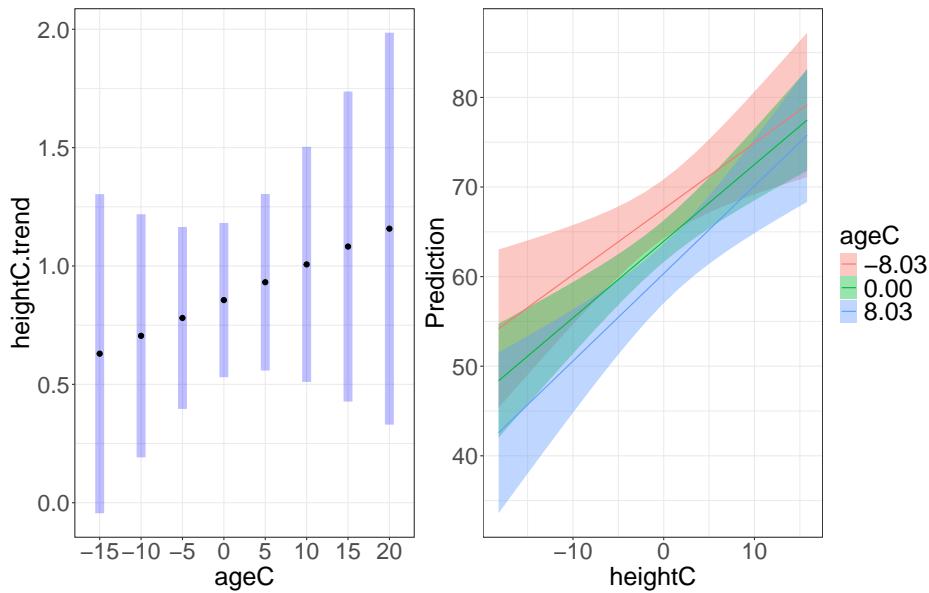


Abbildung 6.3: Simple slopes und Vorhersagegeraden in der Regression mit Moderator

```
> b1 + b3*(mean(ageC) - sd(ageC))      # für M - 1sd ...
> b1 + b3* mean(ageC)                   # für M ...
> b1 + b3*(mean(ageC) + sd(ageC))      # für M + 1sd ...
```

6.4 Regressionsmodelle auf andere Daten anwenden

`predict()` wendet ein von `lm()` angepasstes Regressionsmodell auf neue Daten an, um für sie die Vorhersage \hat{Y} zu berechnen (vgl. `?predict.lm`).

```
predict(object=<lm-Modell>, newdata=<Datensatz>, se.fit=FALSE,
        interval=NULL, level=<Breite Konfidenzintervall>)
```

Als erstes Argument ist ein von `lm()` erzeugtes Objekt zu übergeben. Werden alle weiteren Argumente weggelassen, liefert die Funktion die Vorhersage für die ursprünglichen Prädiktorwerte zurück, also `fitted(<lm-Modell>)`. Wird unter `newdata` ein Datensatz übergeben, der Variablen mit denselben Namen wie jene der ursprünglichen Prädiktoren enthält, so wird \hat{Y} für die Werte dieser neuen Variablen berechnet.¹⁶ Wenn die Modellformel Transformationen von Variablen – etwa mit `I()` (Abschn. 5.2) oder `poly()` (Abschn. 6.6.3) – enthält, werden diese von `predict()` automatisch aus den Variablen des Datensatzes gebildet. Sollen die Standardabweichungen für \hat{Y} ermittelt werden, ist `se.fit=TRUE` zu setzen.

Mit dem Argument `interval="confidence"` berechnet `predict()` zusätzlich für jeden Wert der Prädiktorvariablen die Grenzen des punktweisen Konfidenzintervalls für den geschätzten

¹⁶Handelt es sich etwa im Rahmen einer Kovarianzanalyse (Abschn. 7.9) um einen kategorialen Prädiktor – ein Objekt der Klasse `factor`, so muss die zugehörige Variable in `newdata` dieselben Stufen in derselben Reihenfolge beinhalten wie die Variable des ursprünglichen Modells – selbst wenn nicht alle Faktorstufen tatsächlich als Ausprägung vorkommen.

Parameter, hier also den bedingten Erwartungswert von Y . Die Breite des Konfidenzintervalls wird mit `level` kontrolliert. Mit `interval="prediction"` erhält man stattdessen die Grenzen des Toleranzintervalls für die Vorhersage einer neuen Beobachtung von Y . Das Toleranzintervall spiegelt einerseits die Unsicherheit der Parameterschätzer, andererseits auch die geschätzte Streuung $\hat{\sigma}$ von Y um den bedingten Erwartungswert (Standardschätzfehler) wider. Die Ausgabe erfolgt 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 des Modells mit nur dem Prädiktor `height` wird für den Aufruf von `predict()` zunächst ein Datensatz mit einer passend benannten Variable erzeugt.

```
> newHeight <- c(177, 150, 192, 189, 181)           # neue Daten
> newDf      <- data.frame(height=newHeight)        # Datensatz
> predict(fitH, newDf, interval="prediction", level=0.95)
    fit      lwr      upr
1 66.00053 41.76303 90.23803
2 43.68101 18.07599 69.28604
3 78.40026 53.47502 103.32550
4 75.92032 51.21385 100.62678
5 69.30713 44.98704 93.62721

# Kontrolle der Vorhersage durch Extrahieren der Regressionsparameter
# und Einsetzen der neuen Daten in die Vorhersagegleichung
> (coeffs <- coef(fitH))                           # Parameter
(Intercept)      height
-80.3163014   0.8266488

> coeffs[2]*newHeight + coeffs[1]                  # Vorhersage
[1] 66.00053 43.68101 78.40026 75.92032 69.30713
```

Der Vertrauensbereich um die Vorhersage für die ursprünglichen Daten lässt sich auch grafisch darstellen (Abb. 6.4).

```
> plot(weight ~ height, pch=20, xlab="Prädiktor",
+       ylab="Kriterium und Vorhersage", xaxs="i",
+       main="Daten und Vorhersage durch Regression")

# Vertrauensintervall um Vorhersage für Originaldaten
> predOrg <- predict(fitH, interval="confidence", level=0.95)
> hOrd     <- order(height)
> polygon(c(height[hOrd],           height[rev(hOrd)]),
+           c(predOrg[hOrd, "lwr"], predOrg[rev(hOrd), "upr"]),
+           border=NA, col=rgb(0.7, 0.7, 0.7, 0.6))

> abline(fitH, col="blue")                          # Regressionsgerade
> legend(x="bottomright", legend=c("Daten", "Vorhersage",
+ "Vertrauensbereich"), pch=c(20, NA, NA), lty=c(NA, 1, 1),
+ lwd=c(NA, 1, 8), col=c("black", "blue", "gray"))
```

Daten und Vorhersage durch Regression

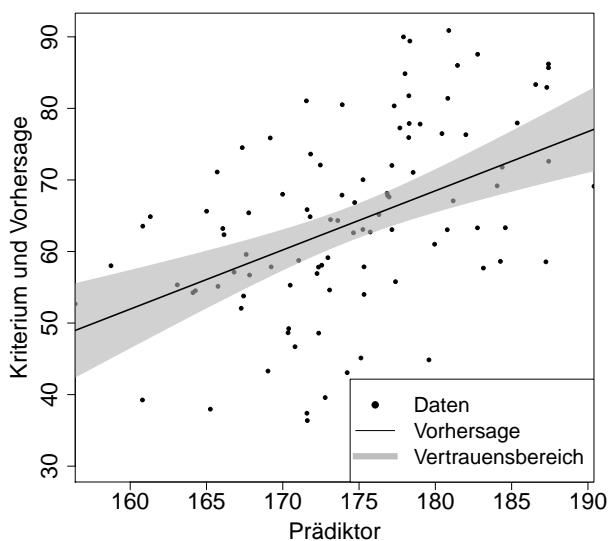


Abbildung 6.4: Lineare Regression: Prädiktor, Kriterium und Vorhersage mit Vertrauensbereich

6.5 Regressionsdiagnostik

Einer konventionellen Regressionsanalyse liegen verschiedene Annahmen zugrunde, deren Gültigkeit vorauszusetzen ist, damit die berechneten Standardfehler der Parameterschätzungen und die p -Werte korrekt sind (Abschn. 12.9.4). Dazu gehören bedingte Normalverteiltheit und gemeinsame Unabhängigkeit der Messfehler des Kriteriums, die zudem unabhängig von den Prädiktoren sein müssen. Hinzu kommt die *Homoskedastizität*, also die Gleichheit aller bedingten Fehlervarianzen (Abb. 12.6).

Mit Hilfe der Regressionsdiagnostik soll zum einen geprüft werden, ob die Daten mit den gemachten Annahmen konsistent sind. Zum anderen kann die Parameterschätzung der konventionellen Regression durch wenige Ausreißer überproportional beeinflusst werden. Daher ist es von Interesse, diese zu identifizieren und den Einfluss einzelner Beobachtungen auf das Ergebnis zu bestimmen. Schließlich ist in der multiplen Regression das Ausmaß der *Multikollinearität* bedeutsam, also die lineare Abhängigkeit der Prädiktoren untereinander. Für eine ausführliche Darstellung vgl. [Fox und Weisberg \(2019\)](#) sowie das zugehörige Paket `car` für Funktionen, mit denen sich eine Vielzahl diagnostischer Diagramme erstellen lassen.

6.5.1 Extremwerte, Ausreißer und Einfluss

Unter einem Ausreißer sind hier Beobachtungen zu verstehen, deren Kriteriumswert stark vom Kriteriumswert anderer Beobachtungen abweicht, die ähnliche Prädiktorwerte besitzen. Extremwerte einer gegebenen Variable zeichnen sich dadurch aus, dass sie weit außerhalb der Verteilung der übrigen Beobachtungen liegen. Die grafische Beurteilung, ob Extremwerte vorliegen, lässt sich getrennt für jede Variable etwa durch Histogramme oder Boxplots der z -standardisierten Variable durchführen (Abb. 6.5; Abschn. 14.6.1, 14.6.3, 15.2.4).

Als numerische Indikatoren eignen sich die z -standardisierten Werte, an denen abzulesen ist, wie viele Streuungseinheiten ein Wert vom Mittelwert entfernt ist. Als numerisches Maß der multivariaten Extremwertanalyse bietet sich die Mahalanobis-Distanz als Verallgemeinerung der z -Transformation an (Abschn. 12.1.4). Sie repräsentiert den Abstand eines Datenpunkts zum Zentroid der Verteilung, wobei der Gesamtform der Verteilung i. S. der Kovarianzmatrix Rechnung getragen wird. Für die grafische Darstellung der gemeinsamen Verteilung von zwei oder drei Variablen s. Abschn. 14.6.8, 14.7.¹⁷

```
> Xpred <- cbind(height, age, sport)      # Prädiktoren als Datenmatrix
> Xz    <- scale(Xpred)                  # z-Transformierte
> boxplot(Xz, main="Verteilung standardisierte Prädiktoren")
> summary(Xz)
  height           age           sport
Min. : -2.534e+00 Min. : -1.952e+00 Min. : -1.796e+00
1st Qu.: -6.040e-01 1st Qu.: -7.110e-01 1st Qu.: -6.884e-01
Median : 1.075e-02 Median : -1.069e-01 Median : -9.467e-02
Mean   : 1.781e-15 Mean   : -8.978e-17 Mean   : 1.087e-16
3rd Qu.: 6.338e-01 3rd Qu.: 7.411e-01 3rd Qu.: 6.519e-01
Max.   : 2.200e+00 Max.   : 2.839e+00 Max.   : 2.727e+00

# Mahalanobis-Distanz der Beobachtungen zum Zentroid der Prädiktoren
> ctrX  <- colMeans(Xpred)            # Zentroid
> sX    <- cov(Xpred)               # Kovarianzmatrix
> mahaSq <- mahalanobis(Xpred, ctrX, sX) # quadrierte M-Distanzen
> summary(sqrt(mahaSq))
  Min. 1st Qu. Median  Mean 3rd Qu.  Max.
0.2855 1.1640 1.5940 1.5970 1.9750 3.3090
```

Durch Extremwerte oder Ausreißer wird die ermittelte Vorhersagegleichung womöglich in dem Sinne verzerrt, dass sie die Mehrzahl der Daten nicht mehr gut repräsentiert. Um den Einfluss der einzelnen Beobachtungen auf die Parameterschätzungen der Regression direkt zu quantifizieren, existieren verschiedene Kennwerte, darunter der Hebelwert h (*leverage*). h wird durch `hatvalues(lm-Modell)` berechnet.¹⁸ Für Modelle, die einen absoluten Term b_0 einschließen, kann h Werte im Intervall $[\frac{1}{n}, 1]$ annehmen, wobei n die Anzahl an Beobachtungen ist. Der Mittelwert ist dann gleich $\frac{p+1}{n}$ mit $p + 1$ als Anzahl zu schätzender Parameter der Regression (p Prädiktoren sowie absoluter Term). Um besonders große Hebelwerte zu identifizieren, kann ihre Verteilung etwa über ein Histogramm oder einen *spike-plot* veranschaulicht werden (Abb. 6.5, Abschn. 14.2). Als numerisches Kriterium für auffällig große Hebelwerte dient bisweilen das Zwei- bis Dreifache seines Mittelwerts.

```
> fitHAS <- lm(weight ~ height + age + sport)  # Regression
> h       <- hatvalues(fitHAS)                  # Hebelwerte
```

¹⁷Da Extremwerte die Lage und Streuung der Daten mit beeinflussen, sollten hierfür evtl. robuste Schätzer in Betracht gezogen werden (Abschn. 2.7.9). Robuste Schätzungen für die Kovarianzmatrix können etwa an das Argument `cov` von `mahalanobis()` übergeben werden. Für fortgeschrittene Tests, ob Ausreißer in multivariaten Daten vorliegen, vgl. `aq.plot()` und `pcout()` aus dem Paket `mvoutlier` (Filzmoser & Gschwandtner, 2018).

¹⁸Zudem ist h_i gleich dem i -ten Eintrag H_{ii} in der Diagonale der Hat-Matrix H (Abschn. 6.3.1).

```
> hist(h, main="Histogramm der Hebelwerte")      # Histogramm
> summary(h)
   Min. 1st Qu. Median Mean 3rd Qu. Max.
0.01082 0.02368 0.03568 0.04000 0.04942 0.12060
```

Zur Kontrolle lässt sich die Beziehung nutzen, dass die quadrierte Mahalanobisdistanz einer Beobachtung i zum Zentroid der Prädiktoren gleich $(n - 1)(h_i - \frac{1}{n})$ ist.

```
> all.equal(mahaSq, (N-1) * (h - (1/N)), check.attributes=FALSE)
[1] TRUE
```

Die Indizes DfFITS bzw. DfBETAS liefern für jede Beobachtung ein standardisiertes Maß, wie stark sich die Vorhersagewerte (DfFITS) bzw. jeder geschätzte Parameter (DfBETAS) tatsächlich ändern, wenn die Beobachtung aus den Daten ausgeschlossen wird. In welchem Ausmaß sich dabei der Standardschätzfehler ändert, wird über das Verhältnis beider resultierenden Werte (mit bzw. ohne ausgeschlossene Beobachtung) ausgedrückt. Cooks Distanz ist ein weiteres Einflussmaß, das sich als $\frac{E^2}{\hat{\sigma}^2(1-h)^2} \cdot \frac{h}{p+1}$ berechnet, wobei E für die Residuen $Y - \hat{Y}$ und $\hat{\sigma}$ für den Standardschätzfehler der Regression mit $p + 1$ Parametern steht.

Mit `influence.measures(lm-Modell)` lassen sich die genannten Kennwerte gleichzeitig berechnen. Auffällige Beobachtungen können aus der zurückgegebenen Liste mit `summary()` extrahiert werden, wobei der Wert der abweichenden diagnostischen Größe durch einen Stern * markiert ist.

```
> inflRes <- influence.measures(fitHAS)      # Diagnosegrößen
> summary(inflRes)                          # auffällige Beobachtungen
Potentially influential observations of
lm(formula = weight ~ height + age + sport) :
    dfb.1 dfb.hght dfb.age dfb.sppt dffit cov.r cook.d hat
6   -0.05     0.07   -0.23    0.25   0.40  0.86_*  0.04  0.03
13  -0.21     0.21   -0.10    0.36   0.41  1.13_*  0.04  0.12_*
30   0.19    -0.23    0.17   -0.01  -0.42  0.74_*  0.04  0.02
31  -0.08     0.11   -0.08   -0.05   0.27  0.86_*  0.02  0.01
67   0.02    -0.02   -0.03    0.03  -0.05  1.17_*  0.00  0.11
72  -0.03     0.03    0.04    0.01   0.05  1.17_*  0.00  0.11
86   0.10    -0.16    0.60   -0.25  0.68_*  0.90  0.11  0.08
97  -0.01     0.01   -0.01    0.01   0.02  1.16_*  0.00  0.10
```

In der Ausgabe beziehen sich die Spalten `dfb.1` auf das DfBETA jedes Prädiktors (inkl. des absoluten Terms 1 in der ersten Spalte), `dffit` auf DfFITS, `cov.r` auf das Verhältnis der Standardschätzfehler, `cook.d` auf Cooks Distanz und `hat` auf den Hebelwert. Für Funktionen zur separaten Berechnung der Maße vgl. `?influence.measures`.

```
> cooksDst <- cooks.distance(fitHAS)          # Cooks Distanz
> plot(cooksDst, main="Cooks Distanz", type="h") # spike-plot

# manuelle Berechnung
> P    <- 3                                     # Anzahl Prädiktoren
> E    <- residuals(fitHAS)                      # Residuen
```

```
> MSE <- sum(E^2) / (N - (P+1))          # quadr. Standardschätzfehler
> CD  <- (E^2 / (MSE * (1-h)^2)) * (h / (P+1))    # Cooks Distanz
> all.equal(cooksDst, CD)                  # Kontrolle
[1] TRUE
```

Grafisch aufbereitete Informationen über den Einfluss einzelner Beobachtungen sowie über die Verteilung der Residuen (s. u.) liefert auch eine mit `plot(<lm-Modell>, which=1:6)` aufzurufende Serie von Diagrammen. Über das Argument `which` können dabei einzelne Grafiken der Serie selektiv gezeigt werden. Vergleiche dazu auch `influencePlot()` und `influenceIndexPlot()` aus dem `car` Paket.

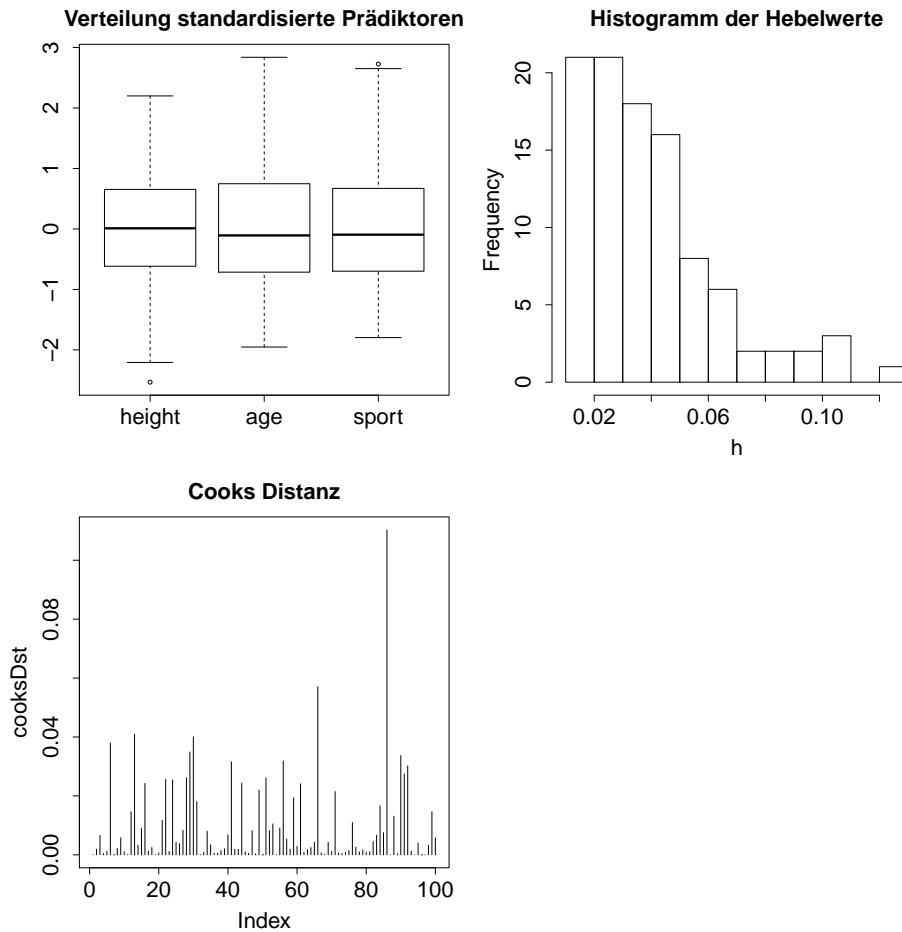


Abbildung 6.5: Beurteilung von Extremwerten und Einflussgrößen in der Regression

6.5.2 Verteilungseigenschaften der Residuen

Die Verteilung der Residuen in Abhängigkeit von einzelnen Prädiktoren kann in einer multiplen Regression das Verständnis der Zusammenhänge erleichtern. Das Paket `car` bietet hierfür die Funktion `residualPlots()` mit zahlreichen Optionen (Abb. 6.6).

```
> library(car) # für residualPlots()
> residualPlots(fitHAS, tests=FALSE)
```

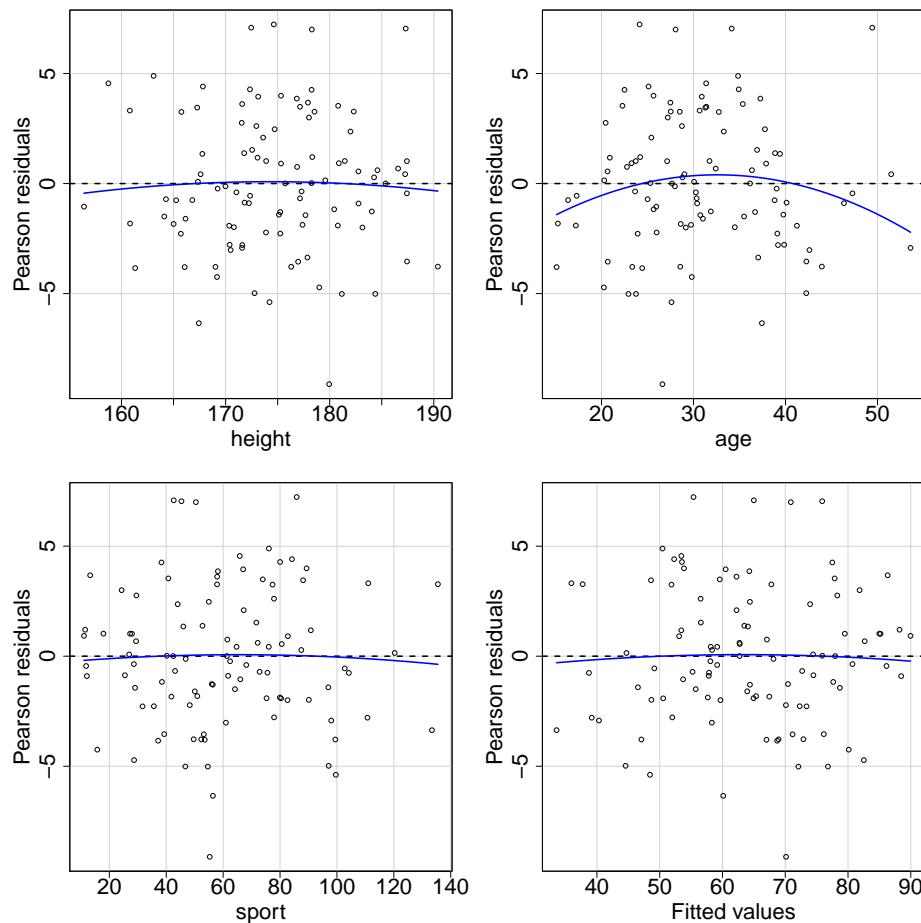


Abbildung 6.6: Prädiktor-Residuen Diagramme in der multiple Regression

Anhand verschiedener grafischer Darstellungen der Residuen einer Regression lässt sich heuristisch beurteilen, ob die vorliegenden Daten mit den Voraussetzungen der Normalverteiltheit, Unabhängigkeit und Homoskedastizität der Messfehler vereinbar sind. Als Grundlage können die Residuen $E = Y - \hat{Y}$ selbst, oder aber zwei Transformationen von ihnen dienen: Die *standardisierten* und *studentisierten* Residuen besitzen eine theoretische Streuung von 1 und ergeben sich als $\frac{E}{\hat{\sigma}\sqrt{1-h}}$, wobei $\hat{\sigma}$ eine Schätzung der theoretischen Fehlerstreuung ist (Abschn. 12.9.4).

Für die standardisierten Residuen wird der Standardschätzfehler als globale Schätzung in der Rolle von $\hat{\sigma}$ verwendet, bei den studentisierten Residuen dagegen eine beobachtungsweise Schätzung $\hat{\sigma}_{(i)}$. Dabei wird im Fall *extern* studentisierter Residuen (*leave-one-out*) $\hat{\sigma}_{(i)}$ für jede Beobachtung i auf Basis des Regressionsmodells berechnet, in das alle Daten bis auf die der i -ten Beobachtung einfließen. Für die im Folgenden aufgeführten Prüfmöglichkeiten werden oft standardisierte oder studentisierte Residuen E gegenüber vorgezogen.

`lm.influence()` speichert $\hat{\sigma}_{(i)}$ in der Komponente `sigma` der ausgegebenen Liste. Die Re-

siduen selbst lassen sich mit `residuals()` für E , `rstandard()` für die standardisierten und `rstudent()` für die extern studentisierten Residuen ermitteln (s. auch Abschn. 13.1.2). Bei allen Funktionen ist als Argument ein von `lm()` erstelltes Modell zu übergeben.

```
> Estnd <- rstandard(fitHAS) # standardisierte Residuen
> all.equal(Estnd, E / sqrt(MSE * (1-h))) # manuelle Kontrolle
[1] TRUE

# studentisierte Residuen und manuelle Kontrolle
> Estud <- rstudent(fitHAS)
> all.equal(Estud, E / (lm.influence(fitHAS)$sigma * sqrt(1-h)))
[1] TRUE
```

Für eine visuell-exploratorische Beurteilung der Normalverteiltheit wird die Verteilung der bevorzugten Residuen-Variante mit einem Histogramm der relativen Klassenhäufigkeiten dargestellt, dem die Dichtefunktion der Standardnormalverteilung hinzugefügt wurde (Abb. 6.7, Abschn. 15.2.3, 15.4). Das Histogramm sollte in seiner Form nicht stark von der Dichtefunktion abweichen. Zudem lässt sich ein Q-Q-plot nutzen, um die empirischen Quantile der Residuen mit jenen der Standardnormalverteilung zu vergleichen. Die Datenpunkte sollten hier auf einer Geraden liegen (Abb. 6.7, Abschn. 14.6.5). Für einen inferenzstatistischen Test mit der H_0 , dass Normalverteilung vorliegt, bietet sich jener nach Shapiro-Wilk an (Abschn. 7.1).

```
> hist(Estud, main="Histogramm studentisierte Residuen", freq=FALSE)

# Dichtefunktion der Standardnormalverteilung hinzufügen
> curve(dnorm(x, mean=0, sd=1), col="red", lwd=2, add=TRUE)
> qqnorm(Estud, main="Q-Q-Plot studentisierte Residuen") # Q-Q-plot
> qqline(Estud, col="red", lwd=2) # Referenzgerade für Normalverteilung

# Shapiro-Wilk-Test auf Normalverteilung der studentisierten Residuen
> shapiro.test(Estud)
Shapiro-Wilk normality test
data: Estud
W = 0.9879, p-value = 0.4978
```

Soll eingeschätzt werden, ob die Annahme von Homoskedastizität plausibel ist, kann die bevorzugte Residuen-Variante auf der Ordinate gegen die Vorhersage auf der Abszisse abgetragen werden (*spread-level-plot*, Abb. 6.7).¹⁹ Die Datenpunkte sollten überall gleichmäßig um die 0-Linie streuen. Anhand desselben Diagramms kann auch die Unabhängigkeit der Messfehler heuristisch geprüft werden: Die Residuen sollten eine Verteilung aufweisen, die nicht systematisch mit der Vorhersage zusammenhängt.²⁰

```
# studentisierte Residuen gegen Vorhersage darstellen
> plot(fitted(fitHAS), Estud, pch=20, xlab="Vorhersage",
```

¹⁹ Mitunter werden hierfür auch die Beträge der Residuen bzw. deren Wurzel gewählt (*scale-location plot*). Der Breusch-Pagan-Test auf Heteroskedastizität kann mit `bptest()` aus dem Paket `lmtest` (Zeileis & Hothorn, 2002) durchgeführt werden.

²⁰ Für den Durbin-Watson-Test auf Autokorrelation der Residuen vgl. `durbinWatsonTest()` aus dem Paket `car`. Das Autokorrelations-Diagramm der Residuen erzeugt `acf(residuals(<lm-Objekt>))`.

```
+     ylab="studentisierte Residuen", main="Spread-Level-Plot")  
  
> abline(h=0, col="red", lwd=2)           # Referenz für Modellgültigkeit
```

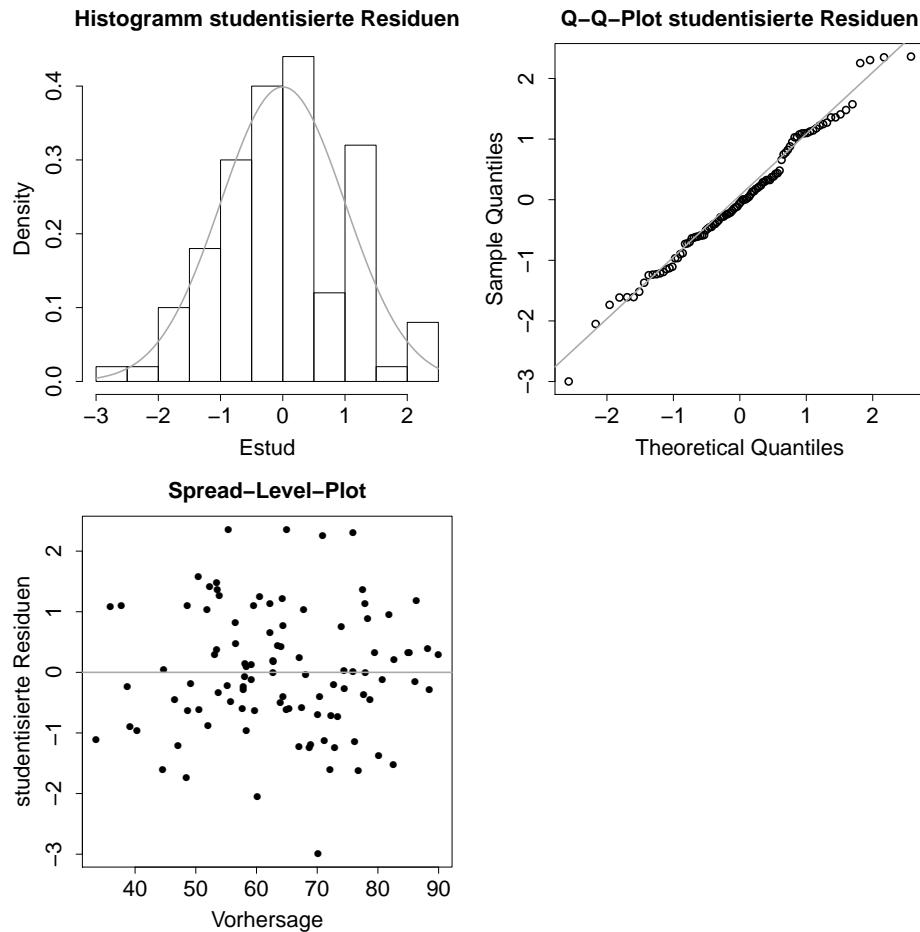


Abbildung 6.7: Grafische Prüfung der Verteilungsvoraussetzungen für eine Regressionsanalyse

Für manche Fälle, in denen die Daten darauf hindeuten, dass die Verteilung einer Variable Y nicht den Voraussetzungen genügt, können streng monotone Transformationen die Verteilung günstig beeinflussen, bei echt positiven Daten etwa Potenzfunktionen der Form Y^λ : Dazu zählen der Kehrwert Y^{-1} , der Logarithmus $\ln Y$ (per definitionem für $\lambda = 0$), die Quadratwurzel $Y^{\frac{1}{2}}$ (z. B. bei absoluten Häufigkeiten), oder der Arkussinus der Quadratwurzel $\arcsin Y^{\frac{1}{2}}$ (z. B. bei Anteilen). In der Regression finden bisweilen Box-Cox-Transformationen $\frac{Y^\lambda - 1}{\lambda}$ für $\lambda \neq 0$ bzw. $\ln Y$ für $\lambda = 0$ Verwendung, die ebenfalls echt positive Daten voraussetzen. Für sie stellt das Paket `car` die in Kombination miteinander zu verwendenden Funktionen `powerTransform()` und `bcPower()` bereit.

```
> powerTransform(<lm-Modell>, family="bcPower")  
> bcPower(<Vektor>, <lambd>)
```

Als erstes Argument von `powerTransform()` kann ein mit `lm()` erstelltes Modell angegeben werden. Für die Maximum-Likelihood-Schätzung des Parameters λ ist das Argument `family`

auf "bcPower" zu setzen. λ erhält man aus dem zurückgegebenen Objekt durch `coef()`. Die Box-Cox-Transformation selbst führt `bcPower()` durch und benötigt dafür als Argument zum einen den Vektor der zu transformierenden Werte, zum anderen den Parameter λ der Transformation.

```
> library(car)                                # für boxCox(), powerTransform()
> lamObj <- powerTransform(fitHAS, family="bcPower")
> (lambda <- coef(lamObj))                  # maximum-likelihood lambda
Y1
1.057033

# transformiertes Kriterium und manuelle Kontrolle
> wTrans <- bcPower(weight, coef(lambda))      # Transformation
> all.equal(wTrans, ((weight^lambda) - 1) / lambda) # manuell
[1] TRUE
```

6.5.3 Multikollinearität

Multikollinearität liegt in einer multiplen Regression dann vor, wenn sich die Werte eines Prädiktors gut aus einer Linearkombination 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 berücksichtigt werden. Andererseits ergeben sich Schwierigkeiten bei der Interpretation der b_j - bzw. der standardisierten b_j^z -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.²¹

Ob paarweise lineare Abhängigkeiten vorliegen, lässt sich anhand der Korrelationsmatrix \mathbf{R}_x der Prädiktoren prüfen.

```
> (Rx <- cor(cbind(height, age, sport)))      # 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
```

Die Diagonalelemente der Inversen \mathbf{R}_x^{-1} liefern den *Varianzinflationsfaktor* VIF_j jedes Prädiktors j als weitere Möglichkeit zur Kollinearitätsdiagnostik: $\sqrt{VIF_j}$ ist der Faktor, um den das Konfidenzintervall für das wahre β_j -Gewicht breiter als im analogen Fall linear unabhängiger Prädiktoren ist. VIF_j berechnet sich alternativ als $\frac{1}{1-R_j^2}$, also als Kehrwert der Toleranz $1 - R_j^2$, wobei R_j^2 der Determinationskoeffizient bei der Regression des Prädiktors j auf alle

²¹Auf numerischer Seite bringt starke Multikollinearität das Problem mit sich, dass die interne Berechnung der Parameterschätzungen anfälliger für Fehler werden kann, die aus der notwendigen Ungenauigkeit der Repräsentation von Gleitkommazahlen in Computern herrühren (Abschn. 1.4.6).

übrigen Prädiktoren ist. Aus dem Paket `car` stammt die Funktion `vif(<lm-Modell>)`, die als Argument ein durch `lm()` erstelltes lineares Modell erwartet.

```
> library(car)                                # für vif()
> vif(fitHAS)
  height      age      sport
1.054409  1.017361  1.059110
```

In der Ausgabe findet sich unter dem Namen jedes Prädiktors der 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. Es folgt die manuelle Kontrolle anhand der Regressionen jeweils eines Prädiktors auf alle übrigen.

```
# Regression jeweils eines Prädiktors auf alle übrigen
> fitHeight <- lm(height ~ age + sport)
> fitAge    <- lm(age    ~ height + sport)
> fitSport   <- lm(sport   ~ height + age)

# VIF_j aus zugehörigem Determinationskoeffizienten R^2
> 1 / (1 - summary(fitHeight)$r.squared)      # VIF height
[1] 1.054409

> 1 / (1 - summary(fitAge)$r.squared)          # VIF age
[1] 1.017361

> 1 / (1 - summary(fitSport)$r.squared)         # VIF sport
[1] 1.05911

# alternativ: Diagonalelemente der Inversen der Korrelationsmatrix
> diag(solve(Rx))
  height      age      sport
1.054409  1.017361  1.059110
```

Ein weiterer Kennwert zur Beurteilung von Multikollinearität ist die Kondition κ der Designmatrix \mathbf{X} des meist mit standardisierten Variablen gebildeten linearen Modells (Abschn. 12.1.5, 12.9.1). Werte von $\kappa > 20$ sprechen einer Faustregel folgend für Multikollinearität. Zur Berechnung dient `kappa(<lm-Modell>, exact=FALSE)`, wobei für eine numerisch aufwendigere, aber präzisere Bestimmung von κ das Argument `exact=TRUE` zu setzen ist. Neben κ eignen sich zur differenzierteren Diagnose auch die Eigenwerte von $\mathbf{X}^\top \mathbf{X}$ selbst sowie ihr jeweiliger Konditionsindex.²²

```
# Regressionsmodell mit standardisierten Prädiktoren
# hier vorausgesetzt: keine fehlenden Werte
> lmScl <- lm(scale(weight) ~ scale(height) + scale(age) + scale(sport))
> kappa(lmScl, exact=TRUE)
```

²²Fortgeschrittene Methoden zur Diagnostik von Multikollinearität enthält das Paket `perturb` ([Hendrickx, 2019](#)).

```
[1] 1.279833
```

```
> X      <- model.matrix(lmScl)          # Designmatrix
> (eigVals <- eigen(t(X) %*% X)$values)    # Eigenwerte von X^t * X
[1] 119.93081 103.85018 100.00000  73.21902

# Konditionsindizes: jeweils Wurzel aus Eigenwert / (Minimum != 0)
> sqrt(eigVals / min(eigVals[eigVals >= .Machine$double.eps]))
[1] 1.279833 1.190945 1.168660 1.000000
```

Wenn von den ursprünglichen Variablen zu zentrierten oder standardisierten Variablen übergegangen wird, ändern sich die VIF_j Werte nur dann, wenn multiplikative Terme, also Interaktionen in der Regression einbezogen sind (Abschn. 6.3.4). Dagegen ändert sich κ bei solchen Variablentransformationen praktisch immer.²³

```
# VIF: Regression mit standardisierten Variablen -> kein Unterschied
> vif(lmScl)
scale(height)  scale(age)  scale(sport)
1.054409     1.017361     1.059110

# kappa: Regression mit ursprünglichen Variablen -> Unterschied
> kappa(lm(weight ~ height + age + sport), exact=TRUE)
[1] 4804.947
```

6.6 Erweiterungen der linearen Regression

6.6.1 Robuste Regression

Die Parameterschätzung *robuster* Regressionsverfahren soll weniger sensibel auf die Verletzung von Voraussetzungen und auf Ausreißer reagieren. Das Paket **robustbase** stellt für zwei Varianten der robusten Regression **lmrob()** und **ltsReg()** bereit. Einen Überblick über weitere Quellen gibt der Abschnitt *Robust Statistical Methods* der CRAN Task Views ([Maechler, 2020](#)).

```
> library(robustbase)                      # für lmrob()
> fitLMR <- lmrob(weight ~ height + age + sport, setting="KS2014")
> summary(fitLMR)                         # gekürzte Ausgabe ...
Residuals:
    Min      1Q  Median      3Q      Max 
-9.1512 -1.9595 -0.1204  2.4366  7.2985
```

Coefficients:

²³Ursache dafür ist die Änderung der Eigenwerte bei Datentransformationen: Ist \mathbf{X} die Designmatrix des ursprünglichen Modells und \mathbf{X}' die Designmatrix des Modells der transformierten Daten, so gehen die Eigenwerte von $(\mathbf{X}')^\top \mathbf{X}'$ nicht auf einfache Weise aus denen von $\mathbf{X}^\top \mathbf{X}$ hervor. Insbesondere verändern sich der größte und kleinste Eigenwert jeweils unterschiedlich, so dass deren Quotient nicht konstant ist.

```

Estimate Std. Error t value Pr(>|t|)
(Intercept) 7.97897   8.41419   0.948   0.345
height       0.51710   0.04732  10.927 < 2e-16 ***
age         -0.29663   0.04175  -7.105 2.11e-10 ***
sport        -0.41546   0.01230 -33.785 < 2e-16 ***

```

Robust residual standard error: 3.21
Multiple R-squared: 0.9458, Adjusted R-squared: 0.9441

Allgemein können Bootstrap-Verfahren geeignet sein, trotz verletzter Modellvoraussetzungen angemessene Standardfehler der Parameterschätzungen zu erhalten (Abschn. 11.1.4).

Speziell für Schätzungen der Standardfehler der Regression unter Heteroskedastizität vgl. `sandwich(<lm-Modell>)` und `vcovHC(<lm-Modell>)` aus dem Paket `sandwich` (Zeileis, 2004). Diese Funktionen bestimmen die Kovarianzmatrix der Parameterschätzer eines mit `lm()` angepassten Modells neu. Wald-Tests der Parameter können dann mit `coeftest(<lm-Modell>, vcov=<Schätzer>)` aus dem Paket `lmtest` durchgeführt werden, wobei für das Argument `vcov` das Ergebnis von `sandwich()` oder `vcovHC()` anzugeben ist.

```

> library(sandwich)                      # für vcovHC()
> library(lmtest)                        # für coeftest()
> hcSE <- vcovHC(fitHAS, type="HC3")
> coeftest(fitHAS, vcov=hcSE)
t test of coefficients:

Estimate Std. Error t value Pr(>|t|)
(Intercept) 8.101233   8.083794   1.0022   0.3188
height       0.514893   0.046507  11.0714 < 2.2e-16 ***
age         -0.286646   0.042196  -6.7932 9.177e-10 ***
sport        -0.415952   0.011293 -36.8330 < 2.2e-16 ***

```

6.6.2 Penalisierte Regression

Bei hoher Multikollinearität und in Situationen mit sehr vielen Prädiktoren (relativ zur Anzahl verfügbarer Beobachtungen), kommen *penalisierte* Regressionsverfahren in Betracht. Sie beschränken den geschätzten Parametervektor $\hat{\beta}$ ohne $\hat{\beta}_0$ in seiner Norm, reduzieren also den Betrag der Parameterschätzungen (*shrinkage*). Die resultierenden Parameterschätzungen sind nicht mehr unverzerrt, für den Preis eines geringen bias können sie jedoch eine deutlich geringere Varianz der Schätzungen aufweisen als die übliche lineare Regression. Penalisierte Regressionen werden üblicherweise mit standardisierten Prädiktoren und Kriterium durchgeführt. Bei kategorialen Prädiktoren kann es sinnvoll sein, von der Dummy-Codierung (Treatment-Kontraste) zur Effektcodierung zu wechseln (Abschn. 12.9.2). Penalisierte Regressionsmodelle sind auch für kategoriale Zielvariablen in verallgemeinerten linearen Modellen wie der logistischen Regression oder der Poisson-Regression anwendbar (Kap. 8).

Die Ridge-Regression wird durch `lm.ridge()` aus dem MASS Paket (Venables & Ripley, 2002) bereitgestellt. Die Funktion akzeptiert eine Modellformel und für das Argument `lambda` den

Wert des Tuning-Parameters λ . Dieser kontrolliert, wie stark das shrinkage ausfällt. Im ersten Schritt empfiehlt es sich, für `lambda` einen Vektor mit vielen Werten zu übergeben, etwa im Intervall [0.01, 10000]. `lm.ridge()` berechnet dann für jeden Wert von λ den Vorhersagefehler aus der verallgemeinerten Kreuzvalidierung (GCV, s. Abschn. 13.1.2 sowie Abb. 6.8).

```
> library(MASS)                                # für lm.ridge(), select()
> lambdas <- 10^(seq(-2, 4, length=100))      # Tuning-Parameter
> ridgeGCV <- lm.ridge(scale(weight) ~ scale(height) + scale(age) +
+                         scale(sport), lambda=lambdas)
```

Im Anschluss erfährt man mit `select(<lm.ridge-Objekt>)`, für welchen Wert von λ der Kreuzvalidierungsfehler sein Minimum erreicht.

```
> select(ridgeGCV)
modified HKB estimator is 0.06879525
modified L-W estimator is 0.06014114
smallest value of GCV at 0.2477076
```

In einem erneuten Aufruf von `lm.ridge()` kann schließlich `lambda` auf den vorher identifizierten Wert mit dem geringsten Kreuzvalidierungsfehler gesetzt werden. Aus dem erzeugten Objekt extrahiert `coef()` dann die Parameterschätzungen.

```
# lambda für geringsten Kreuzvalidierungsfehler
> lambda <- ridgeGCV$lambda[ridgeGCV$GCV == min(ridgeGCV$GCV)]
> ridgeSel <- lm.ridge(scale(weight) ~ scale(height) + scale(age) +
+                         scale(sport), lambda=lambda)

> coef(ridgeSel)                               # Parameterschätzungen
      scale(height)    scale(age)    scale(sport)
-3.765673e-16  2.743740e-01 -1.706931e-01 -8.475898e-01
```

Die Methoden LASSO und elastic net wählen anders als die Ridge-Regression gleichzeitig auch eine Teilmenge von Prädiktoren aus, indem sie abhängig von λ einzelne Parameterschätzungen auf 0 reduzieren. Eine Umsetzung von ridge, LASSO und elastic net liefern `cv.glmnet()` sowie `glmnet()` aus dem Paket `glmnet` (Friedman, Hastie & Tibshirani, 2010).

Anders als die bisher vorgestellten Funktionen akzeptieren `cv.glmnet()` und `glmnet()` keine Modellformel.²⁴ Stattdessen muss für `x` die Matrix der Prädiktoren übergeben werden und für `y` der Vektor der modellierten Zielvariable. Dabei berechnet die Funktion `cv.glmnet()` für von ihr selbst gewählte Werte von λ den Vorhersagefehler aus der verallgemeinerten Kreuzvalidierung für `nfolds` viele Partitionen (Abschn. 13.1.2). Die zurückgegebene Liste speichert in der Komponente `lambda.min` den Wert für λ , der den Kreuzvalidierungsfehler minimiert. Für die Ridge-Regression ist das Argument `alpha=0` zu setzen. Der Regularisierungspfad – hier der Verlauf der Parameterschätzungen $\hat{\beta}_j$ in Abhängigkeit von $\ln \lambda$ – ist in Abb. 6.8 dargestellt.

```
# Matrix der standardisierten Prädiktoren und des Kriteriums
> matScl <- scale(cbind(weight, height, age, sport))
> library(glmnet)                            # für cv.glmnet(), glmnet()
```

²⁴Das Paket `glmnetUtils` (Ooi, 2020) bietet aber eine entsprechende Erweiterung an.

```
# verallgemeinerte Kreuzvalidierung für lambda - Ridge-Regression
> ridgeCV <- cv.glmnet(matScl[ , c("height", "age", "sport")],
+                         matScl[ , "weight"], nfolds=10, alpha=0)
```

Anschließend passt `glmnet()` das Modell für einen konkreten Wert von λ an und gibt ein Objekt zurück, aus dem sich mit `coef()` die Parameterschätzungen extrahieren lassen.

```
> ridge <- glmnet(matScl[ , c("height", "age", "sport")],
+                     matScl[ , "weight"],
+                     lambda=ridgeCV$lambda.min, alpha=0)

> coef(ridge)                                # Parameterschätzungen
(Intercept) -3.837235e-16
height       2.640929e-01
age          -1.618367e-01
sport         -7.806060e-01

# oder direkt ohne Anpassung mit glmnet()
> coef(ridgeCV, s=ridgeCV$lambda.min)        # ...
```

Für das LASSO-Verfahren ist in `cv.glmnet()` und `glmnet()` das Argument `alpha=1` zu setzen. Der Regularisierungspfad – hier der Verlauf der Parameterschätzungen in Abhängigkeit von der L_1 -Norm $\sum_j |\hat{\beta}_j|$ des Vektors $\hat{\beta}$ – ist in Abb. 6.8 dargestellt.

```
# verallgemeinerte Kreuzvalidierung für lambda - LASSO
> lassoCV <- cv.glmnet(matScl[ , c("height", "age", "sport")],
+                         matScl[ , "weight"], nfolds=10, alpha=1)

> lasso <- glmnet(matScl[ , c("height", "age", "sport")],
+                     matScl[ , "weight"],
+                     lambda=lassoCV$lambda.min, alpha=1)

> coef(lasso)                                # Parameterschätzungen
(Intercept) -3.886238e-16
height       2.708041e-01
age          -1.667786e-01
sport         -8.466213e-01

# oder direkt ohne Anpassung mit glmnet()
> coef(lassoCV, s=lassoCV$lambda.min)        # ...
```

Für das elastic-net-Verfahren ist in `cv.glmnet()` und `glmnet()` das Argument `alpha=0.5` zu setzen. Der Regularisierungspfad – hier der Verlauf der Parameterschätzungen $\hat{\beta}_j$ in Abhängigkeit vom Anteil der aufgeklärten Devianz – ist in Abb. 6.8 dargestellt.

```
# verallgemeinerte Kreuzvalidierung für lambda - elastic net
> elNetCV <- cv.glmnet(matScl[ , c("height", "age", "sport")],
+                         matScl[ , "weight"], nfolds=10, alpha=0.5)
```

```
> elNet <- glmnet(matScl[, c("height", "age", "sport")],
+                   matScl[, "weight"],
+                   lambda=elNetCV$lambda.min, alpha=0.5)

> coef(elNet)                                # Parameterschätzungen
(Intercept) -3.899363e-16
height       2.713961e-01
age         -1.674823e-01
sport        -8.447744e-01

# oder direkt ohne Anpassung mit glmnet()
> coef(elNetCV, s=elNetCV$lambda.min)          # ...
```

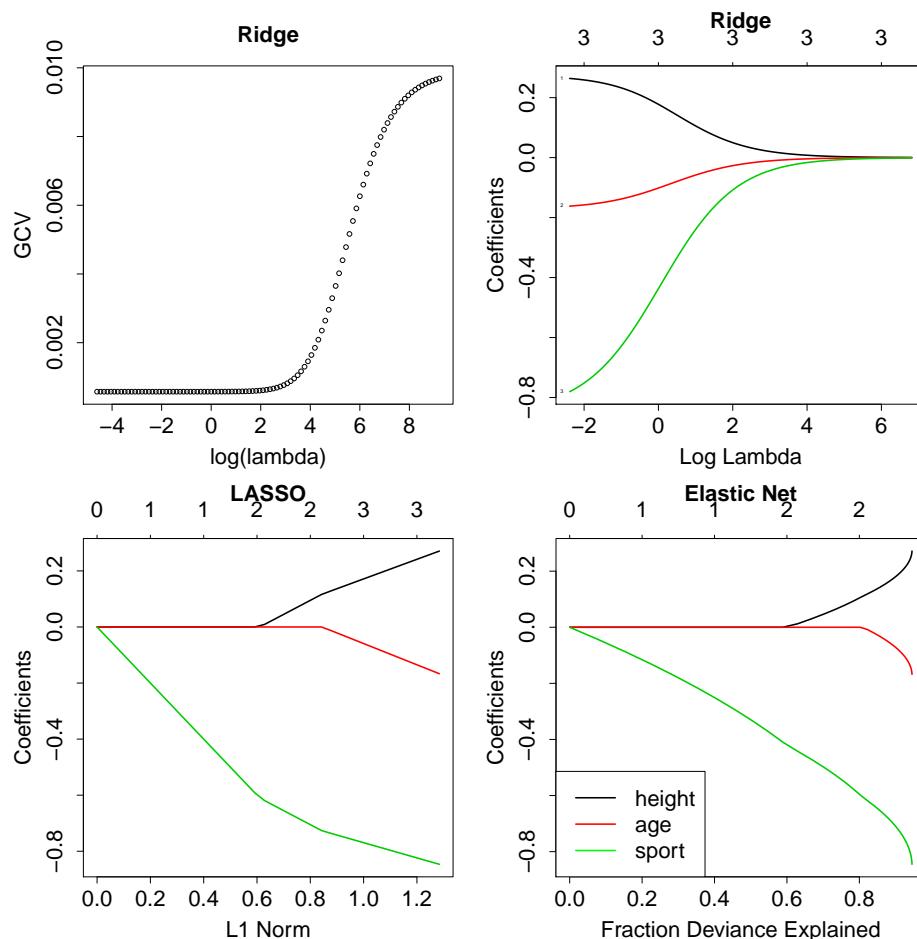


Abbildung 6.8: Penalisierte Regression: Kreuzvalidierungsfehler in Abhängigkeit von $\ln \lambda$ in der Ridge-Regression mit `lm.ridge()`. Regularisierungspfade für Ridge-Regression, LASSO und elastic net mit `glmnet()`. Die obere x-Achse zeigt die Anzahl ausgewählter Parameter $\hat{\beta}_j \neq 0$.

6.6.3 Nichtlineare Zusammenhänge

Ein Regressionsmodell, das linear in den Parametern ist, kann dennoch das Kriterium als nicht-lineare Funktion einzelner Prädiktoren beschreiben. So erleichtert es die Funktion `poly(<Prädiktor>, degree=<Grad>)`, orthogonale Polynome für eine polynomiale Regression mit quadratischen Termen oder Polynomen höheren Grades zu erstellen. Analog stellt das im Basisumfang von R enthaltene Paket `splines` mit `ns(<Prädiktor>, df=<Grad>)` natürliche splines (außerhalb des beobachteten Wertebereichs linear) und mit `bs(<Prädiktor>, df=<Grad>)` B-splines (mit Bernstein-Polynomen als Basis-Funktionen) zur Verfügung. Beide Funktionen können innerhalb der Modellformel von `lm()` verwendet werden und besitzen das Argument `df`, um ihre Variabilität zu kontrollieren.

Verallgemeinerte additive Modelle werden durch die Funktion `gam()` aus dem Paket `mgcv` (Wood, 2017) unterstützt. Sie kann automatisch die Variabilitätsparameter einer Vielzahl von Spline-Typen auswählen und lässt auch Interaktionen von Spline-Termen mit anderen Kovariaten zu.

Für die Bestimmung von Parametern in nichtlinearen Vorhersagmodellen anhand der Methode der kleinsten quadrierten Abweichungen vgl. `nls()` sowie Ritz und Streibig (2009).

6.6.4 Abhängige Fehler bei Messwiederholung oder Clustering

Liefert dieselbe Person mehrere Beobachtungen zu unterschiedlichen Zeitpunkten, sind ihre einzelnen Messwerte nicht unabhängig voneinander. Abhängige Messwerte entstehen auch, wenn verschiedene Personen aufgrund eines gemeinsamen Merkmals cluster bilden, z.B. durch eine familiäre Verwandtschaft. Eine wesentliche Voraussetzung der linearen Regressionsanalyse ist dann verletzt. Verallgemeinerte kleinste Quadrate Modelle (*generalized least squares*, GLS) (Pinheiro & Bates, 2000), gemischte lineare Modelle (*linear mixed effects models*) (West et al., 2022) und verallgemeinerte Schätzgleichungen (*generalized estimating equations*, GEE) (Yan & Fine, 2004) kommen für die Modellierung abhängige Daten in Betracht. Diese Herangehensweisen sind auch für kategoriale Zielvariablen in verallgemeinerten linearen Modellen wie der logistischen Regression oder der Poisson-Regression anwendbar (Kap. 8).

Die genannten Modelle eignen sich auch für Situationen, in denen nicht alle Personen dieselbe Anzahl von Messwerten zu denselben Zeitpunkten liefern. Die Abstände zwischen Zeitpunkten können variieren. Prädiktoren können kategorial oder kontinuierlich sein. Außerdem dürfen sie zeitabhängig, d. h. innerhalb einer Person veränderlich sein. In varianzanalytischen Designs mit Messwiederholung (Abschn. 7.5, 7.7, 7.8) bieten diese Modelle deshalb häufig eine alternative Möglichkeit zur Auswertung, die die Abhängigkeitsstruktur der Messungen besser als die klassische Varianzanalyse berücksichtigen kann.

GLS Modelle werden von den Paketen `nlme` (Pinheiro, Bates, DebRoy, Sarkar & R Core Team, 2020) und `mmrm` (Sabanes Bove et al., 2024) unterstützt. Gemischte Modelle (Pinheiro & Bates, 2000; West et al., 2022) lassen sich mit Funktionen der Pakete `lme4` (Bates, Maechler, Bolker & Walker, 2015) sowie `nlme` anpassen. Verallgemeinerte Schätzgleichungen (GEE) modellieren die Abhängigkeit des bedingten Erwartungswerts der Zielvariable von den Prädiktoren getrennt von der Abhängigkeitsstruktur der Daten derselben Person oder desselben clusters. GEE sind im Paket `geepack` (Højsgaard, Halekoh & Yan, 2006) implementiert. Hinweise auf weitere

Pakete finden sich im Abschnitt *Mixed, Multilevel, and Hierarchical Models in R* der CRAN Task Views ([Bolker, Piaskowski, Tanaka, Alday & Viechtbauer, 2024](#)).

6.6.5 Beta-Regression für natürliche Anteile

Die im Paket `betareg` ([Cribari-Neto & Zeileis, 2010; Grün, Kosmidis & Zeileis, 2012](#)) umgesetzte Beta-Regression erlaubt es stetige Variablen zu modellieren, deren Beobachtungen im offenen Intervall $(0, 1)$ liegen. Dies können etwa natürliche Anteile sein, z. B. eines Volumens. Hier hat die Beta-Regression Vorteile gegenüber der Transformation der Daten mittels des Arkussinus der Quadratwurzel, um die Verteilung einer Normalverteilung ähnlicher zu machen (Abschn. 6.5.2). Für dichotome Variablen, deren Erfolgswahrscheinlichkeit im Intervall $[0, 1]$ liegt, eignet sich dagegen die logistische Regression besser (Abschn. 8.1) – ggf. in der Formulierung für binomiale Daten als Summe dichotomer Variablen (Abschn. 8.1.2).

6.6.6 Regressionsmodelle für mehrere Verteilungsparameter

In der einfachen linearen Regression wird nur der Erwartungswert als Lageparameter der bedingten Verteilung der vorhergesagten Variable als Linearkombination der Kovariaten modelliert. Mit Funktionen aus dem Paket `gamlss` ([Rigby, Stasinopoulos, Heller & de Bastiani, 2019](#)) ist es möglich, gleichzeitig auch ein Modell für die Streuung als zusätzlichen Parameter der bedingten Verteilung anzupassen und als Funktion von Kovariaten zu schätzen. Dies ist in verallgemeinerten linearen Modellen (Kap. 8) auch für bedingte Verteilungen möglich, die keine Normalverteilung sind und auch noch über einen ebenfalls modellierbaren Formparameter verfügen können.

Die vom Paket `quantreg` ([Koenker et al., 2024](#)) unterstützte Quantilregression erweitert das Modell der linearen Regression auf die Vorhersage von bedingten Quantilen der Verteilung der vorhergesagten Variable.

6.7 Partialkorrelation und Semipartialkorrelation

Ein von `lm()` erzeugtes Objekt kann auch zur Ermittlung der Partialkorrelation $r_{(XY).Z}$ zweier Variablen X und Y ohne eine dritte Variable Z verwendet werden (Abschn. 2.7.8): $r_{(XY).Z}$ ist gleich der Korrelation der Residuen der Regressionen jeweils von X und Y auf Z .

```
# Partialkorrelation weight mit height, beide ohne sport
> weight.S <- residuals(lm(weight ~ sport)) # Resid.: weight ohne sport
> height.S <- residuals(lm(height ~ sport)) # Resid.: height ohne sport
> cor(weight.S, height.S)
[1] 0.6595016
```

Zur Kontrolle lässt sich die konventionelle Formel umsetzen oder der Umstand nutzen, dass Partialkorrelationen aus der invertierten Korrelationsmatrix \mathbf{R}^{-1} der drei beteiligten Variablen

berechnet werden können: Ist \mathbf{D} die Diagonalmatrix aus den Diagonalelementen von \mathbf{R}^{-1} , stehen die Partialkorrelationen außerhalb der Diagonale der Matrix $-(\mathbf{D}^{-\frac{1}{2}} \mathbf{R}^{-1} \mathbf{D}^{-\frac{1}{2}})$ (s. Abschn. 12.1 für Multiplikation und Invertierung von Matrizen).

```
# Kontrolle über herkömmliche Formel
> (cor(weight, height) - (cor(weight, sport) * cor(height, sport))) /
+   sqrt((1-cor(weight, sport)^2) * (1-cor(height, sport)^2))
[1] 0.6595016

# Kontrolle über Inverse der Korrelationsmatrix
> R           <- cor(cbind(weight, height, sport))      # Korrelationsmatrix
> Rinv        <- solve(R)                                # ihre Inverse R^(-1)
> DsqrtInv   <- diag(1/sqrt(diag(Rinv)))            # D^(-1/2)
> Rpart       <- -(DsqrtInv %*% Rinv %*% DsqrtInv)
> Rpart[upper.tri(Rpart)]                            # alle Partialkorrelationen
[1] 0.6595016 -0.9468826 0.5738578
```

Gleichzeitig beinhaltet $-(\mathbf{D}^{-1} \mathbf{R}^{-1})$ außerhalb der Diagonale die standardisierten b_j^z -Gewichte der Regressionen mit jeweils dem durch die Zeile definierten Kriterium und den durch die Spalten definierten Prädiktoren.

```
> Dinv <- diag(1/diag(Rinv))                      # D^(-1)
> bz   <- -(Dinv %*% Rinv)                         # -(D^(-1) * R^(-1))
> bz[upper.tri(bz)]                                # alle bz-Gewichte
[1] 0.2589668 -0.8691255  1.3414126

# standardisierte Regression weight auf height und sport
> fit <- lm(scale(weight) ~ scale(height) + scale(sport))
> zapsmall(coef(fit))                             # bz-Gewichte
(Intercept)  scale(height)  scale(sport)
0.0000000    0.2589668   -0.8691255
```

Die Semipartialkorrelation $r_{(X.Z)Y} = \frac{r_{XY} - (r_{XZ} \cdot r_{YZ})}{\sqrt{1 - r_{XZ}^2}}$ einer Variable Y mit einer Variable X ohne Z lässt sich analog zur Partialkorrelation als Korrelation von Y mit den Residuen der Regression von X auf Z berechnen.

```
# Semipartialkorrelation weight mit height ohne sport
> cor(weight, height.S)
[1] 0.2532269

# Kontrolle über herkömmliche Formel
> (cor(weight, height) - (cor(weight, sport) * cor(height, sport))) /
+   sqrt(1-cor(height, sport)^2)
[1] 0.2532269
```

Partialkorrelation und Semipartialkorrelation lassen sich auf die Situation verallgemeinern, dass nicht nur eine Variable Z auspartialisiert wird, sondern mehrere Variablen Z_j . Die Partialkorrelation $r_{(XY).Z_j}$ ist dann die Korrelation der Residuen der multiplen Regression von X

auf alle Z_j mit den Residuen der multiplen Regression von Y auf alle Z_j . Ebenso ist die Semipartialkorrelation $r_{(X.Z_j)Y}$ die Korrelation von Y mit den Residuen der multiplen Regression von X auf alle Z_j .

```
# Residuen Regression von weight bzw. height auf verbleibende Variablen
> weight.AS <- residuals(lm(weight ~ age + sport))
> height.AS <- residuals(lm(height ~ age + sport))

# Partialkorrelation (weight mit height) ohne verbleibende Variablen
> (pcorWH.AS <- cor(weight.AS, height.AS))
[1] 0.7531376

# Semipartialkorrelation weight mit (height ohne restliche Variablen)
> (spcorWH.AS <- cor(weight, height.AS))
[1] 0.2674680
```

Die Semipartialkorrelation $r_{(X.Z_j)Y}$ weist folgenden Zusammenhang zur multiplen Regression des Kriteriums Y auf die Prädiktoren X und Z_j auf: $r_{(X.Z_j)Y}^2$ ist gleich der Erhöhung des Determinationskoeffizienten R^2 beim Übergang von der Regression von Y auf Z_j hin zur Regression von Y auf die Prädiktoren X und Z_j . In diesem Sinne ist die quadrierte Semipartialkorrelation des Kriteriums mit einem Prädiktor ohne alle übrigen gleich dem Varianzanteil, den der Prädiktor zusätzlich zu allen übrigen Prädiktoren aufklärt.

```
> spcorWH.AS^2 # quadrierte Semipartialkorrelation
[1] 0.07153911

# R^2 Differenz: Regressionen mit vs. ohne height inkl. übrige Prädikt.
> fitAS <- lm(weight ~ age + sport) # ohne height
> fitHAS <- lm(weight ~ height + age + sport) # mit height
> summary(fitHAS)$r.squared - summary(fitAS)$r.squared
[1] 0.07153911
```

Ist b_x das Regressionsgewicht des Prädiktors X in der Regression des Kriteriums Y auf die Prädiktoren X und Z_j , und ist weiterhin $X.Z_j$ die Variable der Residuen der Regression von X auf alle Z_j , so gilt analog zur einfachen linearen Regression $b_x = \frac{K_{(X.Z_j)Y}}{s_{X.Z_j}^2}$, wenn K für die Kovarianz und s^2 für die Varianz von Variablen steht.

```
> fitHAS # multiple Regression, gekürzte Ausgabe ...
Coefficients:
(Intercept) height      age      sport
8.1012    0.5149   -0.2866   -0.4160

# b-Gewicht von height
> cov(weight, height.AS) / var(height.AS)
[1] 0.5148935
```

Kapitel 7

***t*-Tests und Varianzanalysen**

Häufig bestehen in empirischen Untersuchungen Hypothesen über Erwartungswerte von Variablen. Viele der für solche Hypothesen geeigneten Tests gehen davon aus, dass bestimmte Annahmen über die Verteilungen der Variablen erfüllt sind, etwa in allen Bedingungen Normalverteilungen mit derselben Varianz vorliegen. Bevor auf Tests zum Vergleich von Erwartungswerten eingegangen wird, sollen deshalb zunächst Verfahren vorgestellt werden, die sich mit der Prüfung statistischer Voraussetzungen befassen. Bei ihrer Anwendung ist zu berücksichtigen, dass i. d. R. die H_0 den gewünschten Zustand darstellt. Angemessene Herangehensweisen beschreibt Wellek (2010, Kap. 9).

Für Details der statistischen Grundlagen vgl. Eid et al. (2017), Kirk (2013) sowie Maxwell, Delaney und Kelley (2017). Meier (2022) konzentriert sich auf die Umsetzung von Varianzanalysen unterschiedlicher Designs in R.

7.1 Tests auf Normalverteilung

Der Kolmogorov-Smirnov-Test auf eine bestimmte Verteilung (Abschn. 10.1.3) setzt voraus, dass die Verteilung unter H_0 durch konkrete Werte für ihre Parameter vollständig spezifiziert ist. Dies ist in der Praxis oft unrealistisch.

Der abgewandelte Test auf Normalverteilung mit Lilliefors-Schranken ist in `lillie.test()` aus dem Paket `nortest` (Gross & Ligges, 2015) implementiert und macht diese Voraussetzung nicht. Dies gilt auch für den Anderson-Darling-Test mit der Funktion `ad.test()` aus demselben Paket. Der über `shapiro.test()` aufzurufende Shapiro-Wilk-Test auf Normalverteilung ist eine weitere Alternative.

```
> DV <- rnorm(100, mean=100, sd=15)^2      # Daten Zielgröße
> library(nortest)                         # für lillie.test(), ad.test()
> lillie.test(DV)
Lilliefors (Kolmogorov-Smirnov) normality test
data:  DV
D = 0.06792, p-value = 0.3086

> ad.test(DV)
Anderson-Darling normality test
data:  DV
A = 0.62999, p-value = 0.09796
```

```
> shapiro.test(DV)
Shapiro-Wilk normality test
data: DV
W = 0.97519, p-value = 0.0587
```

Anstelle inferenzstatistischer Tests kann die Verteilungsform auch mit einem Q-Q-Diagramm beurteilt werden (Abschn. 14.6.5, 15.2.5). Das Paket MVN ([Korkmaz, Goksuluk & Zarasiz, 2021](#)) stellt Funktionen für einen Test auf multivariate Normalverteilung bereit.

7.2 Tests auf Varianzhomogenität

Die ab Abschn. 7.3.2 und 7.4 vorgestellten Tests auf Erwartungswertunterschiede setzen voraus, dass die Variable in allen Bedingungen normalverteilt mit derselben Varianz ist.

7.2.1 *F*-Test auf Varianzhomogenität für zwei Stichproben

Um festzustellen, ob die empirischen Varianzen einer Variable in zwei unabhängigen Stichproben mit der H_0 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 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 zu `x` und `y` kann auch eine Modellformel $\langle AV \rangle \sim \langle UV \rangle$ angegeben werden. Dabei ist $\langle UV \rangle$ ein Faktor mit zwei Stufen und derselben Länge wie die Zielgröße $\langle AV \rangle$, der für jede Beobachtung die Gruppenzugehörigkeit codiert. Stammen die in der Modellformel verwendeten Variablen aus einem Datensatz, ist dieser unter `data` zu nennen. Die Argumente `alternative` und `conf.level` beziehen sich auf die Größe des *F*-Bruchs unter H_1 im Vergleich zu seiner Größe unter H_0 (dann gleich 1) bzw. auf die Breite seines Vertrauensintervalls.

```
> n1      <- 110                      # Gruppengröße 1
> n2      <- 90                       # Gruppengröße 2
> DV1     <- rnorm(n1, mean=100, sd=15) # Daten Gruppe 1
> DV2     <- rnorm(n2, mean=100, sd=13) # Daten Gruppe 2
> varDf   <- stack(list(grp1=DV1, grp2=DV2)) # Gesamtdaten
> var.test(values ~ ind, data=varDf)
F test to compare two variances
data: values by ind
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:
```

¹Als Alternative für den Fall nicht normalverteilter Variablen existieren die nonparametrischen Tests nach Mood bzw. Ansari-Bradley (Abschn. 10.4).

```
1.124894 2.495117
sample estimates:
ratio of variances
1.682140
```

Die Ausgabe des Tests umfasst den empirischen *F*-Wert (*F*), der gleich dem Quotienten der zu testenden Varianzen ist (*ratio of variances*). Zusammen mit den zugehörigen Freiheitsgraden (Gruppeneffekt: *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:

```
> var1 <- var(DV1)                                # korrig. Varianz Gruppe 1
> var2 <- var(DV2)                                # korrig. Varianz Gruppe 2
> Fval <- var1 / var2                             # Teststatistik
> (pVal <- 2*pf(Fval, n1-1, n2-1, lower.tail=FALSE)) # p-Wert
[1] 0.0115746

# kritischer Wert links/rechts für zweiseitiges 95%-Vertrauensintervall
> Fcrit <- qf(c(0.025, 0.975), n1-1, n2-1, lower.tail=FALSE)
> (ci <- var1 / (Fcrit*var2))                      # Konfidenzintervall
[1] 1.124894 2.495117
```

7.2.2 Levene-Test für mehr als zwei Stichproben

Ein Test auf Varianzhomogenität für auch mehr als zwei Gruppen ist jener nach Levene, für den das Paket *car* benötigt wird. Der Levene-Test reagiert robust auf Verletzungen der Voraussetzung von Normalverteiltheit.

```
leveneTest(y=<Daten>, group=<Faktor>, data=<Datensatz>)
```

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 Modellformel $\langle \text{AV} \rangle \sim \langle \text{UV} \rangle$ möglich. Dabei ist $\langle \text{UV} \rangle$ ein Faktor mit zwei Stufen und derselben Länge wie die Zielgröße $\langle \text{AV} \rangle$, der für jede Beobachtung die Gruppenzugehörigkeit codiert. In einer Modellformel verwendete Variablen müssen immer aus einem Datensatz stammen, der unter *data* anzugeben ist. Schließlich ist es auch möglich, nur das Objekt zu übergeben, das ein mit *lm()* angepasstes lineares Modell speichert.

```
> Nj <- c(22, 18, 20)                            # Gruppengrößen
> N   <- sum(Nj)                                 # Gesamt-N
> P   <- length(Nj)                             # Anzahl Gruppen
> DV  <- sample(0:100, N, replace=TRUE)          # Daten Zielgröße

# Personen zufällig auf Gruppen aufteilen
> IV   <- factor(sample(rep(1:P, Nj), N, replace=FALSE))
> levDf <- data.frame(IV=IV, DV=DV)             # Datensatz
```

```
> library(car)                                # für leveneTest()
> leveneTest(DV ~ IV, data=levDf)
Levene's Test for Homogeneity of Variance
  Df   F value  Pr(>F)
IV    2     2.113  0.1302
57
```

Da der Levene-Test letztlich eine Varianzanalyse ist, beinhaltet die Ausgabe einen empirischen *F*-Wert (*F value*) mit den Freiheitsgraden von Effekt- und Residual-Quadratsumme (*Df*) sowie den zugehörigen *p*-Wert (*Pr(>F)*). In der beim Levene-Test durchgeführten Varianzanalyse gehen statt der ursprünglichen Werte der Zielgröße die jeweiligen Beträge ihrer Differenz zum zugehörigen Gruppenmedian ein.² Der Test lässt sich so auch manuell durchführen (Abschn. 7.4).

```
# Berechnung mit den absoluten Abweichungen zum Median jeder Gruppe
> absDiff <- abs(DV - ave(DV, IV, FUN=median))      # Betrag Abweichungen
> anova(lm(absDiff ~ IV))                            # Varianzanalyse ...

# alternativ: absolute Abweichungen zum Gruppenmittelwert
> absErr <- abs(DV - ave(DV, IV, FUN=mean))        # Betrag Residuen
> anova(lm(absErr ~ IV))                            # Varianzanalyse ...
```

7.2.3 Fligner-Killeen-Test für mehr als zwei Stichproben

Der Fligner-Killeen-Test auf Varianzhomogenität für mehr als zwei Gruppen basiert auf den Rängen der absoluten Abweichungen der Daten zu ihrem Gruppenmedian. Er verwendet eine asymptotisch χ^2 -verteilte Teststatistik und gilt als robust gegenüber Verletzungen der Voraussetzung von Normalverteiltheit.

```
fligner.test(formula=<Modellformel>, data=<Datensatz>)
```

Unter *formula* sind Daten und Gruppierungsvariable als Modellformel $\langle AV \rangle \sim \langle UV \rangle$ zu nennen, wobei $\langle UV \rangle$ ein Gruppierungsfaktor derselben Länge wie die Zielgröße $\langle AV \rangle$ ist und für jeden ihrer Werte die zugehörige Faktorstufe angibt. Geschieht dies mit Variablen aus einem Datensatz, muss dieser unter *data* eingetragen werden.

```
> fligner.test(DV ~ IV, data=levDf)
Fligner-Killeen test of homogeneity of variances
data: DV by IV
Fligner-Killeen:med chi-squared = 4.5021, df = 2, p-value = 0.1053

# manuell: Standard-NV-Quantile der transformierten Ränge in der
# Gesamtstichprobe der absoluten Abweichungen zum Gruppenmedian
> absDiff <- abs(DV - ave(DV, IV, FUN=median))      # Betrag Abweichungen
> quants <- qnorm((0.5 + rank(absDiff) / (2*(N+1))), mean=0, sd=1)
```

²Diese Variante wird auch als Brown-Forsythe-Test bezeichnet. Mit *leveneTest(..., center=mean)* können alternativ die Differenzen zum jeweiligen Gruppenmittelwert gewählt werden.

```
> MQj      <- tapply(quants, IV, mean)      # mittlere Quantile pro Gruppe

# asymptotisch chi^2 verteilte Teststatistik
> (FK <- sum(Nj * ((MQj - mean(quants))^2)) / var(quants))
[1] 4.502134

> (pVal <- pchisq(FK, P-1, lower.tail=FALSE))      # p-Wert
[1] 0.1052868
```

7.3 *t*-Tests

Hypothesen über den Erwartungswert einer Variable in einer oder zwei Bedingungen lassen sich mit *t*-Tests prüfen. Den analogen multivariaten T^2 -Tests für mehrere Zielgrößen erläutert Abschn. 12.6.

7.3.1 *t*-Test für eine Stichprobe

Der einfache *t*-Test prüft, ob die in einer Stichprobe ermittelten Werte einer normalverteilten Variable mit der H_0 verträglich sind, dass diese Variable einen bestimmten Erwartungswert μ_0 besitzt.

```
t.test(x=<Vektor>, alternative=c("two.sided", "less", "greater"),
       mu=0, conf.level=0.95)
```

Unter **x** ist der Datenvektor einzutragen. Alternativ zu **x** kann auch eine Modellformel $\langle AV \rangle \sim 1$ angegeben werden. Stammt dabei die Zielgröße $\langle AV \rangle$ aus einem Datensatz, ist dieser unter **data** zu nennen. Mit **alternative** wird festgelegt, ob die H_1 gerichtet oder ungerichtet ist. "less" und "greater" beziehen sich dabei auf die Reihenfolge Erwartungswert unter H_1 "less" bzw. "greater" μ_0 . Das Argument **mu** bestimmt μ_0 . **conf.level** legt die Breite des je nach H_1 ein- oder zweiseitigen Konfidenzintervalls für den Erwartungswert μ fest.

```
> N      <- 100                      # Stichprobengröße
> DV    <- rnorm(N, 5, 20)          # Zielgröße
> muH0 <- 0                         # Erwartungswert unter H0
> t.test(DV ~ 1, alternative="two.sided", mu=muH0)
One Sample t-test
data: DV
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 μ in der gewünschten Breite sowie der Mittelwert genannt. Das Ergebnis lässt sich manuell verifizieren:

```
> M      <- mean(DV)                      # Mittelwert
> s      <- sd(DV)                       # korrigierte Streuung
> (tVal <- (M-muH0) / (s/sqrt(N)))      # Teststatistik t
[1] 2.412462

> (pVal <- 2*pt(tVal, N-1, lower.tail=FALSE)) # p-Wert
[1] 0.01768618

# # kritische t-Werte zweiseitiges 95%-Vertrauensintervall für mu
> tCrit <- qt(c(0.025, 0.975), N-1, lower.tail=FALSE)
> (ci   <- M - tCrit * s/sqrt(N))        # Konfidenzintervall
[1] 0.8989476  9.2292614
```

Als Effektstärkemaß kann Cohens δ herangezogen werden, dessen Schätzung *d* auf der Differenz des Mittelwerts zum Erwartungswert unter H_0 beruht, die an der korrigierten Streuung relativiert wird. Die Berechnung inkl. des Konfidenzintervalls ist mit der Funktion *cohens_d()* aus dem Paket *effectsize* (Ben-Shachar et al., 2024) möglich, die als Argument den Datenvektor benötigt.

```
> library(effectsize) # für cohens_d()
> cohens_d(DV)       # Schätzung Cohens d
Cohen's d | 95% CI
-----
0.24 | [0.04, 0.44]

> (d <- (mean(DV) - muH0) / sd(DV))
[1] 0.2412462
```

7.3.2 *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 H_0 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 auch eine Modellformel $\langle AV \rangle \sim \langle UV \rangle$ angegeben werden. Dabei ist $\langle UV \rangle$ ein Faktor mit zwei Ausprägungen und derselben Länge wie die Zielgröße $\langle AV \rangle$, der für jede Beobachtung die Gruppenzugehörigkeit codiert. Stammen die in der Modellformel verwendeten Variablen aus einem Datensatz, ist dieser unter *data* zu nennen. Mit *alternative* wird festgelegt, ob die H_1 gerichtet oder ungerichtet ist. "less" und "greater" beziehen sich dabei auf

die Reihenfolge x "less" bzw. "greater" y. Bei Verwendung einer Modellformel $\langle AV \rangle \sim \langle UV \rangle$ ist die Reihenfolge der Gruppen über die Reihenfolge der Faktorstufen in $\langle UV \rangle$ bestimmt.

Das Argument `paired=FALSE` bestimmt, dass es sich um unabhängige 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 H_1 ein- oder zweiseitig.

Test mit Annahme von Varianzhomogenität und Schätzung der Effektstärke

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
> DVm <- rnorm(n1, mean=180, sd=10)    # Daten Männer
> DVf <- rnorm(n2, mean=175, sd=6)      # Daten Frauen
> tDf <- stack(list(m=DVm, f=DVf))     # Gesamtdaten, erste Stufe m
> t.test(values ~ ind, alternative="greater", var.equal=TRUE, data=tDf)

Two Sample t-test
data: DV ~ IV
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 in group m mean in group f
180.1867        176.4451
```

Das Ergebnis lässt sich manuell prüfen.

```
# gepoolte Streuung & Schätzung der Streuung der Mittelwertsdifferenz
> sdPool <- sqrt(((n1-1)*var(DVm) + (n2-1)*var(DVf)) / (n1+n2-2))
> estSigDiff <- sqrt((n1+n2) / (n1*n2)) * sdPool
> (tVal <- (mean(DVm)-mean(DVf)) / estSigDiff)           # t-Wert
[1] 1.634606

> (pVal <- pt(tVal, n1+n2-2, lower.tail=FALSE))    # einseitiger p-Wert
[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(DVm)-mean(DVf) - tCrit*estSigDiff)
[1] -0.1201456
```

Als Effektstärkemaß kann Cohens δ herangezogen werden. Seine Schätzung d beruht auf der Mittelwertsdifferenz, die an der gepoolten Streuung relativiert wird, wie sie auch in der *t*-Statistik Verwendung findet. Die Berechnung erfolgt mit `cohens_d()` aus dem Paket `effectsize`.

```
> library(effectsize)                      # für CohenD()
> cohens_d(DVm, DVf)
Cohen's d | 95% CI
-----
0.53 | [-0.12, 1.16]

> (d <- (mean(DVm) - mean(DVf)) / sdPool)      # Schätzung Cohens d
[1] 0.5250485
```

Test ohne Annahme von Varianzhomogenität

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(values ~ ind, alternative="greater", var.equal=FALSE, data=tDf)
Welch Two Sample t-test
data: DV by IV
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 in group m mean in group f
180.1867        176.4451
```

Das Ergebnis lässt sich manuell prüfen.

```
> varM      <- var(DVm)                      # korrigierte Varianz Männer
> varF      <- var(DVf)                      # korrigierte Varianz Frauen
> num       <- (varM/n1 + varF/n2)^2          # Zähler
> denom     <- varM^2/((n1-1)*n1^2) + varF^2/((n2-1)*n2^2)    # Nenner
> (dfWelch <- num/denom)                      # Freiheitsgrade im Welch-Test
[1] 25.72683

> (tValW <- (mean(DVm)-mean(DVf)) / sqrt(varM/n1 + varF/n2)) # t-Wert
[1] 1.568068

> (pValW <- pt(tValW, dfWelch, lower.tail=FALSE)) # einseitiger p-Wert
[1] 0.06454212
```

7.3.3 *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 für `t.test()` das Argument `paired=TRUE` 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. Es kann auch eine Modellformel `Pair(x, ~y) ~ 1` verwendet werden, wobei dann das Argument `paired=TRUE` entfällt. Sind `x` und `y` aus der Modellformel Teil eines Datensatzes, ist dieser unter `data` zu nennen.

Bei im Long-Format vorliegenden Daten kann analog zum Test für zwei unabhängige Stichproben eine Modellformel $\langle AV \rangle \sim \langle UV \rangle$ angegeben werden – ggf. zusammen mit einem Datensatz `data`, aus dem die Variablen stammen. Die Daten in $\langle AV \rangle$ müssen so geordnet sein, dass innerhalb jeder von $\langle UV \rangle$ definierten Gruppe dieselbe Reihenfolge von Beobachtungsobjekten vorliegt, um die paarweise Zuordnung sicherzustellen. Zudem dürfen keine fehlenden Werte vorhanden sein.

Der Test prüft die H_0 , dass die sich aus paarweiser Subtraktion ergebende Differenzvariable von `x` und `y` den Erwartungswert 0 besitzt. Entsprechend bezieht sich das Konfidenzintervall auf den Erwartungswert dieser Differenzvariable.

```
> N      <- 20                                # Stichprobengröße
> DVpre <- rnorm(N, mean=90,   sd=15)        # Daten prä
> DVpost <- rnorm(N, mean=100,  sd=15)        # Daten post
> DV     <- c(DVpre, DVpost)                  # Gesamtdaten Long-Format

# Faktor, der im Long-Format jedem Wert von DV Messzeitpunkt zuordnet
# dabei Kontrolle der Reihenfolge der Faktorstufen: pre vor post
> IV <- factor(rep(0:1, each=N), labels=c("pre", "post"))
> t.test(DV ~ IV, alternative="less", paired=TRUE)

Paired t-test
data: DV by IV
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
> DVdiff <- DVpre-DVpost                      # Differenzvariable
> t.test(DVdiff, mu=0, alternative="less")    # ...
```

Als Effektstärkemaß kann Cohens δ herangezogen werden, dessen Schätzung d mit der Differenzvariable wie im Fall für eine Stichprobe vollzogen wird, wobei $\mu_0 = 0$ gilt. Die Berechnung erfolgt mit `cohens_d()` aus dem Paket `effectsize`.

```
> library(effectsize)                                # für cohens_d()
> cohens_d(DVdiff)
Cohen's d |      95% CI
-----
-0.47    | [-0.92, 0.00]

> (d <- mean(DVdiff) / sd(DVdiff))               # Schätzung Cohens d
[1] -0.4655015
```

7.4 Einfaktorielle Varianzanalyse (CR-*p*)

Bestehen Hypothesen über die Erwartungswerte einer normalverteilten Zielgröße (AV) in $p > 2$ unabhängigen Bedingungen eines Faktors (UV), kann eine einfaktorielle Varianzanalyse (ANOVA, *analysis of variance*) zur Prüfung herangezogen werden. Das Versuchsdesign wird hier mit CR-*p* (*completely randomized design*) abgekürzt.³ Für die analoge multivariate Varianzanalyse s. Abschn. 12.7.1.

Durch die enge Verwandtschaft von linearer Regression und Varianzanalyse ähneln sich die Befehle für beide Analysen in R stark (s. Abschn. 12.9 für eine formalere Darstellung). Zur Unterscheidung ist die Art der Variablen auf der rechten Seite der Modellformel bedeutsam (Abschn. 5.2): 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 Gruppierungsvariablen tatsächlich die Klasse **factor** besitzen.

7.4.1 Auswertung mit `oneway.test()`

In der einfaktoriellen Varianzanalyse wird geprüft, ob die in verschiedenen Bedingungen erhöhenen Werte einer Variable mit der H_0 verträglich sind, dass diese Variable in allen Gruppen denselben Erwartungswert besitzt. Die H_1 ist unspezifisch und lautet, dass sich mindestens zwei Erwartungswerte unterscheiden.

```
oneway.test(formula=<Modellformel>, data=<Datensatz>,
            subset=<Indexvektor>, var.equal=FALSE)
```

Unter **formula** sind Daten und Gruppierungsvariable als Modellformel $\langle \text{AV} \rangle \sim \langle \text{UV} \rangle$ einzugeben, wobei $\langle \text{UV} \rangle$ ein Faktor derselben Länge wie die Zielgröße $\langle \text{AV} \rangle$ ist und für jeden ihrer Werte die Gruppenzugehörigkeit angibt. Geschieht dies mit Variablen aus einem Datensatz, muss dieser unter **data** eingetragen werden. Das Argument **subset** erlaubt es, nur eine Teilmenge der Fälle einfließen zu lassen. Es erwartet einen numerischen oder logischen Indexvektor, der sich auf die Zeilen des Datensatzes bezieht. Mit **var.equal** wird vorgegeben, ob von Varianzhomogenität ausgegangen werden kann (Voreinstellung ist FALSE).

³Hier und im Folgenden wird für varianzanalytische Versuchspläne die Notation von Kirk (2013) übernommen.

Kapitel 7 t-Tests und Varianzanalysen

```
> P <- 4                                # Anzahl Gruppen
> Nj <- c(41, 37, 42, 40)                # Gruppengrößen
> IVeff <- c(0, 0.3, 0.6, 1)             # Gruppen-Effekte
> IV <- factor(rep(LETTERS[1:P], times=Nj)) # Gruppierungsfaktor
> DV <- IVeff[unclass(IV)] + rnorm(sum(Nj), mean=0, sd=1) # Messwerte
> dfCRp <- data.frame(IV, DV)            # Datensatz
> oneway.test(DV ~ IV, data=dfCRp, var.equal=TRUE)
One-way analysis of means
data: DV and IV
F = 4.8619, num df = 3, denom df = 156, p-value = 0.002922
```

Die Ausgabe von `oneway.test()` beschränkt sich auf die wesentlichen Ergebnisse des Hypothesentests: Dies sind der empirische F -Wert (F) mit den Freiheitsgraden der Effekt-Quadratsumme (`num df`) im Zähler des F -Bruchs und jener der Quadratsumme der Residuen (`denom df`) im Nenner des F -Bruchs sowie der zugehörige p -Wert (`p-value`).

Für ausführlichere Informationen zum Modell, etwa zu den Quadratsummen von Effekt und Residuen, sollte auf `aov()` oder `anova()` zurückgegriffen werden (Abschn. 7.4.2, 7.4.3). Die Ergebnisse können manuell überprüft werden.

```
> N <- sum(Nj)                            # Gesamt-N
> Vj <- tapply(DV, IV, var)               # korrig. Gruppenvarianzen
> Mj <- tapply(DV, IV, mean)              # Gruppenmittel
> M <- sum((Nj/N) * Mj)                  # gewichtetes Gesamtmittel

# Quadratsumme Residuen, alternativ sum((DV - ave(DV, IV, FUN=mean))^2)
> SSw <- sum((Nj-1) * Vj)                # Quadratsumme within
> SSb <- sum(Nj * (Mj-M)^2)              # Quadratsumme between
> MSw <- SSw / (N-P)                    # mittlere QS within
> MSb <- SSb / (P-1)                     # mittlere QS between
> (Fval <- MSb / MSw)                   # Teststatistik F-Wert
[1] 4.861867

> (pVal <- pf(Fval, P-1, N-P, lower.tail=FALSE))    # p-Wert
[1] 0.002921932
```

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 (Abschn. 7.3.2), der i. A. etwas konservativer ist – im Beispiel allerdings einen kleineren p -Wert liefert.

```
> oneway.test(DV ~ IV, data=dfCRp, var.equal=FALSE)
One-way analysis of means (not assuming equal variances)
data: DV and IV
F = 5.3235, num df = 3.000, denom df = 85.576, p-value = 0.002065
```

7.4.2 Auswertung mit `aov()`

Wenn Varianzhomogenität anzunehmen ist, kann eine Varianzanalyse auch mit `aov()` berechnet werden.

```
aov(formula=<Modellformel>, data=<Datensatz>, subset=<Indexvektor>)
```

Unter `formula` werden Daten und Gruppierungsvariable als Modellformel $\langle AV \rangle \sim \langle UV \rangle$ einge tragen, wobei $\langle UV \rangle$ ein Faktor derselben Länge wie die Zielgröße $\langle AV \rangle$ ist und für jeden ihrer Werte die Gruppenzugehörigkeit angibt. Unter `data` ist ggf. der Datensatz als Quelle der Variablen anzugeben. Das Argument `subset` erlaubt es, nur eine Teilmenge der Fälle einzufließen zu lassen. Es erwartet einen numerischen oder logischen Indexvektor, der sich auf die Zeilen des Datensatzes bezieht.

Das von `aov()` zurückgegebene Objekt ist wie das Ergebnis von `lm()` eine Liste, die u. a. die berechneten Quadratsummen und Freiheitsgrade enthält.⁴ Um von Gruppeneffekt und Residuen auch ihre inferenzstatistischen Größen zu erhalten, muss `summary()` auf das Ergebnis angewendet werden.

```
> aovCRp <- aov(DV ~ IV, data=dfCRp)
> summary(aovCRp)
      Df   Sum Sq  Mean Sq  F value    Pr(>F)
IV       3   13.782   4.5941   4.8619  0.002922 ***
Residuals 156  147.407   0.9449
```

Die Kennwerte des Gruppeneffekts (Zeile `IV`) und der Residuen (Zeile `Residuals`) finden sich in den Spalten `Df` (Freiheitsgrade), `Sum Sq` (Quadratsumme), `Mean Sq` (mittlere Quadratsumme), `F value` (Teststatistik: *F*-Wert) und `Pr(>F)` (*p*-Wert).

Eine tabellarische Übersicht über den Gesamtmittelwert, die Mittelwerte in den einzelnen Bedingungen und deren Zellbesetzung erzeugt `model.tables()`.

```
model.tables(<aov-Objekt>, type="<Schätzer>", se=FALSE)
```

Im ersten Argument muss das Ergebnis einer durch `aov()` vorgenommenen Modellanpassung stehen. Über das Argument `type` wird bestimmt, ob die Mittelwerte ("means") oder geschätzten Effektgrößen (Voreinstellung "effects") ausgegeben werden sollen. Letztere erhält man auch mit `coef(<aov-Objekt>)`, für ihre Bedeutung s. Abschn. 12.9.2. Sollen Standardfehler genannt werden, ist `se=TRUE` zu setzen.

```
> model.tables(aovCRp, type="means")
```

Tables of means

Grand mean

0.4362142

| | A | B | C | D |
|-----|---------|----------|---------|---------|
| IV | 0.2080 | 0.07393 | 0.6395 | 0.7917 |
| rep | 41.0000 | 37.00000 | 42.0000 | 40.0000 |

⁴Da `aov()` letztlich `lm()` aufruft, lassen sich auf diese Liste dieselben Funktionen zur Extraktion weiterer Informationen anwenden (Abschn. 6.2).

Grafisch aufbereitet werden deskriptive Kennwerte der Messwerte in den einzelnen Gruppen mit `plot.design()`. 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 Gruppierungsfaktor variiert wird (Abschn. 7.6.1, Abb. 7.3).

```
plot.design(<Modellformel>, fun=mean, data=<Datensatz>)

> plot.design(DV ~ IV, fun=mean, data=dfCRp,
+               main="Mittelwerte getrennt nach Gruppen")
```

7.4.3 Auswertung mit `anova()`

Äquivalent zum Aufruf von `summary(<aov-Modell>)` ist die Verwendung der `anova()` Funktion, wenn ein mit `lm()` formuliertes lineares Modell übergeben wird. Das Ergebnis von `anova(<lm-Modell>)` unterscheidet sich auf der Konsole nur wenig von der Ausgabe von `summary(<aov-Modell>)`. `anova()` gibt jedoch einen Datensatz zurück, aus dem sich besonders einfach Freiheitsgrade, Quadratsummen und *p*-Werte für weitere Rechnungen extrahieren lassen. Dagegen hat die von `summary(<aov-Modell>)` zurückgegebene Liste eine komplexere Struktur.

```
> (anovaCRp <- anova(lm(DV ~ IV, data=dfCRp)))
Analysis of Variance Table
Response: DV
      Df  Sum Sq  Mean Sq  F value    Pr(>F)
IV       3   13.782   4.5941   4.8619  0.002922 ***
Residuals 156  147.407   0.9449

> anovaCRp["Residuals", "Sum Sq"]          # Residual-Quadratsumme
[1] 147.4069
```

Analog zur Verwendung von `anova()` zum Vergleich zweier unterschiedlich umfassender Regressionsmodelle (Kap. 6.3.3) lässt sich auch die Varianzanalyse als Vergleich von *nested* Modellen durchführen: Das eingeschränkte Modell `<fitR>` berücksichtigt hier als Effektterm nur den für alle Beobachtungen konstanten Erwartungswert (`(AV) ~ 1`), das umfassendere Modell `<fitU>` zusätzlich den Gruppierungsfaktor (`(AV) ~ (UV)`). Der Modellvergleich erfolgt dann mit `anova(<fitR>, <fitU>)` (Abschn. 12.9.6).

```
> fitR <- lm(DV ~ 1, data=dfCRp)          # eingeschränktes Modell
> fitU <- lm(DV ~ IV, data=dfCRp)         # umfassendes Modell
> anova(fitR, fitU)                      # Modellvergleich
Analysis of Variance Table
Model 1: DV ~ 1
Model 2: DV ~ IV
      Res.Df    RSS  Df  Sum of Sq      F    Pr(>F)
1     159 161.19
2     156 147.41  3      13.782  4.8619  0.002922 **
```

7.4.4 Effektstärke schätzen

Als Maß für die Effektstärke wird oft η^2 herangezogen, in dessen Schätzung $\hat{\eta}^2 = \frac{SS_b}{SS_b+SS_w}$ die Quadratsumme von Effekt (SS_b) und Residuen (SS_w) einfließen. Die Berechnung übernimmt die Funktion `eta_squared(aov-Objekt)` aus dem Paket `effectsize`. Sie eignet sich auch für das – hier identische – partielle $\hat{\eta}_p^2$.

```
> library(effectsize)                                     # für eta_squared()
> eta_squared(aovCRp, partial=FALSE)
Parameter | Eta2 |      95% CI
-----
IV       | 0.09 | [0.02, 1.00]
```

Ein ähnliches Maß ist ω^2 , dessen Schätzung $\hat{\omega}^2 = \frac{df_b \cdot (MS_b - MS_w)}{SS_b + SS_w + MS_w}$ auch die mittleren Quadratsummen von Effekt (MS_b) und Residuen (MS_w) sowie die Freiheitsgrade df_b von SS_b einbezieht. Das Paket `effectsize` besitzt hierfür die Funktion `omega_squared(aov-Objekt)`

```
> omega_squared(aovCRp)
Parameter | Omega2 |      95% CI
-----
IV       | 0.07 | [0.01, 1.00]
```

Alternativ eignet sich $\hat{f} = \sqrt{\frac{\hat{\eta}^2}{1-\hat{\eta}^2}}$.

```
> dfSSb <- anovaCRp["IV",        "Df"]           # df QS between
> SSb   <- anovaCRp["IV",        "Sum Sq"]        # Quadratsumme between
> MSb   <- anovaCRp["IV",        "Mean Sq"]       # mittlere QS between
> SSw   <- anovaCRp["Residuals", "Sum Sq"]        # Quadratsumme within
> MSw   <- anovaCRp["Residuals", "Mean Sq"]       # mittlere QS within

# Schätzungen verschiedener Effektstärken
> (etaSq <- SSb / (SSb + SSw))                  # Schätzung von eta^2
[1] 0.08550311

> (omegaSq <- dfSSb * (MSb-MSw) / (SSb+SSw+MSw)) # Schätzung omega^2
[1] 0.06752082

> (f <- sqrt(etaSq / (1-etaSq)))                 # Schätzung von f
[1] 0.3057735
```

7.4.5 Voraussetzungen grafisch prüfen

Eine Varianzanalyse setzt u. a. voraus, dass die Fehler unabhängige, normalverteilte Variablen mit Erwartungswert 0 und fester Streuung sind.⁵ Ob empirische Daten einer Stichprobe mit

⁵Für alternative Verfahren, die robuster gegenüber der Verletzung bestimmter Voraussetzungen sind, oder weniger Voraussetzungen machen, s. Abschn. 6.6.1, 6.6.4, 10.5.6 und 11.1.5.

diesen Annahmen konsistent sind, kann heuristisch durch eine Reihe von Diagrammen abgeschätzt werden, wie sie auch in der Regressionsdiagnostik Verwendung finden (Abschn. 6.5). Dazu können die (ggf. standardisierten) Residuen gegen die Gruppenmittelwerte abgetragen werden – bei Modellgültigkeit sollten sie unabhängig vom Gruppenmittelwert zufällig um 0 streuen. Ebenso kann die empirische Verteilung der Residuen in jeder Gruppe mit Hilfe von Boxplots veranschaulicht werden (Abschn. 14.6.3) – diese Verteilungen sollten einander ähnlich sein. Schließlich lassen sich die Residuen mittels eines Quantil-Quantil-Plots danach beurteilen, ob sie mit der Annahme von Normalverteiltheit der Fehler verträglich sind (Abb. 7.1, Abschn. 14.6.5).

```
# standardisierte Residuen gegen Vorhersage darstellen
> Estnd <- rstandard(aovCRp) # standardisierte Residuen
> plot(Estnd ~ fitted(aovCRp), pch=20,
+       xlab="Vorhersage", ylab="standardisierte Residuen",
+       main="standardisierte Residuen vs. Vorhersage")

> abline(h=0, col="gray60", lwd=2) # Referenz für Modellgültigkeit

# Verteilung der standardisierten Residuen in den Gruppen
> plot(Estnd ~ aovCRp$model$IV, main="Residuen vs. Stufen")

# Normalverteiltheit der Fehler prüfen
> qqnorm(Estnd, pch=20) # Q-Q-Plot der Residuen
> qqline(Estnd, col="gray60") # Referenzgerade
```

7.4.6 Einzelvergleiche (Kontraste)

Ein *Kontrast* $\psi = \sum_j c_j \cdot \mu_j$ meint im Folgenden eine Linearkombination der p Gruppenerwartungswerte μ_j mit den Koeffizienten c_j , wobei $1 \leq j \leq p$ sowie $\sum_j c_j = 0$ gilt. Solche Kontraste dienen einem spezifischen Vergleich zwischen den p experimentellen Bedingungen. Über die Größe von ψ lassen sich Hypothesen aufstellen (unter H_0 gilt meist $\psi_0 = 0$), deren Test im Folgenden beschrieben wird (für eine formalere Darstellung s. Abschn. 12.9.5).

Beliebige a-priori Kontraste

Der Test beliebiger Kontraste lässt sich mit `glht()` (*general linear hypothesis test*) aus dem Paket `multcomp` (Hothorn, Bretz & Westfall, 2008) durchführen. Hierfür ist zunächst der Kontrastvektor \mathbf{c} aus den Koeffizienten c_j der Linearkombination zu bilden.

```
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, wozu die `mcp()` Funktion dient. Diese erwartet ihrerseits eine Zuweisung

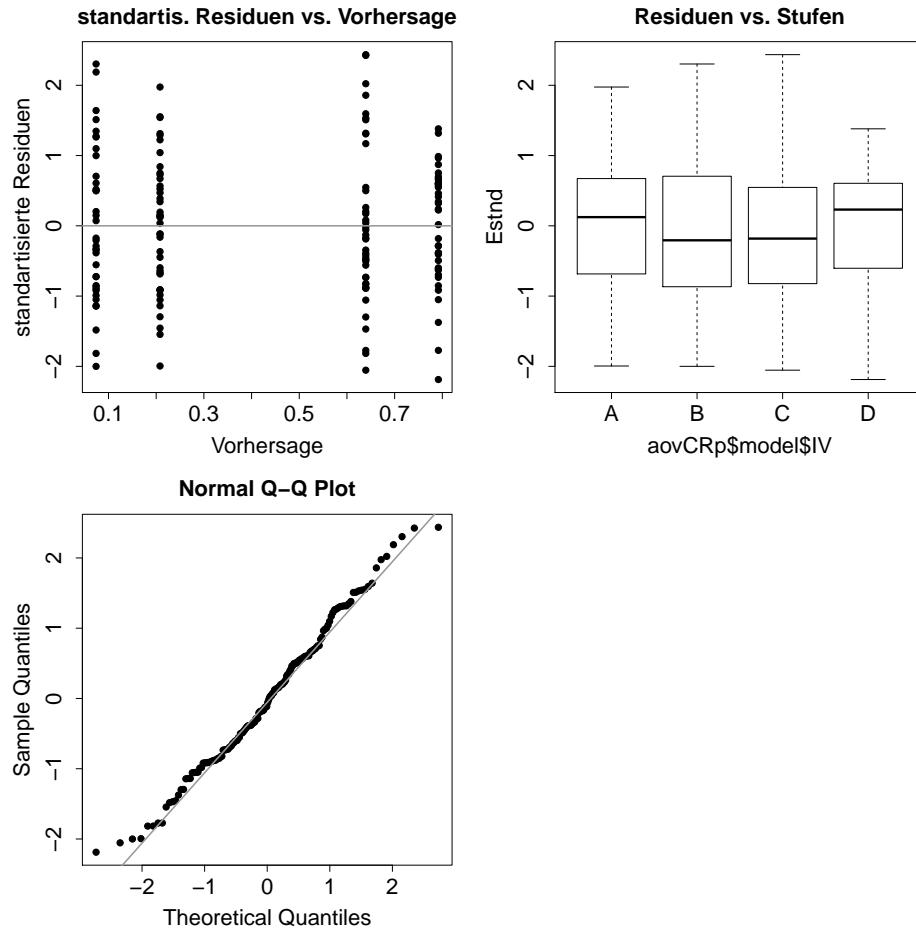


Abbildung 7.1: Grafische Prüfung der Voraussetzungen für die einfaktorielle Varianzanalyse

der Form $\langle UV \rangle = \langle \text{Koeffizienten} \rangle$ als Argument. Dabei ist $\langle UV \rangle$ der Name des Gruppierungsfaktors aus dem mit `aov()` erstellten Modell (ohne Anführungszeichen) und $\langle \text{Koeffizienten} \rangle$ eine zeilenweise aus (ggf. benannten) Kontrastvektoren zusammengestellte Matrix. Dabei muss jeder Koeffizient c_j angegeben werden, also auch solche, die 0 sind und somit nicht in die Linear kombination eingehen. Mit `alternative` wird festgelegt, ob die H_1 gerichtet oder ungerichtet ("two.sided") ist. "less" und "greater" beziehen sich dabei auf die Reihenfolge $\psi < \psi_0$ bzw. $\psi > \psi_0$, wobei die Voreinstellung $\psi_0 = 0$ ist.

Im Beispiel soll zunächst nur das Mittels der ersten beiden mit dem Mittel der verbleibenden Gruppen verglichen werden. Dabei sei $\psi_0 = 0$ und unter $H_1 \psi < 0$.

```
# Matrix der Kontrastkoeffizienten - hier nur eine Zeile
> cntrMat <- rbind("(A+B)-(C+D)"=c(1/2, 1/2, -1/2, -1/2))
> library(multcomp)                                # für glht()
> glhtRes <- glht(aovCRp, linfct=mcp(IV=cntrMat), alternative="less")
```

`summary(<glht-Modell>, test=adjusted("<alpha-Adjustierung>"))` testet das Ergebnis auf Signifikanz. Das Argument `test` erlaubt für den Test mehrerer Kontraste gleichzeitig, eine Methode zur α -Adjustierung auszuwählen (vgl. `?summary.glht`). Soll dies unterbleiben, ist

`test=adjusted("none")` zu setzen. Beim Aufstellen von Kontrasten ist zu beachten, dass die Reihenfolge der Gruppen durch die Reihenfolge der Faktorstufen bestimmt wird (Abschn. 2.6.4).

```
# Matrix der Kontrastkoeffizienten - hier nur eine Zeile
> cntrMat <- rbind("(A+B)-(C+D)"=c(1/2, 1/2, -1/2, -1/2))
> library(multcomp)                                # für glht()
> glhtRes <- glht(aovCRp, linfct=mcp(IV=cntrMat), alternative="less")
> summary(glhtRes)

Simultaneous Tests for General Linear Hypotheses
Multiple Comparisons of Means: User-defined Contrasts
Fit: aov(formula = DV ~ IV, data = dfCRp)
Linear Hypotheses:
Estimate Std. Error t value Pr(<t)
(A+B)-(C+D) >= 0 -0.5746     0.1539 -3.735 0.000132 ***

```

Die Ausgabe führt den Namen des getesteten Kontrastes aus der Matrix der Kontrastkoeffizienten auf, gefolgt von der Schätzung $\hat{\psi}$ (Estimate), für die die Erwartungswerte in der Linearkombination durch die Gruppenmittelwerte ersetzt werden ($\hat{\psi} = \sum_j c_j \cdot M_j$). Es folgt der Standardfehler dieser Schätzung (Std. Error), der Wert der *t*-Teststatistik (t value) und der zugehörige *p*-Wert (Pr(<t)).

Die Differenz $\hat{\psi} - \psi_0$ bildet den Zähler der Teststatistik *t*. Im Nenner wird die quadrierte Länge $\|\mathbf{c}\|^2$ des Kontrastvektors benötigt, für dessen Berechnung eine Gewichtung mit den Zellbesetzungen vorzunehmen ist. Das Produkt $\|\mathbf{c}\|^2 \cdot MS_w$ der quadrierten Länge mit der mittleren Quadratsumme der Residuen aus der zugehörigen Varianzanalyse bildet den quadrierten Nenner der Teststatistik und stellt die Schätzung der Varianz des Kontrastes dar. Die Teststatistik ist im Fall von a-priori Kontrasten unter H_0 zentral *t*-verteilt mit den Freiheitsgraden der Quadratsumme der Residuen in der zugehörigen Varianzanalyse.

```
> P      <- nlevels(dfCRp$IV)                      # Anzahl der Gruppen
> Mj    <- tapply(dfCRp$DV, dfCRp$IV, mean)       # Gruppenmittelwerte
> Nj    <- table(dfCRp$IV)                         # Gruppengrößen
> dfSSw <- sum(Nj) - P                            # df von SS within
> SSw   <- sum((DV - ave(DV, IV, FUN=mean))^2)   # SS within
> MSw   <- SSw / dfSSw                            # MS within
> (psiHat <- sum(cntrMat[1, ] * Mj))            # Kontrastschätzung
[1] -0.5746347

> lenSq <- sum(cntrMat[1, ]^2 / Nj)               # quadrierte Länge
> (tStat <- psiHat / sqrt(lenSq*MSw))           # Teststatistik t
[1] -3.734508

> (pVal <- pt(abs(tStat), dfSSw, lower.tail=FALSE)) # p-Wert einseitig
[1] 0.0001316091
```

Durch `confint(glht-Modell)` erhält man die Vertrauensintervalle der definierten ψ , die sich über `plot(glht-Modell)` grafisch darstellen lassen.

```
> confint(glhtRes)                      # einseitiges Konfidenzintervall
Simultaneous Confidence Intervals      # Ausgabe gekürzt ...
95% family-wise confidence level
Linear Hypotheses:
Estimate lwr upr
(A+B)-(C+D) >= 0 -0.5746 -Inf -0.3200

# obere Grenze des einseitigen 95%-Vertrauensintervalls für psi
> (tCrit <- qt(0.05, dfSSw, lower.tail=FALSE))    # krit. t-Wert eins.
[1] 1.65468

> (ciUp <- psiHat + tCrit*sqrt(lenSq*MSw))
[1] -0.3200264
```

Sollen mehrere Kontraste gleichzeitig getestet werden, verfügt die an das Argument `linfct` übergebene Funktion `mcp()` zum einen über Voreinstellungen für verschiedene Spezialfälle: Mit `mcp(UV)="Dunnett"` erhält man etwa alle $p-1$ paarweisen Vergleiche, in denen ein Gruppenerwartungswert gegen jenen der Referenzstufe (die erste Faktorstufe von `UV`), s. Abschn. 2.6.5) getestet wird. Analog erzeugt `mcp(UV)="Tukey"` alle Tests der paarweisen Gruppenvergleiche (s. u.). Zum anderen sind allgemeine Kontrast-Tests durch Zusammenstellung der zugehörigen Kontrastvektoren als (ggf. benannte) Zeilen einer Matrix möglich. Hier werden drei Kontraste ohne Adjustierung des α -Niveaus gerichtet getestet.

```
> cntrMat <- rbind("A-D"           =c( 1,   0,   0, -1),
+                     "1/3*(A+B+C)-D"=c(1/3, 1/3, 1/3, -1),
+                     "B-C"           =c( 0,   1,  -1,  0))

> glhtRes <- glht(aovCRp, linfct=mcp(IV=cntrMat), alternative="less")
> plot(glhtRes)                  # Diagramm der Konfidenzintervalle
> (sumRes <- summary(glhtRes, test=adjusted("none")))
Simultaneous Tests for General Linear Hypotheses
Multiple Comparisons of Means: User-defined Contrasts
Fit: aov(formula = DV ~ IV, data = dfCRp)
Linear Hypotheses:
Estimate Std. Error t value Pr(<t)
A-D >= 0       -0.5837     0.2160  -2.702 0.00383 **
1/3*(A+B+C)-D >= 0    -0.4845     0.1775  -2.729 0.00354 **
B-C >= 0       -0.5656     0.2192  -2.581 0.00539 **
```

Das Ergebnis lässt sich manuell prüfen (s. Abschn. 12.1.1 für die Matrixmultiplikation).

```
> psiHats <- cntrMat %*% Mj          # Kontrastschätzungen
> lenSqs  <- cntrMat^2 %*% (1/Nj)    # quadrierte Längen
> tStats  <- psiHats / sqrt(lenSqs*MSw) # Teststatistiken
> pVals   <- pt(abs(tStats), dfSSw, lower.tail=FALSE) # p-Werte eins.
> data.frame(psiHats, tStats, pVals)
            psiHats      tStats      pVals
A-D        -0.5836704 -2.701783 0.003830077
```

```
1/3*(A+B+C)-D -0.4845486 -2.729212 0.003539043
B-C             -0.5655991 -2.580620 0.005391977
```

Beliebige post-hoc Kontraste nach Scheffé

Beliebige Kontraste 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 bei Einzeltests eine geeignete α -Adjustierung vorgenommen werden, hier vorgestellt nach der Methode von Scheffé.⁶ Sie ist in `ScheffeTest()` aus dem Paket `DescTools` implementiert.

```
ScheffeTest(<aov-Modell>, which="UV", contrasts=<Kontrastmatrix>,
            conf.level=0.95)
```

Als erstes Argument ist die mit `aov()` angepasste ANOVA zu übergeben. `which` erwartet den Namen des Gruppierungsfaktors, auf den sich die Kontraste beziehen, was hier im einfaktoriellen Fall optional ist. Für `contrasts` ist die Kontrastmatrix anzugeben, wobei die Koeffizienten eines Kontrasts in einer Spalte stehen. Das Argument `conf.level` legt die Breite des Konfidenzintervalls für ψ fest. Hier sollen wieder die drei Kontraste des letzten Abschnitts getestet werden.

```
> library(DescTools)                                # für ScheffeTest()
# transponiere cntrMat mit t(), damit Koeffizienten in Spalten stehen
> ScheffeTest(aovCRp, which="IV", contrasts=t(cntrMat))
Posthoc multiple comparisons of means : Scheffe Test
95% family-wise confidence level
Fit: aov(formula = DV ~ IV, data = dfCRp)
$IV
      diff     lwr.ci     upr.ci    pval
A-D   -0.5836704 -1.194230  0.02688944 0.0671 .
A,B,C-D -0.4845486 -0.986326  0.01722875 0.0629 .
B-C   -0.5655991 -1.185034  0.05383580 0.0880 .
```

In der Ausgabe stehen die Ergebnisse für jeweils einen Kontrast in einer Zeile. Dies sind die Kontrastschätzung $\hat{\psi}$ in der Spalte `diff`, das Konfidenzintervall für ψ in den Spalten `lwr.ci` und `upr.ci` und schließlich der *p*-Wert in der Spalte `pval`.

Bei der manuellen Kontrolle gilt zunächst alles bereits für a-priori Kontraste Ausgeführte. Lediglich die Wahl des kritischen Werts weicht ab und ergibt sich zur α -Adjustierung aus einer *F*-Verteilung. Dieser kritische Wert ist mit der quadrierten a-priori *t*-Teststatistik zu vergleichen – die etwa in der von `summary(glht(...))` zurückgegebenen Liste in der Komponente `test$tstat` steht.

```
> dfSSb <- P-1                                     # df von SS between
> (Fstat <- sumRes$test$tstat^2)                  # quadrierte t-Teststatistik
      A-D 1/3*(A+B+C)-D      B-C
```

⁶Für weitere vgl. `?PostHocTest` nach Laden des Pakets `DescTools`.

```
7.299631      7.448600  6.659600
```

```
# kritischer F-Wert für quadrierte t-Teststatistik
> (Fcrit <- dfSSb*qf(0.05, df1=dfSSb, df2=dfSSw, lower.tail=FALSE))
[1] 7.987706

# p-Wert einseitig
> (pVal <- pf(Fstat/dfSSb, dfSSb, dfSSw), lower.tail=FALSE)
A-D   1/3*(A+B+C)-D          B-C
0.06706707    0.06294284  0.08801514
```

Die Wahl des kritischen Werts 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 Kontrastraums stammen), kann der kritische Wert entsprechend anders gewählt werden. In diesem Fall erfolgt eine gleichzeitige α -Adjustierung 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 $(q-1) * qf(1-0.05, q-1, dfSSw)$.

Paarvergleiche mit *t*-Tests und α -Adjustierung

Taucht die spezifischere Frage auf, welche paarweisen Gruppenunterschiede vorliegen, besteht eine andere Herangehensweise in der Anwendung von $\binom{p}{2} = \frac{p(p-1)}{2}$ *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, lässt sich die Funktion `pairwise.t.test()` nutzen, die alle möglichen Paarvergleiche testet und verschiedene Verfahren 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 Messwerte enthalten, `g` ist der zugehörige Faktor derselben Länge wie `x`, der für jedes Element von `x` die Gruppenzugehörigkeit codiert. Über das Argument `p.adjust.method` wird die α -Adjustierung festgelegt. Als Methoden stehen u. a. jene nach Holm (Voreinstellung) und Bonferroni zur Auswahl, für weitere vgl. `?p.adjust`. Ist von Varianzhomogenität auszugehen und daher `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` anzudeuten ist. Für gerichtete Tests ist das Argument `alternative` auf "less" oder "greater" zu setzen – Voreinstellung ist "two.sided" für ungerichtete Tests.

```
> DV <- dfCRp$DV                      # Daten
> IV <- dfCRp$IV                      # Gruppierungsfaktor

# paarweise t-Tests mit alpha-Adjustierung nach Bonferroni
> pairwise.t.test(DV, IV, p.adjust.method="bonferroni")
```

```
Pairwise comparisons using t tests with pooled SD
data: DV and IV
      A       B       C
B 1.0000 -      -
C 0.2694 0.0647 -
D 0.0460 0.0088 1.0000
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 separat durchgeführten *t*-Tests ohne α -Adjustierung erhalten hätte.

Simultane Konfidenzintervalle nach Tukey

Die Konstruktion simultaner Vertrauensintervalle nach Tukey für die paarweisen Differenzen der Gruppenerwartungswerte (Tukey *honestly significant differences*) stellt eine weitere Möglichkeit dar, auf Unterschiede zwischen jeweils zwei Gruppen zu testen.

```
TukeyHSD(<aov-Objekt>, which="<Faktor>", conf.level=0.95)
```

Als erstes Argument erwartet `TukeyHSD()` ein von `aov()` erstelltes Objekt. Wurde in ihm 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. Über `conf.level` wird die Breite des Vertrauensintervalls für die jeweilige Differenz zweier Erwartungswerte festgelegt.

```
> (tHSD <- TukeyHSD(aovCRp))
Tukey multiple comparisons of means
95% family-wise confidence level
Fit: aov(formula = DV ~ IV, data = dfCRp)
```

| \$IV | | | | |
|------|------------|--------------|-----------|-----------|
| | diff | lwr | upr | p adj |
| B-A | -0.1341170 | -0.706534340 | 0.4383004 | 0.9292665 |
| C-A | 0.4314821 | -0.122736070 | 0.9857003 | 0.1843991 |
| D-A | 0.5836704 | 0.022649650 | 1.1446911 | 0.0380018 |
| C-B | 0.5655991 | -0.003576610 | 1.1347748 | 0.0521308 |
| D-B | 0.7177873 | 0.141985791 | 1.2935888 | 0.0079573 |
| D-C | 0.1521882 | -0.405524534 | 0.7099010 | 0.8935478 |

Die Ausgabe umfasst für jeden Gruppenvergleich die beobachtete Mittelwertsdifferenz (`diff`), ihr Vertrauensintervall (`lwr`, `upr`) sowie den zugehörigen *p*-Wert (`p adj`). Auch `glht()` aus dem Paket `multcomp` kann alle paarweisen Gruppenvergleiche mit der α -Adjustierung nach Tukey testen, indem das Argument `linfct` auf `mcp(<UV>="Tukey")` gesetzt wird.

```
> library(multcomp) # für glht()
> tukey <- glht(aovCRp, linfct=mcp(IV="Tukey")) # Tukey Kontraste
> summary(tukey) # Tests ...
> confint(tukey) # Konfidenzintervalle ...
```

Die Ergebnisse lassen sich manuell nachvollziehen, wobei hier zu beachten ist, dass ungleiche Gruppengrößen vorliegen. Kritischer Wert und Verteilungsfunktion der Teststatistik sind mit `qtukey()` bzw. `ptukey()` zu berechnen.

```
> Mj <- tapply(dfCRp$DV, dfCRp$IV, mean) # Gruppenmittelwerte
> Nj <- table(dfCRp$IV) # Gruppengrößen

# alle paarweisen Mittelwertsdifferenzen
> diffMat <- outer(Mj, Mj, "-")
> (diffs <- diffMat[lower.tri(diffMat)])
[1] -0.1341170 0.4314821 0.5836704 0.5655991 0.7177873 0.1521882

# Skalierungsfaktor: Wurzel aus Summe der Kehrwerte je zweier Nj
> NjMat <- sqrt(outer(1/Nj, 1/Nj, "+"))
> NjFac <- NjMat[lower.tri(NjMat)]
> qTs <- abs(diffs) / (sqrt(MSw/2) * NjFac) # Teststatistiken
> (pVals <- ptukey(qTs, P, dfSSw, lower.tail=FALSE)) # p-Werte
[1] 0.92926654 0.18439914 0.03800183 0.05213078 0.00795732 0.89354783

# halbe Breite Vertrauensintervall für paarweise Mittelwertsdifferenz
> tWidth <- qtukey(0.05, P, dfSSw, lower.tail=FALSE)*sqrt(MSw/2)*NjFac
> diffs - tWidth # Vertrauensintervalle untere Grenzen
[1] -0.706534340 -0.122736070 0.022649650 -0.003576610
[5] 0.141985791 -0.405524534

> diffs + tWidth # Vertrauensintervalle obere Grenzen
[1] 0.4383004 0.9857003 1.1446911 1.1347748 1.2935888 0.7099010
```

Die Konfidenzintervalle können grafisch veranschaulicht werden, indem das Ergebnis von `TukeyHSD()` einem Objekt zugewiesen und dies an `plot()` übergeben wird (Abb. 7.2). Die gestrichelt gezeichnete senkrechte Linie markiert die 0 – Intervalle, die die 0 enthalten, entsprechen einem nicht signifikanten Paarvergleich.

```
> plot(tHSD)
```

7.5 Einfaktorielle Varianzanalyse mit abhängigen Gruppen (RB-*p*)

In einem RB-*p* Design werden in *p* Bedingungen eines Gruppierungsfaktors (UV) abhängige Beobachtungen einer Zielgröße (AV) gemacht. Diese Situation verallgemeinert jene eines *t*-Tests für abhängige Stichproben auf mehr als zwei Gruppen. Jede Menge aus *p* abhängigen Beobachtungen (eine aus jeder Bedingung) wird als *Block* bezeichnet. Versuchsplanerisch ist

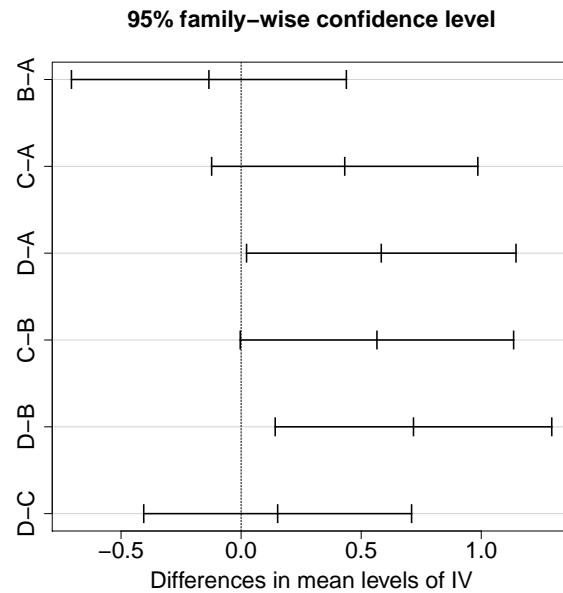


Abbildung 7.2: Grafische Darstellung simultaner Vertrauensintervalle nach Tukey

zu beachten, dass im Fall der Messwiederholung die Reihenfolge der Beobachtungen für jede Person randomisiert wird, weswegen das Design unter *randomized block design* firmiert. Bei gematchten Personen ist zunächst die Blockbildung so vorzunehmen, dass jeder Block p Personen umfasst, die bzgl. relevanter Störvariablen homogen sind. Innerhalb jedes Blocks müssen daraufhin die Personen 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 festen (*fixed*) Gruppenfaktor stellt die Blockzugehörigkeit einen zufälligen (*random*) Faktor dar. Die Bezeichnungen leiten sich daraus ab, dass die Faktorstufen inhaltlich bedeutsam sind, vom Versuchsleiter reproduzierbar festgelegt werden und eine Generalisierung der Ergebnisse über diese Stufen hinaus nicht erfolgt. Die vorliegenden Ausprägungen der Blockzugehörigkeit (im Fall der Messwiederholung die Personen selbst) sind dagegen selbst nicht bedeutsam. Stattdessen sind sie nicht kontrollierte Realisierungen einer Zufallsvariable im statistischen Sinn. Dies erlaubt es den Ergebnissen, über die tatsächlich realisierten Werte hinaus verallgemeinert werden zu können.

7.5.1 Univariat formuliert auswerten und Effektstärke schätzen

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 erhobenen Daten müssen, als Zufallsvektoren aufgefasst, gemeinsam normalverteilt sein. Darüber hinaus muss Zirkularität gelten (Abschn. 7.5.2). Für die Durchführung des Tests mit `aov()` muss der Datensatz so strukturiert sein, dass die Messwerte eines Blocks aus verschiedenen Messzeitpunkten in separaten Zeilen stehen (Long-Format, s. Abschn. 3.3.11). Der Messzeitpunkt muss mit einem Objekt der Klasse `factor` codiert werden. Weiterhin muss es eine

Variable geben, die codiert, von welcher Person, bzw. allgemeiner aus welchem Block ein Messwert stammt. Dieser Blockbildungsfaktor sei im Folgenden als `<Block>` bezeichnet und muss ein Objekt der Klasse `factor` sein.⁷

```
aov(<AV> ~ <UV> + Error(<Block>/<UV>), data=<Datensatz>)
```

Weil die Fehlerstruktur der Messwerte blockweise Abhängigkeiten aufweist, ändert sich die Modellformel in den Aufrufen von `aov()` dahingehend, dass explizit anzugeben ist, aus welchen additiven Komponenten sich die Quadratsumme innerhalb der Gruppen zusammensetzt. Im RB-*p* Design drückt `Error(<Block>/<UV>)` aus, dass `<Block>` in `<UV>` verschachtelt ist. Der durch die Variation des Gruppierungsfaktors entstehende Effekt in der Zielgröße muss deshalb jeweils innerhalb der durch `<Block>` definierten Blöcke analysiert werden.⁸

```
> N      <- 10                      # Zellbesetzung
> P      <- 4                       # Anzahl UV-Stufen
> id     <- factor(rep(1:N, times=P)) # Blockzugehörigkeit
> IV     <- factor(rep(1:P, each=N))  # Intra-Gruppen Faktor
> DV_t1 <- round(rnorm(N, -0.3, 1), 2) # AV zu t1
> DV_t2 <- round(rnorm(N, -0.2, 1), 2) # AV zu t2
> DV_t3 <- round(rnorm(N,  0.1, 1), 2) # AV zu t3
> DV_t4 <- round(rnorm(N,  0.4, 1), 2) # AV zu t4
> DV     <- c(DV_t1, DV_t2, DV_t3, DV_t4) # Gesamtdaten

> dfRBpL <- data.frame(id, IV, DV)      # Datensatz Long-Format
> aovRBp <- aov(DV ~ IV + Error(id/IV), data=dfRBpL)
> summary(aovRBp)

Error: id
    Df  Sum Sq  Mean Sq  F value  Pr(>F)
Residuals  9  15.115   1.6795

Error: id:IV
    Df  Sum Sq  Mean Sq  F value  Pr(>F)
IV          3  12.569   4.1895   3.9695  0.01823 *
Residuals 27  28.496   1.0554
```

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 RB-*p* Design wird die Quadratsumme des festen Effekts (`IV`) gegen die Quadratsumme der Interaktion von festem und Random-Faktor getestet (`Error: id:IV`). Dies wird auch deutlich, wenn die Daten eines RB-*p* Designs mit einer zweifaktoriellen Varianzanalyse im CRF-*pq* Design analysiert werden (Abschn. 7.6), wobei der Random- und der feste Faktor jeweils die Rolle eines Gruppierungsfaktors einnehmen:

⁷Bei Verwendung von `reshape()` werden sowohl die Blockzugehörigkeit als auch der Messzeitpunkt jeweils als numerischer Vektor codiert. Beide Variablen sind deshalb manuell in Faktoren umzuwandeln.

⁸Ausgeschrieben lautet der Term `Error(<Block> + <Block>:<UV>)`. Dies sind die beiden Effekte, deren Quadratsummen sich zur Quadratsumme innerhalb der Gruppen addieren – also zur Quadratsumme der Residuen einer CR-*p* ANOVA, die keinen Effekt von `<Block>` berücksichtigt (`anova(lm(DV ~ IV, data=dfRBpL))`).

```
> anova(lm(DV ~ id + IV + id:IV, data=dfRBpL))
Analysis of Variance Table
  Df  Sum Sq Mean Sq F value Pr(>F)
id       9   15.115  1.6795
IV        3   12.569  4.1895
id:IV    27   28.496  1.0554
Residuals 0   0.000
```

Die sich in dieser Varianzanalyse ergebende Quadratsumme der Interaktion beider Faktoren (`id:IV`) ist identisch zu jener der Residuen beim Test des festen Effekts im RB-*p* Design. Analoges gilt für den Effekt der Variable `id`, dessen Quadratsumme gleich der ersten Residual-Quadratsumme im RB-*p* Design ist. Die Effekt-Quadratsumme von `IV` ist in beiden Varianzanalysen notwendigerweise identisch. Da aus jedem Block nur eine Beobachtung pro Stufe von `IV` vorliegt, beträgt im CRF-*pq* Design die Zellbesetzung 1. Deshalb kann es keine Abweichungen zum Zellmittelwert geben (`Residuals` beträgt 0) – der Vorhersage des Modells mit beiden Haupteffekten und Interaktionseffekt.

Manuelle Kontrolle

Die Ergebnisse lassen sich 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, zudem ist das Gesamtmittel der zentrierten Daten 0.

```
# für Quadratsummen: jeden Wert durch zugehöriges Mittel ersetzen
> MiL   <- ave(DV, id, FUN=mean)                      # Block
> MjL   <- ave(DV, IV, FUN=mean)                      # Messzeitpunkt
> DVctr <- DV - MiL                                    # blockweise zentrierte AV
> MjCtr <- tapply(DVctr, IV, mean)                    # Messzeitpunkt-Mittel
> SSb   <- sum(N * MjCtr^2)                           # Quadratsumme UV
> M     <- mean(DV)                                    # Gesamtmittel

# Interaktion Block:UV, alternativ analog zu SSb im CR-p Design
# IDxIV <- DVctr - ave(DVctr, IV, FUN=mean)
> IDxIV <- DV - MiL - MjL + M                      # Interaktion
> SSE   <- sum(IDxIV^2)                             # Quadratsumme Residuen
> dfSSb <- P-1                                      # Freiheitsgrade UV
> dfSSE <- (N-1) * (P-1)                            # Freiheitsgrade Fehler
> (MSb   <- SSb / dfSSb)                           # mittlere QS UV
[1] 4.189542

> (MSE <- SSE / dfSSE)                             # mittlere QS Residuen
[1] 1.055424

> (Fval <- MSb / MSE)                             # Teststatistik F-Wert
[1] 3.969535

> (pVal <- pf(Fval, dfSSb, dfSSE, lower.tail=FALSE)) # p-Wert
```

```
[1] 0.01823255
```

Effektstärke schätzen

Als Maß für die Effektstärke kann das generalisierte η_g^2 herangezogen werden, zu dessen Schätzung $\hat{\eta}_g^2$ die Effekt-Quadratsumme an der Summe von ihr selbst mit allen Residual-Quadratsummen relativiert wird. Die Berechnung – inkl. des hier identischen einfachen $\hat{\eta}^2$ und des partiellen $\hat{\eta}_p^2$ – übernimmt `eta_squared(aov-Objekt)` aus dem Paket `effectsize`. Alternativ dient auch die Intraklassenkorrelation *ICC1* als Maß der Effektstärke in RB-*p* Designs (Abschn. 10.3.3).

```
> library(effectsize)                      # für eta_squared()
> eta_squared(aovRBp, generalized=TRUE)    # generalisiertes eta^2
Group | Parameter | Eta2 (generalized) | 95% CI
-----
id:IV |           IV |          0.22 | [0.00, 1.00]

> eta_squared(aovRBp, partial=TRUE)          # partielle eta^2
Group | Parameter | Eta2 (partial) | 95% CI
-----
id:IV |           IV |          0.31 | [0.04, 1.00]
```

Da von `summary(aov-Modell)` zurückgegebene Objekte eine recht verschachtelte Struktur besitzen, lassen sich die Quadratsummen zur manuellen Kontrolle hier leichter aus dem von `anova(lm-Modell)` erzeugten Datensatz extrahieren.

```
> anRes <- anova(lm(DV ~ IV:id, data=dfRBpL))

# Residual-Quadratsummen
> SSEid   <- anRes["id",      "Sum Sq"]
> SSEIVid <- anRes["IV:id",   "Sum Sq"]
> SSb     <- anRes["IV",      "Sum Sq"]       # Effekt-QS
> SSEtot  <- SSEid + SSEIVid
> (gEtaSq <- SSb / (SSb + SSEtot))          # generalisiertes eta^2
[1] 0.22372

> (pEtaSq <- SSb / (SSb + SSEIVid))         # partielle eta^2
[1] 0.3060661
```

7.5.2 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 Zielgröße in den einzelnen Gruppen gilt. Sie ist gegeben, wenn

alle aus je zwei unterschiedlichen Variablen gebildeten Differenzvariablen dieselbe theoretische Varianz besitzen.⁹ Ein Test auf Zirkularität ist der von Mauchly (Abschn. 7.5.3, 7.5.4).

Für Situationen ohne gegebene Zirkularität sollen Korrekturformeln verhindern, dass in der 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 getesteten F -Bruchs mit einem Skalar ε , für den $\frac{1}{p-1} \leq \varepsilon \leq 1$ gilt. ε ist genau dann 1, wenn die theoretische Kovarianzmatrix zirkulär ist, andernfalls schwankt ε in Abhängigkeit vom Ausmaß der Abweichung von Zirkularität. Um die zugehörige Schätzung $\hat{\varepsilon}$ zu berechnen, 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, s. Abschn. 7.5.3 für die automatische Berechnung.

```
# Schätzung von epsilon nach Greenhouse & Geisser
> DVmat <- cbind(DV_t1, DV_t2, DV_t3, DV_t4) # Daten im Wide-Format
> N      <- nrow(DVmat)                      # Anzahl Beobachtungen
> S      <- var(DVmat)                       # empirische Kovarianzmatrix
> P      <- nrow(S)                          # Anzahl Variablen (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
> num    <- P^2 * (mdS-mS)^2                  # Zähler
> den    <- (P-1)*(sum(S^2) - 2*P*sum(mSr^2) + P^2*mS^2) # Nenner
> (epsGG <- num/den)
[1] 0.786984

# Schätzung von epsilon nach Huynh & Feldt
> dfId   <- N-1                            # df Blockeffekt = df Fehler
> (epsHF <- ((dfId+1)*(P-1)*epsGG - 2) / ((P-1)*(dfId - (P-1)*epsGG)))
[1] 1.084971

# setze epsilon nach Hyhn & Feldt auf 1, wenn es größer ist
> epsHF <- min(c(1, epsHF))

# Multiplikation der Freiheitsgrade mit epsilon & Vergleich p-Werte
# p-Wert Greenhouse & Geisser
> (pEpsGG <- pf(Fval, dfSSb*epsGG, dfSSE*epsGG, lower.tail=FALSE))
[1] 0.02875036

# p-Wert Huynh & Feldt
> (pEpsHF <- pf(Fval, dfSSb*epsHF, dfSSE*epsHF, lower.tail=FALSE))
[1] 0.01823255
```

Im Beispiel bringt die Korrektur der Freiheitsgrade einen größeren p -Wert mit sich, wobei dies nicht in jeder Situation 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

⁹Dies ist bei nur zwei Gruppen immer der Fall.

nach Huynh und Feldt, deren Schätzung aber auch 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 $\hat{\epsilon}$ pauschal auf das theoretische Minimum $\frac{1}{p-1}$ zu setzen.

Im vorangehenden Abschnitt wurde bereits die Quadratsumme der Interaktion $\langle \text{Block} \rangle : \langle \text{UV} \rangle$ manuell berechnet, die als Quadratsumme der Residuen beim Test des festen Effekts dient. Hier kann zur Berechnung von $\hat{\epsilon}$ nach Greenhouse und Geisser deshalb auch die etwas einfachere Formel verwendet werden, die auf der Kovarianzmatrix der Residuen basiert.

```
# dem Datensatz Interaktion als neue Variable hinzufügen
> dfRBpL <- cbind(dfRBpL, IDxIV)

# Residuen als Matrix im Wide-Format extrahieren
> errMat <- data.matrix(unstack(dfRBpL, IDxIV ~ IV))
> Serr    <- cov(errMat)                      # Kovarianzmatrix der Residuen
> (epsGG <- (1/(P-1)) * sum(diag(Serr))^2 / sum(Serr^2)) # epsilon G&G
[1] 0.786984
```

7.5.3 Multivariat formuliert auswerten mit Anova()

`Anova()` aus dem `car` Paket erlaubt die Durchführung von Varianzanalysen mit Messwiederholung, wobei sowohl die bereits beschriebene univariate wie auch eine multivariate Auswertung möglich ist (Abschn. 12.6.4). Ein Vorteil besteht darin, mit einer leicht nachvollziehbaren Syntax direkt anzugeben, bzgl. welcher Faktoren abhängige Messungen vorliegen. Weiterhin berechnet `Anova()` den Mauchly-Test auf Zirkularität und die Schätzungen $\hat{\epsilon}$ als Maß 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(<lm-Modell>, type=c("II", "III"), idata=<Datensatz>,
      idesign=<Modellformel>)
```

Zunächst ist ein mit `lm()` erstelltes lineares Modell zu übergeben, das bei abhängigen Designs multivariat formuliert werden muss – statt einer einzelnen gibt es also mehrere Zielgrößen. In der Modellformel $\langle \text{AV} \rangle \sim \langle \text{UV} \rangle$ werden dazu auf der linken Seite der \sim die Daten der Zielgröße als Matrix zusammengefügt, wobei jede Spalte die Werte der Zielgröße 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 (Abschn. 7.8). Liegt ein reines Intra-Gruppen Design vor, ist hier nur die Konstante 1 anzugeben. Sind etwa die Werte einer an denselben Personen gemessenen Zielgröße 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 Quadratsummen-Typ beim Test festgelegt werden (Typ II oder Typ III, Abschn. 7.6.2), der im mehrfaktoriellen Fall mit ungleichen Zellbesetzungen relevant ist. Die Argumente `idata` und `idesign` dienen der Spezifizierung der Intra-Gruppen Faktoren. Hierfür erwartet `idata` einen Datensatz, der die Struktur der als Matrix zusammengefassten Daten der Zielgröße beschreibt. Im RB-*p* Design beinhaltet `idata` eine Variable der Klasse `factor`, die die Stufen des Intra-Gruppen Faktors codiert. In jeder Zeile von `idata` gibt dieser Faktor an, aus

welcher Bedingung die zugehörige Spalte der Datenmatrix stammt. Im obigen Design mit drei Messzeitpunkten könnte das Argument `idata` damit `data.frame(IV=factor(1:3))` lauten – die erste Spalte der Datenmatrix entspricht der ersten Zeile von `idata`, also der Bedingung 1, usw. Unter `idesign` wird eine Modellformel mit den Intra-Gruppen Vorhersagetermen auf der rechten Seite der `~` angegeben, die linke Seite bleibt leer. Im RB-*p* Design lautet das Argument also `idesign=~IV`.

```
# Datensatz im Wide-Format
> dfRBpW <- data.frame(DV_t1, DV_t2, DV_t3, DV_t4)

# Zwischen-Gruppen Design (hier kein Zwischen-Gruppen Faktor)
> fitRBp <- lm(cbind(DV_t1, DV_t2, DV_t3, DV_t4) ~ 1, data=dfRBpW)
> inRBp  <- data.frame(IV=gl(P, 1))           # Intra-Gruppen Design
> library(car)                                # für Anova()
> AnovaRBp <- Anova(fitRBp, idata=inRBp, idesign=~IV)
```

Der Test des Modells 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)      # ...
```

`Anova()` setzt hier voraus, dass die Anzahl der Freiheitsgrade des Blockeffekts ($n - 1$) mindestens so groß ist wie jene des Intra-Gruppen Effekts ($p - 1$). Es müssen also mindestens so viele Blöcke wie Bedingungen vorhanden sein. Ist dies nicht der Fall, kann für die automatische Berechnung der $\hat{\epsilon}$ -Korrekturen wie im folgenden Abschnitt beschrieben auf `anova()` ausgewichen werden.

7.5.4 Multivariat formuliert auswerten mit `anova()`

Die Auswertung des RB-*p* Designs in multivariater Formulierung ist auch mit der bereits bekannten `anova()` Funktion möglich, die es jedoch anders als `Anova()` erlaubt, dass weniger Blöcke als Gruppen vorliegen. Wie bei `Anova()` muss das erste Argument ein mit `lm()` multivariat formuliertes Modell sein. Ebenso muss die Struktur des Datensatzes im Wide-Format mit der Zielgröße in den unterschiedlichen Bedingungen als Datensatz `idata` angegeben werden. Die Ergebnisse sind identisch zur univariat formulierten Auswertung, wenn das Argument `test="Spherical"` gesetzt wird.¹⁰ ϵ wird mit den Methoden von Greenhouse und Geisser sowie von Huynh und Feldt geschätzt und samt der korrigierten *p*-Werte ausgegeben.

Die Angabe des zu testenden Effekts erfolgt mit den Argumenten `M` und `X` in Form eines Modellvergleichs. `M` spezifiziert als rechte Seite einer Modellformel das umfassendere Intra-Gruppen Modell mit dem zu testenden Effekt, `X` analog das eingeschränkte Intra-Gruppen Modell ohne diesen Effekt (Abschn. 6.3.3, 7.4.3, 12.9.6 sowie Dalgaard, 2007). Die Kennwerte des von `X` zu `M` hinzugenommenen Effekts stehen in der Ausgabe in der Zeile (Intercept).

¹⁰Bei der multivariaten Formulierung des Modells wird intern aufgrund der generischen `anova()` Funktion automatisch `anova.mlm()` verwendet, ohne dass dies explizit angegeben werden muss (Abschnitt 17.3.7). Ohne das Argument `test="Spherical"` wird multivariat getestet (Abschn. 12.6.4).

```
> anova(fitRBp, M=~IV, X=~1, idata=inRBp, test="Spherical")      # ...
```

Der Mauchly-Test auf Zirkularität wird mit der Funktion `mauchly.test()` durchgeführt, deren Argumente dieselben wie beim obigen Aufruf von `anova()` sind.

```
> mauchly.test(fitRBp, M=~IV, X=~1, idata=inRBp)                  # ...
```

7.5.5 Einzelvergleiche und alternative Auswertungsmöglichkeiten

Die Voraussetzungen einer Varianzanalyse im RB-*p* Design sind recht restriktiv: So müssen alle Personen zu denselben Messzeitpunkten beobachtet werden. Personen mit unvollständigen Daten, bei denen Beobachtungen einzelner Messzeitpunkte fehlen, können nicht in die Auswertung einfließen. Weiter ist die Voraussetzung von Zirkularität in vielen Situationen unplausibel.

Der letztgenannte Punkt ist insbesondere für beliebige Einzelvergleiche relevant, die im Prinzip nach dem Muster jener in Abschn. 7.4.6 für das CR-*p* Design beschriebenen ebenfalls möglich sind. Dabei ist die mittlere Quadratsumme der Interaktion von Gruppierungsfaktors und Blockeffekt in der Rolle der mittleren Residual-Quadratsumme beim CR-*p* Design zu verwenden. Auch die Anzahl der Fehler-Freiheitsgrade ändert sich entsprechend. Dieses Vorgehen gilt aber als anfällig gegenüber Verletzungen der Zirkularitäts-Voraussetzung. Dabei ist die Korrektur der Freiheitsgrade mit einer Schätzung $\hat{\epsilon}$ kein geeignetes Mittel, um die Gefahr einer erhöhten Wahrscheinlichkeit eines α -Fehlers zu verringern. Eine Alternative für Paarvergleiche zwischen den Bedingungen sind *t*-Tests mit `pairwise.t.test(..., paired=TRUE)` (Abschn. 7.4.6).

Bei verletzter Zirkularität bieten verallgemeinerte Schätzgleichungen, GLS-Modelle oder gemischte Modelle (Abschn. 6.6.4) eine flexible Alternative, um die Abhängigkeitsstruktur von Daten aus mehrfacher Beobachtung derselben Personen zu berücksichtigen. Für gemischte Modelle können Einzelvergleiche durchgeführt werden, die wieder in `glht()` aus dem Paket `multcomp` implementiert sind (Abschn. 7.4.6).

Eine multivariate Herangehensweise (Abschn. 7.5.3, 12.6.4) benötigt keine Zirkularitäts-Voraussetzung, ist aber nur für Situationen geeignet, in denen von allen Personen dieselbe Anzahl von Beobachtungen von denselben Messzeitpunkten vorliegen. Eine weitergehende Darstellung alternativer Strategien liefern Kirk (2013) sowie Maxwell et al. (2017).

7.6 Zweifaktorielle Varianzanalyse (CRF-*pq*)

Bei zwei Faktoren (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 Person also in nur einer Bedingungskombination Beobachtungen einer Zielgröße (AV) liefern. Die Zuteilung von Personen auf Bedingungen erfolge randomisiert. In diesem Fall liegt ein CRF-*pq* Design (*completely randomized factorial*) mit *p* Stufen der ersten und *q* Stufen der zweiten UV vor. Abschnitt 12.7.2 zeigt die analoge multivariate Varianzanalyse für mehrere Zielgrößen.

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 i. A. um ein unbalanciertes Design, bei dem sich die Gesamt-Quadratsumme nicht mehr eindeutig in die Quadratsummen der einzelnen Effekte als additive Komponenten zerlegen lässt (Abschn. 7.6.2). Im Folgenden sei ein balanciertes Design vorausgesetzt.

7.6.1 Auswertung und Schätzung der Effektstärke

Für das CRF-*pq* Design erlaubt es das Modell der Varianzanalyse, drei Effekte zu testen: den der ersten UV, den der zweiten UV und den Interaktionseffekt. Jeder dieser Effekte kann in die Modellformel im Aufruf von `lm()` oder `aov()` als modellierender Term eingehen. Hierbei sei daran erinnert, dass der Interaktionseffekt zweier Variablen durch den Doppelpunkt `:` symbolisiert wird.

```
aov(<AV> ~ <UV1> + <UV2> + <UV1>:<UV2>, data=<Datensatz>)
aov(<AV> ~ <UV1> * <UV2>, data=<Datensatz>) # gleichbedeutend
```

Im Beispiel soll neben den beiden eigentlichen UVn auch ein auf diese Bezug nehmender Faktor in den Datensatz aufgenommen werden: Er ignoriert die zweifaktorielle Struktur und codiert die aus der Kombination beider UVn resultierenden Bedingungen als Ausprägungen einer einzelnen UV i. S. der assoziierten einfaktoriellen Varianzanalyse (Abschn. 2.6.2, 7.6.4).

```
> Njk      <- 8                                # Zellbesetzung
> P        <- 2                                # Anzahl Stufen UV1
> Q        <- 3                                # Anzahl Stufen UV2
> IV1     <- factor(rep(1:P, times=Q*Njk))    # Ausprägungen UV1
> IV2     <- factor(rep(1:Q, each=P*Njk))    # Ausprägungen UV2
> IVcomb  <- interaction(IV1, IV2) # Faktor assoz. einfaktorielle ANOVA

# UV-Effekte 1.1 2.1 1.2 2.2 1.3 2.3: Haupteffekte, keine Interaktion
> IVEff    <- c(0.5, -0.5, 0, -1, 1, 0)
> DV       <- IVEff[unclass(IVcomb)] + rnorm(Njk*P*Q, 0, 1) # Gesamt-AV
> dfCRFpq <- data.frame(IV1, IV2, IVcomb, DV)    # Datensatz
> aovCRFpq <- aov(DV ~ IV1*IV2, data=dfCRFpq)
> summary(aovCRFpq)

   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 varianzanalytische Auswertung eines Designs mit drei vollständig gekreuzten Zwischen-Gruppen Faktoren (CRF-*pqr*) unterscheidet sich nur wenig von der beschriebenen zweifaktoriellen Situation. Lediglich die Modellformel mit den zu berücksichtigenden Effekten ist entsprechend anzupassen. Sollen alle Haupteffekte, Interaktionseffekte erster Ordnung und der Interaktionseffekt zweiter Ordnung eingehen, könnte die Modellformel etwa $\langle AV \rangle \sim \langle UV1 \rangle * \langle UV2 \rangle * \langle UV3 \rangle$ lauten. Um den Interaktionseffekt zweiter Ordnung auszuschließen, wäre die Formulierung $\langle AV \rangle \sim \langle UV1 \rangle * \langle UV2 \rangle * \langle UV3 \rangle - \langle UV1 \rangle : \langle UV2 \rangle : \langle UV3 \rangle$ möglich

Mittelwertsdiagramme

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 grafisch über Mittelwertsdiagramme veranschaulicht werden. Neben der Möglichkeit, dies für die Randmittelwerte mit `plot.design()` zu tun, bietet sich auch `interaction.plot()` für die Zellmittelwerte an.¹¹

```
interaction.plot(x.factor=<UV1>, trace.factor=<UV2>, response=<AV>,
                 fun=mean, col="<Farben>", lwd=<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 Variablenamen 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. 7.3).

```
> plot.design(DV ~ IV1*IV2, data=dfCRFpq, main="Randmittelwerte")
> interaction.plot(IV1, IV2, DV, main="Mittelwertsverläufe",
+                   col=c("red", "blue", "green"), lwd=2)
```

Effektstärke schätzen

Als Maß für die Stärke jedes getesteten Effekts kann das partielle η_p^2 herangezogen werden, zu dessen Schätzung $\hat{\eta}_p^2$ jeweils seine Effekt-Quadratsumme an der Summe von ihr und der Residual-Quadratsumme relativiert wird. Die Berechnung – inkl. des einfachen $\hat{\eta}^2$ – übernimmt `eta_squared(<aov-Objekt>)` aus dem Paket `effectsize`.

```
> library(effectsize)                                # für eta_squared()
> eta_squared(aovCRFpq, partial=FALSE)            # einfaches eta^2
Parameter | Eta2 |      95% CI
-----
IV1       | 0.24 | [0.07, 1.00]
IV2       | 0.03 | [0.00, 1.00]
IV1:IV2   | 0.05 | [0.00, 1.00]

> eta_squared(aovCRFpq, partial=TRUE)              # partielles eta^2
Parameter | Eta2 (partial) |      95% CI
-----
```

¹¹ Aufwendiger gestaltete und sehr flexibel anpassbare Visualisierungen bietet das Paket `emmeans` mit der in Abschn. 6.3.4 vorgestellten Funktion `emmpip()`. Nach seiner Installation zeigt `vignette("interactions", package="emmeans")` Beispiele. Das Paket eignet sich auch für komplexere Modelle und bietet die Möglichkeit Kontraste zu testen.

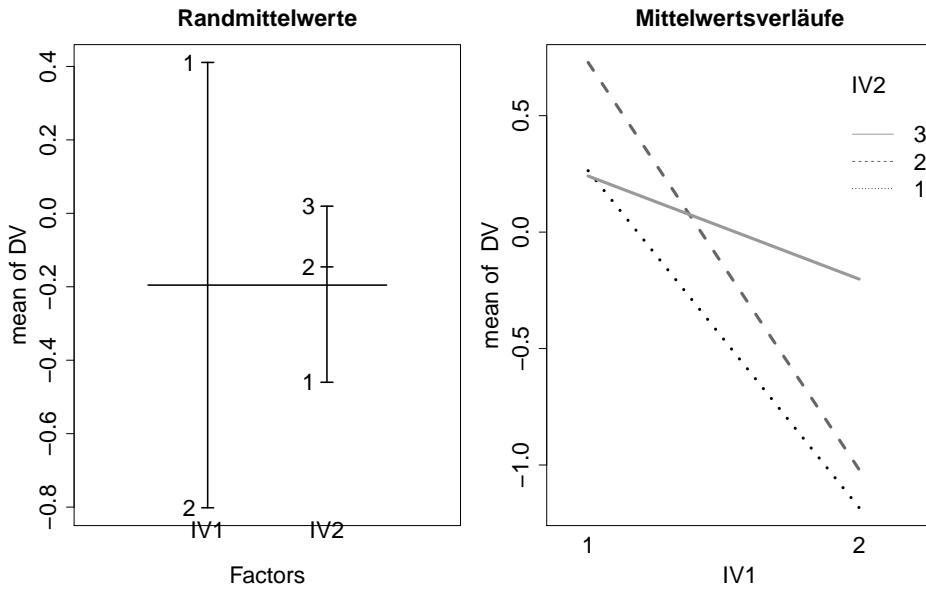


Abbildung 7.3: Darstellung der Randmittelwerte durch `plot.design()` sowie der Gruppenmittelwerte durch `interaction.plot()`

| | | | |
|---------|--|------|--------------|
| IV1 | | 0.26 | [0.09, 1.00] |
| IV2 | | 0.04 | [0.00, 1.00] |
| IV1:IV2 | | 0.07 | [0.00, 1.00] |

Für die manuelle Kontrolle wird das Modell zunächst mit `anova(lm(...))` angepasst, aus dessen Ergebnis sich die Quadratsummen leicht extrahieren lassen.

```
> anRes <- anova(lm(DV ~ IV1*IV2, data=dfCRFpq))
> SS1   <- anRes["IV1",           "Sum Sq"]      # Quadratsumme UV1
> SS2   <- anRes["IV2",           "Sum Sq"]      # Quadratsumme UV2
> SSI    <- anRes["IV1:IV2",     "Sum Sq"]      # Quadratsumme Interaktion
> SSE    <- anRes["Residuals",   "Sum Sq"]      # Quadratsumme Residuen

> (etaSq1 <- SS1 / (SS1 + SS2 + SSI + SSE))  # eta^2 UV1
[1] 0.238161

> (etaSq2 <- SS2 / (SS1 + SS2 + SSI + SSE))  # eta^2 UV2
[1] 0.02561867

> (etaSqI <- SSI / (SS1 + SS2 + SSI + SSE))  # eta^2 Interaktion
[1] 0.05048956

> (pEtaSq1 <- SS1 / (SS1 + SSE))              # partielle eta^2 UV1
[1] 0.2577802

> (pEtaSq2 <- SS2 / (SS2 + SSE))              # partielle eta^2 UV2
[1] 0.03601418
```

```
> (pEtaSqI <- SSI / (SSI + SSE)) # part. eta^2 Interaktion
[1] 0.06857941
```

7.6.2 Quadratsummen vom Typ I, II und III

In mehrfaktoriellen Designs mit unbalancierten Zellbesetzungen offenbart sich, dass R eine andere Methode zur Berechnung der Quadratsummen der Haupteffekte heranzieht, als es der Voreinstellung etwa von SAS, Stata oder SPSS entspricht.¹² Für eine ausführlichere Darstellung s. Abschn. 12.9.2, 12.9.6 sowie Maxwell et al. (2017).

R berechnet sequentielle Quadratsummen vom Typ I, für die bei unbalancierten Zellbesetzungen die Reihenfolge der im Modell berücksichtigten Terme bedeutsam ist. Die Quadratsumme eines Effekts wird hier als Reduktion der Quadratsumme der Residuen (*residual sum of squares*, RSS) beim Wechsel zwischen den folgenden beiden Modellen berechnet: dem Modell mit allen Vorhersagetermen, die in der Modellformel vor dem zu testenden Effekt auftauchen und dem Modell, in dem zusätzlich dieser Effekt selbst berücksichtigt wird (Abschn. 6.3.3). Der resultierende Test ist bei Haupteffekten äquivalent zur 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 ($\langle AV \rangle \sim 1$).

Quadratsummen vom Typ II für Haupteffekte berechnen sich als RSS-Differenz beim Wechsel zwischen den zwei folgenden Modellen: dem Modell mit allen Haupteffekten, aber ohne deren Interaktion sowie demselben Modell ohne den jeweils interessierenden Haupteffekt (egal wo er in der Modellformel steht). Die Reihenfolge der Effekte im Modell ist dabei irrelevant. Für den Test der Interaktion wird das vollständige Modell (beide Haupteffekte und Interaktion) mit dem Modell verglichen, das nur beide Haupteffekte berücksichtigt.

SAS und SPSS dagegen verwenden in der Voreinstellung 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 Modellformel stehend) und dem vollständigen Modell mit allen Termen. Die Reihenfolge der Effekte im Modell ist dabei irrelevant. Die Summe aller einzelnen Effekt-Quadratsummen hat hier bei unbalancierten Zellbesetzungen keine Bedeutung. Der so berechnete Test ist bei Haupteffekten äquivalent zur Frage, ob die gleichgewichteten Randerwartungswerte übereinstimmen.¹³

Unabhängig vom Quadratsummen-Typ ist die Fehler-Quadratsumme beim Test einer Hypothese immer jene des vollständigen Modells mit allen Vorhersagetermen der Modellformel. Es

¹²Die hier verwendete Terminologie von *Typen von Quadratsummen* wurde ursprünglich mit dem Programm SAS eingeführt und bezeichnet letztlich unterschiedliche Hypothesen in varianzanalytischen Designs mit mehreren Faktoren (Blair & Higgins, 1978).

¹³Die genannte Äquivalenz von Modellvergleichen und Hypothesen über ungewichtete Randerwartungswerte setzt voraus, dass ein passendes Codierschema für kategoriale Variablen verwendet wird, z. B. die Effektcodierung (Venables, 2018; Abschn. 12.9.2).

handelt sich bei Quadratsummen vom Typ I, II und III letztlich nicht um verschiedene Berechnungsmethoden desselben Kennwerts, stattdessen dienen sie Tests inhaltlich unterschiedlicher Hypothesen (Blair & Higgins, 1978), die sich bei Haupteffekten auf unterschiedliche Modellvergleiche beziehen. Der Test der Interaktion stimmt hingegen bei allen Typen überein. Bei proportional ungleichen Zellbesetzungen¹⁴ ist zudem der Test der Haupteffekte beim Typ I gleich jenem beim Typ II. Im Spezialfall gleicher Zellbesetzungen liefern die Quadratsummen vom Typ I, II und III dieselben Ergebnisse.

Das folgende Beispiel eines CRF-33 Designs mit unbalancierten Zellbesetzungen ist jenes aus Maxwell et al. (2017, p. 375). 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) # Gesamtdaten

# 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()
> 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 dem
# Vergleich der sequentiell aufeinander aufbauenden Modelle
> SSI1 <- anova(lm(DV ~ 1,           dfMD), lm(DV ~ IV1,           dfMD))
```

¹⁴Proportional ungleiche Zellbesetzungen liegen vor, wenn $\frac{n_{jk}}{n_{jk'}} = \frac{n_{j'k}}{n_{j'k'}}$ sowie $\frac{n_{jk}}{n_{jk'}} = \frac{n_{jk'}}{n_{j'k'}}$ für alle j, j', k, k' gilt.

Kapitel 7 t-Tests und Varianzanalysen

```

> 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[2, "Sum of Sq"]                      # QS Typ I von UV1
[1] 101.1111

> SSI2[2, "Sum of Sq"]                      # QS Typ I von UV2
[1] 1253.189

> SSIi[2, "Sum of Sq"]                      # 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[2, "Sum of Sq"]                      # QS Gesamtmodell
[1] 1368.487

# Summe aller einzelnen Quadratsummen vom Typ I
> SSI1[2, "Sum of Sq"] + SSI2[2, "Sum of Sq"] + SSIi[2, "Sum of Sq"]
[1] 1368.487

```

In R gibt es im wesentlichen zwei Möglichkeiten, Quadratsummen vom Typ III zu erhalten. Zum einen kann mit `drop1()` (Abschn. 6.3.3) die RSS-Änderung berechnet werden, die sich jeweils ergibt, wenn ein Vorhersageterm aus der Modellformel gestrichen wird und alle anderen beibehalten werden. Diese Vergleiche liegen gerade Quadratsummen vom Typ III zugrunde. Die Quadratsumme der Residuen steht in der Ausgabe in der Zeile `<none>`. Zum anderen testet `Anova()` aus dem Paket `car` mit Quadratsummen vom Typ III, wenn das Argument `type="III"` gesetzt ist (analog Quadratsummen vom Typ II, Abschn. 7.5.3). In beiden Fällen ist das `contrasts` Argument für `lm()` notwendig, um von der voreingestellten Dummy-Codierung (Treatment-Kontraste) zur Effektcodierung der Faktoren zu wechseln (Abschn. 12.9.2).

```

# lineares Modell mit Effektcodierung
> fitIII <- lm(DV ~ IV1 + IV2 + IV1:IV2, data=dfMD,
+                 contrasts=list(IV1=contr.sum, IV2=contr.sum))

> drop1(fitIII, ~ ., 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)                                # für Anova()
> Anova(fitIII, type="III")                  # QS Typ III ...
```

Die Quadratsumme der Interaktion unterscheidet sich nicht zwischen Typ I, II und III, die hier alle dieselben Modellvergleiche verwenden: Das eingeschränkte Modell ist immer das mit beiden Haupteffekten, das vollständige bzw. sequentiell folgende Modell jenes mit beiden Haupteffekten und ihrer Interaktion. Es folgt die manuelle Berechnung der Quadratsummen vom Typ III der Haupteffekte.

```
> (MjkMD <- tapply(mdDV, list(mdIV1, mdIV2), mean))    # Zellmittelwerte
      1       2       3
1 44.66667 49.40 56.85714
2 48.60000 56.00 61.25000
3 48.00000 54.25 62.00000

> (NjkMD <- table(mdIV1, mdIV2))                      # Zellbesetzungen
  1 2 3
1 3 5 7
2 5 6 4
3 5 4 6

> Mj <- rowMeans(MjkMD)                                # ungewichtete Zeilen-M
> Mk <- colMeans(MjkMD)                                # ungewichtete Spalten-M

# effektive Rand-Ns aus harmonischem Mittel der Zellen-Ns
> effNj <- 1 / rowMeans(1/NjkMD)                      # harm. Mittel Zeilen-N
> effNk <- 1 / colMeans(1/NjkMD)                      # harm. Mittel Spalten-N

# Gesamtmittel aus gewichteten Zeilen-Ms, bzw. gewichteten Spalten-Ms
> gM1 <- sum(effNj*Mj) / sum(effNj)                  # aus Zeilen-Ms
> gM2 <- sum(effNk*Mk) / sum(effNk)                  # aus Spalten-Ms

# QS Typ III - mdP bzw. mdQ * quadrierte Differenzen der Randmittel
# zum zugehörigen Gesamtmittel, gewichtet mit dem effektiven Rand-N
> (SSIII1 <- mdP * sum(effNj * (Mj-gM1)^2))    # QS Typ III UV1
[1] 204.7617

> (SSIII2 <- mdQ * sum(effNk * (Mk-gM2)^2))    # QS Typ III UV2
[1] 1181.105

# QS Residuen - Summe der quadrierten Differenzen zum Zellenmittel
> (SSE <- sum((mdDV - ave(mdDV, mdIV1, mdIV2, 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 Residuen des vollständigen Modells dividiert durch ihre Freiheitsgrade.

```
> dfSS1 <- mdP - 1 # Freiheitsgrade QS UV1
> dfSS2 <- mdQ - 1 # Freiheitsgrade QS UV2
> dfSSE <- sum(NjkMD - 1) # Freiheitsgrade QS Residuen vollst. Modell
> (Fval1 <- (SSIII1/dfSS1) / (SSE/dfSSE)) # Teststatistik F-Wert UV1
[1] 3.665829

> (Fval2 <- (SSIII2/dfSS2) / (SSE/dfSSE)) # Teststatistik F-Wert UV2
[1] 21.14520

> (pVal1 <- pf(Fval1, dfSS1, dfSSE, lower.tail=FALSE)) # p-Wert UV1
[1] 0.03555901

> (pVal2 <- pf(Fval2, dfSS2, dfSSE, lower.tail=FALSE)) # p-Wert UV2
[1] 8.44678e-07
```

7.6.3 Bedingte Haupteffekte testen

Im Folgenden sei wieder das balancierte Design aus Abschn. 7.6.1 betrachtet.

Dem Test der bedingten Haupteffekte (*simple effects*) der ersten UV im CRF-*pq* Design liegt folgende Frage zugrunde: Stimmen in einer festen Stufe k der zweiten UV alle Zellerwartungswerte μ_{jk} mit $1 \leq j \leq p$ überein und sind damit gleich dem zugehörigen Randerwartungswert $\mu_{.k}$ (Tab. 7.1)? Für jede der q Stufen der zweiten UV lässt sich ein solcher bedingter Haupteffekt der ersten UV testen. Analog soll der Test eines bedingten Haupteffekts der zweiten UV in einer festen Stufe j der ersten UV prüfen, ob alle Zellerwartungswerte μ_{jk} mit $1 \leq k \leq q$ übereinstimmen und damit gleich dem zugehörigen Randerwartungswert $\mu_{j.}$ sind. Für die zweite UV können p bedingte Haupteffekte getestet werden.

Tabelle 7.1: 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 |

Für den Test bedingter Haupteffekte stellt das Paket `emmeans` die Funktion `joint_tests()` bereit.

```
joint_tests(<aov-Modell>, by="<Faktor>",
            adjust="<alpha-Adjustierung>")
```

Zunächst ist ein mit `aov()` angepasstes Modell zu übergeben. Für `by` ist der Faktor mit den Stufen zu nennen, für die bedingte Haupteffekte des anderen Faktors berechnet werden sollen. Für den Test der bedingten Haupteffekte der UV1 auf allen Stufen der UV2 wäre also die UV2 an `by` zu übergeben. Wie der Gefahr eines erhöhten Fehlers erster Art durch wiederholtes Testen zu begegnen ist, wird in der Literatur uneinheitlich beurteilt (für verschiedene Strategien vgl. Kirk, 2013, p. 377 ff.). Über `adjust` können unterschiedliche Methoden gewählt werden.

```
> library(emmeans)                                # für joint_tests()
> joint_tests(aovCRFpq, by="IV2", adjust="none")
IV2 = 1:
model term df1 df2 F.ratio p.value
IV1           1   42    6.924  0.0118

IV2 = 2:
model term df1 df2 F.ratio p.value
IV1           1   42   10.108  0.0028

IV2 = 3:
model term df1 df2 F.ratio p.value
IV1           1   42    0.647  0.4256
```

Die Ergebnisse lassen sich manuell prüfen, hier am Beispiel des bedingten Haupteffekts der ersten UV in der ersten Stufe der zweiten UV. Dafür ist zum einen die einfaktorielle Varianzanalyse der ersten UV notwendig, durchgeführt für die erste Stufe der zweiten UV. Die Beschränkung der Daten auf die feste Stufe einer UV lässt sich mit dem `subset` Argument von `aov()` oder `lm()` erreichen, dem ein geeigneter Indexvektor zu übergeben ist. Die mittlere Effekt-Quadratsumme dieser Varianzanalyse bildet den Zähler der *F*-Teststatistik. Wird die Voraussetzung der Varianzhomogenität als gegeben erachtet, ist der Nenner des *F*-Bruchs für alle Tests identisch und besteht aus der mittleren Quadratsumme der Residuen der zweifaktoriellen Varianzanalyse.

```
# einfaktorielle ANOVAs für UV1 in 1. Stufe der UV2
> CRFp1 <- anova(lm(DV ~ IV1, data=dfCRFpq, subset=(IV2=="1")))

# extrahiere Effekt-Quadratsumme für bedingten Haupteffekt der UV1
> SSp1 <- CRFp1["IV1", "Sum Sq"]          # in Stufe 1 der UV2

# zweifaktorielle ANOVA: QS und df UV1, Interaktion, Residuen
> CRFpq <- anova(lm(DV ~ IV1*IV2, data=dfCRFpq))
> SSA   <- CRFpq["IV1",      "Sum Sq"]      # QS UV1
> SSE   <- CRFpq["Residuals", "Sum Sq"]     # QS Residuen
> dfSSA <- CRFpq["IV1",      "Df"]          # Freiheitsgrade QS UV1
> dfSSE <- CRFpq["Residuals", "Df"]         # Freiheitsgrade Fehler

# F-Wert des bedingten Haupteffekts der UV1
> Fp1 <- (SSp1/dfSSA) / (SSE/dfSSE)       # in Stufe 1 der UV2
> (pP1 <- pf(Fp1, dfSSA, dfSSE, lower.tail=FALSE))    # p-Wert
[1] 0.01184106
```

7.6.4 Beliebige a-priori Kontraste

Im Vergleich zum einfaktoriellen Fall im CR-*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 *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. 7.4.6 formulieren und testen.

Die mittlere Quadratsumme der Residuen 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 Tab. 7.1 dargestellten Schema vorliegen. Als assoziiertes einfaktorielles Schema ergibt sich das in Tab. 7.2 aufgeführte.

Tabelle 7.2: Assoziiertes einfaktorielles Schema zum CRF-23 Design

| UVcomb – 1 | UVcomb – 2 | UVcomb – 3 | UVcomb – 4 | UVcomb – 5 | UVcomb – 6 | M |
|------------|------------|------------|------------|------------|------------|-------|
| μ_{11} | μ_{21} | μ_{12} | μ_{22} | μ_{13} | μ_{23} | μ |

```
# assoziierte einfaktorielle sowie die zweifaktorielle ANOVA
> CRFpq1 <- aova(lm(DV ~ IVcomb, data=dfCRFpq))
> CRFpq2 <- aova(lm(DV ~ IV1*IV2, data=dfCRFpq))

# prüfe MS within = MS error
> all.equal(CRFpq1[["Residuals", "Mean Sq"]],
+            CRFpq2[["Residuals", "Mean Sq"]])
[1] TRUE
```

Mit `glht()` aus dem Paket `multcomp` soll im Beispiel das Mittel der Erwartungswerte μ_{11} und μ_{12} gegen das Mittel der verbleibenden vier Gruppenerwartungswerte getestet werden. Unter H_0 soll der Kontrast gleich 0 sein, unter H_1 größer als 0.

```
> library(multcomp)                                # für glht()
> aovComb <- aov(DV ~ IVcomb, data=dfCRFpq)      # aov-Modell

# Matrix der Kontrastkoeffizienten - hier nur eine Zeile
> cntrMat <- rbind("contr 01"=c(1/2, -1/4, 1/2, -1/4, -1/4, -1/4))
> summary(glht(aovComb, 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)
contr 01 <= 0     1.0372      0.3368     3.08  0.00182 **
```

Da die Gruppengrößen hier gleich sind, vereinfacht sich die Formel für die Teststatistik in der manuellen Berechnung, da nicht mehr gruppenweise mit der Anzahl der Beobachtungen gewichtet werden muss.

```

> Mjk      <- tapply(DV, IVcomb, mean)                      # Zellenmittel
> dfSSE    <- (Njk-1)*P*Q                                    # df von QS error
> SSE      <- sum((DV - ave(DV, IVcomb, FUN=mean))^2) # QS error
> MSE      <- SSE / dfSSE                                     # MS error
> (psiHat <- sum(cntrMat[1, ] * Mjk))                      # Kontrastschätzer
[1] 1.03724

> lenSq   <- sum(cntrMat[1, ]^2 / Njk)                      # quadrierte Länge
> (tStat <- psiHat / sqrt(lenSq*MSE))                      # Teststatistik t
[1] 3.079714

> (tCrit <- qt(0.05, dfSSE, lower.tail=FALSE)) # krit. t-Wert einseitig
[1] 1.681952

> (pVal <- pt(abs(tStat), dfSSE, lower.tail=FALSE)) # p-Wert einseitig
[1] 0.001822898

# untere Grenze des einseitigen 95%-Vertrauensintervalls für psi
> (ciLo <- psiHat - tCrit*sqrt(lenSq*MSE))
[1] 0.4707625

```

Sollen mehrere Kontraste gleichzeitig getestet werden, ist dies durch Zusammenstellung der zugehörigen Kontrastvektoren als (ggf. benannte) Zeilen einer Matrix möglich. Hier werden drei Kontraste ohne Adjustierung des α -Niveaus getestet.

```

# Matrix der Kontrastkoeffizienten
> cntrMat <- rbind("contr 01"=c( 1/2, -1/4, 1/2, -1/4, -1/4, -1/4),
+                     "contr 02"=c( 0,     0,     1,     0,    -1,     0),
+                     "contr 03"=c(-1/2, -1/2, 1/4, 1/4, 1/4, 1/4))

> library(multcomp)                                         # für glht()
> (sumRes <- summary(glht(aovComb, 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)
contr 01 <= 0     1.0372     0.3368   3.080  0.00182 ** 
contr 02 <= 0     0.4878     0.5500   0.887  0.19009
contr 03 <= 0     0.3969     0.3368   1.178  0.12264

```

Das Ergebnis lässt sich manuell prüfen (s. Abschn. 12.1.1 für die Matrixmultiplikation).

```
> psiHats <- cntrMat %*% Mjk # Kontrastschätzungen
> lenSqs <- cntrMat^2 %*% (1/rep(Njk, ncol(cntrMat))) # quadr. Längen
> tStats <- psiHats / sqrt(lenSqs*MSE) # Teststatistiken
> pVals <- pt(abs(tStats), dfSSE, lower.tail=FALSE) # p-Werte eins.
> data.frame(psiHats, tStats, pVals)
   psiHats     tStats      pVals
contr 01  1.0372399  3.0797137  0.001822898
contr 02  0.4877878  0.8869062  0.190090003
contr 03  0.3968684  1.1783590  0.122643406
```

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 $(\frac{1}{3}, -\frac{1}{3}, \frac{1}{3}, -\frac{1}{3}, \frac{1}{3}, -\frac{1}{3})$ 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 $(\frac{1}{2}, \frac{1}{2}, -\frac{1}{4}, -\frac{1}{4}, -\frac{1}{4}, -\frac{1}{4})$ den mittleren Erwartungswert $\mu_{.1}$ gegen das Mittel der mittleren Erwartungswerte $\mu_{.2}$ und $\mu_{.3}$ 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 $(\frac{1}{2}, -\frac{1}{2}, -\frac{1}{4}, \frac{1}{4}, -\frac{1}{4}, \frac{1}{4})$ 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 <- sum(cntrVecA^2 / (Q*Njk)) # quadrierte Länge
> (statA <- psiHatA / sqrt(lenSqA*MSE)) # Teststatistik
```

```
[1] 3.819294
```

```
# 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   <- sum(cntrVecB^2 / (P*Njk))             # quadrierte Länge
> (statB   <- psiHatB / sqrt(lenSqB*MSE))           # Teststatistik
[1] -1.178359
```

7.6.5 Beliebige post-hoc Kontraste nach Scheffé

Beliebige Kontraste 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.¹⁵ Sie ist in `ScheffeTest()` aus dem Paket `DescTools` implementiert (Abschn. 7.4.6).

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.

```
# transponiere cntrMat mit t(), damit Koeffizienten in Spalte stehen
> aovComb <- aov(DV~IVcomb, data=dfCRFpq)    # assozierte 1-fakt. ANOVA
> library(DescTools)                           # für ScheffeTest()
> ScheffeTest(aovComb, which="IVcomb", contrasts=t(cntrMat))
Posthoc multiple comparisons of means : Scheffe Test
95% family-wise confidence level
Fit: aov(formula = DV ~ IVcomb, data = dfCRFpq)
$IVcomb
      diff      lwr.ci     upr.ci     pval
1.1,1.2-2.1,2.2,1.3,2.3 1.0372399 -0.1385869 2.213067 0.1153
1.2-1.3                  0.4877878 -1.4323293 2.407905 0.9766
1.2,2.2,1.3,2.3-1.1,2.1 0.3968684 -0.7789584 1.572695 0.9228
```

Für die manuelle Kontrolle gilt zunächst alles bereits für beliebige a-priori Kontraste Ausgeführt. Lediglich die Wahl des kritischen Werts weicht ab und ergibt sich zur α -Adjustierung aus einer *F*-Verteilung. Dieser kritische Wert ist mit dem Quadrat der a-priori *t*-Teststatistik zu vergleichen – die etwa in der von `summary(glht(...))` zurückgegebenen Liste in der Komponente `test$tstat` steht.

¹⁵Für weitere vgl. `?PostHocTest` aus dem Paket `DescTools`.

```
> (Fstat <- sumRes$test$tstat^2)                      # quadrierte Teststatistik
contr 01    contr 02    contr 03
9.4846366  0.7866025  1.3885299

# df von SS between aus der assoziierten einfaktoriellen Varianzanalyse
> dfSSba <- P*Q - 1

# kritischer F-Wert für quadrierte Teststatistik
> (Fcrit <- dfSSba * qf(0.05, df1=dfSSba, df2=dfSSE, lower.tail=FALSE))
[1] 12.18846

# p-Wert einseitig
> (pVal <- pf(Fstat/dfSSba, dfSSba, dfSSE, lower.tail=FALSE))
contr 01    contr 02    contr 03
0.1152858  0.9766012  0.9227677
```

Die Wahl des kritischen Werts erfolgte hier so, dass alle möglichen Kontraste aus der assoziierten einfaktoriellen Varianzanalyse zugelassen sind. Sollen die Kontraste dagegen nur aus einem Unterraum des Kontrastraums stammen, etwa weil jeweils nur Kontraste aus einer Familie (*A*, *B* oder *I*) von Interesse sind, kann der kritische *F*-Wert wie folgt gewählt werden. In diesem Fall erfolgt eine gleichzeitige α -Adjustierung nur für Kontraste aus der gewählten Familie, woraus ein geringerer kritischer Wert resultiert:

- *A*-Kontraste (*p* Stufen): $(P-1) * qf(1-0.05, P-1, dfSSE))$
- *B*-Kontraste (*q* Stufen): $(Q-1) * qf(1-0.05, Q-1, dfSSE))$
- *I*-Kontraste: $(P-1) * (Q-1) * qf(1-0.05, (P-1)*(Q-1), dfSSE))$

7.6.6 Marginale Paarvergleiche nach Tukey

Alternativ zu beliebigen Kontrasten können im Anschluss an eine Varianzanalyse auch die Randerwartungswerte jeweils eines Faktors mit Tukey-Kontrasten paarweise miteinander verglichen werden (Abschn. 7.4.6). Dafür ist in TukeyHSD() für das Argument which der Faktor zu nennen, dessen Stufen verglichen werden sollen. Die so durchgeführten Tests ignorieren allerdings eine ggf. im Modell vorhandene Interaktion.

```
> aovCRF <- aov(DV ~ IV1+IV2, data=dfCRFpq)    # Modell ohne Interaktion
> TukeyHSD(aovCRF, which="IV2")
Tukey multiple comparisons of means
95% family-wise confidence level
Fit: aov(formula = DV ~ IV1 + IV2, data = dfCRFpq)
$IV2
      diff      lwr      upr      p adj
2-1 0.3142383 -0.6406700 1.269147 0.7061067
3-1 0.4794984 -0.4754099 1.434407 0.4490851
3-2 0.1652602 -0.7896482 1.120168 0.9076526
```

Auch mit `glht()` aus dem Paket `multcomp` können Tukey Paarvergleiche der Randerwartungswerte getestet werden (Abschn. 7.4.6).

```
> library(multcomp)                                # für glht()
> tukey <- glht(aovCRF, linfct=mcp(IV2="Tukey")) # Tukey Kontraste
> summary(tukey)                                 # Tests ...
> confint(tukey)                                 # Konfidenzintervalle
```

7.7 Zweifaktorielle Varianzanalyse mit zwei Intra-Gruppen Faktoren (RBF-*pq*)

Liefert jede Person in jeder der von zwei Faktoren (UVn) gebildeten Bedingungskombinationen Beobachtungen einer Zielgröße (AV), spricht man von einem RBF-*pq* Design (*randomized block factorial*) mit *p* Stufen der ersten und *q* Stufen der zweiten UV. Die Reihenfolge, in der die $p \cdot q$ Bedingungen durchlaufen werden, ist für jede Person 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 Personen stammen. Hierbei ist innerhalb eines Blocks die Zuteilung von Personen 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.

7.7.1 Univariat formuliert auswerten und Effektstärke schätzen

Wie im RB-*p* Design (Abschn. 7.5) müssen die Daten im Long-Format (Abschn. 3.3.11) inkl. eines Blockbildungsfaktors `Block` der Klasse `factor` vorliegen. Nicht alle möglichen Blöcke sind auch experimentell realisiert, sondern nur eine Zufallsauswahl – `Block` ist also ein Random-Faktor. In der Modellformel muss explizit angegeben werden, aus welchen additiven Komponenten sich die Quadratsumme innerhalb der Zellen zusammensetzt. Im RBF-*pq* Design drückt `Error(Block)/(<UV1>*<UV2>)` aus, dass `Block` in den durch die Kombination von `<UV1>` und `<UV2>` entstehenden Bedingungen verschachtelt ist: Die durch die kombinierte Variation von UV1 und UV2 entstehenden Effekte in der AV (beide Haupteffekte und der Interaktionseffekt) sind daher jeweils innerhalb der durch `Block` definierten Blöcke zu analysieren.¹⁶

```
aov(<AV> ~ <UV1>*<UV2> + Error(<Block>/(<UV1>*<UV2>)),
     data=<Datensatz>)
```

¹⁶Der Term lautet `Error(<Block> + <Block>:<UV1> + <Block>:<UV2> + <Block>:<UV1>:<UV2>)`, wenn er ausgeschrieben wird. Dies sind die vier Effekte, deren Quadratsummen sich zur Quadratsumme innerhalb der Zellen addieren – also zur Quadratsumme der Residuen einer CRF-*pq* ANOVA, die keinen Effekt von `<Block>` berücksichtigt (`anova(lm(DV ~ IV1*IV2, data=dfRBFpqL))`).

```

> N <- 10                                     # Anzahl Personen
> P <- 2                                      # Messzeitpunkte UV1
> Q <- 3                                      # Messzeitpunkte UV2
> id <- factor(rep(1:N, times=P*Q))          # Blockzugehörigkeit
> IV1 <- factor(rep(rep(1:P, each=N), times=Q)) # Intra-Gruppen UV1
> IV2 <- factor(rep(rep(1:Q, each=P*N)))      # Intra-Gruppen UV2

# Simulation der AV mit Effekt beider UVn, ohne Interaktion
> DV_t11 <- round(rnorm(N, -0.8, 1), 2)       # AV zu t1-1
> DV_t12 <- round(rnorm(N, -0.7, 1), 2)       # AV zu t1-2
> DV_t13 <- round(rnorm(N, 0.0, 1), 2)         # AV zu t1-3
> DV_t21 <- round(rnorm(N, 0.2, 1), 2)         # AV zu t2-1
> DV_t22 <- round(rnorm(N, 0.3, 1), 2)         # AV zu t2-2
> DV_t23 <- round(rnorm(N, 1.0, 1), 2)         # AV zu t2-3
> DV <- c(DV_t11, DV_t21, DV_t12, DV_t22, DV_t13, DV_t23)

> dfRBFpqL <- data.frame(id, IV1, IV2, DV)      # Datensatz long
> aovRBFpq <- aov(DV ~ IV1*IV2 + Error(id/(IV1*IV2)), data=dfRBFpqL)
> summary(aovRBFpq)
Error: id
      Df  Sum Sq  Mean Sq  F value  Pr(>F)
Residuals  9  3.6306  0.4034

Error: id:IV1
      Df  Sum Sq  Mean Sq  F value  Pr(>F)
IV1        1 16.865  16.8646   11.209  0.00855 **
Residuals  9 13.541   1.5045

Error: id:IV2
      Df  Sum Sq  Mean Sq  F value  Pr(>F)
IV2        2 13.797   6.8987   4.5641  0.02493 *
Residuals 18 27.207   1.5115

Error: id:IV1:IV2
      Df  Sum Sq  Mean Sq  F value  Pr(>F)
IV1:IV2    2  2.2468   1.1234   0.9608  0.4014
Residuals 18 21.0473   1.1693

```

Die Ausgabe unterscheidet sich von jener im CRF-*pq* Design, da hier nicht mehr dieselbe Residual-Quadratsumme für den Test jedes 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 **IV1** wird mit der Quadratsumme aus der Interaktion von **IV1** und dem Random-Faktor **id** in der Rolle der Residual-Quadratsumme verglichen. Analoges gilt für den festen Faktor **IV2**. Die Quadratsumme der Interaktion der beiden festen Faktoren wird entsprechend mit der Quadratsumme der Interaktion zweiter Ordnung vom Random-Faktor und beiden festen Faktoren getestet (vgl. mit der Ausgabe von **anova(lm(DV ~ IV1*IV2*id, **

$\rightarrow \text{data}=\text{dfRBFpqL})$).

Die varianzanalytische Auswertung eines dreifaktoriellen Designs mit drei Intra-Gruppen Faktoren (RBF-*pqr*) unterscheidet sich nur wenig von der beschriebenen zweifaktoriellen Situation. Lediglich die Modellformel mit den zu berücksichtigenden Effekten ist entsprechend anzupassen. Sollen alle Haupteffekte, Interaktionseffekte erster Ordnung und der Interaktionseffekt zweiter Ordnung eingehen, könnte die Modellformel etwa $\langle \text{AV} \rangle \sim \langle \text{UV1} \rangle * \langle \text{UV2} \rangle * \langle \text{UV3} \rangle + \rightarrow \text{Error}(\langle \text{Block} \rangle) / (\langle \text{UV1} \rangle * \langle \text{UV2} \rangle * \langle \text{UV3} \rangle)$ lauten.

Manuelle Kontrolle

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(DV ~ id + IV1, data=dfG, FUN=mean)

# Datensatz aus blockweise über Stufen der UV1 gemittelten Daten
> mDf2 <- aggregate(DV ~ id + IV2, data=dfG, FUN=mean)

# hier: Datensatz aus blockweisen Differenzen zwischen Stufen der UV1
> dDfI <- aggregate(DV ~ id + IV2, data=dfG, FUN=diff)

# äquivalent zum Test des Haupteffekts der UV1 im RBF-pq Design
> summary(aov(DV ~ IV1 + Error(id/IV1), data=mDf1)) # ...

# äquivalent zum Test des Haupteffekts der UV2 im RBF-pq Design
> summary(aov(DV ~ IV2 + Error(id/IV2), data=mDf2)) # ...

# hier: äquivalent zum Test der Interaktion UV1:UV2 im RBF-pq Design
> summary(aov(DV ~ IV2 + Error(id/IV2), data=dDfI)) # ...
```

Im allgemeinen Fall wäre der Test der Interaktion wie folgt manuell durchzuführen.

```
# für Quadratsumme Interaktion: Mittel der Zellen, UV1, UV2, gesamt
> Mjk <- with(dfG, tapply(DV, list(IV1, IV2), mean)) # Zellen-M
> Mj <- with(dfG, tapply(DV, IV1, mean)) # UV1-Mittel
> Mk <- with(dfG, tapply(DV, IV2, mean)) # UV2-Mittel
```

```

> M    <- mean(Mjk)                                # Gesamtmittel

# Interaktion UV1:UV2
> IV1xIV2 <- c(sweep(sweep(Mjk, 1, Mj, "-"), 2, Mk, "-")) + M
> (SSI     <- N * sum(IV1xIV2^2))               # Quadratsumme Interaktion
[1] 2.246813

# QS Residuen: jeden Wert durch zugehöriges Mittel ersetzen
> MjkL <- with(dfG, ave(DV,      IV1, IV2, FUN=mean))
> MijL <- with(dfG, ave(DV, id,   IV1,           FUN=mean))
> MikL <- with(dfG, ave(DV, id,   IV2,           FUN=mean))
> MiL  <- with(dfG, ave(DV, id,           FUN=mean))
> MjL  <- with(dfG, ave(DV,      IV1,           FUN=mean))
> MkL  <- with(dfG, ave(DV,      IV2,           FUN=mean))

# Interaktion Block:UV1:UV2 -> Residuen bei Test Interaktion UV1:UV2
> IDxIV1xIV2 <- dfG$DV - MijL - MikL - MjkL + MiL + MjL + MkL - M
> (SSE      <- sum(IDxIV1xIV2^2))              # Kontrolle: QS Residuen
[1] 21.04732

> dfSSI  <- (P-1) * (Q-1)                      # Freiheitsgrade Interaktion
> dfSSE  <- (P-1) * (Q-1) * (N-1)                # Freiheitsgrade Residuen
> (FvalI <- (SSI/dfSSI) / (SSE/dfSSE))          # F-Wert Interaktion
[1] 0.9607551

# p-Wert Interaktion
> (pValI <- pf(FvalI, dfSSI, dfSSE, lower.tail=FALSE))
[1] 0.4013768

```

Effektstärke schätzen

Als Maß für die Stärke jedes getesteten Effekts kann das generalisierte η_g^2 herangezogen werden, zu dessen Schätzung $\hat{\eta}_g^2$ jeweils seine Effekt-Quadratsumme an der Summe von ihr selbst mit allen Residual-Quadratsummen relativiert wird. Die Berechnung – inkl. des einfachen $\hat{\eta}^2$ und des partiellen $\hat{\eta}_p^2$ – übernimmt `eta_squared(aov-Objekt)` aus dem Paket `effectsize`.

```

> library(effectsize)                            # für eta_squared()
> eta_squared(aovRBFpq, partial=FALSE)          # einfaches eta^2
Group      | Parameter | Eta2 |      95% CI
-----
id:IV1     |       IV1 | 0.17 | [0.00, 1.00]
id:IV2     |       IV2 | 0.14 | [0.00, 1.00]
id:IV1:IV2 | IV1:IV2 | 0.02 | [0.00, 1.00]

> eta_squared(aovRBFpq, partial=TRUE)            # partielle eta^2
Group      | Parameter | Eta2 (partial) |      95% CI

```

```
-----
id:IV1 | IV1 | 0.55 | [0.14, 1.00]
id:IV2 | IV2 | 0.34 | [0.03, 1.00]
id:IV1:IV2 | IV1:IV2 | 0.10 | [0.00, 1.00]

> eta_squared(aovRBFpq, generalized=TRUE) # generalisiertes eta^2
Group | Parameter | Eta2 (generalized) | 95% CI
-----
id:IV1 | IV1 | 0.20 | [0.00, 1.00]
id:IV2 | IV2 | 0.17 | [0.00, 1.00]
id:IV1:IV2 | IV1:IV2 | 0.03 | [0.00, 1.00]
```

Da von `summary(aov-Modell)` zurückgegebene Objekte eine recht verschachtelte Struktur besitzen, lassen sich die Quadratsummen zur manuellen Kontrolle hier leichter aus dem von `anova(lm-Modell)` erzeugten Datensatz extrahieren.

```
> anRes <- anova(lm(DV ~ IV1*IV2*id, data=dfRBFpqL))

> SSEid      <- anRes["id",           "Sum Sq"]
> SSEIV1id   <- anRes["IV1:id",       "Sum Sq"]
> SSEIV2id   <- anRes["IV2:id",       "Sum Sq"]
> SSEIV1IV2id <- anRes["IV1:IV2:id", "Sum Sq"]
> SSEtot <- SSEid + SSEIV1id + SSEIV2id + SSEIV1IV2id
> SS1      <- anRes["IV1",          "Sum Sq"]
> SS2      <- anRes["IV2",          "Sum Sq"]
> SSI      <- anRes["IV1:IV2",      "Sum Sq"]

> (etaSq1 <- SS1 / (SS1 + SS2 + SSI + SSEtot))    # eta^2 UV1
[1] 0.1715017

> (etaSq2 <- SS2 / (SS1 + SS2 + SSI + SSEtot))    # eta^2 UV2
[1] 0.1403111

> (etaSqI <- SSI / (SS1 + SS2 + SSI + SSEtot))    # eta^2 Interaktion
[1] 0.02284859

> (pEtaSq1 <- SS1 / (SS1 + SSEIV1id))    # partielle eta^2 UV1
[1] 0.5546596

> (pEtaSq2 <- SS2 / (SS2 + SSEIV2id))    # partielle eta^2 UV2
[1] 0.336484

> (pEtaSqI <- SSI / (SSI + SSEIV1IV2id)) # part. eta^2 Interaktion
[1] 0.09645404

> (gEtaSq1 <- SS1 / (SS1 + SSEtot))        # generalisiertes eta^2 UV1
[1] 0.2049396
```

```
> (gEtaSq2 <- SS2 / (SS2 + SSEtot))      # generalisiertes eta^2 UV2
[1] 0.174159

> (gEtaSqI <- SSI / (SSI + SSEtot))      # general. eta^2 Interaktion
[1] 0.03320113
```

7.7.2 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 (Abschn. 7.5.2). Meist erfolgt daher separat für jeden Test eine Korrektur der Freiheitsgrade auf Basis der Schätzung $\hat{\epsilon}$ nach Greenhouse und Geisser bzw. nach Huynh und Feldt.

Bei der Schätzung $\hat{\epsilon}$ für die Tests der Haupteffekte ist zunächst jeweils 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 $\hat{\epsilon}$ berechnet wird. Mit zwei Stufen der UV1 ist hier keine ϵ -Korrektur des Tests der UV1 notwendig, 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 $\hat{\epsilon}$ auf Basis der Kovarianzmatrix der Residuen ermitteln.

```
# epsilon-Schätzung für Test der UV2: Daten als Matrix im Wide-Format
> DVmat <- with(dfG, tapply(DV, list(id, IV2), mean))
```

Die weiteren Berechnungen würden mit jenen in Abschn. 7.5.2 übereinstimmen.

```
# epsilon-Schätzung für Test der Interaktion UV1:UV2 aus Residuen
> dfG <- cbind(dfG, IDxIV1xIV2)      # dem Datensatz Residuen hinzufügen

# extrahiere Residuen als Matrix im Wide-Format
> errMat <- data.matrix(unstack(dfG, IDxIV1xIV2 ~ IV2))
> Serr     <- cov(errMat)            # Kovarianzmatrix der Residuen
> (epsGGi <- (1 / (Q-1)) * sum(diag(Serr))^2 / sum(Serr^2))
[1] 0.9934177

> dfId     <- N-1                  # df Blockeffekt = df Fehler
> (epsHFi <- ((dfId+1)*(Q-1)*epsGGi - 2) /
+   ((Q-1)*(dfId - ((Q-1)*epsGGi))))
[1] 1.273915
```

7.7.3 Multivariat formuliert auswerten

Für eine Beschreibung von `Anova()` aus dem `car` Paket s. Abschn. 7.5.3. 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 Modellformel beide Haupteffekte und deren Interaktion 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
> fitRBFpq <- lm(cbind(DV_t11, DV_t21,
+                         DV_t12, DV_t22,
+                         DV_t13, DV_t23) ~ 1,
+                         data=dfRBFpqW)

# Intra-Gruppen Design: Aufbau der Datenmatrix in fitRBFpq
> (inRBFpq <- expand.grid(IV1=g1(P, 1), IV2=g1(Q, 1)))
  IV1  IV2
1    1    1
2    2    1
3    1    2
4    2    2
5    1    3
6    2    3

> library(car)                      # für Anova()
> AnovaRBFpq <- Anova(fitRBFpq, idata=inRBFpq, idesign=~IV1*IV2)
> summary(AnovaRBFpq, multivariate=FALSE, univariate=TRUE)      # ...


```

`Anova()` setzt hier voraus, dass die Anzahl der Freiheitsgrade des Blockeffekts ($n - 1$) mindestens so groß ist wie jene der Interaktion beider Intra-Gruppen Faktoren $((p - 1) \cdot (q - 1))$. Es müssen also mindestens $(p - 1) \cdot (q - 1) + 1$ viele Blöcke vorhanden sein. Ist dies nicht der Fall, kann für die automatische Berechnung der $\hat{\varepsilon}$ -Korrekturen auf `anova()` ausgewichen werden (Abschn. 7.5.4).

Für jeden Test ist mit den Argumenten `M` und `X` das passende Paar vom umfassenderen und eingeschränkten Intra-Gruppen Modell als rechte Seite einer Modellformel zu formulieren. Hier soll dabei wie bei Quadratsummen vom Typ I sequentiell vorgegangen werden (Abschn. 7.6.2, 12.9.6). Die Kennwerte des von `X` zu `M` hinzugenommenen Effekts stehen in der Ausgabe in der Zeile (`Intercept`). Für `mauchly.test()` s. Abschn. 7.5.4.

```
> anova(fitRBFpq, M=~IV1,                  # Test IV1 ...
+        X=~1,                      idata=inRBFpq, test="Spherical")
```

```
> anova(fitRBFpq, M=~IV1 + IV2,                      # Test IV2 ...
+           X=~IV1,          idata=inRBFpq, test="Spherical")

> anova(fitRBFpq, M=~IV1 + IV2 + IV1:IV2,          # Test IV1:IV2 ...
+           X=~IV1 + IV2, idata=inRBFpq, test="Spherical")

# Mauchly-Tests auf Zirkularität (P=2 -> IV1 hier nicht notwendig)
> mauchly.test(fitRBFpq, M=~IV1 + IV2,              # IV2 ...
+               X=~IV1,          idata=inRBFpq)

> mauchly.test(fitRBFpq, M=~IV1 + IV2 + IV1:IV2,   # IV1:IV2 ...
+               X=~IV1 + IV2, idata=inRBFpq)
```

7.7.4 Einzelvergleiche (Kontraste)

Prinzipiell ist es möglich, auch für ein RBF-*pq* Design beliebige Einzelvergleiche als Tests von Linearkombinationen von Erwartungswerten durchzuführen. Dabei ist die Situation analog zu jener im CRF-*pq* Design, wie sie in Abschn. 7.6.4 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 Residuen: Während diese im CRF-*pq* Design für jeden Test dieselbe ist, wird im RBF-*pq* Design jeder Test mit einer anderen Residual-Quadratsumme durchgeführt. Ein *A*-Kontrast wäre im RBF-*pq* Design gegen die Quadratsumme der Interaktion des Blocks mit der UV1 zu testen, ein *B*-Kontrast entsprechend gegen die Quadratsumme der Interaktion des Blocks mit der UV2, ein *I*-Kontrast gegen die Quadratsumme der Interaktion zweiter Ordnung des Blocks mit der UV1 mit der UV2. Die Anzahl der Fehler-Freiheitsgrade ändert sich entsprechend der verwendeten Quadratsumme. Für Vergleiche einzelner Zellen müsste das assoziierte einfaktorielle RB-*p* Design herangezogen werden.

Wie im RB-*p* Design stellt sich die Frage, ob es angemessen ist, Einzelvergleiche in abhängigen Designs auf diese Weise durchzuführen (Abschn. 7.5.5). Eine Alternative bieten verallgemeinerte Schätzgleichungen, GLS-Modelle oder gemischte Modelle (Abschn. 6.6.4), die auch Einzelvergleiche erlauben – etwa mit `glht()` aus dem Paket `multcomp` (Abschn. 7.4.6).

7.8 Zweifaktorielle Varianzanalyse mit Split-Plot-Design (SPF-*p · q*)

Ein Split-Plot-Design liegt im zweifaktoriellen Fall vor, wenn die Bedingungen eines Zwischen-Gruppen Faktors (UV1) mit jenen eines Faktors (UV2) kombiniert werden, bzgl. dessen Stufen abhängige Beobachtungen einer Zielgröße (AV) resultieren – etwa weil es sich um einen Messwiederholungsfaktor handelt. Hat die UV1 *p* und die UV2 *q* Stufen, wobei jede mögliche Stufenkombination auch realisiert wird, spricht man von einem SPF-*p · q* Design (*split plot factorial*).

7.8.1 Univariat formuliert auswerten und Effektstärke schätzen

Versuchsplanerisch müssen zunächst Blöcke aus jeweils q Beobachtungen gebildet werden. 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 sollte. Als weiterer Schritt der Randomisierung wird im Fall der Messwiederholung innerhalb jedes Blocks die Reihenfolge der Beobachtungen bzgl. der q Stufen der UV2 randomisiert. Ein Block kann sich auch aus Beobachtungen unterschiedlicher, z. B. bzgl. relevanter Störvariablen homogener Personen zusammensetzen. In diesem Fall ist die Zuordnung der q Personen 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 enthält.

Wie beim RBF- pq Design (Abschn. 7.7) müssen die Daten im Long-Format (Abschn. 3.3.11) inkl. eines Blockbildungsfaktors $\langle\text{Block}\rangle$ der Klasse **factor** vorliegen. Nicht alle möglichen Blöcke sind auch experimentell realisiert, sondern nur eine Zufallsauswahl – $\langle\text{Block}\rangle$ ist also ein Random-Faktor. In der Modellformel muss explizit angegeben werden, aus welchen additiven Komponenten sich die Quadratsumme innerhalb der Zellen zusammensetzt. Im SPF- $p \cdot q$ Design drückt **Error**($\langle\text{Block}\rangle/\langle\text{UV2}\rangle$) aus, dass $\langle\text{Block}\rangle$ in $\langle\text{UV2}\rangle$ verschachtelt ist: Der Effekt der UV2 ist daher jeweils innerhalb der durch $\langle\text{Block}\rangle$ definierten Blöcke zu analysieren.¹⁷

```
aov(<AV> ~ <UV1>*<UV2> + Error(<Block>/<UV2>), data=<Datensatz>)

> Nj      <- 10                                # Gruppengröße
> P       <- 3                                 # Stufen Zwischen-UV
> Q       <- 3                                 # Zeitpunkte Intra-UV
> id      <- factor(rep(1:(P*Nj), times=Q))    # Blockzugehörigkeit
> IVbtw   <- factor(rep(LETTERS[1:P], times=Q*Nj)) # Zwischen-UV between
> IVwth   <- factor(rep(1:Q, each=P*Nj))        # Intra-UV: within

# Simulation ohne Effekt IVbtw, mit Effekt IVwth, ohne Interaktion
> DV_t1 <- round(rnorm(P*Nj, -0.5, 1), 2)      # AV zu t1
> DV_t2 <- round(rnorm(P*Nj, 0, 1), 2)          # AV zu t2
> DV_t3 <- round(rnorm(P*Nj, 0.5, 1), 2)        # AV zu t3
> DV     <- c(DV_t1, DV_t2, DV_t3)              # Gesamtdaten

> dfSPFpqL <- data.frame(id, IVbtw, IVwth, DV)  # Daten Long-Format
> aovSPFpq <- aov(DV ~ IVbtw*IVwth + Error(id/IVwth), data=dfSPFpqL)
> summary(aovSPFpq)
Error: id
      Df  Sum Sq  Mean Sq  F value  Pr(>F)
IVbtw     2  1.2424  0.6212  0.5533  0.5814
Residuals 27 30.3135  1.1227

Error: id:IVwth
```

¹⁷ Ausgeschrieben lautet der Term **Error**($\langle\text{Block}\rangle + \langle\text{Block}\rangle:\langle\text{UV2}\rangle$). Dies sind die beiden Effekte, deren Quadratsummen sich zur Quadratsumme innerhalb der Zellen addieren – also zur Quadratsumme der Residuen einer CRF- pq ANOVA, die keinen Effekt von $\langle\text{Block}\rangle$ berücksichtigt: `anova(lm(DV ~ IVbtw*IVwth, data=dfSPFpqL))`.

| | Df | Sum Sq | Mean Sq | F value | Pr(>F) |
|-------------|----|--------|---------|---------|-------------|
| IVwth | 2 | 9.494 | 4.7470 | 6.2443 | 0.003635 ** |
| IVbtw:IVwth | 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**. Die Ausgabe macht deutlich, dass die Quadratsumme des Effekts des Zwischen-Gruppen Faktors **IVbtw** mit der Quadratsumme des Random-Faktors **id** in der Rolle der Residual-Quadratsumme verglichen wird. Sowohl die Quadratsumme des Intra-Gruppen Faktors **IVwth** als auch die der Interaktion beider UVn wird mit der Quadratsumme aus der Interaktion von **IVwth** und **id** in der Rolle der Residual-Quadratsumme verglichen (vgl. mit der Ausgabe von **anova(lm(DV ~ IVbtw*IVwth*id, data=dfSPFpqL))**).

Die in Abschn. 7.8.3 verwendete **Anova()** Funktion bietet die Möglichkeit, bei ungleichen Zellbesetzungen bzgl. des Zwischen-Gruppen Faktors Quadratsummen vom Typ II und III zu berechnen, während **aov()** nur Quadratsummen vom Typ I ermittelt (Abschn. 7.6.2).

Effektstärke schätzen

Als Maß für die Stärke jedes getesteten Effekts kann das generalisierte η_g^2 herangezogen werden, zu dessen Schätzung $\hat{\eta}_g^2$ jeweils seine Effekt-Quadratsumme an der Summe von ihr selbst mit allen Residual-Quadratsummen relativiert wird. Die Berechnung – inkl. des einfachen $\hat{\eta}^2$ und des partiellen $\hat{\eta}_p^2$ – übernimmt **eta_squared()** aus dem Paket **effectsize**.

```
> library(effectsize)                                # für eta_squared()
> eta_squared(aovSPFpq, partial=FALSE)              # einfaches eta^2
Group      | Parameter | Eta2 |      95% CI
-----
id         |       IVbtw | 0.01 | [0.00, 1.00]
id:IVwth   |       IVwth | 0.11 | [0.00, 1.00]
id:IVwth   | IVbtw:IVwth | 0.06 | [0.00, 1.00]

> eta_squared(aovSPFpq, partial=TRUE)                # partielles eta^2
Group      | Parameter | Eta2 (partial) |      95% CI
-----
id         |       IVbtw |          0.04 | [0.00, 1.00]
id:IVwth   |       IVwth |          0.19 | [0.04, 1.00]
id:IVwth   | IVbtw:IVwth |          0.11 | [0.00, 1.00]

> eta_squared(aovSPFpq, generalized=TRUE)           # generalisiertes eta^2
Group      | Parameter | Eta2 (generalized) |      95% CI
-----
id         |       IVbtw |          0.02 | [0.00, 1.00]
id:IVwth   |       IVwth |          0.12 | [0.01, 1.00]
id:IVwth   | IVbtw:IVwth |          0.06 | [0.00, 1.00]
```

Da von `summary(aov-Modell)` zurückgegebene Objekte eine recht verschachtelte Struktur besitzen, lassen sich die Quadratsummen zur manuellen Kontrolle hier leichter aus dem von `anova(lm-Modell)` erzeugten Datensatz extrahieren.

```
> anRes <- anova(lm(DV ~ IVbtw*IVwth*id, data=dfSPFpqL))

# Residual-Quadratsummen
> SSEid      <- anRes["id",           "Sum Sq"]
> SSEIVwid   <- anRes["IVwth:id",    "Sum Sq"]
> SSEtot     <- SSEid + SSEIVwid
> SSbtw      <- anRes["IVbtw",       "Sum Sq"]  # QS Zwischen-UV
> SSwth      <- anRes["IVwth",       "Sum Sq"]  # QS Intra-UV
> SSI        <- anRes["IVbtw:IVwth", "Sum Sq"]  # QS Interaktion

> (etaSqB  <- SSbtw / (SSbtw + SSwth + SSI + SSEtot)) # eta^2 Zw.-UV
[1] 0.01429186

> (etaSqW  <- SSbtw / (SSbtw + SSwth + SSI + SSEtot)) # eta^2 Intra-UV
[1] 0.01429186

> (etaSqI  <- SSbtw / (SSbtw + SSwth + SSI + SSEtot)) # eta^2 Interak.
[1] 0.01429186

> (pEtaSqB <- SSbtw / (SSbtw + SSEid))  # partielle eta^2 Zwischen-UV
[1] 0.0393716

> (pEtaSqW <- SSwth / (SSwth + SSEIVwid)) # partielle eta^2 Intra-UV
[1] 0.1878299

> (pEtaSqI <- SSI   / (SSI   + SSEIVwid)) # part. eta^2 Interaktion
[1] 0.1052614

> (gEtaSqB <- SSbtw / (SSbtw + SSEtot))  # general. eta^2 Zwischen-UV
[1] 0.01711126

> (gEtaSqW <- SSwth / (SSwth + SSEtot))  # general. eta^2 Intra-UV
[1] 0.1174138

> (gEtaSqI <- SSI / (SSI + SSEtot))      # general. eta^2 Interaktion
[1] 0.06338389
```

7.8.2 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 Details vgl. [Kirk, 2013](#), p. 541 ff.). 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* · *q* Design äquivalent ist.

```
# Datensatz aus blockweise über UV2 gemittelten Werten
> mDf <- aggregate(DV ~ id + IVbtw, data=dfSPFpqL, FUN=mean)

# einfaktorielle ANOVA -> äquivalent zum Test UV1 im SPF-p.q Design
> summary(aov(DV ~ IVbtw, 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 der UV1 gebildeten theoretischen Kovarianzmatrizen der UV2 müssen identisch und zudem zirkulär sein (Abschn. 7.5.2). Da im Gegensatz zum RBF-*pq* Design beide Tests auf derselben Residual-Quadratsumme basieren, gibt es nur eine für beide Tests gültige Korrektur der Freiheitsgrade auf Basis der Schätzung $\hat{\epsilon}$ nach Greenhouse und Geisser bzw. nach Huynh und Feldt.

Für die manuelle Berechnung von $\hat{\epsilon}$ ist es hier notwendig, zunächst explizit die Interaktion des Blockeffekts mit dem Zwischen-Gruppen Faktor zu berechnen, da sie die Residual-Quadratsumme in den Tests des Intra-Gruppen Faktors und der Interaktion von Zwischen- und Intra-Gruppen Faktor im Nenner des *F*-Bruchs liefert.

```
# ersetze Zielgröße durch zugehörige Mittelwerte
> Mjk <- with(dfSPFpqL, ave(DV, IVbtw, IVwth, FUN=mean)) # Zellen
> Mi <- with(dfSPFpqL, ave(DV, id, FUN=mean)) # Blöcke
> Mj <- with(dfSPFpqL, ave(DV, IVbtw, FUN=mean)) # UV1-Stufen

# Interaktion Block:UV2 - Residuen bei Test UV2, Interaktion UV1:UV2
> IDxIV <- dfSPFpqL$DV - Mi - Mjk + Mj
> sum(IDxIV^2) # Kontrolle: QS Residuen
[1] 41.05164

# Interaktion dfSPFpqL anfügen, als Matrix im Wide-Format extrahieren
> dfSPFpqL <- cbind(dfSPFpqL, IDxIV)
> errMat <- data.matrix(unstack(dfSPFpqL, IDxIV ~ IVwth))

# Schätzung epsilon nach Greenhouse & Geisser auf Basis der Residuen
> Serr <- cov(errMat) # Kovarianzmatrix der Residuen
> (epsGG <- (1 / (Q-1)) * sum(diag(Serr))^2 / sum(Serr^2))
[1] 0.9125422

# Schätzung von epsilon nach Huynh + Feldt
> dfId <- (Nj-1) * P # df Blockeffekt = df Fehler
> (epsHF <- ((dfId+1)*(Q-1)*epsGG - 2) / ((Q-1)*(dfId - (Q-1)*epsGG)))
[1] 0.975224
```

7.8.3 Multivariat formuliert auswerten

Für eine Beschreibung von `Anova()` aus dem `car` Paket s. Abschn. 7.5.3. Als wesentlicher Unterschied zur RB- p Situation ergibt sich im SPF- $p \cdot q$ Design die Änderung des mit `lm()` multivariat spezifizierten Zwischen-Gruppen Designs. In der Modellformel ist nun auf der rechten Seite der \sim der Zwischen-Gruppen Faktor zu nennen.

```
# Datensatz im Wide-Format
> IVbtwW <- factor(rep(LETTERS[1:P], Nj))
> dfSPFpqW <- data.frame(IVbtwW, DV_t1, DV_t2, DV_t3)

# Zwischen-Gruppen Design
> fitSPFpq <- lm(cbind(DV_t1, DV_t2, DV_t3) ~ IVbtwW, data=dfSPFpqW)
> intraSPFpq <- data.frame(IVwth=gl(Q, 1)) # Intra-Gruppen Design
> library(car) # für Anova()
> AnovaSPFpq <- Anova(fitSPFpq, idata=intraSPFpq, idesign=~IVwth)
> summary(AnovaSPFpq, multivariate=FALSE, univariate=TRUE) # ...
```

`Anova()` setzt hier voraus, dass die Anzahl der Freiheitsgrade des Blockeffekts ($(n_j - 1) \cdot p$) bei gleichen Gruppengrößen n_j mindestens so groß wie die des Intra-Gruppen Effekts ($q - 1$) ist. Es müssen also mindestens $\frac{q-1}{p} + 1$ viele Blöcke pro Gruppe vorhanden sein. Ist dies nicht der Fall, kann für die automatische Berechnung der $\hat{\epsilon}$ -Korrekturen auf `anova()` ausgewichen werden (Abschn. 7.5.4).

Hier ist zu beachten, dass sich die mit `X` und `M` definierten Modelle nur auf die Intra-Gruppen Effekte beziehen. Taucht ein Zwischen-Gruppen Faktor in der Formel von `lm()` auf, testet `anova()` seine Interaktion mit dem von `X` zu `M` hinzugenommenen Intra-Gruppen Effekt, wobei die Quadratsumme des umfassenderen Intra-Gruppen Modells die Rolle der Residual-Quadratsumme einnimmt.

Die Quadratsumme des Zwischen-Gruppen Effekts wird im SPF- $p \cdot q$ Design gegen jene des Blockeffekts getestet. Das mit `M` definierte umfassendere Intra-Gruppen Modell muss also jenes mit nur diesem Effekt sein (~ 1). Mit `X` ist entsprechend das eingeschränkte Modell ohne jeden Effekt zu definieren (~ 0). Da von `X` zu `M` kein Intra-Gruppen Effekt hinzu kommt, bleibt es beim Test der Quadratsumme des Zwischen-Gruppen Faktors gegen jene des umfassenderen Intra-Gruppen Modells.

```
# Test des Zwischen-Gruppen Faktors IVbtw
> anova(fitSPFpq, M=~1, X=~0, idata=intraSPFpq, test="Spherical") # ...
```

Der Intra-Gruppen Faktor lässt sich testen, indem er beim Modellwechsel von `X` zu `M` hinzugenommen wird. Die Kennwerte dieses Effekts stehen in der Ausgabe in der Zeile (Intercept). Zusätzlich testet `anova()` wie erwähnt die Quadratsumme der Interaktion des Zwischen-Gruppen Faktors mit diesem hinzugenommenen Effekt gegen jene des vollständigen Intra-Gruppen-Modells. Für `mauchly.test()` s. Abschn. 7.5.4.

```
# Test IVwth (Zeile Intercept), Interaktion IVbtw:IVwth (Zeile IVbtw)..
> anova(fitSPFpq, M=~IVwth, X=~1, idata=intraSPFpq, test="Spherical")
> mauchly.test(fitSPFpq, M=~IVwth, X=~1, idata=inSPFpq) # ...
```

7.8.4 Einzelvergleiche (Kontraste)

Im SPF- $p \cdot q$ Design lassen sich beliebige Einzelvergleiche als Tests von Linearkombinationen von Erwartungswerten analog zu jenen im CRF- pq Design testen (Abschn. 7.6.4). Auch hier sind Vergleiche der mittleren Erwartungswerte des Zwischen-Gruppen Faktors (A -Kontraste) von solchen des Intra-Gruppen Faktors (B -Kontraste) und von Interaktionskontrasten (I -Kontraste) zu unterscheiden.

Im Vergleich zum CRF- pq Design ergeben sich jedoch Unterschiede bei der jeweils zu verwendenden Quadratsumme der Residuen, die dort immer dieselbe ist. Die Effekt-Quadratsumme eines A -Kontrasts ist im SPF- $p \cdot q$ Design gegen die Quadratsumme des Blockeffekts zu testen. Alternativ lassen sich A -Kontraste wie Kontraste im CR- p Design testen, nachdem zu blockweise gemittelten Daten übergegangen wurde. B - sowie Interaktionskontraste sind gegen die Quadratsumme der Interaktion vom Blockeffekt und Intra-Gruppen Faktor mit den zugehörigen Freiheitsgraden $(n_j - p) \cdot (q - 1)$ zu testen (mit gleichen Gruppengrößen n_j). Allerdings stellt sich hier die Frage, ob sich auf Stufen des Intra-Gruppen Faktors beziehende Einzelvergleiche sinnvollerweise durchgeführt werden sollten (Abschn. 7.5.5).

Im Beispiel soll der A -Kontrast $-0.5(\mu_1 + \mu_2) + \mu_3$ im zugehörigen CR- p Design nach blockweiser Mittelung gerichtet getestet werden (Abschn. 7.4.6).

```
> aovRes <- aov(DV ~ IVbtw, data=mDf)
> cntrMat <- rbind("-0.5*(A+B)+C"=c(-1/2, -1/2, 1))      # Kontrastmatrix
> library(multcomp)                                         # für glht()
> summary(glht(aovRes, linfct=mcp(IVbtw=cntrMat),
+                alternative="greater"))
Simultaneous Tests for General Linear Hypotheses
Multiple Comparisons of Means: User-defined Contrasts
Fit: aov(formula = DV ~ IVbtw, data = mDf)
Linear Hypotheses:
Estimate Std. Error t value Pr(>t)
-0.5*(A+B)+C <= 0    0.2480     0.2369    1.047   0.152

# manuelle Kontrolle
> P   <- nlevels(mDf$IVbtw)                                # Anzahl Gruppen
> Nj  <- table(mDf$IVbtw)                                    # Gruppengrößen
> Mj  <- tapply(mDf$DV, mDf$IVbtw, mean)                  # Gruppenmittel
> SSw <- sum((mDf$DV - ave(mDf$DV, mDf$IVbtw, FUN=mean))^2) # QS within
> MSw <- SSw / (sum(Nj) - P)                                # mittlere QS within
> (psiHat <- sum(cntrMat[1, ] * Mj))                      # Kontrastschätzung
[1] 0.248

> lenSq <- sum(cntrMat[1, ]^2 / Nj)                         # quadrierte Länge
> (tStat <- psiHat / sqrt(lenSq*MSw))                      # Teststatistik t
[1] 1.04672

> (pVal <- pt(tStat, sum(Nj) - P), lower.tail=FALSE) # p-Wert
[1] 0.1522546
```

7.8.5 Erweiterung auf dreifaktorielles SPF- $p \cdot qr$ Design

Univariat formulierte Auswertung

Die varianzanalytische Auswertung eines dreifaktoriellen Designs mit einem Zwischen-Gruppen Faktor 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 Modellformel mit den zu berücksichtigenden Effekten anzupassen.

```
> Nj <- 10                                # Gruppengröße
> P  <- 2                                  # Stufen Zwischen-UV
> Q  <- 3                                  # Zeitpunkte Intra-UV1
> R  <- 2                                  # Zeitpunkte Intra-UV2
> id <- factor(rep(1:(P*Nj), times=Q*R))    # Blockzugehörigkeit
> IVbtw <- factor(rep(LETTERS[1:P], times=Q*R*Nj)) # Zw.-UV: between
> IVwth1 <- factor(rep(1:Q, each=P*R*Nj))      # Intra-UV1: within
> IVwth2 <- factor(rep(rep(1:R, each=P*Nj), times=Q)) # In.-UV2: within

# Simulation: kein Effekt IVbtw, Effekte IVwth1, IVwth2, keine Interakt.
> DV_t11 <- round(rnorm(P*Nj, 8, 2), 2)       # AV zu t1-1
> DV_t21 <- round(rnorm(P*Nj, 13, 2), 2)       # AV zu t2-1
> DV_t31 <- round(rnorm(P*Nj, 13, 2), 2)       # AV zu t3-1
> DV_t12 <- round(rnorm(P*Nj, 10, 2), 2)       # AV zu t1-2
> DV_t22 <- round(rnorm(P*Nj, 15, 2), 2)       # AV zu t2-2
> DV_t32 <- round(rnorm(P*Nj, 15, 2), 2)       # AV zu t3-2
> DV     <- c(DV_t11, DV_t12, DV_t21, DV_t22, DV_t31, DV_t32)

# Datensatz im Long-Format und Auswertung mit aov() ...
> dfSPFp.qrL <- data.frame(id, IVbtw, IVwth1, IVwth2, DV)
> summary(aov(DV ~ IVbtw*IVwth1*IVwth2 + Error(id/(IVwth1*IVwth2)),
+               data=dfSPFp.qrL))                  # ...
```

Als Maß für die Stärke jedes Effekts dient wie im SPF- $p \cdot q$ Design das generalisierte η_g^2 (Abschn. 7.8), das hier analog mit `eta_squared()` aus dem Paket `effectsize` geschätzt wird.

Multivariat formulierte Auswertung

Wird `Anova()` aus dem `car` Paket eingesetzt, ist zunächst zu Daten im Wide-Format überzugehen. Zudem sind Zwischen-Gruppen Design und Intra-Gruppen Struktur zu benennen.

```
# Datensatz im Wide-Format
> IVbtwW    <- factor(rep(LETTERS[1:P], Nj))        # Zwischen-UV
> dfSPFp.qrW <- data.frame(IVbtwW, DV_t11, DV_t21, DV_t31, DV_t12,
+                           DV_t22, DV_t32)

# Zwischen-Gruppen Design
> fitSPFp.qr <- lm(cbind(DV_t11, DV_t21, DV_t31, DV_t12, DV_t22,
```

```

+ DV_t32) ~ IVbtwW, data=dfSPFp.qrW)

# Intra-Gruppen Design
> inSPFp.qr <- expand.grid(IVwth1=gl(Q, 1), IVwth2=gl(R, 1))
> library(car) # für Anova()
> AnovaSPFp.qr <- Anova(fitSPFp.qr, idata=inSPFp.qr,
+ idesign=~IVwth1*IVwth2)

> summary(AnovaSPFp.qr, multivariate=FALSE, univariate=TRUE) # ...

Anova() setzt hier voraus, dass die Anzahl der Freiheitsgrade des Blockeffekts  $((n_j - 1) \cdot p$  bei gleichen Gruppengrößen  $n_j$ ) mindestens so groß wie die der Interaktion beider Intra-Gruppen Effekte  $((q - 1) \cdot (r - 1))$  ist. Es müssen also mindestens  $\frac{(q-1) \cdot (r-1)}{p} + 1$  viele Blöcke pro Gruppe vorhanden sein. Ist dies nicht der Fall, kann für die automatische Berechnung der  $\hat{\epsilon}$ -Korrekturen auf anova() ausgewichen werden (Abschn. 7.5.4, 7.8.3).

# Test Zwischen-Gruppen Faktor IVbtw ...
> anova(fitSPFp.qr, M=~1, X=~0, idata=inSPFp.qr, test="Spherical") # ...

# Test Intra-Gruppen Faktor IVwth1, Interaktion IVbtw:IVwth1 ...
> anova(fitSPFp.qr, M=~IVwth1, X=~1, idata=inSPFp.qr, test="Spherical")

# Test Intra-Gruppen Faktor IVwth2, Interaktion IVbtw:IVwth2 ...
> anova(fitSPFp.qr, M=~IVwth1 + IVwth2, X=~IVwth1,
+ idata=inSPFp.qr, test="Spherical") # ...

# Test Interaktionen IVwth1:IVwth2, IVbtw:IVwth1:IVwth2
> anova(fitSPFp.qr, M=~IVwth1 + IVwth2 + IVwth1:IVwth2,
+ X=~IVwth1 + IVwth2,
idata=inSPFp.qr, test="Spherical") # ...

# R=2 -> Mauchly-Test IVwth2 hier unnötig
> mauchly.test(fitSPFp.qr, M=~IVwth1, X=~1, idata=inSPFp.qr) # ...
> mauchly.test(fitSPFp.qr, M=~IVwth1 + IVwth2 + IVwth1:IVwth2,
+ X=~IVwth1 + IVwth2, idata=inSPFp.qr) # ...

```

7.8.6 Erweiterung auf dreifaktorielles SPF- $pq \cdot r$ Design

Univariat formulierte Auswertung

Die varianzanalytische Auswertung eines dreifaktoriellen Designs mit zwei Zwischen-Gruppen Faktoren und einem Intra-Gruppen Faktor (SPF- $pq \cdot r$) mit `aov()` bringt ebenfalls eine etwas komplexere Datenstruktur mit sich und macht eine Anpassung der Modellformel notwendig.

```

> Njk     <- 10                      # Gruppengröße
> P       <- 2                       # Stufen Zwischen-UV1
> Q       <- 2                       # Stufen Zwischen-UV2

```

```
> R      <- 3                                # Zeitpunkte Intra-UV
> id     <- factor(rep(1:(P*Q*Njk), times=R))    # Blockzugehörigkeit
> IVbtw1 <- factor(rep(1:P, times=Q*R*Njk))      # Zwischen-UV1: between
> IVbtw2 <- factor(rep(rep(1:Q, each=P*Njk), times=R))  # Zwischen-UV2
> IVwth  <- factor(rep(1:R, each=P*Q*Njk))        # Intra-UV: within

# Simulation: kein Effekt IVbtw1, IVbtw2, Effekt IVwth, keine Interakt.
> DV_t1 <- round(rnorm(P*Q*Njk, -3, 2), 2)       # AV zu t1
> DV_t2 <- round(rnorm(P*Q*Njk, 1, 2), 2)         # AV zu t2
> DV_t3 <- round(rnorm(P*Q*Njk, 2, 2), 2)         # AV zu t3
> DV     <- c(DV_t1, DV_t2, DV_t3)                 # Gesamtdaten

# Datensatz im Long-Format und Auswertung mit aov() ...
> dfSPFpq.rL <- data.frame(id, IVbtw1, IVbtw2, IVwth, DV)
> summary(aov(DV ~ IVbtw1*IVbtw2*IVwth + Error(id/IVwth),
+               data=dfSPFpq.rL))
```

Als Maß für die Stärke jedes Effekts dient wie im SPF- $p \cdot q$ Design das generalisierte η_g^2 (Abschn. 7.8), das hier analog mit `eta_squared()` aus dem Paket `effectsize` geschätzt wird.

Multivariat formulierte Auswertung

Wird `Anova()` aus dem `car` Paket eingesetzt, ist zunächst zu Daten im Wide-Format überzugehen. Zudem sind Zwischen-Gruppen Design und Intra-Gruppen Struktur zu benennen.

```
# Datensatz im Wide-Format
> IVbtw1W    <- factor(rep(LETTERS[1:P], times=Q*Njk))
> IVbtw2W    <- factor(rep(c("+", "-"), each=P*Njk))
> dfSPFpq.rW <- data.frame(IV1W, IV2W, DV_t1, DV_t2, DV_t3)

# Zwischen-Gruppen Design
> fitSPFpq.r <- lm(cbind(DV_t1,DV_t2,DV_t3) ~ IVbtw1W*IVbtw2W,
+                     data=dfSPFpq.rW)

> inSPFpq.r  <- data.frame(IVwth=gl(R, 1))          # Intra-Gruppen Design
> library(car)                                         # für Anova()
> AnovaSPFpq.r <- Anova(fitSPFpq.r, idata=inSPFpq.r, idesign=~IVwth)
> summary(AnovaSPFpq.r, multivariate=FALSE, univariate=TRUE) # ...
```

`Anova()` setzt hier voraus, dass die Anzahl der Freiheitsgrade des Blockeffekts ($(n_j - 1) \cdot p \cdot q$ bei gleichen Gruppengrößen n_j) mindestens so groß wie die des Intra-Gruppen Effekts ($r - 1$) ist. Es müssen also mindestens $\frac{r-1}{p \cdot q} + 1$ viele Blöcke pro Gruppe vorhanden sein. Ist dies nicht der Fall, kann für die automatische Berechnung der $\hat{\epsilon}$ -Korrekturen auf `anova()` ausgewichen werden (Abschn. 7.5.4, 7.8.3).

```
# Test IVbtw1, IVbtw2, IVbtw1:IVbtw2
> anova(fitSPFpq.r, M=~1, X=~0, idata=inSPFpq.r, test="Spherical") # ..
```

```
# Test IVwth, IVbtw1:IVwth, IVbtw2:IVwth, IVbtw1:IVbtw2:IVwth ...
> anova(fitSPFpq.r, M=~IVwth, X=~1, idata=inSPFpq.r, test="Spherical")
> mauchly.test(fitSPFpq.r, M=~IVwth, X=~1, idata=inSPFpq.r)      # ...
```

7.9 Kovarianzanalyse

Bei der Kovarianzanalyse (ANCOVA) werden die Daten einer Zielgröße (AV) 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 $\langle AV \rangle \sim \langle UV \rangle$ Modellformel in der Rolle von $\langle UV \rangle$ sowohl quantitative Variablen wie Faktoren als Vorhersageterme auftauchen.

7.9.1 Test der Effekte von Gruppenzugehörigkeit und Kovariate

Im Folgenden sei die Situation mit einem quantitativen und einem kategorialen Prädiktor vorausgesetzt. Aus Perspektive der Regression kann die Kovarianzanalyse als Prüfung betrachtet werden, ob ein Zusammenhang zwischen dem kontinuierlichen Prädiktor und der Zielgröße über unterschiedliche Gruppen hinweg besteht.

Aus Perspektive der Varianzanalyse ist man an der Wirkung des Gruppierungsfaktors interessiert, hält jedoch den Einfluss einer quantitativen Störvariable für möglich, der innerhalb jeder Gruppe als linear angenommen wird. Da der Einfluss der Störvariable die Heterogenität der Werte der Zielgröße innerhalb jeder Gruppe erhöht, scheint es erstrebenswert, die Einflüsse des Faktors und der Kovariate voneinander trennen zu können und so die power des Tests auf Gruppenunterschiede zu erhöhen. Meist wird die Fragestellung dahingehend eingeschränkt, dass in den Gruppen nur unterschiedliche y -Achsenabschnitte der Regressionsgleichung zugelassen sind, die Steigungen dagegen als identisch vorausgesetzt werden. Die Gruppenunterschiede wären dann in den y -Achsenabschnitten der Regressionsgeraden erkennbar.

Als Beispiel diene jenes aus [Maxwell et al. \(2017, p. 483\)](#): 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 SSRI-Wirkstoff, 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
> 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
> P         <- 3                                         # Anzahl Gruppen
```

```
> Nj      <- rep(length(SSRIpre), times=P)           # Gruppengrößen

# Faktor der Gruppenzugehörigkeiten
> IV       <- factor(rep(1:3, Nj), labels=c("SSRI", "Placebo", "WL"))
> DVpre    <- c(SSRIpre, PlacPre, WLpre)          # alle prä-Messungen
> DVpost   <- c(SSRIpost, PlacPost, WLpost)        # alle post-Messungen
> dfAncova <- data.frame(id=1:sum(Nj), IV, DVpre, DVpost) # Datensatz
```

Als Veranschaulichung wird die Verteilung der Vorher- und Nachher-Werte in den Gruppen durch Boxplots dargestellt.

```
> plot(DVpre ~ IV, main="Prä-Scores je Gruppe")      # Boxplot Prä
> plot(DVpost ~ IV, main="Post-Scores je Gruppe")    # Boxplot Post
```

Anschließend führt `anova()` die Varianzanalyse zunächst ohne, dann die Kovarianzanalyse mit Kovariate durch (Abb. 7.4, Abschn. 14.6.3). Dabei sei vorausgesetzt, dass der Steigungsparameter in allen Gruppen identisch ist, das Modell berücksichtigt also keinen Interaktionsterm von Treatment-Variable und Kovariate. Modelle dieser Art lassen sich sehr flexibel und anschaulich mit dem Paket `emmeans` visualisieren (Abschn. 7.6.1, Fußnote 11).

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.

```
> fitFull <- lm(DVpost ~ IV + DVpre, data=dfAncova) # vollst. Modell
> fitGrp  <- lm(DVpost ~ IV,             data=dfAncova) # ohne Kovariate
> fitRegr <- lm(DVpost ~      DVpre, data=dfAncova) # ohne Gruppe
> anova(fitGrp)                                     # 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
```

```
# 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 `Anova()` aus dem `car` Paket zurückgegriffen werden.¹⁸ Zum anderen lassen sich diese Quadratsummen mit `anova()` durch den Test zweier geeigneter Modelle gegeneinander ermitteln (Abschn. 7.6.2): Für den Effekt der Kovariate sind dies auf der einen Seite das Modell ohne Kovariate als Vorhersage-
term, auf der anderen Seite das vollständige Modell. Für den Effekt des Gruppierungsfaktors

¹⁸Da keine Interaktion von Gruppierungsfaktor und Kovariate berücksichtigt wird, sind die Quadratsummen vom Typ II und III hier identisch.

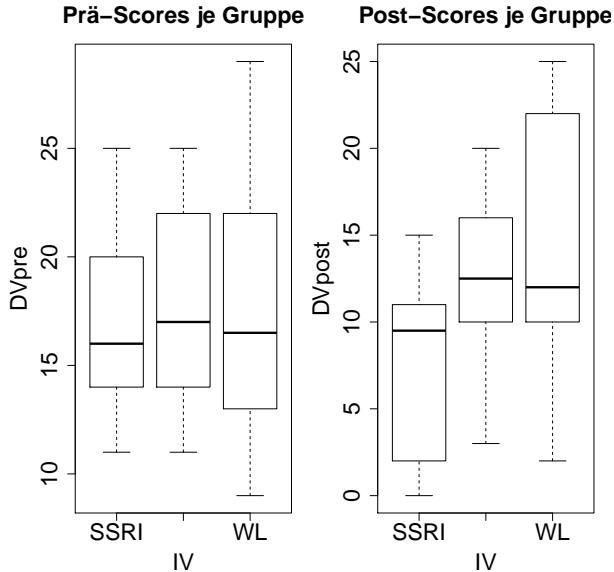


Abbildung 7.4: Kovarianzanalyse: Boxplots zum Vergleich der Verteilungen der Zielgröße in den Gruppen

entsprechend auf der einen Seite das Modell ohne Gruppierungsfaktor als Vorhersageterm, auf der anderen Seite das vollständige Modell.

```
# Effektcodierung für Quadratsummen vom Typ III
> fitIII <- lm(DVpost ~ IV + DVpre,
+                 contrasts=list(IV=contr.sum), data=dfAncova)

> library(car)                                     # für Anova()
> Anova(fitIII, type="III")                         # QS Typ III
Anova Table (Type II 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 II (hier = III) über Modellvergleiche
> anova(fitRegr, fitFull)                         # Test Treatment ...
> anova(fitGrp,  fitFull)                         # Test Kovariate ...
```

Die Ergebnisse lassen sich manuell nachvollziehen, indem die Residual-Quadratsummen für das vollständige Modell, das Regressionsmodell ohne Gruppierungsfaktor und das ANOVA-Modell ohne Kovariate berechnet werden. Die Effekt-Quadratsummen vom Typ III ergeben sich dann jeweils als Differenz der Residual-Quadratsumme des Modells, in dem dieser Effekt nicht berücksichtigt wird und der Residual-Quadratsumme des vollständigen Modells.

```
> X <- DVpre                                         # kürzerer Name Kovariate
```

```

> Y      <- DVpost                                # kürzerer Name AV
> XMj   <- tapply(X, IV, mean)                  # Gruppenmittel Kovariate
> YMj   <- tapply(Y, IV, mean)                  # Gruppenmittel AV
> N     <- length(Y)                            # Gesamtanzahl Personen

# Residual-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   <- YMj - bFull*XMj                   # y-Achsenabschnitte
> YhatFull <- bFull*X + aFull[IV]              # Vorhersage
> SSEfull  <- sum((Y-YhatFull)^2)             # Residual-QS
> dfSSEfull <- N-P-1                           # Freiheitsgrade Residual-QS
> MSEfull  <- SSEfull / dfSSEfull            # mittlere Residual-QS

# Residual-Quadratsumme Regressionsmodell ohne Treatment-Variablen
> 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)             # Residual-QS
> dfSSEregr <- N-2                            # df Fehler
> MSEregr  <- SSEregr / dfSSEregr            # mittlere Residual-QS

# Residual-Quadratsumme ANOVA-Modell ohne Kovariate
> Vj      <- tapply(Y, IV, var)                # Gruppenvarianzen
> M       <- sum((Nj/N) * YMj)                 # Gesamt-Mittel
> SSEgrp  <- sum((Nj-1) * Vj)                 # Residual-QS
> dfSSEgrp <- N-P                          # df Fehler
> MSEgrp  <- SSEgrp / dfSSEgrp            # mittlere Residual-QS

# Effekt-Quadratsummen und F-Werte der zugehörigen Tests
> SSregr <- SSEgrp - SSEfull                # Effekt-QS Kovariate
> dfRegr <- dfSSEgrp - dfSSEfull           # df Kovariate -> 1
> MSregr <- SSregr / dfRegr                 # MS Kovariate
> (Fregr <- MSregr / MSEfull)               # F-Wert Kovariate
[1] 10.77234

> SSgrp <- SSEregr - SSEfull                # Effekt-QS Treatment
> dfGrp <- dfSSEregr - dfSSEfull           # df Treatment -> P-1
> MSgrp <- SSgrp / dfGrp                   # MS Treatment
> (Fgrp <- MSgrp / MSEfull)                # F-Wert Treatment
[1] 3.732399

```

Mit `summary(`*lm*-Modell`)` lassen sich die in den verschiedenen Gruppen angepassten Regressionsparameter ausgeben und einzeln auf Signifikanz testen.

```
> (sumRes <- summary(fitFull)) # gekürzte Ausgabe ...
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 (Abschn. 2.6.5, 12.9.2). 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 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.¹⁹

Eine grafische Veranschaulichung des linearen Zusammenhangs zwischen Vorher- und Nachher-Messwert in den einzelnen Gruppen erfolgt in Abb. 7.5.

```
# Steigung und y-Achsenabschnitte der Regressionsgeraden extrahieren
> coeffs <- coef(sumRes) # alle Koeffizienten
> iCptSSRI <- coeffs[1, 1] # b0 SSRI (Referenzgruppe)
> iCptPlac <- coeffs[2, 1] + iCptSSRI # b0 Placebo
> iCptWL <- coeffs[3, 1] + iCptSSRI # b0 Warteliste
> slopeAll <- coeffs[4, 1] # gemeinsame Steigung

# Darstellungsbereich für x- & y-Achse wählen und Punktwolke darstellen
> xLims <- c(0, max(dfAncova$DVpre))
> yLims <- c(min(c(iCptSSRI, iCptPlac, iCptWL)),
+           max(dfAncova$DVpost))

> plot(DVpost ~ DVpre, data=dfAncova, xlim=xLims, ylim=yLims,
+       pch=rep(c(3,17,19), Nj), col=rep(c("red", "green", "blue"), Nj),
+       main="Rohdaten und Regressionsgerade pro Gruppe")
```

¹⁹Die absolute Höhe der Gruppe der *y*-Achsenabschnitte lässt sich aus den Daten nicht unabhängig schätzen, während ihre Abstände untereinander eindeutig bestimmt sind. Die in Maxwell et al. (2017, Kap. 9) gezeigte Lösung fixiert den *y*-Achsenabschnitt der **WL** Gruppe auf 0 und berichtet die übrigen als Differenz dazu.

```
> legend(x="topleft", legend=levels(IV), 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")
```

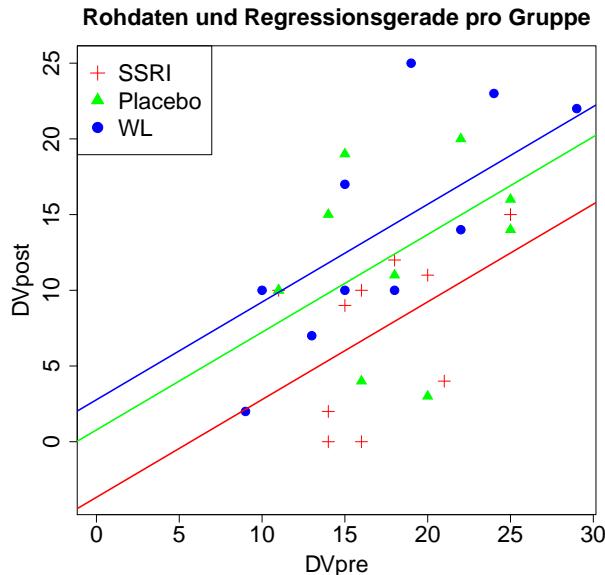


Abbildung 7.5: Kovarianzanalyse: nach Gruppen getrennte Regressionsgeraden

Als Maß für die Stärke jedes getesteten Effekts kann das partielle η_p^2 herangezogen werden, zu dessen Schätzung $\hat{\eta}_p^2$ jeweils seine Effekt-Quadratsumme an der Summe von ihr und der Residual-Quadratsumme relativiert wird. $\hat{\eta}_p^2$ der Kovariate ist hier gleich der quadrierten Partialkorrelation von Kovariate und Kriterium ohne den Gruppierungsfaktor (Abschn. 6.7).

```
> SSEfull <- deviance(fitFull)           # Fehler-Quadratsumme full model
> SSEgrp  <- deviance(fitGrp)           # Fehler-Quadratsumme Treatment
> SSEregr <- deviance(fitRegr)           # Fehler-Quadratsumme Kovariate
> SSregr  <- SSEgrp - SSEfull           # Effekt-Quadratsumme Kovariate
> SSgrp   <- SSEregr - SSEfull           # Effekt-Quadratsumme Treatment
> SSregr / (SSregr + SSEfull)            # partielle eta^2 Kovariate
[1] 0.2929469

# Kontrolle: Partialkorrelation Kovariate mit Kriterium ohne Faktor
> cor(residuals(lm(DVpre ~ IV)), residuals(lm(DVpost ~ IV)))^2
[1] 0.2929469

> SSgrp / (SSgrp + SSEfull)              # partielle eta^2 Treatment
[1] 0.2230642
```

Sollen in der Kovarianzanalyse die Steigungen der Regressionsgeraden in den Gruppen nicht als identisch festgelegt, sondern auch bzgl. dieses Parameters Gruppenunterschiede i. S. einer Moderation (Abschn. 6.3.4) zugelassen werden, lautet das Modell:

```
> summary(lm(DVpost ~ IV + DVpre + IV:DVpre, data=dfAncova))      # ...
```

7.9.2 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 (Abschn. 7.4.6). Die Kovariate findet dabei Berücksichtigung, indem hier der Vergleich zwischen korrigierten Erwartungswerten der Zielgröße stattfindet: Auf empirischer Ebene müssen für deren Schätzung zunächst die Regressionsparameter pro Gruppe bestimmt werden, wobei wie oben das *b*-Gewicht 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 der Zielgröße zu erwarten wäre, wenn alle Personen denselben Wert auf der Kovariate (nämlich deren Gesamtmittelwert) hätten und sich nur in der Gruppenzugehörigkeit unterscheiden würden.

Im Beispiel werden drei Kontraste ohne α -Adjustierung getestet. 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),
+                      "SSRI-0.5(P+WL)" = c(-2, 1, 1))

> library(multcomp)                                # für glht()
> aovAncova <- aov(DVpost ~ IV + DVpre, data=dfAncova)
> (sumRes <- 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 **
SSRI-0.5(P+WL) <= 0 10.890     4.183   2.603  0.00753 **
```

Die korrigierten Gruppenmittel lassen sich mit `emmeans(aov-Modell, ~Faktor)` aus dem gleichnamigen Paket berechnen. In der als zweites Argument zu nennenden Modellformel ist auf der rechten Seite der `~` der Faktor zu nennen, für dessen Gruppen die korrigierten Mittelwerte gebildet werden sollen.

```
> library(emmeans)                                # für emmeans()
> emmeans(aovAncova, ~IV)
IV      emmean   SE df lower.CL upper.CL
SSRI      7.54 1.71 26     4.03    11.0
Placebo  11.98 1.71 26     8.48    15.5
WL       13.98 1.71 26    10.47    17.5
```

Die Ergebnisse für korrigierte Gruppenmittel und Kontraste können manuell geprüft werden (s. Abschn. 12.1.1 für die Matrixmultiplikation).

```
> (YMjAdj <- bFull*mean(X) + aFull)           # korrig. Gruppenmittel
SSRI      Placebo          WL
7.536616 11.984895 13.978489

> psiHats  <- cntrMat    %*% YMjAdj          # Kontrastschätzungen
> lenSqs   <- cntrMat^2  %*% (1/Nj)         # quadrierte Längen
> Xctr     <- X - ave(X, IV, FUN=mean)      # zentrierte Kovariate
> fracs    <- (cntrMat %*% XMj)^2 / sum(Xctr^2)
> varPsiHats <- MSEfull * (lenSqs + fracs)   # Varianz der Schätzungen
> tStats    <- psiHats / sqrt(varPsiHats)    # Teststatistiken
> pVals    <- pt(abs(tStats), dfSSEfull, lower.tail=FALSE) # p-Werte
> data.frame(psiHats, tStats, pVals)
            psiHats      tStats      pVals
SSRI-Placebo  4.448279  1.8411994  0.038515271
SSRI-WL        6.441874  2.6692923  0.006461541
SSRI-0.5(P+WL) 10.890153  2.6031958  0.007529061
```

7.9.3 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 Werts weicht ab und ergibt sich zur α -Adjustierung aus einer *F*-Verteilung. Dieser kritische Wert ist mit dem Quadrat der a-priori *t*-Teststatistik zu vergleichen – die etwa in der von `summary(glht(...))` zurückgegebenen Liste in der Komponente `test$tstat` steht. Hier sollen dieselben Kontraste wie im a-priori Fall gerichtet getestet werden.

```
> dfSSgrp <- P-1                                # Freiheitsgrade Treatment-QS
> Fstats  <- sumRes$test$tstat^2                # quadrierte t-Teststatistiken

# p-Werte einseitig
> (pVals <- pf(Fstats/dfSSgrp, dfSSgrp, dfSSEfull, lower.tail=FALSE))
```

| SSRI-Placebo | SSRI-WL | SSRI-0.5(P+WL) |
|--------------|------------|----------------|
| 0.20326188 | 0.04291472 | 0.04923999 |

7.10 Power, Effektstärke und notwendige Stichprobengröße

Mit Hilfe der Funktionen von Zufallsvariablen (Abschn. 5.3) lässt sich die power der vorgestellten inferenzstatistischen Tests berechnen, sofern eine exakte H_1 vorliegt. Hierfür ist es notwendig, zunächst auf Basis der Verteilung der Teststatistik unter H_0 mit der Quantilfunktion $q(\text{Funktionsfamilie})()$ den kritischen Wert für das gewünschte α -Niveau zu bestimmen. Mit Hilfe der zugehörigen Verteilungsfunktion $p(\text{Funktionsfamilie})()$ kann dann unter Gültigkeit der H_1 die power berechnet werden.

Analog lässt sich auch die notwendige Stichprobengröße (Fallzahl) ermitteln, für die der Test bei einem als gegeben vorausgesetzten Effekt eine gewisse power erreicht. Hierfür bedarf es meist eines Nonzentralitätsparameters der Verteilung der Teststatistik unter H_1 , der anhand der aus der Statistik bekannten Formeln zu berechnen ist und sich aus der theoretischen Effektstärke ergibt.

Die im Basisumfang von R enthaltenen Funktionen zur Bestimmung von power und Stichprobengröße besitzen eine recht eingeschränkte Funktionalität, so berücksichtigen sie nur Binomialtest, *t*-Test und Varianzanalyse. Mehr Möglichkeiten bietet das Paket *pwr* (Champely & De Rosario, 2020). Insbesondere besitzt jedoch das kostenlose Programm G*Power (Faul, Erdfelder, Lang & Buchner, 2007) einen deutlich breiteren Einsatzbereich hinsichtlich der unterstützten Tests. Zudem bietet es vielfältige Möglichkeiten zur Visualisierung der Zusammenhänge von Effektstärke, power und Fallzahl.

7.10.1 Binomialtest

Im Beispiel soll zunächst der Fall eines rechtsseitigen Binomialtests betrachtet werden (Abschn. 10.1.1). Die Punktwahrscheinlichkeiten der einzelnen Ereignisse bei Gültigkeit von H_0 und H_1 sind zusammen mit dem kritischen Wert in Abb. 7.6 dargestellt.

```
> N      <- 7                      # Stichprobengröße
> pH0    <- 0.25                  # Wahrscheinlichkeit Treffer unter H0
> pH1    <- 0.7                   # Wahrscheinlichkeit Treffer unter H1
> alpha   <- 0.05                 # Signifikanzniveau
> (critB <- qbinom(alpha, N, pH0, lower.tail=FALSE)) # krit. Wert
[1] 4

# power für konkrete H0, H1, N
> (powB <- pbinom(critB, N, pH1, lower.tail=FALSE))
[1] 0.6470695

> sum(dbinom((critB+1):N, N, pH1))    # Kontrolle: Summe Einzelwkt.
[1] 0.6470695
```

```
# Säulendiagramm: 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="Binomialvert. unter H0 und H1 (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, labels="kritischer Wert")
```

Beim Test mit der diskreten Binomialverteilung 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 Werts, die zusammen mit der Powerfunktion in Abb. 7.6 abgebildet ist.

```
> Nvec <- 2:15 # betrachteter Bereich für N

# kritische Werte
> critBvec <- qbinom(alpha, size=Nvec, prob=pH0, lower.tail=FALSE)

# power
> powBvec <- pbinom(critBvec, size=Nvec, prob=pH1, lower.tail=FALSE)

# veranschauliche Verlauf der kritischen Werte
> 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="Power und kritischer Wert",
+ 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)
```

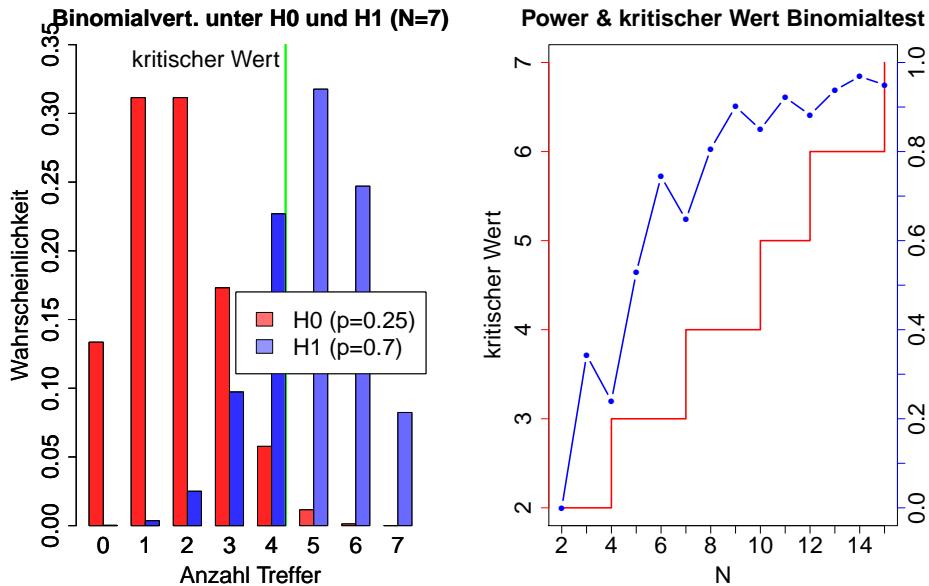


Abbildung 7.6: Binomialverteilung: Wahrscheinlichkeiten von Treffern unter H_0 und H_1 sowie kritischer Wert. Powerfunktion und kritischer Wert in Abhängigkeit von der Stichprobengröße

7.10.2 t-Test

Für den t -Test mit einer Stichprobe ist zur Bestimmung der Verteilung der Teststatistik unter H_1 die Berechnung des Nonzentralitätsparameters δ erforderlich, für den die theoretische Streuung sowie der Erwartungswert unter H_0 und H_1 bekannt sein muss.²⁰ Abbildung 7.7 zeigt die Verteilungen von t für das gegebene Hypothesenpaar und kennzeichnet die Flächen, deren Größe α , β und power bei einem rechtsseitigen Test entsprechen.

```
> N      <- 10                                # Stichprobengröße
> muH0   <- 0                                 # Erwartungswert unter H0
> muH1   <- 1.6                               # Erwartungswert unter H1
> alpha   <- 0.05                             # Signifikanzniveau
> sigma   <- 2                                 # theoretische Streuung
> (d      <- (muH1-muH0) / sigma)            # Effektstärke d
[1] 0.8

> (delta <- (muH1-muH0) / (sigma/sqrt(N)))    # NZP, oder: sqrt(N)*d
[1] 2.529822
```

²⁰Für zwei unabhängige Stichproben mit Gruppengrößen n_1 und n_2 , Streuung σ und Erwartungswerten μ_1 und μ_2 unter H_1 ist $\delta = \sqrt{\frac{n_1 n_2}{n_1 + n_2}} \frac{\mu_2 - \mu_1}{\sigma}$. Für zwei abhängige Stichproben des jeweiligen Umfangs n mit theoretischen Streuungen σ_1 und σ_2 sowie der theoretischen Korrelation ρ ist $\delta = \sqrt{n} \frac{\mu_2 - \mu_1}{\sqrt{\sigma_1^2 + \sigma_2^2 + 2\rho\sigma_1\sigma_2}}$.

```

> (tCrit <- qt(1-alpha, N-1, lower.tail=FALSE))      # kritischer t-Wert
[1] 1.833113

> (powT <- pt(tCrit, N-1, delta, lower.tail=FALSE)) # power
[1] 0.7544248

# bestimme Werte der t-Verteilungen
> xLims <- c(-5, 10)
> tLeft <- seq(xLims[1], tCrit, length.out=100)
> tRight <- seq(tCrit, xLims[2], length.out=100)
> yH0r <- dt(tRight, N-1, 0)
> yH1l <- dt(tLeft, N-1, delta)
> yH1r <- dt(tRight, N-1, delta)

# markiere Flächen für alpha, beta und power
> curve(dt(x, N-1, 0), xlim=xLims, lwd=2, col="red", xlab="t",
+        ylab="Dichte", main="Verteilung von t unter H0 und H1",
+        ylim=c(0, 0.4), xaxs="i")

> curve(dt(x, N-1, delta), lwd=2, col="blue", add=TRUE)
> 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, lwd=2, angle=45, col="darkgray")

# zusätzliche Beschriftungen
> abline(v=tCrit, lty=1, lwd=3, col="red")
> text(tCrit+0.2, 0.4, adj=0, labels="kritischer Wert")
> text(tCrit-2.8, 0.3, adj=1, labels="Verteilung unter H0")
> text(tCrit+1.5, 0.3, adj=0, labels="Verteilung unter H1")
> text(tCrit+1.0, 0.08, adj=0, labels="Power")
> text(tCrit-0.7, 0.05, expression(beta))
> text(tCrit+0.5, 0.015, expression(alpha))

```

Wie für Binomial- und *t*-Tests demonstriert, kann die power analog für viele andere Tests manuell ermittelt 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.²¹

²¹Die Funktionen sind nicht vektorisiert, akzeptieren für jedes Argument also nur jeweils einen Wert.

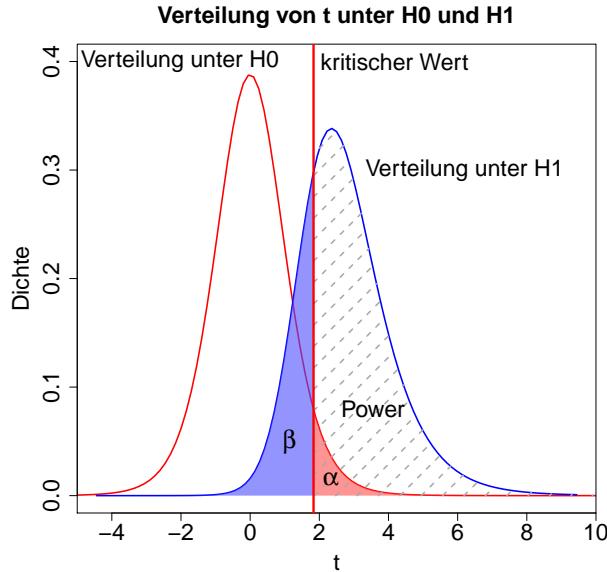


Abbildung 7.7: Rechtsseitiger *t*-Test für eine Stichprobe: Verteilung von *t* unter H_0 und H_1 , kritischer Wert, α , β und power

```
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 das α -Niveau 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 Gruppe – es werden also auch bei unabhängigen Stichproben gleiche Gruppengrößen vorausgesetzt.²²

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.

²²Für 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 $\text{sqrt}(n/2) * (\text{delta}/\text{sd})$ korrekt ist. Statt mit der richtigen Zahl der Freiheitsgrade n_1+n_2-2 rechnet **power.t.test()** dann aber mit $4 \frac{n_1 n_2}{n_1+n_2} - 2$. Der so entstehende Fehler wächst zwar mit der Differenz von n_1 und n_2 , bleibt jedoch absolut gesehen gering.

Im Beispiel soll für die oben gegebene Situation herausgefunden werden, wie viele Beobachtungsobjekte 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, alle anderen Argumente sind zu spezifizieren.

7.10.3 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)
```

Von den Argumenten `groups` für die Anzahl der Gruppen, `n` für die Gruppengröße, `between.var` für die Varianz der Erwartungswerte unter H_1 ,²³ `within.var` für die theoretische Fehlervarianz, `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.

```
> muJ <- c(100, 110, 115)                      # Erwartungswerte unter H1
> sigma <- 15                                     # theoretische Streuung
> power.anova.test(groups=3, n=NULL, between.var=var(muJ),
+                     within.var=sigma^2, sig.level=0.05, power=0.9)
Balanced one-way analysis of variance power calculation
groups = 3
n = 25.4322
between.var = 58.33333
within.var = 225
sig.level = 0.05
```

²³Enthält der Vektor `muJ` 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(muJ)`, für das Argument `between.var` übergeben werden.

```
power = 0.9
```

NOTE: n is number in each group

Für das gegebene Beispiel folgt die manuelle Berechnung der power und der Maße für die Effektstärke einer Varianzanalyse im CR-*p* Design mit ungleichen Zellbesetzungen.

```
> P      <- length(muJ)           # Anzahl Gruppen
> Nj    <- c(21, 17, 19)         # Gruppengrößen
> N      <- sum(Nj)             # Gesamt-N
> mu    <- sum(Nj * muJ) / N     # mittlerer Erwartungswert, gewichtet
> alphaJ <- muJ - mu            # Gruppeneffekte
> varMUj <- sum(Nj * alphaJ^2)/N # Varianz der Erwartungswerte

# Maße für die Effektstärke - Bezeichnungen uneinheitlich
> (fSq <- varMUj / sigma^2)       # f^2
[1] 0.1826887

> (etaSq <- varMUj / (sigma^2 + varMUj))   # eta^2 bzw. omega^2
[1] 0.1544690

# Nonzentralitätsparam. lambda bzw. delta^2 - Bezeichnung uneinheitlich
> (lambda <- sum(Nj * alphaJ^2) / sigma^2) # oder: N * fSq
[1] 10.41326

# kritischer Wert, alpha=0.05
> (Fcrit <- qf(0.05, P-1, N-P, lower.tail=FALSE))
[1] 3.168246

> (powF <- pf(Fcrit, P-1, N-P, lambda, lower.tail=FALSE)) # power
[1] 0.8090387
```

Liegt keine exakte H_1 vor, ist die power als Funktion der Effektstärke f darstellbar (Abb. 7.8).

```
> fVals <- seq(0, 1.2, length.out=100) # Effektstärken f auf x-Achse
> nn     <- seq(10, 25, by=5)      # Stichprobengrößen: separate Kurven

# Funktion: power für verschiedene Stichprobengrößen und f-Werte
> getFpow <- function(n) {
+   Fcrit <- qf(0.05, P-1, P*n - P, lower.tail=FALSE)
+   pf(Fcrit, P-1, P*n - P, P*n * fVals^2, lower.tail=FALSE)
+ }

> powsF <- sapply(nn, getFpow) # Power-Werte für jede Stichprobengröße

# Beschriftungen vorbereiten
> yStr <- "Wahrscheinlichkeit der Annahme von H1"
```

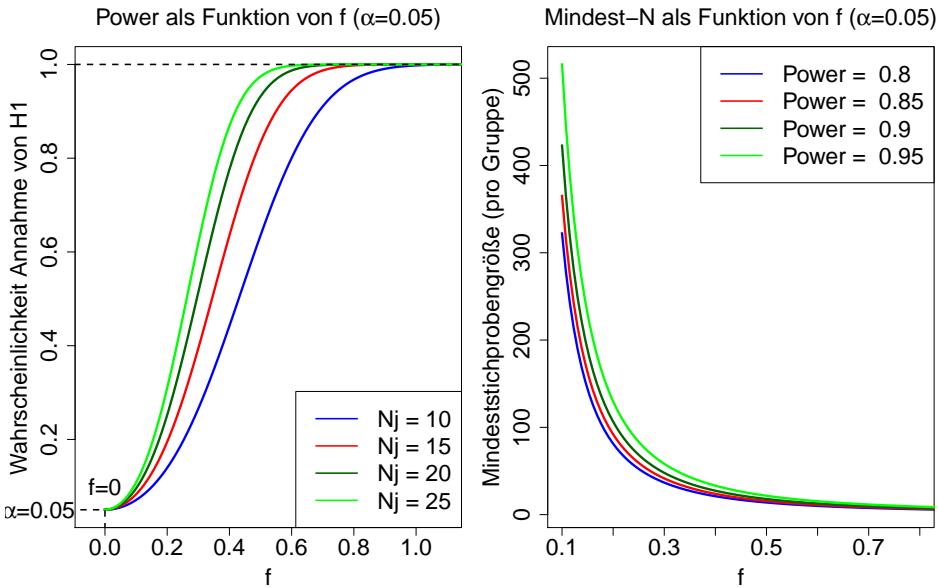


Abbildung 7.8: Einfaktorielle Varianzanalyse: power als Funktion der Effektstärke f für verschiedene Gruppengrößen sowie Mindeststichprobengröße als Funktion von f für verschiedene Power-Werte

```
> mStr <- substitute(paste("Power - Funktion von f (",alpha,"=0.05)"))

# 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=.05, las=1)
> legend(x="bottomright", legend=paste("Nj =", c(10, 15, 20, 25)),
+         lwd=2, col=c("blue", "red", "darkgreen", "green"))
```

Liegt keine exakte H_1 vor, lässt sich auch die Mindeststichprobengröße als Funktion der Effektstärke f darstellen (Abb. 7.8).

```
# Funktion: Mindeststichprobengröße für gegebene power und var.between
> 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
+ }

# Mindeststichprobengröße für mehrere Effektstärken f und Power-Werte
```

```
# 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: eigene Kurven
> minN <- sapply(fVals, getManyFn, pows) # Mindeststichprobengrößen

# Beschriftungen vorbereiten
> yStr <- "Mindeststichprobengröße (pro Gruppe)"
> mStr <- substitute(paste("Mindest-N als Funktion von 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 8

Regressionsmodelle für kategoriale Daten und Zähldaten

Das Modell der linearen Regression und Varianzanalyse (Abschn. 6.3, 7.4, 12.9.1) lässt sich zum verallgemeinerten linearen Modell (GLM, *generalized linear model*) erweitern, das auch für Daten einer kategorialen vorherzusagenden Variable Y geeignet ist.¹ Details erläutern Faraway, 2016; Fox & Weisberg, 2019.

Als Prädiktoren lassen sich sowohl kontinuierliche Variablen als auch Gruppierungsfaktoren einsetzen. Ein Spezialfall ist die logistische Regression für dichotome Y (codiert als 0 und 1). Im Vergleich zur Vorhersage quantitativer Variablen in der linearen Regression wird an diesem Beispiel zunächst folgende Schwierigkeit deutlich: Eine lineare Funktion von p Prädiktoren X_j der Form $\beta_0 + \beta_1 X_1 + \dots + \beta_j X_j + \dots + \beta_p X_p$ kann bei einem unbeschränkten Definitionsbereich beliebige Werte im Intervall $(-\infty, +\infty)$ annehmen. Es können sich durch die Modellgleichung also Werte ergeben, die weder von Y selbst, noch vom Erwartungswert $E(Y)$ angenommen werden können. Dabei ist $E(Y)$ für dichotome Y die Wahrscheinlichkeit eines Treffers $P(Y = 1)$, die im Intervall $[0, 1]$ liegt.

Dagegen ist die lineare Modellierung von $\text{logit}(P) = \ln \frac{P}{1-P}$ möglich, dem natürlichen Logarithmus des Wettquotienten, der Werte im Intervall $(-\infty, +\infty)$ annimmt.² Man setzt nun $\ln \frac{P}{1-P} = \beta_0 + \beta_1 X_1 + \dots + \beta_p X_p$ als gültiges Modell voraus und fasst die rechte Seite der Gleichung als *linearen Prädiktor $\mathbf{X}\boldsymbol{\beta}$* zusammen (Abschn. 12.9.1). Dann ist P mit der Verteilungsfunktion der standardisierten logistischen Verteilung als Umkehrfunktion der Logit-Funktion als $P = \frac{e^{\mathbf{X}\boldsymbol{\beta}}}{1+e^{\mathbf{X}\boldsymbol{\beta}}} = \frac{1}{1+e^{-\mathbf{X}\boldsymbol{\beta}}}$ identifizierbar.³ Solche Transformationen des eigentlich vorherzusagenden Parameters, die eine lineare Modellierung ermöglichen, heißen *Link-Funktion $g(\cdot)$* . Sie müssen u. a. eine Umkehrfunktion $g^{-1}(\cdot)$ besitzen, so dass $g(E(Y)) = \mathbf{X}\boldsymbol{\beta}$ und $E(Y) = g^{-1}(\mathbf{X}\boldsymbol{\beta})$ gilt.

Verkürzt gesprochen wird im GLM anders als im allgemeinen linearen Modell nur $E(Y)$ über die Link-Funktion linear modelliert, nicht Y selbst. Es ist deshalb notwendig, die angenommene Form der bedingten Verteilung von Y für gegebene Prädiktorwerte explizit anzugeben (Abschn. 12.9). Mit dieser Form liegt auch die Varianz der Verteilung in Abhängigkeit von $E(Y)$ fest. Die bedingte Verteilung muss aus der natürlichen Exponentialfamilie stammen und wird mit einer Link-Funktion kombiniert, die $E(Y)$ mit dem linearen Prädiktor $\mathbf{X}\boldsymbol{\beta}$ in Beziehung setzt. Sind mehrere Link-Funktionen mit einer bedingten Verteilungsform kombinierbar, ist eine davon die *kanonische Link-Funktion* mit besonderen statistischen Eigenschaften.

¹ Abschnitt 6.6.4 gibt Hinweise auf gemischte Regressionsmodelle und verallgemeinerte Schätzgleichungen (GEE) für abhängige Daten – etwa durch Messwiederholung oder Clusterung, die analog auf kategoriale Zielgrößen übertragen werden können.

² Diese Funktion ist in `qlogis()` implementiert.

³ Diese Funktion ist in `plogis()` implementiert.

Die lineare Regression ist ein Spezialfall des GLM, bei dem von bedingter Normalverteilung von Y ausgegangen wird (Abschn. 12.9.4, Abb. 12.6) und $E(Y)$ zudem nicht durch eine Link-Funktion transformiert werden muss, um linear modellierbar zu sein. Anders als in der linearen Regression werden die Parameter β_j im GLM über die Maximum-Likelihood-Methode geschätzt (Abschn. 8.1.8). In den folgenden Abschnitten sollen die logistische, ordinale, multinomiale und Poisson-Regression ebenso vorgestellt werden wie log-lineare Modelle. Für die Visualisierung kategorialer Daten s. Abschn. 14.4, 15.2.2 sowie das Paket `vcf` (D. Meyer, Zeileis & Hornik, 2020).

8.1 Logistische Regression

In der logistischen Regression für dichotome Daten wird als bedingte Verteilung von Y die Binomialverteilung angenommen, die kanonische Link-Funktion ist die Logit-Funktion. Die bedingten Verteilungen sind dann durch $E(Y) = P$ vollständig festgelegt, da ihre Varianz gleich $n_i P(1 - P)$ ist. Dabei ist n_i die Anzahl der dichotomen Messwerte für dieselbe Kombination von Prädiktorwerten.

8.1.1 Modell für dichotome Daten anpassen

Die Anpassung einer logistischen Regression geschieht mit der `glm()` Funktion, mit der allgemein GLM-Modelle spezifiziert werden können.⁴ Hier sei zunächst der Fall betrachtet, dass die erhobenen Daten als dichotome Variable vorliegen.

```
glm(formula=<Modellformel>, family=<Verteilung>, data=<Datensatz>,
subset=<Indexvektor>)
```

Unter `formula` wird eine Modellformel `<Zielgröße> ~ <Prädiktoren>` wie mit `lm()` formuliert, wobei sowohl quantitative wie kategoriale Variablen als Prädiktoren möglich sind. Stammen die Variablen aus einem Datensatz, ist dieser unter `data` zu nennen. Die Zielgröße muss ein Objekt der Klasse `factor` mit zwei Stufen sein, die Auftretenswahrscheinlichkeit P bezieht sich auf die zweite Faktorstufe. Weiter ist unter `family` ein Objekt anzugeben, das die für die Zielgröße angenommene bedingte Verteilung sowie die Link-Funktion benennt (vgl. `?family`). Im Fall der logistischen Regression ist dieses Argument auf `binomial(link="logit")` zu setzen. Das Argument `subset` erlaubt es, nur eine Teilmenge der Fälle einfließen zu lassen. Es erwartet einen numerischen oder logischen Indexvektor, der sich auf die Zeilen des Datensatzes bezieht.

Als Beispiel sei jenes aus der Kovarianzanalyse herangezogen (Abschn. 7.9), wobei hier vorgesagt werden soll, ob die Depressivität nach der Behandlung über dem Median liegt. Die Beziehung zwischen der Depressivität vor und nach der Behandlung in den drei Gruppen soll dabei zunächst über ein Diagramm mit dem nonparametrisch geschätzten Verlauf der Trefferwahrscheinlichkeit veranschaulicht werden, das `cdplot(<Modellformel>)` erzeugt (*conditional density plot*, Abb. 8.1).

⁴Für die bedingte logistische Regression bei Stratifizierung der Beobachtungen vgl. `clogit()` aus dem Paket `survival` (Therneau, 2020).

```
# Median-Split: Umwandlung quantitative Zielgröße in dichotomen Faktor
> dfAncova <- transform(dfAncova,
+   postFac=cut(DVpost, breaks=c(-Inf, median(DVpost), Inf),
+   labels=c("lo", "hi")))

# conditional density plot der Zielgröße in den drei Gruppen
> cdplot(postFac ~ DVpre, data=dfAncova, subset=(IV == "SSRI"),
+   main="Geschätzte Kategorien-Wkt. SSRI")

> cdplot(postFac ~ DVpre, data=dfAncova, subset=(IV == "Placebo"),
+   main="Geschätzte Kategorien-Wkt. Placebo")

> cdplot(postFac ~ DVpre, data=dfAncova, subset=(IV == "WL"),
+   main="Geschätzte Kategorien-Wkt. WL")
```

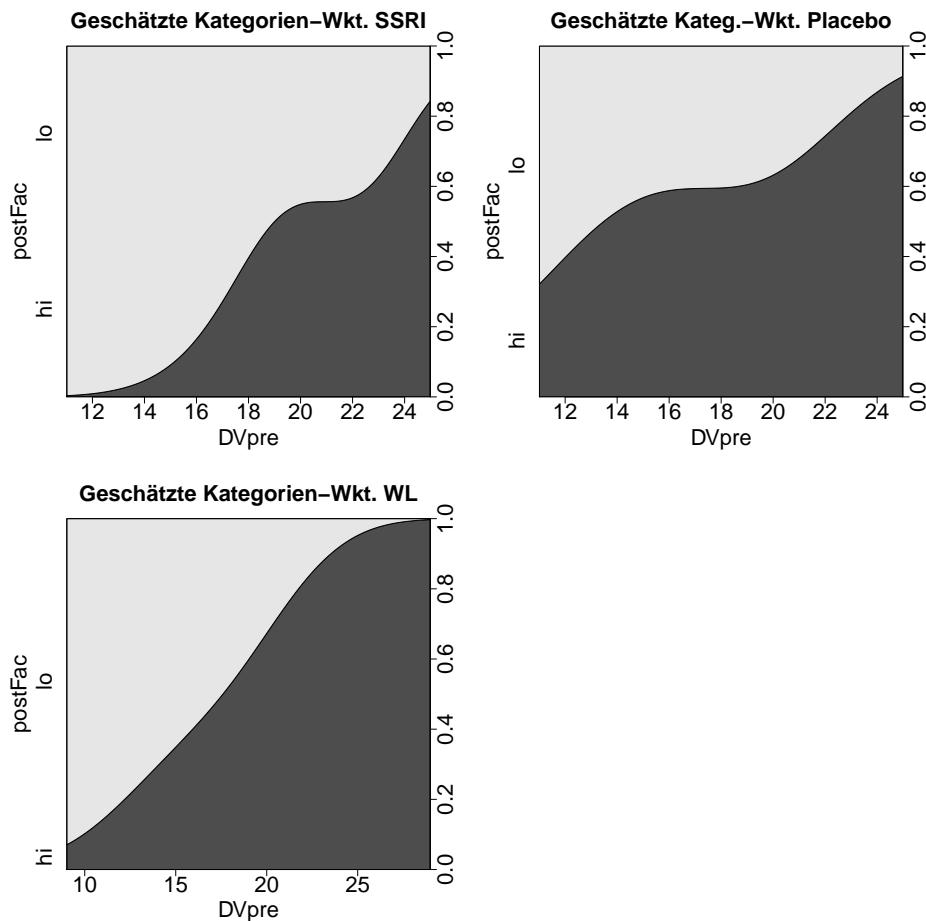


Abbildung 8.1: Nonparametrisch geschätzte Kategorienwahrscheinlichkeiten in Abhängigkeit vom Prädiktor in drei Behandlungsgruppen

```
# Anpassung des logistischen Regressionsmodells
> (glmFit <- glm(postFac ~ DVpre + IV, family=binomial(link="logit"),
+   data=dfAncova)) # gekürzte Ausgabe ...
```

Coefficients:

| | | | |
|-------------|--------|-----------|--------|
| (Intercept) | DVpre | IVPlacebo | IVWL |
| -8.4230 | 0.4258 | 1.7306 | 1.2027 |

```
Degrees of Freedom: 29 Total (i.e. Null); 26 Residual
Null Deviance: 41.46
Residual Deviance: 24.41 AIC: 32.41
```

Die Ausgabe nennt unter der Überschrift **Coefficients** zunächst die Schätzungen b_j der Modellparameter β_j der logistischen Regression, wobei der in der Spalte (**Intercept**) aufgeführte Wert die Schätzung b_0 ist. Der Parameter eines Prädiktors ist als Ausmaß der Änderung der Vorhersage $\ln \frac{\hat{P}}{1-\hat{P}}$ zu interpretieren, wenn der Prädiktor X_j um eine Einheit wächst, also als Differenz der Logits (s. Abschn. 7.9.1 zur Bedeutung der zwei mit dem Faktor IV assoziierten Parameter).

Einfacher ist die Bedeutung eines exponenzierten Parameters e^{b_j} zu erfassen: Dieser Koeffizient gibt an, um welchen Faktor der vorhergesagte Wettquotient $\frac{\hat{P}}{1-\hat{P}}$ zunimmt, wenn sich X_j um eine Einheit vergrößert.⁵ Dies ist das Verhältnis des vorhergesagten Wettquotienten nach der Änderung um eine Einheit zum Wettquotienten vor dieser Änderung, also ihr odds ratio (Abschn. 10.2.6). Dagegen besitzt e^{b_0} keine intuitive Bedeutung. Wie bei linearen Modellen extrahiert `coef(<glm-Modell>)` die Parameterschätzungen.

```
> exp(coef(glmFit))          # exponenzierte Koeffizienten = odds ratios
(Intercept)      DVpre      IVPlacebo      IVWL
0.0002197532  1.5308001795  5.6440022784  3.3291484767
```

Wie bei der linearen Regression lassen sich die Konfidenzintervalle für die wahren Parameter mit `confint(<glm-Modell>)` berechnen.⁶ Für die Konfidenzintervalle der odds ratios e^{β_j} sind die Intervallgrenzen zu exponenzieren.

```
> exp(confint(glmFit))          # zugehörige Konfidenzintervalle
2.5 %      97.5 %
(Intercept) 1.488482e-07  0.0251596
DVpre       1.193766e+00  2.2446549
IVPlacebo   5.343091e-01  95.1942030
IVWL        2.916673e-01  52.2883653
```

8.1.2 Modell für binomiale Daten anpassen

Logistische Regressionen können mit `glm()` auch dann angepasst werden, wenn pro Kombination i von Prädiktorwerten mehrere dichotome Werte erhoben und bereits zu Ausprägungen

⁵Für einen Prädiktor X : $\left(\frac{\hat{P}}{1-\hat{P}}\right)_{X+1} = e^{b_0+b_1(X+1)} = e^{b_0} e^{b_1(X+1)} = e^{b_0} e^{b_1 X} e^{b_1} = e^{b_1} e^{b_0+b_1 X} = e^{b_1} \left(\frac{\hat{P}}{1-\hat{P}}\right)_X$.

⁶Die so ermittelten Konfidenzintervalle basieren auf der Profile-Likelihood-Methode und sind asymmetrisch. Demgegenüber berechnet `confint.default(<glm-Modell>)` symmetrische Wald-Konfidenzintervalle, die asymptotische Normalverteilung der Parameterschätzungen voraussetzen.

einer binomialverteilten Variable aufsummiert wurden ($n_i \geq 1$). In diesem Fall ist das Kriterium in Form einer Matrix an die Modellformel zu übergeben: Jede Zeile der Matrix steht für eine Kombination von Prädiktorwerten i , für die n_i dichotome Werte vorhanden sind. Die erste Spalte der Matrix nennt die Anzahl der Treffer, die zweite Spalte die Anzahl der Nicht-Treffer. Beide Spalten summieren sich pro Zeile also zu n_i .

```
> N      <- 100                      # Anzahl Kombinationen
> x1    <- rnorm(N, 100, 15)        # Prädiktor 1
> x2    <- rnorm(N, 10, 3)          # Prädiktor 2
> total  <- sample(40:60, N, replace=TRUE) # n_i
> hits   <- rbinom(N, total, prob=0.4) # Treffer pro Kombination
> hitMat <- cbind(hits, total-hits)  # Matrix Treffer, Non-Treffer

# logistische Regression für absolute Häufigkeiten anpassen
> glm(hitMat ~ x1 + x2, family=binomial(link="logit"))      # ...
```

Liegt die Zielgröße als Vektor relativer Häufigkeiten eines Treffers vor, sind zusätzlich die n_i als Vektor an das Argument `weights` von `glm()` zu übergeben.

```
> relHits <- hits/total            # relative Häufigkeiten ...
> glm(relHits ~ x1 + x2, weights=total, family=binomial(link="logit"))
```

8.1.3 Anpassungsgüte

Als Maß für die Güte der Modellpassung wird von `glm()` die Residual-Devianz D als Summe der quadrierten Devianz-Residuen ausgegeben.⁷ Mit \hat{L} als geschätzter likelihood des Modells, die an der likelihood eines Modells mit perfekter Vorhersage normalisiert wurde, gilt $D = -2 \ln \hat{L}$. Durch die Maximum-Likelihood-Schätzung der Parameter wird \hat{L} maximiert, D also minimiert – analog zur Fehlerquadratsumme in der linearen Regression (Abschn. 6.2).⁸ Weiter erhält man den Wert des Informationskriteriums $AIC = D + 2(p+1)$, bei dem ebenfalls kleinere Werte für eine bessere Anpassung sprechen (Abschn. 6.3.3). Dabei ist $p+1$ die Anzahl zu schätzender Parameter (p Gewichte β_j sowie β_0).⁹

Die Residual-Devianz eines Modells ermittelt `deviance(<glm-Modell>)`, die logarithmierte geschätzte likelihood eines Modells `logLik(<glm-Modell>)` und den AIC-Wert `AIC(<glm-Modell>)`.

```
> (Dev <- deviance(glmFit))           # Devianz
[1] 24.40857

> sum(residuals(glmFit)^2)            # Devianz
[1] 24.40857

> as.vector(-2 * logLik(glmFit))      # Devianz
```

⁷In der Voreinstellung gibt `residuals(<glm-Modell>, type="<Typ>")` Devianz-Residuen aus. Für andere Residuen-Varianten kann das Argument `type` verwendet werden (vgl. `?residuals.glm`).

⁸Für die gewöhnliche lineare Regression stimmen Devianz und Fehlerquadratsumme überein.

⁹Bei der gewöhnlichen linearen Regression wie auch bei der logistischen Regression mit der quasi-binomial Familie (s. u.) ist zusätzlich ein Varianzparameter zu schätzen. Hier beträgt die Anzahl also $p+1+1$.

[1] 24.40857

```
> all.equal(AIC(glmFit), Dev + 2*(3+1)) # AIC
[1] TRUE
```

Weitere Maße der Anpassungsgüte sind pseudo- R^2 -Koeffizienten, die an den Determinationskoeffizienten in der linearen Regression angelehnt sind (Abschn. 6.2.2).¹⁰ Die Varianten nach McFadden, Cox & Snell (Maddala) und Nagelkerke (Cragg-Uhler) können neben weiteren von `PseudoR2(<glm-Modell>, which="<Variante>")` aus dem Paket `DescTools` berechnet werden.¹¹ Hier soll \hat{L}_0 die geschätzte likelihood des 0-Modells ohne Prädiktoren X_j mit nur dem Parameter β_0 sein. Analog sei \hat{L}_f die geschätzte likelihood des Modells mit allen berücksichtigten Prädiktoren. Ferner bezeichne D_0 bzw. D_f die jeweils zugehörige Devianz.

- $R_{\text{McFadden}}^2 = 1 - \frac{\ln \hat{L}_f}{\ln \hat{L}_0} = 1 - \frac{D_f}{D_0}$
- $R_{\text{Cox \& Snell}}^2 = 1 - \left(\frac{\hat{L}_0}{\hat{L}_f} \right)^{\frac{2}{N}} = 1 - e^{(\ln \hat{L}_0 - \ln \hat{L}_f) \frac{2}{N}}$
Das Maximum von $R_{\text{Cox \& Snell}}^2$ beträgt $1 - \hat{L}_0^{\frac{2}{N}} < 1$.
- $R_{\text{Nagelkerke}}^2 = R_{\text{Cox \& Snell}}^2 / (1 - \hat{L}_0^{\frac{2}{N}})$

```
# Pseudo R^2 nach McFadden, Cox & Snell und Nagelkerke
> library(DescTools) # für PseudoR2()
> PseudoR2(glmFit, which=c("McFadden", "CoxSnell", "Nagelkerke"))
  McFadden CoxSnell Nagelkerke
  0.4112090  0.4334714  0.5788220
```

Auf Basis der Ausgabe von `glm()` können pseudo- R^2 -Koeffizienten auch manuell ermittelt werden. Wie bei linearen Modellen lässt sich ein bereits angepasstes Modell dazu mit `update(<glm-Modell>)` ändern (Abschn. 6.3.2), etwa alle Prädiktoren bis auf den absoluten Term entfernen.

```
> glm0 <- update(glmFit, . ~ 1) # 0-Modell
> LL0  <- logLik(glm0)    # gesch. log-likelihood 0-Modell
> LLf  <- logLik(glmFit)  # gesch. log-likelihood vollständiges Modell
> as.vector(1 - (LLf / LL0)) # R^2 McFadden
[1] 0.411209

> N <- nobs(glmFit)      # Anzahl der Beobachtungen
> as.vector(1 - exp((2/N) * (LL0 - LLf))) # R^2 Cox & Snell
[1] 0.4334714
```

¹⁰ Anders als in der linearen Regression lassen sich die pseudo- R^2 -Maße jedoch nicht als Verhältnis von Variabilitäten verstehen. Ihre Vergleichbarkeit über verschiedene Datensätze hinweg ist zudem eingeschränkt – so beziehen etwa $R_{\text{Cox \& Snell}}^2$ sowie $R_{\text{Nagelkerke}}^2$ neben der absoluten Anpassung auch die Stichprobengröße ein.

¹¹ Für weitere Gütemaße der Modellanpassung vgl. die Funktion `lrm()` aus dem Paket `rms`, die auch die Fläche unter der ROC-Kurve AUC (Abschn. 10.2.7, äquivalent zu Harrels C) ebenso bestimmt wie Somers' d , Goodman und Kruskals γ sowie Kendalls τ für die vorhergesagten Wahrscheinlichkeiten und beobachteten Werte (Abschn. 10.3.1).

```
# R^2 Nagelkerke
> as.vector((1 - exp((2/N) * (LL0 - LLf))) / (1 - exp(LL0)^(2/N)))
[1] 0.578822
```

Eine Alternative zu pseudo- R^2 -Koeffizienten ist der Diskriminationsindex von Tjur, den man mit `PseudoR2(<glm-Modell>, which="Tjur")` erhält. Er berechnet sich als Differenz der jeweils mittleren vorhergesagten Trefferwahrscheinlichkeit für die Beobachtungen mit Treffer und Nicht-Treffer. Der Diskriminationsindex nimmt maximal den Wert 1 an, ist jedoch anders als die pseudo- R^2 -Koeffizienten nicht auf andere Regressionsmodelle für diskrete Kriterien verallgemeinerbar.

```
> PseudoR2(glmFit, which="Tjur")
  Tjur
0.4597767
```

Zur manuellen Berechnung erhält man die vorhergesagte Wahrscheinlichkeit \hat{P} mit `fitted(<glm-Modell>)` (Abschn. 8.1.4).

```
> Phat <- fitted(glmFit)      # vorhergesagte Trefferwahrscheinlichkeiten

# mittlere vorhergesagte Wahrscheinlichkeit für wFac = lo und wFac = hi
> (PhatLoHi <- aggregate(Phat ~ postFac, FUN=mean, data=dfAncova))
  postFac      Phat
1      lo 0.2521042
2      hi 0.7118809

# Tjur Diskriminationsindex
> abs(diff(PhatLoHi$Phat))  # Differenz mittlere vorherges. Wkt.
[1] 0.4597767
```

Für die Kreuzvalidierung verallgemeinerter linearer Modellen s. Abschn. 13.2. Methoden zur Diagnose von Ausreißern in den Prädiktoren können aus der linearen Regression ebenso übernommen werden wie Cooks Distanz und der Index DfBETAS zur Identifikation einflussreicher Beobachtungen (Abschn. 6.5.1). Anders als im Modell der linearen Regression hängt im GLM die Varianz vom Erwartungswert ab. Daher sollte eine Residuen-Diagnostik mit spread-level oder scale-location plots (Abschn. 6.5.2) standardisierte Devianz- oder Pearson-Residuen verwenden, wie es mit `glm.diag(<glm-Modell>)` aus dem Paket [boot Carty & Ripley, 2020](#) möglich ist. Methoden zur ausführlichen Diagnostik der Residuen bietet auch das Paket DHARMA ([Hartig, 2022](#)). Für Methoden zur Einschätzung von Multikollinearität der Prädiktoren s. Abschn. 6.5.3.

8.1.4 Vorhersage, Klassifikation, Kalibrierung und Anwendung auf neue Daten

Die vorhergesagte Wahrscheinlichkeit $\hat{P} = \frac{1}{1+e^{-\mathbf{x}\beta}}$ berechnet sich für jede Beobachtung durch Einsetzen der Parameterschätzungen b_j in $\mathbf{X}\hat{\beta} = b_0 + b_1X_1 + \dots + b_pX_p$. Man erhält sie mit `fitted(<glm-Modell>)` oder mit `predict(<glm-Modell>, type="response")`. In der Voreinstellung `type="link"` werden die vorhergesagten Logits ausgegeben.

```
> Phat <- fitted(glmFit)           # vorhergesagte Wahrscheinlichkeit
> predict(glmFit, type="response") # äquivalent ...
> logitHat <- predict(glmFit, type="link")   # vorhergesagte Logits
> all.equal(logitHat, qlogis(Phat))          # Kontrolle: log-odds
[1] TRUE
```

Mit der gewählten Herangehensweise ist die mittlere vorhergesagte Wahrscheinlichkeit gleich der empirischen relativen Häufigkeit eines Treffers.¹²

```
> mean(Phat)                      # mittlere vorhergesagte W'keit
[1] 0.4666667

# relative Trefferhäufigkeit
> proportions(xtabs(~ postFac, data=dfAncova))
postFac
      lo          hi
0.5333333 0.4666667
```

Die *Kalibrierung* einer logistischen Regression ist ein Maß dafür, wie gut die globale Entsprechung zwischen mittlerer vorhergesagter Trefferwahrscheinlichkeit und beobachteter relativer Trefferhäufigkeit auch in Untergruppen gilt, die bzgl. der vorhergesagten Trefferwahrscheinlichkeit gebildet werden. Die Gruppen sollten nicht zu klein gewählt werden, um die relative Häufigkeit zumindest grob abschätzen zu können. Ein Kalibrierungs-Plot ist ein Streudiagramm (Abschn. 14.2) mit nonparametrischem Glätter (Abschn. 16.1.3), das die vorhergesagte Trefferwahrscheinlichkeit gegen die beobachtete Trefferhäufigkeit abträgt. Es eignet sich, um Größe und Muster systematische Unter- bzw. Überschätzungen von Trefferwahrscheinlichkeiten zu erkennen. Das Paket **CalibrationCurves** ([De Cock, Nieboer, Van Calster, Steyerberg & Vergouwe, 2024](#)) erstellt aufwendige Kalibrierungs-Plots und erläutert Details in der zugehörigen Vignette.

```
> n_grp_calib <- 3                # Anzahl Gruppen für Kalibrierung
> Phat_cut <- cut(Phat,           # gleich große Gruppen bilden
+                   breaks=quantile(Phat,
+                   probs=seq(from=0, to=1,
+                           length.out=n_grp_calib+1)),
+                   include.lowest=TRUE)

# mittlere vorhergesagte Trefferwahrscheinlichkeit in Gruppen
> Phat_cut_avg <- tapply(Phat, Phat_cut, FUN=mean)

# mittlere beobachtete Trefferhäufigkeit in Gruppen
> obs_cut_avg <- tapply(dfAncova$postFac == "hi", Phat_cut, FUN=mean)
> cbind(p_observed=obs_cut_avg,      # Vergleich: Kalibrierung
+       p_predicted=Phat_cut_avg)
      p_observed p_predicted
[0.0232,0.167] 0.0000000 0.1003394
```

¹²Dies ist der Fall, wenn die kanonische Link-Funktion und Maximum-Likelihood-Schätzungen der Parameter gewählt werden und das Modell einen absoluten Term β_0 beinhaltet.

```
(0.167,0.653]    0.5555556   0.4403421
(0.653,0.994]    0.9000000   0.8933188
```

Die vorhergesagte Trefferwahrscheinlichkeit soll nun als Grundlage für eine dichotome Klassifikation mit der Schwelle $\hat{P} = 0.5$ verwendet werden. Diese Klassifikation kann mit den tatsächlichen Kategorien in einer Konfusionsmatrix verglichen und etwa die Rate der korrekten Klassifikation berechnet werden.¹³

```
# Klassifikation auf Basis der Vorhersage
> thresh <- 0.5                                # Schwelle bei P=0.5
> facHat <- cut(Phat, breaks=c(-Inf, thresh, Inf), labels=c("lo","hi"))

# Kontingenztafel: tatsächliche vs. vorhergesagte Kategorie
> cTab <- xtabs(~ postFac + facHat, data=dfAncova)
> addmargins(cTab)
      facHat
postFac lo hi Sum
  lo  12  4 16
  hi  4 10 14
  Sum 16 14 30

> (CCR <- sum(diag(cTab)) / sum(cTab)) # Rate der korrekten Klass.
[1] 0.7333333
```

Inwieweit die Vorhersage der logistischen Regression zutrifft, kann auf unterschiedliche Weise grafisch veranschaulicht werden. Eine Möglichkeit stellt die vorhergesagten Logits $\ln \frac{\hat{P}}{1-\hat{P}}$ dar und hebt die tatsächlichen Treffer farblich hervor (Abb. 8.2). Vorhergesagte Logits > 0 sind bei einer Schwelle von $\hat{P} = 0.5$ äquivalent zur Vorhersage eines Treffers, so dass die tatsächlichen Treffer hier bei einer guten Klassifikation oberhalb einer Referenzlinie bei 0 liegen sollten.

```
> plot(logitHat, pch=c(1, 16)[unclass(regDf$wFac)], cex=1.5, lwd=2,
+       main="(Fehl-) Klassifikation durch Vorhersage",
+       ylab="vorhergesagte Logits")

> abline(h=0)                                     # Referenzlinie
> legend(x="bottomright", legend=c("lo", "hi"), pch=c(1, 16), cex=1.5,
+         lty=NA, lwd=2, bg="white")
```

An das Argument `newdata` von `predict()` kann zusätzlich ein Datensatz übergeben werden, der neue Daten für Variablen mit denselben Namen, und bei Faktoren zusätzlich denselben Stufen wie jene der ursprünglichen Prädiktoren enthält. Als Ergebnis erhält man die vorhergesagten Trefferwahrscheinlichkeiten für die neuen Prädiktorwerte (Abschn. 6.4).

```
> Nnew <- 3                                     # Anzahl neuer Daten
```

¹³Vergleiche Abschn. 13.2 für die Kreuzvalidierung zur Abschätzung der Vorhersagegüte in neuen Stichproben sowie Abschn. 10.2.6, 10.2.7, 10.3.3 für weitere Möglichkeiten, Klassifikationen zu analysieren. Siehe Abschn. 12.8 für die Diskriminanzanalyse sowie die dortige Fußnote 45 für Hinweise zu weiteren Klassifikationsverfahren.

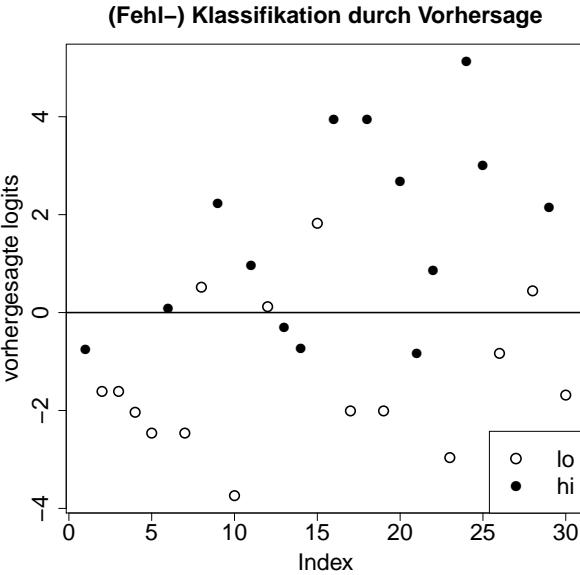


Abbildung 8.2: Vorhersage und Daten einer logistischen Regression. Daten, die auf Basis der Vorhersage falsch klassifiziert würden, sind farblich hervorgehoben

```
> dfNew <- data.frame(DVpre=rnorm(Nnew, 20, sd=7),
+                         IV=factor(rep("SSRI", Nnew),
+                                   levels=levels(dfAncova$IV)))

> predict(glmFit, newdata=dfNew, type="response") # Vorhersage W'keit
      1          2          3
0.947324911 0.008185519 0.123296946
```

8.1.5 Signifikanztests für Parameter und Modell

Die geschätzten Gewichte b lassen sich mit `summary(glm-Modell)` einzeln einem Wald-Signifikanztest unterziehen. In der Ausgabe finden sich dazu unter der Überschrift `Coefficients` die Gewichte b in der Spalte `Estimate`, deren geschätzte Streuungen $\hat{\sigma}_b$ in der Spalte `Std. Error` und die zugehörigen z -Werte $\frac{b}{\hat{\sigma}_b}$ in der Spalte `z value`. Der Test setzt voraus, dass z unter der Nullhypothese asymptotisch standardnormalverteilt ist. Die zugehörigen p -Werte stehen in der Spalte `Pr(>|z|)`.

```
> summary(glmFit) # Ausgabe gekürzt ...
Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept) -8.4230    2.9502 -2.855   0.0043 **
DVpre        0.4258    0.1553  2.742   0.0061 **
```

```
IVPlacebo      1.7306      1.2733      1.359      0.1741
IWVL          1.2027      1.2735      0.944      0.3450
```

(Dispersion parameter for binomial family taken to be 1)

```
Null deviance: 41.455 on 29 degrees of freedom
Residual deviance: 24.409 on 26 degrees of freedom
AIC: 32.409
```

Number of Fisher Scoring iterations: 5

Die geschätzte Streuung der geschätzten odds ratios e^b erhält man mit $\sqrt{(e^b)^2 \cdot \hat{\sigma}_b^2}$.

```
> OR      <- exp(coef(glmFit))      # odds ratios
> varB    <- diag(vcov(glmFit))    # Streuung geschätzte Koeffizienten
> (seOR <- sqrt(OR^2 * varB))[-1] # Streuung geschätzte odds ratios
      DVpre     IVPlacebo        IWVL
0.2377021539 7.1866998892 4.2397702028
```

Geeigneter als Wald-Tests sind oft Likelihood-Quotienten-Tests der Parameter, die auf der asymptotisch χ^2 -verteilten Devianz-Differenz zweier nested Modelle mit demselben Kriterium beruhen:¹⁴ Der Prädiktorensatz des eingeschränkten Modells ist dabei vollständig im Prädiktorensatz des umfassenderen Modells enthalten, das zusätzlich noch weitere Prädiktoren berücksichtigt (Abschn. 6.3.3). Solche Modellvergleiche können mit `anova(<fitR>, <fitU>)` durchgeführt werden, wobei `<fitR>` das eingeschränkte und `<fitU>` das umfassendere Modell ist. Zusätzlich lässt sich analog zur linearen Regression `drop1(<glm-Modell>, test="Chisq")` verwenden (Abschn. 6.3.3).

Um das Gesamtmodell mit einem Likelihood-Quotienten-Test auf Signifikanz zu prüfen, muss somit `anova(<0-Modell>, <glm-Modell>)` aufgerufen werden (Abschn. 6.3.3). Der durchgeführte χ^2 -Test beruht auf dem Vergleich des angepassten Modells mit dem 0-Modell, das nur eine Konstante als Prädiktor beinhaltet. Teststatistik ist die Devianz-Differenz beider Modelle mit der Differenz ihrer Freiheitsgrade als Anzahl der Freiheitsgrade der asymptotisch gültigen χ^2 -Verteilung.

```
> glm0 <- update(glmFit, . ~ 1)                      # 0-Modell
> anova(glm0, glmFit)                                # Modellvergleich
Analysis of Deviance Table
Model 1: postFac ~ 1
Model 2: postFac ~ DVpre + IV
  Resid. Df Resid. Dev Df Deviance  Pr(>Chi)
1         29     41.455
2         26     24.409  3   17.047 0.0006912 ***
# manuelle Kontrolle
```

¹⁴Bei Wald-Tests kann etwa das *Hauck-Donner-Phänomen* auftreten: Bei starken Effekten (sehr große β_j) sind die berechneten Streuungen $\hat{\sigma}_b$ dann deutlich zu groß, wodurch Wald-Tests der Parameter fälschlicherweise nicht signifikant werden.

```
> chisqStat <- glmFit$null.deviance - deviance(glmFit)
> chisqDf   <- glmFit$df.null      - df.residual(glmFit)
> (pVal     <- pchisq(chisqStat, chisqDf, lower.tail=FALSE))
[1] 0.0006912397
```

Da der hier im Modell berücksichtigte Faktor IV mit mehreren Parametern β_j assoziiert ist, muss seine Signifikanz insgesamt über einen Modellvergleich gegen das vollständige Modell getestet werden.

```
> glmPre <- update(glmFit, . ~ . - IV) # eingeschränktes Modell ohne IV
> anova(glmPre, glmFit)           # Modellvergleich
Analysis of Deviance Table
Model 1: postFac ~ DVpre
Model 2: postFac ~ DVpre + IV
  Resid. Df Resid. Dev Df Deviance Pr(>Chi)
1       28    26.566
2       26    24.409  2    2.1572   0.3401
```

Analog erfolgt für Quadratsummen vom Typ I (Abschn. 7.6.2) der Test des kontinuierlichen Prädiktors DVpre als Vergleich des Modells nur mit DVpre mit dem 0-Modell.

```
> anova(glm0, glmPre)           # Modellvergleich
Analysis of Deviance Table
Model 1: postFac ~ 1
Model 2: postFac ~ DVpre
  Resid. Df Resid. Dev Df Deviance Pr(>Chi)
1       29    41.455
2       28    26.566  1    14.89  0.000114 ***
```

8.1.6 Andere Link-Funktionen

Eine meist zu ähnlichen Ergebnissen führende Alternative zur logistischen Regression ist die Probit-Regression. Sie verwendet als Link-Funktion $\Phi^{-1}(\cdot)$, die Umkehrfunktion der Verteilungsfunktion $\Phi(\cdot)$ der Standardnormalverteilung, wofür das Argument `family` von `glm()` auf `binomial(link="probit")` zu setzen ist. Hier ist also $P = \Phi(\mathbf{X}\boldsymbol{\beta})$. Eine weitere Link-Funktion, die mit bedingter Binomialverteilung von Y kombiniert werden kann, ist die komplementäre log-log-Funktion $\ln(-\ln(1-P))$. Man erhält sie mit `binomial(link="cloglog")`. Mit ihrer Umkehrfunktion, der Verteilungsfunktion der Gumbel-Verteilung, gilt $P = 1 - \exp(-\exp(\mathbf{X}\boldsymbol{\beta}))$ – dabei steht $\exp(\cdot)$ aus typografischen Gründen für $e^{(\cdot)}$.

Den Spezialfall einer linearen Regression erhält man mit `gaussian(link="identity")`. Die Maximum-Likelihood-Schätzungen der Parameter stimmen dann mit den durch `lm()` ermittelten Schätzern der linearen Regression überein.

8.1.7 Modelle für overdispersion

Mitunter streuen empirische Residuen stärker, als dies bei bedingter Binomialverteilung mit der Varianz $n_i P(1 - P)$ zu erwarten wäre (*overdispersion*), wobei n_i die Anzahl an Beobachtungen mit gleichen Prädiktorwerten ist. Ein Hinweis auf overdispersion ist ein Verhältnis von Residual-Devianz zu Residual-Freiheitsgraden, das deutlich größer als 1 ist. Die modellbasierten Streuungsschätzungen unterschätzen dann die wahre Streuung, was zu liberale Signifikanztests der Parameter zur Folge hat. Eine mögliche Ursache für overdispersion ist ein unvollständiges Vorhersagemodell, das tatsächlich relevante Prädiktoren nicht berücksichtigt. Bei overdispersion kann ein Binomial-ähnliches Modell verwendet werden, das einen zusätzlichen, aus den Daten zu schätzenden Streuungsparameter ϕ besitzt, mit dem für die bedingte Varianz $\sigma^2 = \phi n_i P(1 - P)$ gilt: Hierfür ist `family` auf `quasibinomial(link="logit")` zu setzen. Die Parameterschätzungen b sind dann identisch zur logistischen Regression, die geschätzten Streuungen $\hat{\sigma}_b$ jedoch unterschiedlich, was zu anderen Ergebnissen der inferenzstatistischen Tests führt.

Da die bedingte Verteilung der Daten in der quasi-binomial Familie bis auf Erwartungswert und Varianz unspezifiziert bleibt, ist die Likelihood-Funktion der Daten für gegebene Parameter unbekannt. Trotzdem kann jedoch die IWLS-Methode Parameter schätzen (Abschn. 8.1.8), wobei dann alle likelihood-basierten Kennwerte wie AIC oder pseudo- R^2 nicht zur Verfügung stehen. Durch die Schätzung des Streuungs-Parameters sind die Wald-Teststatistiken keine z -, sondern t -Werte. Analog werden Modellvergleiche mit `anova()` als F -Test statt als χ^2 -Test wie bei bekannter Varianz durchgeführt.

8.1.8 Mögliche Probleme bei der Modellanpassung

Die Maximum-Likelihood-Schätzung der Parameter ist nicht in geschlossener Form darstellbar, sondern muss mit einem numerischen Optimierungsverfahren gefunden werden (Abschn. 16.4.2) – typischerweise über die IWLS-Methode (*iterative weighted least squares*). Diese numerische Suche nach dem Maximum der Likelihood-Funktion kann in seltenen Fällen fehlschlagen, auch wenn es ein eindeutiges Maximum gibt. Ursache einer nicht konvergierenden Suche ist häufig, dass sie an einem Punkt von Schätzungen beginnt, der zu weit vom tatsächlichen Maximum entfernt ist.

Konvergenz-Probleme können durch Warnmeldungen angezeigt werden (etwa *algorithm did not converge*), sich in sehr großen Schätzungen bei gleichzeitig sehr großen Standardfehlern äußern oder in einer Residual-Devianz, die größer als die Devianz des Null-Modells ist. In diesen Fällen ist das Ergebnis von `glm()` nicht gültig. Ob Konvergenz erreicht wurde, speichert die von `glm()` zurückgegebene Liste in der Komponente `converged`.

Um die Konvergenz der Suche zu begünstigen, lässt sich über das Argument `start` manuell ein Vektor von Start-Werten für alle Parameter vorgeben. Eine Strategie für ihre Wahl besteht darin, eine gewöhnliche lineare Regression der logit-transformierten Variable Y durchzuführen und deren Parameter-Schätzungen zu wählen. Konvergenzprobleme lassen sich u. U. auch über eine höhere maximale Anzahl von Suchschritten beheben, die in der Voreinstellung 25 beträgt und so geändert werden kann:

```
glm(..., control=glm.control(maxit=(Anzahl)))
```

Die Maximum-Likelihood-Schätzung der Parameter setzt voraus, dass keine (quasi-) vollständige Separierbarkeit von Prädiktoren durch die Kategorien von Y vorliegt (Allison, 2008, 2003). Dies ist der Fall, wenn Y stellenweise perfekt aus einer Linearkombination der Prädiktoren vorhersagbar ist. Eine mögliche Ursache sind schwach besetzte Gruppen bei kategorialen Kovariaten. Ein Symptom für (quasi-) vollständige Separierbarkeit ist die Warnmeldung, dass geschätzte Trefferwahrscheinlichkeiten von 0 bzw. 1 aufgetreten sind. Ein weiteres Symptom für Separierbarkeit sind sehr große Parameterschätzungen, die mit großen Standardfehlern assoziiert sind und daher im Wald-Test kleine z -Werte liefern.

In diesen Fällen kann auf Bayes-Methoden mit schwach informativer a-priori-Verteilung (etwa mit den Paketen `rstanarm` bzw. `brms`) oder auf penalisierte Verfahren ausgewichen werden. Dazu zählt die logistische Regression mit Firth-Korrektur, die im Paket `logistf` (Heinze & Ploner, 2020) umgesetzt wird (Abschn. 6.6.2). Eine andere Möglichkeit besteht darin, Kategorien in relevanten Prädiktorvariablen zusammenzufassen.

8.2 Ordinale Regression

In der ordinalen Regression soll eine kategoriale Variable Y mit k geordneten Kategorien $1, \dots, g, \dots, k$ mit p Prädiktoren X_j vorhergesagt werden. Dazu führt man die Situation auf jene der logistischen Regression zurück (Abschn. 8.1), indem man zunächst $k - 1$ dichotome Kategorisierungen $Y \geq g$ vs. $Y < g$ mit $g = 2, \dots, k$ vornimmt. Mit diesen dichotomen Kategorisierungen lassen sich nun $k - 1$ *kumulative Logits* bilden:

$$\text{logit}(P(Y \geq g)) = \ln \frac{P(Y \geq g)}{1 - P(Y \geq g)} = \ln \frac{P(Y = g) + \dots + P(Y = k)}{P(Y = 1) + \dots + P(Y = g - 1)}$$

Die $k - 1$ kumulativen Logits geben jeweils die logarithmierte Chance dafür an, dass Y mindestens die Kategorie g erreicht. Sie werden in $k - 1$ separaten logistischen Regressionen mit dem Modell $\text{logit}(P(Y \geq g)) = \beta_{0_g} + \beta_1 X_1 + \dots + \beta_p X_p$ linear vorhergesagt ($g = 2, \dots, k$). Die Parameterschätzungen erfolgen dabei für alle Regressionen simultan mit der Nebenbedingung, dass die Menge der β_j für alle g identisch ist und $\beta_{0_2} > \dots > \beta_{0_k}$ gilt. In diesem Modell führt ein höherer Prädiktorwert X_j bei positivem β_j zu einer höheren Chance, dass eine höhere Kategorie von Y erreicht wird.¹⁵

Kern des Modells ist die Annahme, dass eine additive Erhöhung des Prädiktorwerts X_j um den Wert c dazu führt, dass die Chance für eine höhere Kategorie unabhängig von g um den festen Faktor $e^{\beta_j c}$ wächst: $\frac{P(Y \geq g | X_j + c)}{P(Y < g | X_j + c)} = e^{\beta_j c} \frac{P(Y \geq g | X_j)}{P(Y < g | X_j)}$. Aus diesem Grund heißt es auch *proportional odds* Modell. Wie in der logistischen Regression ist damit e^{β_j} das odds ratio, also der Faktor, um den der vorhergesagte Wettquotient zunimmt, wenn X_j um eine Einheit wächst.¹⁶

¹⁵ Andere Formulierungen des Modells sind möglich. So legt etwa SPSS das Modell $\text{logit}(P(Y \leq g)) = \beta_{0_g} - (\beta_1 X_1 + \dots + \beta_p X_p)$ mit der Nebenbedingung $\beta_{0_1} < \dots < \beta_{0_{k-1}}$ zugrunde, das jedoch nur zu umgedrehten Vorzeichen der Schätzungen für die β_{0_g} führt. Mit derselben Nebenbedingung ließe sich das Modell auch als $\text{logit}(P(Y \leq g)) = \beta_{0_g} + \beta_1 X_1 + \dots + \beta_p X_p$ formulieren. In diesem Modell führt ein höherer Prädiktorwert X_j bei positivem β_j zu einer höheren Chance, dass eine *niedrigere* Kategorie von Y erreicht wird. Entsprechend haben hier die Schätzungen für alle Parameter umgekehrte Vorzeichen.

¹⁶ Alternative proportional odds Modelle sind zum einen mit *adjacent category Logits* $\ln \frac{P(Y=g)}{P(Y=g-1)}$ möglich, zum anderen mit *continuation ratio (sequentiellen) Logits* $\ln \frac{P(Y=g)}{P(Y < g)}$.

Wegen $P(Y = g) = P(Y \geq g) - P(Y \geq g+1)$ hat die Link-Funktion sowie ihre Umkehrfunktion hier eine zusammengesetzte Form. Mit der logistischen Funktion als Umkehrfunktion der Logit-Funktion sind die theoretischen Parameter $P(Y = g)$ identifizierbar als:

$$P(Y = g) = \frac{e^{\beta_{0g} + \beta_1 X_1 + \dots + \beta_p X_p}}{1 + e^{\beta_{0g} + \beta_1 X_1 + \dots + \beta_p X_p}} - \frac{e^{\beta_{0g+1} + \beta_1 X_1 + \dots + \beta_p X_p}}{1 + e^{\beta_{0g+1} + \beta_1 X_1 + \dots + \beta_p X_p}}$$

Die Modellannahmen lassen sich grafisch veranschaulichen (Abb. 8.3): Bei einem Prädiktor X sind die $k - 1$ kumulativen Logits lineare Funktionen von X mit derselben Steigung und geordneten y -Achsenabschnitten. Die Wahrscheinlichkeit dafür, dass Y mindestens die Kategorie $g = 2, \dots, k$ erreicht, ergibt sich aus $k - 1$ parallelen logistischen Funktionen mit der horizontalen Verschiebung $\frac{\beta_{0g+1} - \beta_{0g}}{\beta_1}$, d. h. $P(Y \geq g+1|X) = P(Y \geq g|X - \frac{\beta_{0g+1} - \beta_{0g}}{\beta_1})$.

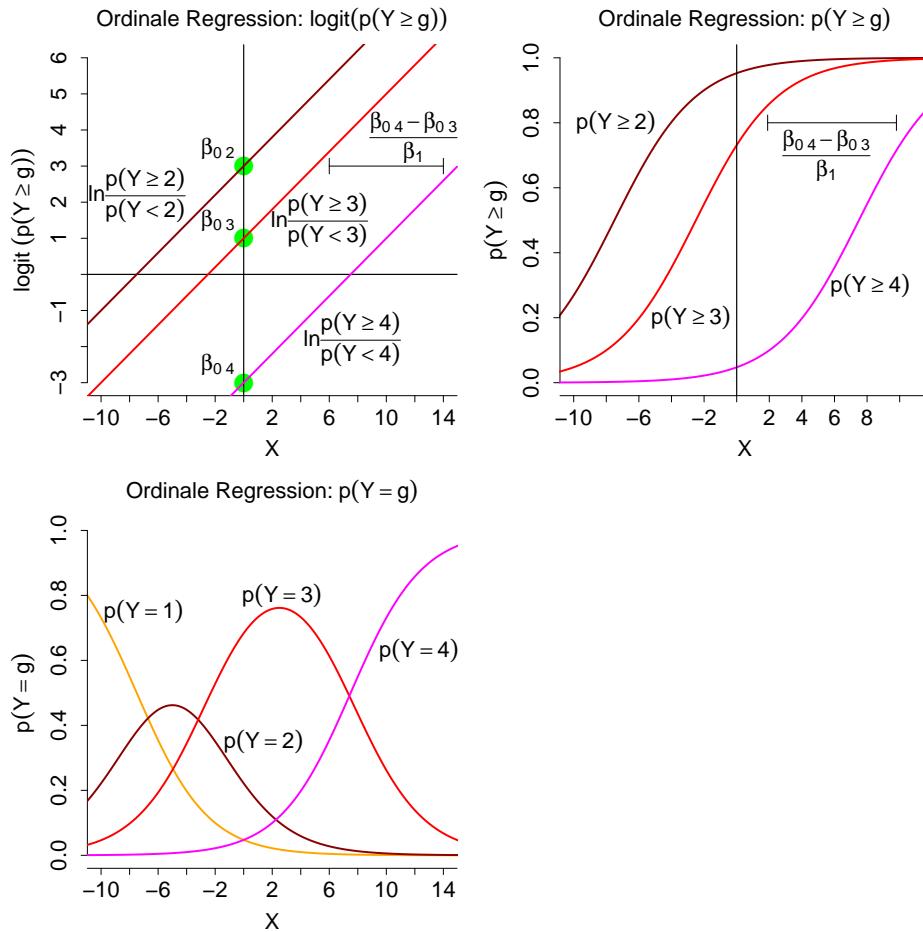


Abbildung 8.3: Modellannahmen der ordinalen Regression bei einem Prädiktor und 4 Kategorien des Kriteriums Y : Linearität der 3 kumulativen Logits mit identischer Steigung; parallel verschobene logistische Funktionen jeweils für die Wahrscheinlichkeit, mindestens Kategorie g zu erreichen; stochastisch geordnete Funktionen für die Wahrscheinlichkeit von $Y = g$.

8.2.1 Modellanpassung

Das proportional odds Modell mit kumulativen Logits kann mit `vglm()` aus dem Paket VGAM (Yee, 2010) angepasst werden. Diese Funktion hat gegenüber anderen, später ebenfalls erwähnten Funktionen den Vorteil, dass sie sich für alle in diesem Kapitel behandelten Modelle eignet und damit eine konsistente Herangehensweise an verwandte Fragestellungen ermöglicht.

```
vglm(<Modellformel>, family=<Modell>, data=<Datensatz>)
```

Die Modellformel ist wie bei `glm()` zu formulieren, ihre linke Seite muss ein geordneter Faktor sein (Abschn. 2.6.4). Für das proportional odds Modell ist `family=propodds` zu setzen.¹⁷ Schließlich benötigt `data` den Datensatz mit den Variablen aus der Modellformel.

Als Beispiel soll eine Variable mit $k = 4$ geordneten Kategorien anhand von $p = 2$ Prädiktoren vorhergesagt werden. Die kategoriale Zielgröße soll sich dabei aus der Diskretisierung einer kontinuierlichen Variable ergeben.

```
> N      <- 100
> X1    <- rnorm(N, 175, 7)                      # Prädiktor 1
> X2    <- rnorm(N, 30, 8)                        # Prädiktor 2
> Ycont <- 0.5*X1 - 0.3*X2 + 10 + rnorm(N, 0, 6) # kontinuierlich

# geordneter Faktor aus Diskretisierung der kontinuierlichen Zielgröße
> Yord <- cut(Ycont, breaks=quantile(Ycont), include.lowest=TRUE,
+               labels=c("--", "-", "+", "++"), ordered=TRUE)

# zugehöriger ungeordneter Faktor für spätere Auswertungen
> Ycateg <- factor(Yord, ordered=FALSE)
> dfOrd  <- data.frame(X1, X2, Yord, Ycateg)     # Datensatz

> library(VGAM)                                    # für vglm()
> vglmFit <- vglm(Yord ~ X1 + X2, family=propodds, data=dfOrd)
```

Die $k - 1$ Schätzungen der Parameter β_{0g} sowie die p Schätzungen der Parameter β_j können mit `coef(<vglm-Modell>)` extrahiert werden. Die b_j sind wie in der logistischen Regression zu exponentiellieren, um die geschätzten odds ratios zu erhalten (Abschn. 8.1.1): e^{b_j} gibt an, um welchen Faktor die vorhergesagte Chance wächst, eine höhere Kategorie zu erreichen, wenn sich X_j um eine Einheit erhöht. Die proportional odds Annahme bedeutet, dass sich die Wechselchance bei allen Kategorien im selben Maße ändert.

```
> exp(coef(vglmFit))                            # odds ratios
(Intercept):1 (Intercept):2 (Intercept):3          X1           X2
3.554552e-11 9.225479e-12 2.195724e-12 1.170903e+00 9.355142e-01
```

¹⁷ Mit `vglm()` ist es möglich, auch die proportional odds Modelle mit adjacent category Logits bzw. continuation ratio Logits anzupassen (Abschn. 8.2, Fußnote 16). Dazu ist `family` auf `acat(parallel=TRUE)` bzw. auf `sratio(parallel=TRUE)` zu setzen. Eine weitere Option für `acat()` bzw. `sratio()` ist dabei das Argument `reverse`, das die Vergleichsrichtung dieser Logits bzgl. der Stufen von Y kontrolliert und auf `TRUE` oder `FALSE` gesetzt werden kann.

8.2.2 Anpassungsgüte

Wie in der logistischen Regression dienen verschiedene Maße als Anhaltspunkte zur Anpassungsgüte des Modells (Abschn. 8.1.3). Dazu zählt die Devianz ebenso wie das Informationskriterium AIC sowie die pseudo- R^2 Kennwerte.¹⁸

```
> deviance(vglmFit)          # Devianz
[1] 244.1915

> AIC(vglmFit)              # Informationskriterium AIC
[1] 254.1915

> library(DescTools)         # für PseudoR2()
> PseudoR2(vglmFit, which="Nagelkerke")
Nagelkerke
0.3003312
```

Für potentiell auftretende Probleme bei der Modellanpassung s. Abschn. 8.1.8.

8.2.3 Signifikanztests für Parameter und Modell

Mit `summary(<vglm-Modell>)` werden neben den Parameterschätzungen b auch ihre geschätzten Streuungen $\hat{\sigma}_b$ sowie die zugehörigen z -Werte $\frac{b}{\hat{\sigma}_b}$ berechnet. Mit der Annahme, dass z unter der Nullhypothese asymptotisch standardnormalverteilt ist, stehen die zweiseitigen p -Werte in der Spalte `Pr(>|z|)`. Die genannten Werte lassen sich mit `coef(summary(<vglm-Modell>))` extrahieren. Profile-Likelihood Konfidenzintervalle erzeugt `confint(<vglm-Modell>, method="profile")`.

```
> sumOrd <- summary(vglmFit)
> (coefOrd <- coef(sumOrd))
      Estimate Std. Error   z value   Pr(>|z|)
(Intercept):1 -24.06020695 5.82571121 -4.130003 3.627579e-05
(Intercept):2 -25.40905205 5.87912090 -4.321914 1.546818e-05
(Intercept):3 -26.84450928 5.94234461 -4.517495 6.257564e-06
X1            0.15777487 0.03385696  4.660042 3.161443e-06
X2           -0.06665895 0.02499298 -2.667107 7.650735e-03

> confint(vglmFit, method="profile")
      2.5 %    97.5 %
(Intercept):1 -35.73662098 -12.97698381
(Intercept):2 -37.20085293 -14.23533935
(Intercept):3 -38.78768133 -15.55313940
X1            0.09392565  0.22569264
X2           -0.11661985 -0.01890649
```

¹⁸Weitere Gütemaße der Modellanpassung erzeugt `orm()` aus dem Paket `rms` (Abschn. 8.1.3, Fußnote 11).

Eine oft geeignetere Alternative zu Wald-Tests sind Likelihood-Quotienten-Tests eines umfassenderen Modells `<fitU>` gegen ein eingeschränktes Modell `<fitR>` mit `anova(<fitR>, <fitU>, type="I")` aus dem Paket VGAM (Abschn. 8.1.5). Der p -Wert steht in der Ausgabe in der Spalte `Pr(>Chi)`.

```
# eingeschränktes Modell ohne Prädiktor X2
> vglmR <- vglm(Yord ~ X1, family=propodds, data=df0rd)
> anova(vglmR, vglmFit, type="I")          # Likelihood-Quotienten-Test
Analysis of Deviance Table
Model 1: Yord ~ X1
Model 2: Yord ~ X1 + X2
  Resid. Df Resid. Dev Df Deviance Pr(>Chi)
1       296    251.75
2       295    244.19  1    7.5568 0.005978 **
```

Auf diese Weise lässt sich auch das Gesamtmodell gegen das 0-Modell ohne Prädiktoren X_j testen.

```
# Null-Modell ohne Prädiktoren
> vglm0 <- vglm(Yord ~ 1, family=propodds, data=df0rd)
> anova(vglm0, vglmFit, type="I")          # Test des Gesamtmodells
Analysis of Deviance Table
Model 1: Yord ~ 1
Model 2: Yord ~ X1 + X2
  Resid. Df Resid. Dev Df Deviance Pr(>Chi)
1       297    277.26
2       295    244.19  2   33.067  6.6e-08 ***
```

Ohne die proportional odds Annahme, dass die Menge der β_j in allen $k-1$ separaten Regressions der kumulativen Logits identisch ist, erhält man ein umfassenderes Modell. Die Parameter β_{jg} sind dann von g abhängig, die linearen Funktionen für die Logits also nicht mehr parallel (Abb. 8.3). Dieses Modell lässt sich ebenfalls mit `vglm()` anpassen, indem man `family` auf `cumulative(parallel=FALSE)` setzt. Der Likelihood-Quotienten-Test des umfassenderen Modells gegen das eingeschränkte Modell mit proportional odds Annahme erfolgt wieder mit `anova()`. Fällt der Test signifikant aus, ist dies ein Hinweis darauf, dass die Daten gegen die proportional odds Annahme sprechen.

```
# eingeschränktes Modell - äquivalent zu vglm(..., family=propodds)
> vglmP <- vglm(Yord ~ X1 + X2, family=cumulative(parallel=TRUE,
+               reverse=TRUE), data=df0rd)

# umfassenderes Modell ohne proportional odds Annahme
> vglmNP <- vglm(Yord ~ X1 + X2, family=cumulative(parallel=FALSE,
+               reverse=TRUE), data=df0rd)

> anova(vglmNP, vglmP, type="I")          # Likelihood-Quotienten-Test
Analysis of Deviance Table
Model 1: Yord ~ X1 + X2
Model 2: Yord ~ X1 + X2
```

```
Resid. Df Resid. Dev Df Deviance Pr(>Chi)
1      291     239.79
2      295     244.19 -4   -4.4033    0.3542
```

8.2.4 Vorhersage, Klassifikation und Anwendung auf neue Daten

Die vorhergesagten Kategorienwahrscheinlichkeiten $\hat{P}(Y = g)$ erhält man wie in der logistischen Regression mit `predict(vglm-Modell, type="response")`. Die ausgegebene Matrix enthält für jede Beobachtung (Zeilen) die vorhergesagte Wahrscheinlichkeit für jede Kategorie (Spalten).

```
> PhatCateg <- predict(vglmFit, type="response")
> head(PhatCateg, n=3)
  Yord=--   Yord=-   Yord=+   Yord=++
1 0.1433775 0.2486796 0.3383702 0.26957271
2 0.2785688 0.3194630 0.2640555 0.13791266
3 0.5278107 0.2837526 0.1360684 0.05236823
```

Die vorhergesagten Kategorien selbst lassen sich aus den vorhergesagten Kategorienwahrscheinlichkeiten bestimmen, indem pro Beobachtung die Kategorie mit der maximalen vorhergesagten Wahrscheinlichkeit herangezogen wird. Den jeweiligen Spaltenindex des zeilenweisen Maximums einer Matrix gibt `max.col(Matrix)` aus.

```
> categHat <- levels(dfOrd$Yord) [max.col(PhatCateg)]
> head(categHat)
[1] "+"  "-"  "--"  "+"  "--"  "+"
```

Die Kontingenztafel tatsächlicher und vorhergesagter Kategorien eignet sich als Grundlage für die Berechnung von Übereinstimmungsmaßen wie der Rate der korrekten Klassifikation. Hier ist darauf zu achten, dass die Kategorien identisch geordnet sind.

```
# Faktor mit gleich geordneten Kategorien
> facHat <- factor(categHat, levels=levels(dfOrd$Yord))

# Kontingenztafel tatsächlicher und vorhergesagter Kategorien
> cTab <- table(dfOrd$Yord, facHat, dnn=c("Yord", "facHat"))
> addmargins(cTab)                                # Randsummen hinzufügen
  facHat
Yord  --  -  +  ++ Sum
  --  10  7  7  1  25
  -   10  7  4  4  25
  +   5   7  7  6  25
  ++  0   2  9  14 25
  Sum 25 23 27 25 100

> (CCR <- sum(diag(cTab)) / sum(cTab)) # Rate der korrekten Klass.
[1] 0.38
```

An das Argument `newdata` von `predict()` kann zusätzlich ein Datensatz übergeben werden, der neue Daten für Variablen mit denselben Namen, und bei Faktoren zusätzlich denselben Stufen wie jene der ursprünglichen Prädiktoren enthält. Als Ergebnis erhält man die vorhergesagten Kategorienwahrscheinlichkeiten für die neuen Prädiktorwerte (Abschn. 6.4).

```
> Nnew <- 3 # Anzahl neuer Daten
> dfNew <- data.frame(X1=rnorm(Nnew, 175, 7), # neue Daten
+ X2=rnorm(Nnew, 30, 8))

> predict(vglmFit, newdata=dfNew, type="response") # Vorhersage
Yord--- Yord=- Yord=+ Yord=++
1 0.10563202 0.2071139 0.3438480 0.3434060
2 0.17385833 0.2739122 0.3253090 0.2269204
3 0.31326643 0.3240997 0.2433696 0.1192643
```

8.3 Multinomiale Regression

In der multinomialen Regression soll eine kategoriale Variable Y mit k ungeordneten Kategorien $1, \dots, g, \dots, k$ mit p Prädiktoren X_j vorhergesagt werden. Dazu führt man die Situation auf jene der logistischen Regression zurück (Abschn. 8.1), indem man zunächst eine Referenzkategorie r von Y festlegt und $k - 1$ Logits der Form $\ln \frac{P(Y=g)}{P(Y=r)}$ bildet. Diese $k - 1$ *baseline category* Logits geben die logarithmierte Chance dafür an, dass Y die Kategorie g annimmt – verglichen mit der Referenzkategorie r . Für r wird in der Voreinstellung typischerweise die erste (so hier im Folgenden) oder die letzte Kategorie von Y gewählt.

Die baseline category Logits werden in $k - 1$ separaten logistischen Regressionen mit dem Modell $\ln \frac{P(Y=g)}{P(Y=1)} = \beta_{0_g} + \beta_{1_g} X_1 + \dots + \beta_{p_g} X_p$ linear vorhergesagt ($g = 1, \dots, k$).¹⁹ Die Parameterschätzungen erfolgen für alle Regressionen simultan, wobei anders als in der ordinalen Regression sowohl die β_{0_g} als auch die Gewichte β_{j_g} als von g abhängig betrachtet werden (Abb. 8.4). In diesem Modell führt ein höherer Prädiktorwert X_j bei positivem β_j zu einer höheren Chance, dass die Kategorie g von Y angenommen wird – verglichen mit der Referenzkategorie.²⁰ Mit den gewählten baseline category Logits sind auch alle verbleibenden logarithmierten Chancen beim Vergleich von je zwei Kategorien a, b von Y festgelegt:

$$\begin{aligned}\ln \frac{P(Y=a)}{P(Y=b)} &= \ln \frac{P(Y=a)/P(Y=1)}{P(Y=b)/P(Y=1)} = \ln \frac{P(Y=a)}{P(Y=1)} - \ln \frac{P(Y=b)}{P(Y=1)} \\ &= \beta_{0_a} + \beta_{1_a} X_1 + \dots + \beta_{p_a} X_p \\ &\quad - (\beta_{0_b} + \beta_{1_b} X_1 + \dots + \beta_{p_b} X_p) \\ &= (\beta_{0_a} - \beta_{0_b}) + (\beta_{1_a} - \beta_{1_b}) X_1 + \dots + (\beta_{p_a} - \beta_{p_b}) X_p\end{aligned}$$

¹⁹Kurz $\ln \frac{P(Y=g)}{P(Y=1)} = \mathbf{X} \boldsymbol{\beta}_g$. In der Referenzkategorie 1 sind die Parameter wegen $\ln \frac{P(Y=1)}{P(Y=1)} = \ln 1 = 0$ festgelegt, und es gilt $\beta_{0_1} = \beta_{j_1} = 0$ (mit $j = 1, \dots, p$) sowie $e^{\mathbf{X} \boldsymbol{\beta}_g} = e^0 = 1$.

²⁰Dabei wird *Unabhängigkeit von irrelevanten Alternativen* angenommen: Für die Chance beim paarweisen Vergleich von g mit der Referenzkategorie soll die Existenz weiterer Kategorien irrelevant sein. Ohne diese Annahme kommen etwa Bradley-Terry-Modelle aus, von denen eine eingeschränkte Variante mit `brat()` aus dem Paket `VGAM` angepasst werden kann.

Mit der logistischen Funktion als Umkehrfunktion der Logit-Funktion sind die theoretischen Parameter $P(Y = g)$ identifizierbar als:

$$P(Y = g) = \frac{e^{\beta_{0g} + \beta_{1g}X_1 + \dots + \beta_{pg}X_p}}{\sum_{c=1}^k e^{\beta_{0c} + \beta_{1c}X_1 + \dots + \beta_{pc}X_p}} = \frac{e^{\beta_{0g} + \beta_{1g}X_1 + \dots + \beta_{pg}X_p}}{1 + \sum_{c=2}^k e^{\beta_{0c} + \beta_{1c}X_1 + \dots + \beta_{pc}X_p}}$$

Insbesondere bestimmt sich die Wahrscheinlichkeit der Referenzkategorie 1 als:

$$P(Y = 1) = \frac{1}{1 + \sum_{c=2}^k e^{\beta_{0c} + \beta_{1c}X_1 + \dots + \beta_{pc}X_p}}$$

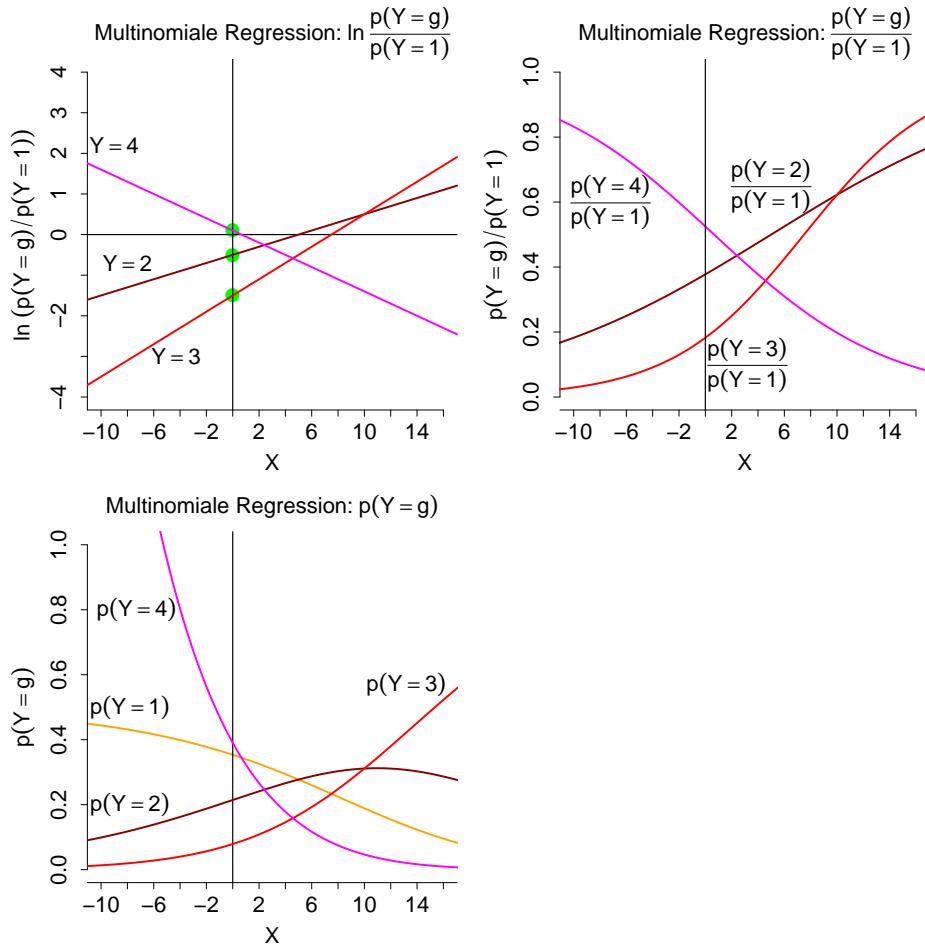


Abbildung 8.4: Multinomiale Regression mit einem Prädiktor und 4 Kategorien der Zielgröße Y : Linearität der 3 baseline category Logits mit unterschiedlichen Steigungen; logistische Funktionen für die Chance, verglichen mit der baseline Kategorie eine Kategorie g zu erhalten; zugehörige Funktionen für die Wahrscheinlichkeit einer Kategorie g von Y .

8.3.1 Modellanpassung

Als Beispiel sollen die Daten der ordinalen Regression in Abschn. 8.2.1 herangezogen werden, wobei als kategoriale Zielgröße nun der dort bereits erstellte ungeordnete Faktor dient. Die Anpassung des Modells erfolgt wieder mit `vglm()` aus dem Paket VGAM. Dafür ist das Argument `family` auf `multinomial(refLevel=1)` zu setzen, womit gleichzeitig die Referenzkategorie `refLevel` auf die erste Stufe von Y festgelegt werden kann. Wie in der logistischen Regression sind die Schätzungen b_{jg} zu exponenzieren, um die geschätzten odds ratios zu erhalten: $e^{b_{jg}}$ gibt bezogen auf die Referenzkategorie an, um welchen Faktor die vorhergesagte Chance wächst, Kategorie g zu erhalten, wenn sich X_j um eine Einheit erhöht (Abschn. 8.1.1).

```
> library(VGAM)                                     # für vglm()
> vglmFitMN <- vglm(Ycateg ~ X1 + X2, family=multinomial(refLevel=1),
+                      data=dfOrd)

> exp(coef(vglmFitMN))                           # odds ratios
(Intercept):1 (Intercept):2 (Intercept):3          X1:1           X1:2
5.812237e-02 1.579953e-11 8.460664e-20 1.019506e+00 1.162001e+00
X1:3          X2:1           X2:2          X2:3
1.316564e+00 9.854338e-01 9.636944e-01 8.647246e-01
```

8.3.2 Anpassungsgüte

Wie in der logistischen Regression dienen verschiedene Maße als Anhaltspunkte zur Anpassungsgüte des Modells (Abschn. 8.1.3). Dazu zählt die Devianz ebenso wie das Informationskriterium AIC sowie die pseudo- R^2 Kennwerte.

```
> deviance(vglmFitMN)                          # Devianz
[1] 237.148

> AIC(vglmFitMN)                             # Informationskriterium AIC
[1] 255.148

> library(DescTools)                         # für PseudoR2()
> PseudoR2(vglmFitMN, which="Nagelkerke")
Nagelkerke
0.3524506
```

Für potentiell auftretende Probleme bei der Modellanpassung s. Abschn. 8.1.8.

8.3.3 Signifikanztests für Parameter und Modell

Mit `summary(⟨vglm-Modell⟩)` werden neben den Parameterschätzungen b auch ihre geschätzten Streuungen $\hat{\sigma}_b$ sowie die zugehörigen z -Werte $\frac{b}{\hat{\sigma}_b}$ berechnet. Mit der Annahme, dass z unter der Nullhypothese asymptotisch standardnormalverteilt ist, stehen die zweiseitigen p -Werte in

der Spalte `Pr(>|z|)`. Die genannten Werte lassen sich mit `coef(summary(<vglm-Modell>))` extrahieren. Profile-Likelihood Konfidenzintervalle erzeugt `confint(<vglm-Modell>, method="profile")`.

```
> sumMN <- summary(vglmFitMN)
> (coefMN <- coef(sumMN))
      Estimate Std. Error   z value Pr(>|z|)
(Intercept):1 -2.84520460 8.86963583 -0.3207803 7.483769e-01
(Intercept):2 -24.87104089 9.67909358 -2.5695630 1.018269e-02
(Intercept):3 -43.91627419 11.63358071 -3.7749576 1.600349e-04
X1:1           0.01931815 0.05232332  0.3692072 7.119733e-01
X1:2           0.15014353 0.05662132  2.6517138 8.008439e-03
X1:3           0.27502499 0.06780602  4.0560557 4.990836e-05
X2:1           -0.01467332 0.03810166 -0.3851098 7.001561e-01
X2:2           -0.03698110 0.04095569 -0.9029539 3.665504e-01
X2:3           -0.14534420 0.04957956 -2.9315347 3.372917e-03

> confint(vglmFitMN, method="profile")
      2.5 %    97.5 %
(Intercept):1 -20.56920320 14.58795777
(Intercept):2 -44.96340309 -6.70036945
(Intercept):3 -68.62626636 -22.65578477
X1:1           -0.08343008 0.12414067
X1:2           0.04419509 0.26805364
X1:3           0.15163779 0.41929882
X2:1           -0.09083943 0.06011880
X2:2           -0.11967388 0.04239173
X2:3           -0.24960539 -0.05352069
```

Da jeder Prädiktor mit mehreren Parametern β_{jg} assoziiert ist, müssen Prädiktoren selbst über Modellvergleiche auf Signifikanz getestet werden (Abschn. 8.1.5). Dazu dienen Likelihood-Quotienten-Tests, die auf der asymptotisch χ^2 -verteilten Devianz-Differenz zweier nested Modelle mit demselben Kriterium beruhen: Der Prädiktorensatz des eingeschränkten Modells `<fitR>` ist dabei vollständig im Prädiktorensatz des umfassenderen Modells `<fitU>` enthalten, das zusätzlich noch weitere Prädiktoren berücksichtigt. Der Test erfolgt dann mit `anova(<fitR>, <fitU>, type="I")`. Der p -Wert steht in der Ausgabe in der Spalte `Pr(>Chi)`.

```
# eingeschränktes Modell ohne Prädiktor X2
> vglmFitR <- vglm(Ycateg ~ X1, family=multinomial(refLevel=1),
+                      data=dfOrd)

> anova(vglmFitR, vglmFitMN, type="I")          # Modellvergleich
Analysis of Deviance Table
Model 1: Ycateg ~ X1
Model 2: Ycateg ~ X1 + X2
Resid. Df Resid. Dev Df Deviance Pr(>Chi)
1       294     249.00
2       291     237.15  3     11.852 0.007906 **
```

8.3.4 Vorhersage, Klassifikation und Anwendung auf neue Daten

Die vorhergesagten Kategorienwahrscheinlichkeiten $\hat{P}(Y = g)$ erhält man wie in der logistischen Regression mit `predict(<vglm-Modell>, type="response")`. Die ausgegebene Matrix enthält für jede Beobachtung (Zeilen) die vorhergesagte Wahrscheinlichkeit für jede Kategorie (Spalten). Mit der gewählten Herangehensweise ist die mittlere vorhergesagte Wahrscheinlichkeit für jede Kategorie gleich der empirischen relativen Häufigkeit der Kategorie.²¹

```
> PhatCateg <- predict(vglmFitMN, type="response")
> head(PhatCateg, n=3)
--      -      +      ++
1 0.1946834 0.2255730 0.2982816 0.28146203
2 0.3219964 0.3219154 0.2675170 0.08857120
3 0.4473732 0.3769135 0.1596527 0.01606071

# mittlere vorhergesagte Kategorien-Wahrscheinlichkeiten
> colMeans(PhatCateg)
--      -      +      ++
0.25 0.25 0.25 0.25

# empirische relative Kategorien-Häufigkeiten
> proportions(xtabs(~ Ycateg, data=dfOrd))
--      -      +      ++
0.25 0.25 0.25 0.25
```

Die vorhergesagten Kategorien selbst lassen sich aus den vorhergesagten Kategorienwahrscheinlichkeiten bestimmen, indem pro Beobachtung die Kategorie mit der maximalen vorhergesagten Wahrscheinlichkeit herangezogen wird. Den jeweiligen Spaltenindex des zeilenweisen Maximums einer Matrix gibt `max.col(<Matrix>)` aus.

```
> categHat <- levels(dfOrd$Ycateg)[max.col(PhatCateg)]
> head(categHat)
[1] "+"  "--"  "---" "+"  "--"  "+"
```

Die Kontingenztafel tatsächlicher und vorhergesagter Kategorien eignet sich als Grundlage für die Berechnung von Übereinstimmungsmaßen wie der Rate der korrekten Klassifikation. Hier ist darauf zu achten, dass die Kategorien identisch geordnet sind.

```
> facHat <- factor(categHat, levels=levels(dfOrd$Ycateg))
> cTab   <- table(dfOrd$Ycateg, facHat, dnn=c("Ycateg", "facHat"))
> addmargins(cTab)                      # Randsummen hinzufügen
                                         facHat
Ycateg  --  -  +  ++ Sum
--    9  5  8  3  25
-     11 5  5  4  25
+     5  5  8  7  25
```

²¹Dies ist der Fall, wenn die kanonische Link-Funktion und Maximum-Likelihood-Schätzungen der Parameter gewählt werden und das Modell die absoluten Terme β_{0g} besitzt.

| | | | | | |
|-----|----|----|----|----|-----|
| ++ | 1 | 2 | 8 | 14 | 25 |
| Sum | 26 | 17 | 29 | 28 | 100 |

```
> (CCR <- sum(diag(cTab)) / sum(cTab)) # Rate der korrekten Klass.
[1] 0.36
```

An das Argument `newdata` von `predict()` kann zusätzlich ein Datensatz übergeben werden, der neue Daten für Variablen mit denselben Namen, und bei Faktoren zusätzlich denselben Stufen wie jene der ursprünglichen Prädiktoren enthält. Als Ergebnis erhält man die vorhergesagten Kategorienwahrscheinlichkeiten für die neuen Prädiktorwerte (Abschn. 6.4).

```
> Nnew <- 3 # Anzahl neuer Daten
> dfNew <- data.frame(X1=rnorm(Nnew, 175, 7), # neue Daten
+ X2=rnorm(Nnew, 30, 8))

> predict(vglmFitMN, newdata=dfNew, type="response") # Vorhersage
-- - + ++
1 0.05150391 0.06762012 0.2943623 0.58651367
2 0.45228458 0.39748171 0.1311896 0.01904412
3 0.15948745 0.18932511 0.3110646 0.34012280
```

8.4 Regression für Zähldaten

Das GLM bietet verschiedene Möglichkeiten zur Modellierung einer Variable Y , die nur ganzzahlige nichtnegative Werte annehmen kann und keine obere Schranke aufweist, wie es für Zähldaten charakteristisch ist. Die von Y gezählten Ereignisse sollen dabei unabhängig voneinander eintreten. Handelt es sich bei den Prädiktoren X_j um kontinuierliche Variablen, spricht man von Regressionsmodellen, bei der Modellierung einer festen Gesamtzahl von Ereignissen ausschließlich durch Gruppierungsfaktoren meist von log-linearen Modellen (Abschn. 8.5).

8.4.1 Poisson-Regression

Bei der Poisson-Regression wird als bedingte Verteilung von Y eine Poisson-Verteilung $P(Y = y) = \frac{\mu^y e^{-\mu}}{y!}$ angenommen, mit dem Erwartungswert $\mu = E(Y)$. Die kanonische Link-Funktion ist der natürliche Logarithmus, das lineare Modell ist also $\ln \mu = \beta_0 + \beta_1 X_1 + \dots + \beta_p X_p = \mathbf{X}\boldsymbol{\beta}$. Die bedingten Verteilungen von Y sind dann bereits vollständig festgelegt, da ihre Varianz ebenfalls gleich μ ist. Der Erwartungswert ist als $\mu = e^{\mathbf{X}\boldsymbol{\beta}} = e^{\beta_0} e^{\beta_1 X_1} \dots e^{\beta_p X_p}$ identifizierbar. Einem exponenzierten Parameter e^{β_j} kommt deshalb die Bedeutung des multiplikativen Faktors zu, mit dem μ wächst, wenn sich der Prädiktor X_j um eine Einheit vergrößert.²²

Die Anpassung einer Poisson-Regression geschieht wie in der logistischen Regression mit `glm()`, wobei das Argument `family` auf `poisson(link="log")` zu setzen ist (Abschn. 8.1.1).²³ Die Daten der vorherzusagenden Variable müssen aus ganzzahligen Werten ≥ 0 bestehen.

²²Für einen Prädiktor X : $\mu_{X+1} = e^{\beta_0 + \beta_1(X+1)} = e^{\beta_0} e^{\beta_1(X+1)} = e^{\beta_0} e^{\beta_1 X} e^{\beta_1} = e^{\beta_1} e^{\beta_0 + \beta_1 X} = e^{\beta_1} \mu_X$.

²³Bei der Verwendung von `vglm()` aus dem Paket `VGAM` ist das Argument `family` auf `poissonff` zu setzen.

Für ein Beispiel sollen zunächst Zähldaten simuliert werden, die mit zwei Prädiktoren korrelieren. Dazu wird die `rmvnorm()` Funktion des `mvtnorm` Pakets verwendet, die Zufallsvektoren einer multinormalverteilten Variable simuliert. 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 Daten einer der erzeugten Variablen werden zusätzlich gerundet und ihre negativen Werte auf Null gesetzt, damit sie als Zählvariable dienen kann.

```
> library(mvtnorm)                                # für rmvnorm()
> N      <- 200
> sigma <- matrix(c(4,2,-3, 2,16,-1, -3,-1,8), byrow=TRUE, ncol=3)
> mu     <- c(-3, 2, 4)                         # Erwartungswerte
> XY     <- rmvnorm(N, mean=mu, sigma=sigma)    # Zufallsdaten
> Y      <- round(XY[, 3] - 1.5)                 # runde Zähldaten
> Y[Y < 0] <- 0                                  # negative Werte -> 0
> dfCount <- data.frame(X1=XY[, 1], X2=XY[, 2], Y) # Datensatz

# Poisson-Regression
> glmFitP <- glm(Y ~ X1 + X2, family=poisson(link="log"), data=dfCount)
```

Wie in der logistischen Regression erhält man die Parameterschätzungen b_j mit `coef()` und die Konfidenzintervalle für die Parameter β_j mit `confint()` (Abschn. 8.1.1, Fußnote 6). Die geschätzten Änderungsfaktoren für $E(Y)$ ergeben sich durch Exponenzieren als e^{b_j} .

```
> exp(coef(glmFitP))                            # Änderungsfaktoren
(Intercept)          X1           X2
1.1555238   0.7692037  1.0205773

> exp(confint(glmFitP))                         # zugehörige Konfidenzintervalle
      2.5 %    97.5 %
(Intercept) 0.9479450 1.3993779
X1          0.7380968 0.8015816
X2          0.9992731 1.0424328
```

Wald-Tests der Parameter und Informationen zur Modellpassung insgesamt liefert `summary()` (Abschn. 8.1.3, 8.1.5).

```
> summary(glmFitP)    # Parametertests + Modellpassung, gekürzte Ausgabe
Deviance Residuals:
    Min      1Q  Median      3Q      Max
-3.1695 -1.4608 -0.2707  0.7173  3.4757

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept) 0.14455   0.09935   1.455    0.146
X1         -0.26240   0.02105 -12.466   <2e-16 ***
X2          0.02037   0.01079   1.888    0.059 .
(Dispersion parameter for poisson family taken to be 1)
```

```
Null deviance: 536.93 on 199 degrees of freedom
Residual deviance: 374.78 on 197 degrees of freedom
AIC: 825.22
```

```
Number of Fisher Scoring iterations: 5
```

Wie in der logistischen Regression erhält man die vorhergesagten Häufigkeiten mit `predict(glm-Modell, type="response")` (Abschn. 8.1.4). Über das Argument `newdata` lässt sich das angepasste Modell dabei auch auf neue Werte für dieselben Prädiktoren anwenden, um Vorhersagen zu gewinnen.

8.4.2 Ereignisraten analysieren

Die Poisson-Regression erlaubt es auch, Ereignisraten zu analysieren. Die absoluten Häufigkeiten Y sind dann auf eine bestimmte Referenzgröße t bezogen, die die Menge der potentiell beobachtbaren Ereignisse bestimmt. Bei t kann es sich etwa um die Länge eines Zeitintervalls handeln, in dem Ereignisse gezählt werden. In einer anderen Situation kann t die Größe einer Fläche sein, auf der die Anzahl von Elementen mit einer bestimmten Eigenschaft zu zählen sind, die jeweils einem Ereignis entsprechen. t wird als *exposure* bezeichnet und ist echt positiv.

Bezeichnet t die Zeitdauer, nimmt man an, dass der Abstand zwischen zwei aufeinander folgenden Ereignissen unabhängig von t exponentialverteilt mit Erwartungswert $\frac{1}{\lambda}$ ist. Dann folgt $E(Y) = \lambda t$ mit der Konstante λ als echt positiver Ereignisrate, mit der ein Ereignis pro Zeiteinheit auftritt.

Ist in einer Untersuchung die zu den beobachteten Häufigkeiten Y gehörende exposure t variabel, stellt die Ereignisrate $\lambda = \frac{\mu}{t}$ die zu modellierende Variable dar. Mit dem Logarithmus als Link-Funktion folgt $\mathbf{X}\beta = \ln \lambda = \ln \frac{\mu}{t} = \ln \mu - \ln t$, als lineares Vorhersagemodell ergibt sich $\ln \mu = \mathbf{X}\beta + \ln t$. Im linearen Prädiktor $1 \ln t + \beta_0 + \beta_1 X_1 + \dots + \beta_p X_p$ ist $1 \ln t$ eine additive Konstante wie β_0 – allerdings mit dem Unterschied, dass t durch die Daten festgelegt ist und das zugehörige, auf 1 fixierte, Regressionsgewicht nicht geschätzt werden muss. $\ln t$ wird als *offset* bezeichnet. Die Ereignisrate λ ist identifizierbar als $\lambda = e^{\mathbf{X}\beta + \ln t} = t e^{\mathbf{X}\beta}$.

In der Anpassung der Poisson-Regression für Ereignisraten mit `glm()` ist das Argument `offset=log(t)` zu verwenden. Dabei ist `(t)` ein Vektor mit den Werten der exposure.

```
# simulierte Zähldaten, hier ohne Zusammenhang Prädiktor - Zielgröße
> Nt    <- 100                                # Beobachtungsobjekte
> Ti    <- sample(20:40, Nt, replace=TRUE)      # exposure
> Xt    <- rnorm(Nt, 100, 15)                  # Prädiktor
> Yt    <- rbinom(Nt, size=Ti, prob=0.5)        # beobachtete Häufigkeiten
> fitT <- glm(Yt ~ Xt, family=poisson(link="log"), offset=log(Ti))
> summary(fitT)                                # ...
```

8.4.3 Adjustierte Poisson-Regression und negative Binomial-Regression

Mitunter streuen empirische Residuen deutlich stärker, als dies bei bedingter Poisson-Verteilung von Y zu erwarten wäre, bei der die Streuung gleich dem bedingten Erwartungswert ist (*overdispersion*). Ein Hinweis auf overdispersion ist ein Verhältnis von Residual-Devianz zu Residual-Freiheitsgraden, das deutlich größer als 1 ist. Bei overdispersion kann ein Poisson-ähnliches Modell verwendet werden, das einen zusätzlichen, aus den Daten zu schätzenden Streuungsparameter ϕ besitzt, mit dem für die bedingte Varianz $\sigma^2 = \phi\mu$ gilt. Hierfür ist das Argument `family` von `glm()` auf `quasipoisson(link="log")` zu setzen. Dies führt zu identischen Parameterschätzungen b , jedoch zu anderen geschätzten Streuungen $\hat{\sigma}_b$ und damit zu anderen Ergebnissen der inferenzstatistischen Tests. Abschnitt 8.1.7 erläutert mögliche Ursachen von overdispersion und Konsequenzen der Verwendung von quasi-Familien.

```
> glmFitQP <- glm(Y ~ X1 + X2, family=quasipoisson(link="log"),
+                     data=dfCount)

> summary(glmFitQP)                      # gekürzte Ausgabe ...
Deviance Residuals:
    Min      1Q  Median      3Q      Max 
-3.1695 -1.4608 -0.2707  0.7173  3.4757 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) 0.14455   0.12943   1.117   0.265    
X1          -0.26240   0.02742  -9.569  <2e-16 ***  
X2          0.02037   0.01405   1.450   0.149    
                                                        
(Dispersion parameter for quasipoisson family taken to be 1.697012)

Null deviance: 536.93 on 199 degrees of freedom
Residual deviance: 374.78 on 197 degrees of freedom
AIC: NA
```

Für eine adjustierte Poisson-Regression kann alternativ auch ein separater robuster Streuungsschätzer verwendet werden, wie ihn etwa das Paket `sandwich` mit `vcovHC()` bereitstellt (Abschn. 6.6.1). Wald-Tests der Parameter lassen sich dann mit `coeftest(<glm-Modell>, vcov=<Schätzer>)` aus dem Paket `lmtest` durchführen, wobei für das Argument `vcov` das Ergebnis von `vcovHC()` anzugeben ist.

```
> library(sandwich)                      # für vcovHC()
> hcSE <- vcovHC(glmFitP, type="HC0")    # robuste SE-Schätzung
> library(lmtest)                        # für coeftest()
> coeftest(glmFitP, vcov=hcSE)           # Parametertests
z test of coefficients:

            Estimate Std. Error z value Pr(>|z|)    
(Intercept) 0.144554   0.135709  1.0652   0.2868    
X1          -0.262399   0.028753 -9.1259  <2e-16 ***
```

| | | | | |
|----|----------|----------|--------|--------|
| X2 | 0.020368 | 0.013097 | 1.5552 | 0.1199 |
|----|----------|----------|--------|--------|

Eine weitere Alternative ist die Regression mit der Annahme, dass die bedingte Verteilung von Y eine negative Binomialverteilung ist, die einen eigenen Dispersionsparameter θ besitzt. Sie verallgemeinert die Poisson-Verteilung mit demselben Erwartungswert μ , ihre Varianz ist jedoch mit $\mu + \frac{\mu^2}{\theta} = \mu(1 + \frac{\mu}{\theta})$ um den Faktor $1 + \frac{\mu}{\theta}$ größer. Die negative Binomial-Regression lässt sich mit `glm.nb()` aus dem Paket MASS anpassen.²⁴

```
> library(MASS)                                # für glm.nb()
> glmFitNB <- glm.nb(Y ~ X1 + X2, data=dfCount)
> summary(glmFitNB)                          # gekürzte Ausgabe ...
Deviance Residuals:
    Min      1Q  Median      3Q     Max 
-2.6146 -1.2540 -0.2149  0.5074  2.7341 

Coefficients:
            Estimate Std. Error z value Pr(>|z|)    
(Intercept) 0.003337  0.130007  0.026   0.9795    
X1          -0.302121  0.029972 -10.080  <2e-16 ***  
X2          0.025419  0.014649   1.735   0.0827 .  
                                                        
(Dispersion parameter for Negative Binomial(3.85) family taken to be 1)

Null deviance: 353.34 on 199 degrees of freedom
Residual deviance: 250.31 on 197 degrees of freedom
AIC: 801.11

Number of Fisher Scoring iterations: 1

Theta:  3.85
Std. Err.: 1.12
2 x log-likelihood: -793.113
```

Das Ergebnis nennt neben den bekannten Kennwerten unter `Theta` eine Schätzung für den Dispersionsparameter θ . Das von `glm.nb()` erzeugte Objekt kann an die Funktion `odTest()` aus dem Paket `pscl` (Zeileis, Kleiber & Jackman, 2008) übergeben werden. Sie führt dann einen Test auf Overdispersion durch, der auf dem Vergleich der Anpassungsgüte von Poisson- und negativem Binomial-Modell beruht.

```
> library(pscl)                                # für odTest()
> odTest(glmFitNB)                            # overdispersion-Test
Likelihood ratio test of H0: Poisson, as restricted NB model:
n.b., the distribution of the test-statistic under H0 is non-standard
e.g., see help(odTest) for details/references

Critical value of test statistic at the alpha= 0.05 level: 2.7055
Chi-Square Test Statistic = 26.1033 p-value = 1.618e-07
```

²⁴Bei der Verwendung von `vglm()` aus dem Paket VGAM ist das Argument `family` auf `negbinomial` zu setzen.

8.4.4 Zero-inflated Poisson-Regression

Eine mögliche Quelle für starke Streuungen von Zähldaten sind gehäuft auftretende Nullen. Sie können das Ergebnis eines zweistufigen Prozesses sein, mit dem die Daten zustande kommen. Wird etwa erfasst, wie häufig Personen im Beruf befördert werden oder ihren Arbeitsplatz wechseln, ist zunächst relevant, ob sie überhaupt jemals eine Arbeitsstelle erhalten haben. Diese Eigenschaft wird vermutlich durch qualitativ andere Prozesse bestimmt als jene, die bei tatsächlich Erwerbstätigen die Anzahl der unterschiedlichen Positionen beeinflussen. Die Daten können damit als Ergebnis einer Mischung von zwei Verteilungen verstanden werden: Zum einen kommen Nullen durch Personen zustande, die ein notwendiges Kriterium für einen auch zu echt positiven Häufigkeiten führenden Prozess nicht erfüllen. Zum anderen verursachen Personen, die ein solches Kriterium erfüllen, ihrerseits Nullen sowie zusätzlich echt positive Häufigkeiten. Für die Daten dieser zweiten Gruppe von Personen kann nun wieder angenommen werden, dass sie sich durch eine Poisson- oder negative Binomialverteilung beschreiben lassen.

In der geschilderten Situation kommen *zero-inflated* Modelle in Betracht. Sie sorgen durch die Annahme einer Mischverteilung letztlich dafür, dass im Modell eine deutlich höhere Auftretenswahrscheinlichkeit von Nullen möglich ist, als es zur Verteilung der echt positiven Häufigkeiten passt.

Die zero-inflated Poisson-Regression eignet sich für Situationen, in denen keine starke overdispersion zu vermuten ist und kann mit `zeroinfl()` aus dem Paket `pscl` angepasst werden (für Details vgl. `vignette("countreg")`).²⁵

```
zeroinfl(<Modellformel>, dist=<Verteilung>, offset=<offset>,
         data=<Datensatz>)
```

Die Modellformel besitzt hier die Form $\langle AV \rangle \sim \langle UV \rangle \mid \langle 0\text{-Anteil} \rangle$. Für $\langle 0\text{-Anteil} \rangle$ ist ein Prädiktor zu nennen, der im Modell separat den Anteil der „festen“ Nullen an den Daten kontrolliert, die von jenen Personen stammen, die prinzipiell keine echt positiven Daten liefern können. Im einfachsten Fall ist dies der absolute Term 1, der zu einem Binomialmodell passt, das für alle Beobachtungen dieselbe Wahrscheinlichkeit vorsieht, zu dieser Gruppe zu gehören. In komplizierteren Situationen könnte die Gruppenzugehörigkeit analog zu einer separaten logistischen Regression durch einen eigenen Prädiktor vorhergesagt werden. Das Argument `dist` ist auf "poisson" zu setzen, `data` erwartet einen Datensatz mit den Variablen aus der Modellformel. Wie bei `glm()` existiert ein Argument `offset` für die Analyse von Ereignisraten (Abschn. 8.4.2).

```
> library(pscl)                                     # für zeroinfl()
> ziFitP <- zeroinfl(Y ~ X1 + X2 | 1, dist="poisson", data=dfCount)
> summary(ziFitP)                                 # gekürzte Ausgabe ...
Pearson residuals:
    Min     1Q   Median     3Q     Max 
-1.4805 -0.9447 -0.1574  0.6397  3.8149 

Count model coefficients (poisson with log link):
Estimate Std. Error z value Pr(>|z|)
```

²⁵Bei der Verwendung von `vglm()` aus dem Paket `VGAM` ist das Argument `family` auf `zipoissonff` zu setzen.

```
(Intercept) 0.50809   0.13063   3.890   0.0001 ***
X1          -0.20443   0.02658  -7.692  1.45e-14 ***
X2          0.01781   0.01156   1.541   0.1233
```

Zero-inflation model coefficients (binomial with logit link):

| | Estimate | Std. Error | z value | Pr(> z) |
|-------------|----------|------------|---------|--------------|
| (Intercept) | -1.4957 | 0.2627 | -5.693 | 1.25e-08 *** |

Number of iterations in BFGS optimization: 10

Log-likelihood: -393.1 on 4 Df

Separate Wald-Tests der Parameter lassen sich mit `coeftest()` aus dem Paket `lmtest` durchführen, Likelihood-Quotienten-Tests für nested Modelle können mit `lrtest()` aus demselben Paket vorgenommen werden.

Die zero-inflated negative Binomial-Regression kann in Situationen mit gehäuft auftretenden Nullen und einer deutlichen overdispersion zum Einsatz kommen. Auch für sie eignet sich `zeroinfl()`, wobei das Argument `dist="negbin"` zu setzen ist.²⁶

```
> ziFitNB <- zeroinfl(Y ~ X1 + X2 | 1, dist="negbin", data=dfCount)
```

```
> summary(ziFitNB)
```

Pearson residuals:

| Min | 1Q | Median | 3Q | Max |
|---------|---------|---------|--------|--------|
| -1.4548 | -0.9007 | -0.1515 | 0.6482 | 3.9289 |

Count model coefficients (negbin with log link):

| | Estimate | Std. Error | z value | Pr(> z) |
|-------------|----------|------------|---------|-----------|
| (Intercept) | 0.36993 | 0.17740 | 2.085 | 0.0370 * |
| X1 | -0.23375 | 0.03738 | -6.254 | 4e-10 *** |
| X2 | 0.02111 | 0.01320 | 1.599 | 0.1098 |
| Log(theta) | 2.65338 | 0.81911 | 3.239 | 0.0012 ** |

Zero-inflation model coefficients (binomial with logit link):

| | Estimate | Std. Error | z value | Pr(> z) |
|-------------|----------|------------|---------|--------------|
| (Intercept) | -1.7486 | 0.3924 | -4.457 | 8.33e-06 *** |

Theta = 14.2019

Number of iterations in BFGS optimization: 20

Log-likelihood: -392 on 5 Df

Der Vuong-Test vergleicht die Anpassungsgüte eines von `glm()` erstellten Poisson-Modells mit jener des durch `zeroinfl()` erstellten zero-inflated Poisson-Modells und kann mit `vuong()` aus dem Paket `pscl` durchgeführt werden. Analog testet `vuong()` auch ein mit `glm.nb()` angepasstes negatives Binomial-Modell gegen das zero-inflated negative Binomial-Modell aus `zeroinfl()`.

²⁶Bei der Verwendung von `vglm()` aus dem Paket `VGAM` ist das Argument `family` auf `zinegbinomial` zu setzen.

```
# Poisson-Modell vs. zero-inflated Poisson-Modell
> library(pscl) # für vuong()
> vuong(ziFitP, glmFitP)
Vuong Non-Nested Hypothesis Test-Statistic: -0.4582521
(test-statistic is asymptotically distributed N(0,1) under the
null that the models are indistinguishable)
in this case:
model2 > model1, with p-value 0.32339

# negatives Binomial- vs. zero-inflated negatives Binomial-Modell
> vuong(ziFitNB, glmFitNB)
Vuong Non-Nested Hypothesis Test-Statistic: -1.382096
(test-statistic is asymptotically distributed N(0,1) under the
null that the models are indistinguishable)
in this case:
model2 > model1, with p-value 0.083471
```

Eine Alternative zu zero-inflated Modellen bei gehäuft auftretenden Nullen ist die Hurdle-Regression, die von `hurdle()` aus dem Paket `pscl` umgesetzt wird.

8.4.5 Zero-truncated Poisson-Regression

Im Gegensatz zum Umstand gehäuft auftretender Nullen erzwingt der Aufbau mancher Untersuchungen, dass *keine* Nullen beobachtet werden können. Dies ist etwa der Fall, wenn mindestens ein Ereignis vorliegen muss, damit eine Untersuchungseinheit in der Erhebung berücksichtigt wird: So ließe sich an bereits stationär aufgenommenen Patienten die Anzahl der Tage erheben, die sie im Krankenhaus verbringen, oder es könnte die Anzahl geborener Hundewelpen eines Wurfes erfasst werden. Zur Modellierung solcher Variablen kommen beschränkte (*zero-truncated*) Verteilungen in Frage, die den Wert 0 nur mit Wahrscheinlichkeit Null annehmen. Dazu zählen die zero-truncated Poisson-Verteilung – für Situationen ohne overdispersion – sowie die zero-truncated negative Binomialverteilung – bei overdispersion. Um sie bei der Modellanpassung mit `vglm()` aus dem Paket `VGAM` zu verwenden, ist das Argument `family` auf `pospoisson` bzw. auf `posnegbinomial` zu setzen.

8.5 Log-lineare Modelle

Log-lineare Modelle analysieren den Zusammenhang mehrerer kategorialer Variablen auf Basis ihrer gemeinsamen Häufigkeitsverteilung (Agresti, 2018). Sie sind für Daten geeignet, die sich als mehrdimensionale Kontingenztafeln absoluter Häufigkeiten darstellen lassen und verallgemeinern damit klassische nonparametrische Tests auf Unabhängigkeit bzw. auf Gleichheit von bedingten Verteilungen (Abschn. 10.2, 10.3.2).

8.5.1 Modell

Aus Sicht der Datenerhebung können Kontingenztafeln absoluter Häufigkeiten aus drei Situationen entstehen. Sie sollen hier an einer $(P \times Q)$ -Kontingenztafel der kategorialen Variablen X_1 und X_2 erläutert werden.

| | $x_{2,1}$ | \dots | $x_{2,q}$ | \dots | $x_{2,Q}$ | Summe |
|-----------|-----------|----------|-----------|----------|-----------|----------|
| $x_{1,1}$ | f_{11} | \dots | f_{1k} | \dots | f_{1q} | f_{1+} |
| \vdots | \vdots | \ddots | \vdots | \ddots | \vdots | \vdots |
| $x_{1,p}$ | f_{j1} | \dots | f_{jk} | \dots | f_{jq} | f_{p+} |
| \vdots | \vdots | \ddots | \vdots | \ddots | \vdots | \vdots |
| $x_{1,P}$ | f_{p1} | \dots | f_{pk} | \dots | f_{pq} | f_{P+} |
| Summe | f_{+1} | \dots | f_{+q} | \dots | f_{+Q} | N |

- Im *produkt-multinomialen* Erhebungsschema ist X_1 ein fester Faktor mit vorgegebenen Gruppenhäufigkeiten f_{p+} , aus deren Summe sich die feste Anzahl N von Beobachtungsobjekten ergibt – etwa in einem Experiment mit kontrollierter Zuweisung zu Versuchsbedingungen. Erhoben wird die kategoriale Variable X_2 , deren Verteilung über die Gruppen j i. S. der Auftretenshäufigkeiten ihrer Kategorien f_{+q} zufällig ist.
- Im *multinomialen* Erhebungsschema werden mehrere kategoriale Variablen X_j gleichzeitig an einer festen Anzahl N von Beobachtungsobjekten erhoben, etwa im Rahmen einer Befragung. Die Kontingenztafel entsteht aus der Kreuzklassifikation der X_j und gibt die Häufigkeiten an, mit der Kombinationen ihrer Kategorien auftreten. Dabei sind die f_{p+} und f_{+q} zufällig. In diesem Schema werden keine Einflussbeziehungen der X_j auf eine separate Variable untersucht, sondern Zusammenhänge zwischen den X_j selbst.
- Im *Poisson*-Erhebungsschema wird gleichzeitig mit den Ausprägungen kategorialer Einflussgrößen X_j eine separate Variable Y als Anzahl bestimmter Ereignisse erhoben. Die Kontingenztafel stellt die Anzahl der Ereignisse in den Gruppen dar, die durch die Kombination der Stufen der X_j entstehen. Dabei ist die Gesamthäufigkeit der Ereignisse N wie auch ihre Verteilung in den Gruppen der Häufigkeiten f_{p+} und f_{+q} zufällig. In diesem Schema nimmt man eine Poisson-Verteilung für N an, womit jedes Ereignis unabhängig voneinander eintritt und als einzelne Beobachtung der Merkmalskombinationen der X_j zählt. Bedingt auf N liegt dann wieder ein multinomiales Schema vor.

Das log-lineare Modell ist ein lineares Modell für $\ln \mu_{jk}$, den logarithmierten Erwartungswert einer Zellhäufigkeit f_{jk} . Formuliert als Regression mit Treatment-Kontrasten hat es dieselbe Form wie das der Poisson-Regression (Abschn. 8.4.1). Es ist jedoch üblicher, es wie die mehrfaktorielle Varianzanalyse (Abschn. 7.6) mit Effektcodierung (Abschn. 12.9.2) zu parametrisieren. Mit der Stichprobengröße N , der Zellwahrscheinlichkeit p_{jk} und den Randwahrscheinlichkeiten p_{p+}, p_{+q} analog zu den Häufigkeiten gilt dafür zunächst:

$$\begin{aligned} \mu_{jk} &= N p_{jk} = N p_{p+} p_{+q} \frac{p_{jk}}{p_{p+} p_{+q}} \\ \ln \mu_{jk} &= \ln N + \ln p_{jk} = \ln N + \ln p_{p+} + \ln p_{+q} + (\ln p_{jk} - (\ln p_{p+} + \ln p_{+q})) \end{aligned}$$

Das Modell für $\ln \mu_{jk}$ hat nun für eine zweidimensionale Kontingenztafel dieselbe Form wie jenes der zweifaktoriellen Varianzanalyse mit Effektcodierung. Dabei ist $\ln N$ analog zu μ , $\ln p_{p+}$ analog zu α_j (Zeileneffekt von Gruppe j des Faktors X_1), $\ln p_{+q}$ analog zu β_k (Spalteneffekt von Stufe k des Faktors X_2) und $\ln p_{jk} - (\ln p_{p+} + \ln p_{+q})$ analog zu $(\alpha\beta)_{jk}$ (Interaktionseffekt).²⁷

$$\begin{aligned}\mu_{jk} &= e^\mu e^{\alpha_j} e^{\beta_k} e^{(\alpha\beta)_{jk}} \\ \ln \mu_{jk} &= \mu + \alpha_j + \beta_k + (\alpha\beta)_{jk}\end{aligned}$$

Als Abweichung der logarithmierten Zellwahrscheinlichkeit von Additivität der logarithmierten Randwahrscheinlichkeiten drückt der Interaktionseffekt $(\alpha\beta)_{jk}$ den Zusammenhang von X_1 und X_2 aus, da bei Unabhängigkeit $p_{jk} = p_{p+} p_{+q}$ gilt – logarithmiert also $\ln p_{jk} = \ln p_{p+} + \ln p_{+q}$. In zweidimensionalen Kreuztabellen sind alle beobachtbaren Zellhäufigkeiten mit dem vollständigen Modell (Zeilen-, Spalten- und Interaktionseffekt) verträglich, das deswegen als *saturiert* bezeichnet wird und sich nicht testen lässt. Dagegen sind eingeschränkte Modelle wie das der Unabhängigkeit statistisch prüfbar. Für höherdimensionale Kontingenztafeln gilt dies analog, wobei kompliziertere Abhängigkeitsbeziehungen durch die Interaktionen erster und höherer Ordnung ausgedrückt werden können.

Für den Spezialfall zweidimensionaler Kreuztabellen ist das beschriebene Unabhängigkeitsmodell im multinomialen Erhebungsschema dasselbe, das vom χ^2 -Test auf Unabhängigkeit geprüft wird (Abschn. 10.2.1). Im produkt-multinomialen Erhebungsschema ist es analog dasselbe, das der χ^2 -Test auf Gleichheit von bedingten Verteilungen testet (Abschn. 10.2.2).

8.5.2 Modellanpassung mit `loglm()`

Log-lineare Modelle lassen sich mit der Funktion `loglm()` aus dem Paket MASS testen, in der die Modelle analog zu `lm()` formuliert werden können.

```
loglm(<Modellformel>, data=<Kreuztabelle>)
```

Stammen die für `data` zu übergebenden Daten aus einer Kreuztabelle absoluter Häufigkeiten, kann das erste Argument eine Modellformel ohne Variable links der `~` sein. Besitzt die Kreuztabelle Namen für Zeilen und Spalten, sind diese mögliche Vorhersageterme in der Modellformel. Alternativ stehen die Ziffern 1, 2, ... für die Effekte der Zeilen, Spalten, etc.

Als Beispiel soll die $(2 \times 2 \times 6)$ -Kreuztabelle `UCBAdmissions` dienen, die im Basisumfang von R enthalten ist. Aufgeschlüsselt nach Geschlecht (Variable `Gender`) vermerkt sie die Häufigkeit, mit der Bewerber für die sechs größten Fakultäten (Variable `Dept`) an der University of California Berkeley 1973 angenommen bzw. abgelehnt wurden (Variable `Admit`).

```
> str(UCBAdmissions)          # Struktur der Kreuztabelle
table [1:2, 1:2, 1:6] 512 313 89 19 353 207 17 8 120 205 ...
- attr(*, "dimnames")=List of 3
```

²⁷ Anders als in der Varianzanalyse gibt es jedoch im log-linearen Modell nur eine Beobachtung pro Zelle, die Rolle der abhängigen Variable der Varianzanalyse hat im log-linearen Modell die logarithmierte Auftretenshäufigkeit der zur Zelle gehörenden Kombination von Faktorstufen.

```
..$ Admit : chr [1:2] "Admitted" "Rejected"
..$ Gender: chr [1:2] "Male" "Female"
..$ Dept  : chr [1:6] "A" "B" "C" "D" ...
# teste Modell der vollständigen Unabhängigkeit = Additivität
> (llFit <- loglm(~ Admit + Dept + Gender, data=UCBAdmissions))
Statistics:
X^2 df P(> X^2)
Likelihood Ratio 2097.671 16      0
Pearson         2000.328 16      0
```

Abschnitt 8.5.3 zeigt, wie Standardfehler und Konfidenzintervalle berechnet werden können.

Im Aufruf von `loglm()` kann `data` auch ein Datensatz sein, der Spalten für die kategorialen Variablen X_j sowie für die Auftretenshäufigkeit Y jeder ihrer Stufenkombination besitzt (Abschn. 2.12.5). Dann bildet Y die linke Seite der Modellformel und eine Kombination der X_j die rechte Seite.

```
> UCBAdf <- as.data.frame(UCBAdmissions) # Kontingenztafel -> Datensatz
> head(UCBAdf, n=3)
  Admit Gender Dept Freq
1 Admitted   Male    A  512
2 Rejected   Male    A  313
3 Admitted Female   A   89

> loglm(Freq ~ Admit + Dept + Gender, data=UCBAdf)      # äquivalent ...
```

Die Ausgabe von `loglm()` nennt die Kennwerte des Likelihood-Quotienten-Tests sowie des Pearson χ^2 -Tests der Hypothese, dass das angegebene Modell stimmt. Der Wert der Teststatistik steht in der Spalte X^2 , die zugehörigen Freiheitsgrade unter `df` und der p -Wert unter `P(> X^2)`. Mit Hilfe von `coef(loglm-Modell)` erhält man zusätzlich Schätzungen für die durch die Modellformel spezifizierten Koeffizienten α_j, β_k, \dots . Dabei ist zu beachten, dass `loglm()` die Effektcodierung verwendet, die geschätzten Zeilen-, Spalten- und Schichten-Parameter summieren sich also pro Variable zu 0 (Abschn. 12.9.2).

```
> (llCoef <- coef(llFit))
$`(Intercept)`
[1] 5.177567
```

```
$Admit
  Admitted  Rejected
-0.2283697  0.2283697
```

```
$Gender
  Male     Female
  0.1914342 -0.1914342
```

```
$Dept
```

| | | | | | |
|------------|-------------|------------|------------|-------------|-------------|
| A | B | C | D | E | F |
| 0.23047857 | -0.23631478 | 0.21427076 | 0.06663476 | -0.23802565 | -0.03704367 |

Die gemeinsamen Häufigkeiten mehrerer kategorialer Variablen lassen sich mit `mosaic()` aus dem Paket `vcd` in einem Mosaik-Diagramm darstellen, einer Erweiterung des `splineplot` für die gemeinsame Verteilung zweier kategorialer Variablen (Abschn. 14.4.2). Zusammen mit dem Argument `shade=TRUE` sind die Argumente dieselben wie für `loglm()`, die zellenweisen Pearson-Residuen bzgl. des durch die Modellformel definierten Models werden dann farbcodiert dargestellt (Abb. 8.5).

```
# Mosaik-Diagramm der Häufigkeiten und Pearson-Residuen
> library(vcd) # für mosaic()
> mosaic(~ Admit + Dept + Gender, shade=TRUE, data=UCBAdmissions)
```

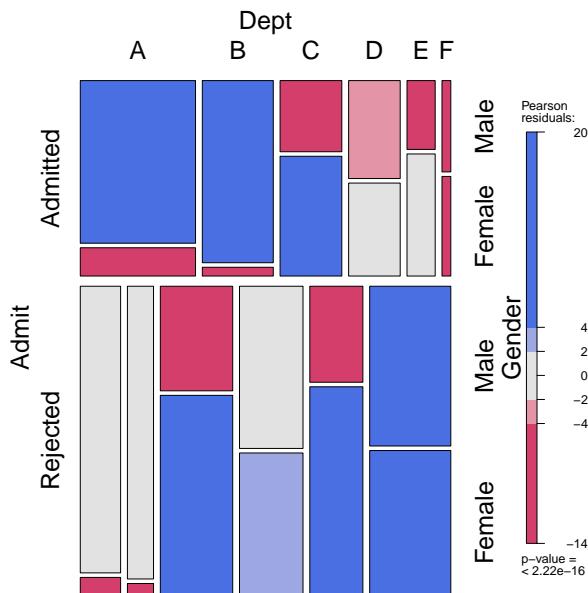


Abbildung 8.5: Mosaik-Diagramm der gemeinsamen Häufigkeiten der Kontingenztafel `UCBAdmissions` inkl. Farbcodierung der Pearson-Residuen vom Modell der Unabhängigkeit.

Wie in der Poisson-Regression erhält man mit `residuals(<loglm-Modell>, type="<Typ>")` die Residuen, hier allerdings nicht pro Beobachtung, sondern pro Zelle der Kontingenztafel. Mit `type="pearson"` sind dies die Pearson-Residuen, deren Quadratsumme gleich der Pearson-Teststatistik in der Ausgabe von `loglm()` ist. Für zweidimensionale Kontingenztafeln ist diese Quadratsumme gleich der Teststatistik des χ^2 -Tests auf Unabhängigkeit mit `chisq.test()` (Abschn. 10.2.1). Analog erhält man mit `type="deviance"` die Devianz-Residuen, deren Quadratsumme gleich der Likelihood-Ratio-Teststatistik in der Ausgabe von `loglm()` ist.

```
> sum(residuals(llFit, type="deviance"))^2
[1] 2097.671

> sum(residuals(llFit, type="pearson"))^2
[1] 2000.328
```

8.5.3 Modellanpassung mit `glm()`

Anders als in der Poisson-Regression liefert ein mit `loglm()` angepasstes Modell keine Standardfehler der Parameterschätzungen. Ebenso fehlt die Möglichkeit, Konfidenzintervalle mit `confint()` zu ermitteln. Um beides zu erhalten, lässt sich das Modell jedoch auch als Poisson-Regression mit `glm()` formulieren. Da `glm()` in der Voreinstellung Treatment-Kontraste verwendet, sind die Parameterschätzungen zunächst andere, mit Effektcodierung jedoch dieselben. Vorhersage und Residuen stimmen mit der des log-linearen Modells überein.

```
# Poisson-Regression mit Treatment-Kontrasten
> glmFitT <- glm(Freq ~ Admit+Dept+Gender, family=poisson(link="log"),
+                   data=UCBAdf)

> (glmTcoef <- coef(glmFitT))                                # Parameterschätzungen
Coefficients:
(Intercept)  AdmitRejected      DeptB      DeptC      DeptD      DeptE
      5.37111       0.45674   -0.46679   -0.01621   -0.16384   -0.46850

      DeptF  GenderFemale
      -0.26752      -0.38287

# prüfe Gleichheit der Vorhersage mit log-linearem Modell
> all.equal(c(fitted(llFit)), fitted(glmFitT), check.attributes=FALSE)
[1] TRUE
```

Auch die Teststatistik des Likelihood-Quotienten-Tests zum saturierten Modell ist im Prinzip identisch. Da `loglm()` und `glm()` jedoch andere numerische Optimierungsverfahren zur Maximum-Likelihood-Schätzung verwenden, sind kleine Abweichungen möglich.

```
# saturiertes Modell mit allen Interaktionen
> glmFitT_sat <- glm(Freq ~ Admit*Dept*Gender,
+                      family=poisson(link="log"), data=UCBAdf)

> anova(glmFitT, glmFitT_sat)
Analysis of Deviance Table
Model 1: Freq ~ Admit + Dept + Gender
Model 2: Freq ~ Admit * Dept * Gender
  Resid. Df Resid. Dev Df Deviance Pr(>Chi)
1          16    2097.7
2          0     0.0 16    2097.7 < 2.2e-16
```

Bei Treatment-Kontrasten ist der absolute Term gleich der Vorhersage in der Zelle, die sich als Kombination aller Referenzkategorien der beteiligten Faktoren (in der Voreinstellung jeweils die erste Faktorstufe) ergibt. Die geschätzten Koeffizienten geben jeweils die geschätzte Abweichung für eine Faktorstufe von der Referenzkategorie an.

```
# Umrechnung mu_jk Effektcodierung aus loglm() in Treatment-Kontraste
# Zelle aus Kombination der Referenz-Kategorien
> glmTcoef["(Intercept)"]                                # Treatment-Kontraste
```

```
(Intercept)
5.37111

> llCoef$`(Intercept)` + llCoef$Admit["Admitted"] + # Effektcodierung
+     llCoef$Gender["Male"] + llCoef$Dept["A"]
Admitted
5.37111

# mu_jk für Admit = "Admit", Gender = "Female", Dept = "C"
# Treatment-Kontraste
> glmTcoef[["(Intercept)"]] + glmTcoef[["DeptC"]]+glmTcoef[["GenderFemale"]]
(Intercept)
4.972034

> llCoef$`(Intercept)` + llCoef$Admit["Admitted"] + # Effektcodierung
+     llCoef$Dept["C"] + llCoef$Gender["Female"]
Admitted
4.972034
```

Auch mit `glm()` lassen sich die Parameter mit Effektcodierung schätzen (Abschn. 7.6.2). Dabei fehlt die Parameterschätzung für die jeweils letzte Faktorstufe in der Ausgabe, da sie sich aus der Nebenbedingung ergibt, dass sich die Parameter pro Faktor zu Null summieren. Die Standardfehler der Parameterschätzungen sowie den zugehörigen Wald-Test extrahiert man mit `coef(summary(glm-Modell))`. Die Einträge für den absoluten Term `(Intercept)` sind hier ohne Bedeutung, da $\ln N$ im log-linearen Modell fest vorgegeben ist und nicht geschätzt werden muss. Konfidenzintervalle lassen sich wie gewohnt mit `confint()` ermitteln.

```
# Poisson-Regression mit Effektcodierung
> glmFitE <- glm(Freq ~ Admit+Dept+Gender, family=poisson(link="log"),
+                   contrasts=list(Admit=contr.sum,
+                                     Dept=contr.sum,
+                                     Gender=contr.sum), data=UCBAdf)

# Parameterschätzungen mit Standardfehlern und Wald-Tests
> coef(summary(glmFitE))    # identisch zu loglm() Parameterschätzungen
   Estimate Std. Error   z value Pr(>|z|)
(Intercept) 5.17756677 0.01577888 328.132716 0.000000e+00
Admit1      -0.22836970 0.01525343 -14.971696 1.124171e-50
Dept1        0.23047857 0.03071783   7.503086 6.233240e-14
Dept2        -0.23631478 0.03699431  -6.387869 1.682136e-10
Dept3        0.21427076 0.03090740   6.932668 4.129780e-12
Dept4        0.06663476 0.03272311   2.036321 4.171810e-02
Dept5        -0.23802565 0.03702165  -6.429363 1.281395e-10
Gender1      0.19143420 0.01513732   12.646505 1.169626e-36

# Konfidenzintervalle für die Relativen Raten
> exp(confint(glmFitE))    # ...
```

Kapitel 9

Survival-Analyse

Die Survival-Analyse modelliert Überlebenszeiten (Hosmer Jr, Lemeshow & May, 2008; Klein & Moeschberger, 2003). Diese geben allgemein an, wieviel Zeit bis zum Eintreten eines bestimmten Ereignisses verstrichen ist und sollen hier deshalb gleichbedeutend mit *Ereigniszeiten* sein. Es kann sich dabei etwa um die Zeidauer handeln, die ein Patient nach einer Behandlung weiter am Leben ist, um die verstrichene Zeit, bis ein bestimmtes Bauteil im Gebrauch einen Defekt aufweist, oder um die Dauer, die ein Kleinkind benötigt, um ein vordefiniertes Entwicklungsziel zu erreichen – z. B. einen Mindestwortschatz besitzt. Bei der Analyse von Ereigniszeiten kann sowohl die Form ihres grundsätzlichen Verlaufs von Interesse sein, als auch inwiefern ihr Verlauf systematisch von Einflussgrößen abhängt.

Die folgenden Auswertungen verwenden Funktionen des Pakets `survival`, das im Basisumfang von R enthalten ist. Seine Anwendung wird vertiefend in Harrell Jr (2015) behandelt.

9.1 Verteilung von Ereigniszeiten

Ereigniszeiten lassen sich äquivalent durch verschiedene Funktionen beschreiben, die jeweils andere Eigenschaften ihrer Verteilung hervortreten lassen: Die Ereigniszeit selbst sei T mit Werten ≥ 0 und einer – hier als stetig vorausgesetzten – Dichtefunktion $f(t)$.¹ Die zugehörige Verteilungsfunktion $F(t) = P(T \leq t)$ liefert die Wahrscheinlichkeit, mit der T höchstens den Wert t erreicht. Die monoton fallende Survival-Funktion $S(t) = P(T > t) = 1 - F(t)$ gibt an, mit welcher Wahrscheinlichkeit die Ereigniszeit größer als t ist. Dabei wird $S(0) = 1$ vorausgesetzt, zum Zeitpunkt $t = 0$ soll das Ereignis also noch nicht eingetreten sein. Schließlich drückt die Hazard-Funktion $\lambda(t)$ die unmittelbare Ereignisrate zum Zeitpunkt t aus.

$$\lambda(t) = \lim_{\Delta_t \rightarrow 0^+} \frac{P(t \leq T < t + \Delta_t | T \geq t)}{\Delta_t} = \frac{P(t \leq T < t + \Delta_t) / \Delta_t}{P(T > t)} = \frac{f(t)}{S(t)}, \quad t \geq 0$$

$\lambda(t)$ ist bei kleiner werdender Intervallbreite Δ_t der Grenzwert für die bedingte Wahrscheinlichkeit pro Zeiteinheit, dass das Ereignis unmittelbar eintritt, wenn es bis zum Zeitpunkt t noch nicht eingetreten ist. Bezeichnet $\# \text{Personen} | \text{Ereignis} \in [t, t + \Delta_t)$ die Anzahl der Personen, bei denen das Ereignis im Intervall $[t, t + \Delta_t)$ eintritt und $\# \text{Personen} | \text{Ereignis} \geq t$ die

¹ $f(t) = \lim_{\Delta_t \rightarrow 0^+} \frac{P(t \leq T < t + \Delta_t)}{\Delta_t}$ in Abhängigkeit von der Zeit t und der Intervallbreite Δ_t .

Anzahl der Personen, bei denen das Ereignis zum Zeitpunkt t noch eintreten kann, lässt sich das hazard auf empirischer Ebene wie folgt formulieren:²

$$\hat{\lambda}(t) = \frac{\#\text{Personen} \mid \text{Ereignis} \in [t, t + \Delta_t)}{\#\text{Personen} \mid \text{Ereignis} \geq t} \cdot \frac{1}{\Delta_t}$$

$\hat{\lambda}(t)$ gibt also an, bei welchem Anteil verbleibender Personen ohne Ereignis bis zum Zeitpunkt t das Ereignis pro Zeiteinheit eintritt. Bei Werten der monoton steigenden kumulativen Hazard-Funktion $\Lambda(t) = -\ln S(t)$ handelt es sich um das bis zum Zeitpunkt t kumulierte Risiko, dass sich das Ereignis unmittelbar ereignet. Umgekehrt gilt $S(t) = e^{-\Lambda(t)}$.

Hazard- und Survival-Funktion bedingen einander. Nimmt man eine über die Zeit konstante Ereignisrate $\lambda(t) = \frac{1}{b}$ an (wie etwa beim radioaktiven Zerfall), impliziert dies eine exponentiell verteilte Ereigniszeit T (Abschn. 9.5). Die bedingte Wahrscheinlichkeit, dass ein noch nicht eingetretenes Ereignis unmittelbar auf t folgt, wäre damit unabhängig von der bereits verstrichenen Zeit t . Mit der Annahme, dass die logarithmierte Ereignisrate linear von t abhängt ($\ln \lambda(t) = \alpha + \rho t$ mit y -Achsenabschnitt α und Steigung ρ), ergibt sich entsprechend eine Gompertz-Verteilung von T . Analog führt die Annahme, dass die logarithmierte Ereignisrate linear mit der logarithmierten Zeit zusammenhängt ($\ln \lambda(t) = \alpha + \rho \ln t$), zu einer Weibull-Verteilung von T (Abschn. 9.5). Bei einem positivem ρ würde in beiden Fällen das hazard mit der Zeit ansteigen, was oft in Situationen angemessen ist, in denen das Eintreten des Ereignisses mit kontinuierlichen Reifungs- oder Abnutzungsprozessen zusammenhängt.

9.2 Zensierte und gestützte Ereigniszeiten

Um festzustellen, wann ein Zielereignis eintritt, werden die untersuchten Objekte über eine gewisse Zeit hinweg beobachtet – etwa wenn bei aus einer stationären Behandlung entlassenen Patienten mit Substanzmissbrauch erhoben wird, ob sich innerhalb eines Zeitraums ein Rückfall ereignet. Meist weisen empirisch erhobene Überlebenszeiten dabei die Besonderheit auf, dass von einigen Beobachtungseinheiten die Zeit bis zum Eintreten des Ereignisses unbekannt bleibt, was spezialisierte statistische Modelle notwendig macht.

Der Erhebungszeitraum ist oft begrenzt, so dass nicht für alle Untersuchungseinheiten das Ereignis auch tatsächlich innerhalb des Beobachtungszeitraums auftritt. Für solche *rechts-zensierte* Daten ist also nur bekannt, dass die Überlebenszeit den letzten Beobachtungszeitpunkt überschreitet, nicht aber ihr exakter Wert. Eine andere Ursache für rechts-zensierte Daten kann ein frühzeitiger dropout aus der Studie sein. *Links-zensierte* Daten entstehen, wenn das Ereignis bekanntermaßen bereits an einem unbekannten Zeitpunkt vor Beginn der Erhebung eingetreten ist. Daten werden als *links-gestützt* bezeichnet, wenn sich das Ereignis bei manchen potentiellen Beobachtungseinheiten bereits vor Erhebungsbeginn ereignet und sie deswegen nicht mehr in der Studie berücksichtigt werden können. Während die Häufigkeit zensierter Beobachtungen in der Stichprobe bekannt ist, fehlt über gestützte Daten jede Information.

² $\#\text{Personen} \mid \text{Ereignis} \geq t$ ist die Größe des *risk set* bzw. die Anzahl der Beobachtungsobjekte *at risk* zum Zeitpunkt t .

Wichtig für die Survival-Analyse ist die Annahme, dass der zur Zensierung führende Mechanismus unabhängig von Einflussgrößen auf die Überlebenszeit ist, Beobachtungsobjekte mit zensierter Überlebenszeit also kein systematisch anderes hazard haben. Diese Bedingung wäre etwa dann erfüllt, wenn zensierte Daten dadurch entstehen, dass eine Studie zu einem vorher festgelegten Zeitpunkt endet, bis zu dem nicht bei allen Untersuchungseinheiten das Ereignis eingetreten ist. Bewirkt eine Ursache dagegen sowohl das nahe bevorstehende Eintreten des Ereignisses selbst als auch den Ausfall von Beobachtungsmöglichkeiten, wäre die Annahme nicht-informativer Zensierung verletzt. Wenn selektiv Beobachtungseinheiten mit erhöhtem hazard nicht mehr beobachtet werden können, besteht die Gefahr verzerrter Schätzungen des Verlaufs der Überlebenszeiten.

9.2.1 Zeitlich konstante Prädiktoren

Survival-Daten beinhalten Angaben zum Beobachtungszeitpunkt, zu den Prädiktoren der Überlebenszeit sowie eine Indikatorvariable dafür, ob das Ereignis zum angegebenen Zeitpunkt beobachtet wurde. Für die Verwendung in späteren Analysen sind Beobachtungszeitpunkt und Indikatorvariable zunächst in einem Survival-Objekt zusammenzuführen, das Informationen zur Art der Zensierung der Daten berücksichtigt. Dies geschieht für potentiell rechts-zensierte Daten mit `Surv()` aus dem Paket `survival` in der folgenden Form:

```
Surv(<Zeitpunkt>, <Status>)
```

Als erstes Argument ist ein Vektor der Zeitpunkte $t_i > 0$ zu nennen, an denen das Ereignis bei den Objekten i eingetreten ist. Bei rechts-zensierten Beobachtungen ist dies stattdessen der letzte bekannte Zeitpunkt, zu dem das Ereignis noch nicht eingetreten war. Die dabei implizit verwendete Zeitskala hat ihren Ursprung 0 beim Eintritt in die Untersuchung. Das zweite Argument ist eine numerische oder logische Indikatorvariable, die den Status zu den genannten Zeitpunkten angibt – ob das Ereignis also vorlag (1 bzw. `TRUE`) oder nicht (0 bzw. `FALSE` bei rechts-zensierten Beobachtungen).³

Für die folgende Simulation von Überlebenszeiten soll eine Weibull-Verteilung mit Annahme proportionaler hazards bzgl. der Einflussfaktoren (Abschn. 9.4) zugrunde gelegt werden (Abb. 9.1). Dafür sei der lineare Effekt der Einflussgrößen durch $\mathbf{X}\beta = \beta_1 X_1 + \dots + \beta_j X_j + \dots + \beta_p X_p$ gegeben (ohne absoluten Term β_0 , s. Abschn. 12.9.1). Hier soll dafür ein kontinuierlicher Prädiktor sowie ein Gruppierungsfaktor mit 3 Stufen verwendet werden, wobei beide Variablen nicht über die Zeit variieren. Zusätzlich sei die Schichtung hinsichtlich des Geschlechts berücksichtigt.

```
> N    <- 180                                # Anzahl Personen
> P    <- 3                                    # Anzahl Stufen Faktor
> sex <- factor(sample(c("f", "m"), N, replace=TRUE))  # Geschlecht
> X    <- rnorm(N, 0, 1)                      # kontinuierl. Prädiktor
> IV   <- factor(rep(LETTERS[1:P], each=N/P)) # Gruppierungsfaktor

# Effekte der Faktorstufen: 1. Stufe = baseline -> Effekt 0
```

³Für Intervall-zensierte Daten vgl. `?Surv`. Vergleiche Abschn. 9.2.2 für zeitabhängige Prädiktoren und Fälle, in denen mehrere Ereignisse pro Beobachtungsobjekt möglich sind.

```
> IVeff <- c(0, -1, 1.5)

# zusammengefasster Effekt der Einflussgrößen mit zufälligem Fehler
> Xbeta <- 0.7*X + IVeff[unclass(IV)] + rnorm(N, 0, 2)
```

Weiter sei $U \sim \mathcal{U}(0, 1)$ eine gleichverteilte Zufallsvariable auf dem Intervall $[0, 1]$. Weibull-verteilte Überlebenszeiten können dann als Realisierung von $T = (-\ln(U) b^a e^{-X\beta})^{\frac{1}{a}}$ simuliert werden (Abschn. 9.5). Die hier getroffene Wahl $a = 1.5$ führt dazu, dass $\ln \lambda(t)$ linear mit $\ln t$ ansteigt. Die Simulation von Überlebenszeiten mit anderen kumulativen Hazard-Funktionen $\Lambda(t)$ unter Annahme proportionaler hazards erfolgt allgemein mit $\Lambda^{-1}(-\ln(U) e^{-X\beta})$ (Bender, Augustin & Blettner, 2005).⁴

```
# Weibull-Verteilung zur Charakterisierung des baseline-hazards
> weibA <- 1.5                                # Formparameter
> weibB <- 100                                 # Skalierungsparameter
> U       <- runif(N, 0, 1)                      # gleichverteilte Variable

# Überlebenszeiten - aufrunden für t > 0
> eventT <- ceiling((-log(U)*(weibB^weibA)*exp(-Xbeta))^(1/weibA))
> obsLen <- 120                                  # Beobachtungsdauer

# stelle kumulierte Verteilung der Überlebenszeiten dar
> plot(ecdf(eventT), xlim=c(0, 200), main="Kumulative
+      Überlebenszeit-Verteilung", xlab="t", ylab="F(t)")

> abline(v=obsLen, col="blue", lwd=2)            # Untersuchungsende
> text(obsLen-5, 0.2, adj=1, labels="Ende Beobachtungszeit")
```

Der zur rechts-Zensierung führende Prozess soll hier ausschließlich das geplante Ende des Beobachtungszeitraums sein. Alle nach dem Endpunkt der Erhebung liegenden Überlebenszeiten werden daher auf diesen Endpunkt zensiert. Entsprechend wird das Ereignis nur beobachtet, wenn die Überlebenszeit nicht nach dem Endpunkt liegt.

```
> censT  <- rep(obsLen, N)                      # Zensierungszeit
> obsT   <- pmin(eventT, censT)                 # zensierte Überlebenszeit
> status <- eventT <= censT                     # Ereignis-Status
> dfSurv <- data.frame(obsT, status, sex, X, IV) # Datensatz
> library(survival)                             # für Surv()
> Surv(obsT, status)                           # Survival-Objekt
[1] 63 25 73 120+ 4 39 4 1 10 11 2 120+ 7 29 95 # ...
```

In der (hier gekürzten) Ausgabe des mit `Surv()` gebildeten Survival-Objekts sind zensierte Überlebenszeiten durch ein + kenntlich gemacht.

⁴Das Paket `simsurv` (Brilleman, 2021) erlaubt es, sehr flexibel Ereigniszeiten unterschiedlicher Verteilung zu simulieren und dabei den Einfluss von Kovariaten zu berücksichtigen.

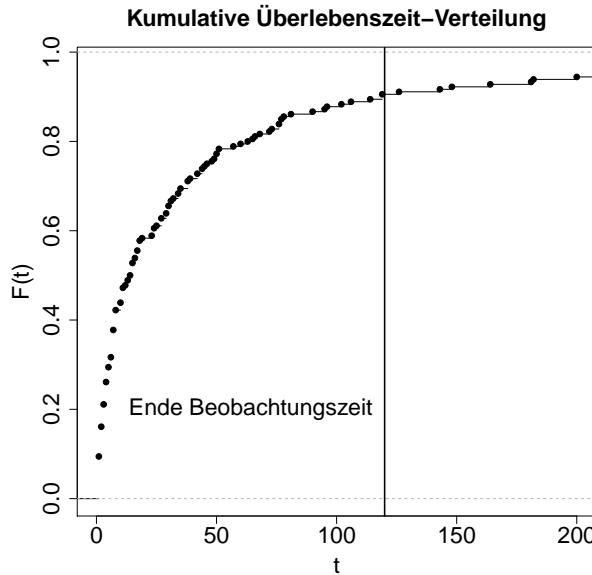


Abbildung 9.1: Kumulative Verteilung simulierter Survival-Daten mit Weibull-Verteilung

9.2.2 Daten in Zählprozess-Darstellung

Überlebenszeiten lassen sich samt der zugehörigen Werte für relevante Prädiktoren auch in der *Zählprozess*-Darstellung notieren. Für jedes Beobachtungsobjekt unterteilt diese Darstellung die Zeit in diskrete Intervalle $(\text{Start}, \text{Stop}]$, für die jeweils angegeben wird, ob sich ein Ereignis im Intervall ereignet hat. Die Intervalle sind links offen und rechts geschlossen, zudem muss $\text{Stop} > \text{Start}$ gelten, so dass keine Intervalle der Länge 0 auftauchen. Der Aufruf von `Surv()` erweitert sich dafür wie folgt:

`Surv(<Start>, <Stop>, <Status>)`

`<Start>` gibt den Beginn eines Beobachtungsintervalls an. `<Stop>` ist wieder der beobachtete Ereignis-Zeitpunkt bzw. bei rechts-zensierten Daten der letzte bekannte Zeitpunkt, zu dem kein Ereignis vorlag. Intervalle für unterschiedliche Personen dürfen dabei unterschiedliche Grenzen besitzen. Anders als in der in Abschn. 9.2.1 vorgestellten Darstellung kann die hier implizit verwendete Zeitskala flexibel etwa das Alter eines Beobachtungsobjekts sein, die (vom Eintritt in die Untersuchung abweichende) Zeit seit einer Diagnose oder die absolute kalendarische Zeit. Ist das Ereignis im Intervall $(\text{Start}, \text{Stop}]$ eingetreten, ist `<Status>` gleich 1 (bzw. `TRUE`), sonst 0 (bzw. `FALSE`) bei rechts-zensierten Beobachtungen.

Die Zählprozess-Darstellung stellt explizit dar, wann ein Beobachtungszeitraum beginnt. Sie gibt daher Auskunft darüber, vor welchem Zeitpunkt keine Informationen darüber vorliegen, ob sich das Ereignis bereits einmal ereignet hat. Die Darstellungsform ist damit für links-gestützte Daten geeignet. Tabelle 9.1 zeigt die Darstellung am Beispiel von vier Beobachtungsobjekten aus zwei Gruppen, deren Lebensalter zu Beginn und zu Ende einer zehnjährigen Untersuchung ebenso erfasst wurde wie der Ereignis-Status zu Ende der Untersuchung.

Die Zählprozess-Darstellung erlaubt es auch Erhebungssituationen abzubilden, in denen der Status von Beobachtungsobjekten an mehreren Zeitpunkten t_m ermittelt wird. So könnte etwa

Tabelle 9.1: Zählprozess-Darstellung für vier links-gestutzte und teilweise rechts-zensierte Beobachtungen mit Überlebenszeiten (Lebensalter) $\{47, 38^+, 41, 55\}$ aus zwei Gruppen in einer zehnjährigen Untersuchung

| Start | Stop | Gruppe | Status |
|-------|------|-----------|--------|
| 37 | 47 | Treatment | 1 |
| 28 | 38 | Treatment | 0 |
| 31 | 41 | Control | 1 |
| 45 | 55 | Control | 1 |

bei regelmäßigen Kontrollterminen nach einer Operation geprüft werden, ob eine bestimmte Krankheit wieder diagnostizierbar ist. Der Aufruf von `Surv()` hat dann die Form wie bei links-gestutzten Beobachtungen, wobei pro Beobachtungsobjekt mehrere $(\text{Start}, \text{Stop}]$ Intervalle vorliegen. Für den Untersuchungszeitpunkt t_m ist dabei $\langle \text{Start} \rangle$ der letzte zuvor liegende Untersuchungszeitpunkt t_{m-1} , während $\langle \text{Stop} \rangle$ t_m selbst ist. Für den ersten Untersuchungszeitpunkt t_1 ist t_0 der Eintritt in die Untersuchung, etwa das zugehörige kalendarische Datum oder das Alter eines Beobachtungsobjekts.

Wiederkehrende Ereignisse

Da der Ereignis-Status in der Zählprozess-Darstellung am Ende mehrerer Zeitintervalle erfasst werden kann, ist es auch möglich Ereignisse abzubilden, die sich pro Untersuchungseinheit potentiell mehrfach ereignen. Die spätere Analyse wiederkehrender Ereignisse muss dabei berücksichtigen, ob diese unabhängig voneinander eintreten, oder etwa in ihrer Reihenfolge festgelegt sind.

Bei mehreren Beobachtungszeitpunkten muss der Datensatz für die Zählprozess-Darstellung im Long-Format vorliegen, d. h. für jede Untersuchungseinheit mehrere Zeilen umfassen (Abschn. 3.3.11). Pro Beobachtungszeitpunkt t_m ist für jede Untersuchungseinheit eine Zeile mit den Werten t_{m-1} , t_m , den Werten der Prädiktoren sowie dem Ereignis-Status zu t_m in den Datensatz aufzunehmen. Zusätzlich sollte jede Zeile den Wert eines Faktors $\langle \text{ID} \rangle$ enthalten, der das Beobachtungsobjekt identifiziert (Tab. 9.2).

Liegen die Daten im $(\langle \text{Zeitpunkt} \rangle, \langle \text{Status} \rangle)$ -Format vor, können sie mit `survSplit()` aus dem Paket `survival` in Zählprozess-Darstellung mit mehreren Zeitintervallen pro Beobachtungsobjekt gebracht werden.

```
survSplit(Modellformel, cut=Grenzen, start="Name",  
         id="Name", zero=Startzeit, data=Datensatz)
```

Als erstes Argument ist eine Modellformel zu übergeben, deren linke Seite ein mit `Surv()` erstelltes Objekt ist (Abschn. 9.2). Die rechte Seite der Modellformel definiert die Variablen, die im erstellten Datensatz beibehalten werden – meist `.` stellvertretend für alle Variablen des Datensatzes `data`. `cut` legt die linken Grenzen t_{m-1} der neu zu bildenden Intervalle fest,

Tabelle 9.2: Zählprozess-Darstellung für zwei Beobachtungsobjekte aus zwei Gruppen am Ende von je drei zweijährigen Beobachtungsintervallen mit potentiell mehrfach auftretenden Ereignissen

| ID | Start (t_{m-1}) | Stop (t_m) | Gruppe | Status |
|----|---------------------|----------------|-----------|--------|
| 1 | 37 | 39 | Treatment | 0 |
| 1 | 39 | 41 | Treatment | 1 |
| 1 | 41 | 43 | Treatment | 1 |
| 2 | 28 | 30 | Control | 0 |
| 2 | 30 | 32 | Control | 1 |
| 2 | 32 | 34 | Control | 0 |

jedoch ohne die erste Grenze t_0 .⁵ **start** ist der Name der zu erstellenden Variable der linken Intervallgrenzen t_{m-1} . Soll eine Variable hinzugefügt werden, die jedem Intervall das zugehörige Beobachtungsobjekt zuordnet, muss deren Name für **id** genannt werden. Mit **zero** lässt sich in Form eines Vektors oder als – dann für alle Beobachtungen identische – Zahl angeben, was der Zeitpunkt t_0 des Beginns der Beobachtungen ist.

```

> library(survival)                                # für survSplit()
> dfSurvCP <- survSplit(Surv(obsT, status) ~ ., cut=seq(30, 90, by=30),
+                         start="start", id="ID", zero=0, data=dfSurv)

# sortiere nach Beobachtungsobjekt und linken Intervallgrenzen
> head(sort_by(dfSurvCP, ~ ID + start), n=7)
  sex          X IV ID start obsT status
1  f -1.3130607  A  1     0   30     0
2  f -1.3130607  A  1    30   60     0
3  f -1.3130607  A  1    60   63     1
4  m -0.1384786  A  2     0   25     1
5  m -0.3846335  A  3     0   30     0
6  m -0.3846335  A  3    30   60     0
7  m -0.3846335  A  3    60   73     1

```

Zeitabhängige Prädiktoren

In der Zählprozess-Darstellung ist es möglich, die Werte von zeitlich variablen Prädiktoren in die Daten aufzunehmen (Tab. 9.3). Eine spätere Analyse setzt dabei voraus, dass der Wert eines zeitabhängigen Prädiktors zum Zeitpunkt t_m nur Informationen widerspiegelt, die bis t_m vorlagen – aber nicht später. Eine zeitlich rückwirkende Kategorisierung von Untersuchungseinheiten in verschiedene Gruppen auf Basis ihres Verhaltens zu Studienende wäre etwa demnach

⁵Bei nicht wiederkehrenden Ereignissen ist die Einteilung der Gesamtbeobachtungszeit in einzelne, bündig aneinander anschließende Intervalle beliebig: Die einzelne Beobachtung im (*Zeitpunkt*, *Status*)-Format (10, TRUE) ist sowohl äquivalent zu den zwei Beobachtungen in Zählprozess-Darstellung (0, 4, FALSE), (4, 10, TRUE) als auch zu den drei Beobachtungen (0, 2, FALSE), (2, 6, FALSE), (6, 10, TRUE).

unzulässig. Ebenso sind meist zeitabhängige Variablen problematisch, die weniger Einflussgröße bzw. Prädiktor, sondern eher Effekt oder Indikator eines Prozesses sind, der zu einem bevorstehenden Ereignis führt.

Tabelle 9.3: Zählprozess-Darstellung für zeitabhängige Prädiktoren bei drei Beobachtungsobjekten mit Überlebenszeiten $\{3, 4^+, 2\}$

| ID | Start (t_{m-1}) | Stop (t_m) | Temperatur | Status |
|----|---------------------|----------------|------------|--------|
| 1 | 0 | 1 | 45 | 0 |
| 1 | 1 | 2 | 52 | 0 |
| 1 | 2 | 3 | 58 | 1 |
| 2 | 0 | 1 | 37 | 0 |
| 2 | 1 | 2 | 41 | 0 |
| 2 | 2 | 3 | 56 | 0 |
| 2 | 3 | 4 | 57 | 0 |
| 3 | 0 | 1 | 35 | 0 |
| 3 | 1 | 2 | 61 | 1 |

9.3 Kaplan-Meier-Analyse

9.3.1 Survival-Funktion schätzen

Die Kaplan-Meier-Analyse liefert als nonparametrische Maximum-Likelihood-Schätzung der Survival-Funktion eine Stufenfunktion $\hat{S}(t)$, deren Stufen bei den empirisch beobachteten Ereignis-Zeitpunkten t_i liegen. Sie wird samt der punktweisen Konfidenzintervalle mit `survfit()` aus dem Paket `survival` berechnet.

```
survfit(<Modellformel>, type="kaplan-meier", conf.type=<'CI-Typ>")
```

Als erstes Argument ist eine Modellformel zu übergeben, deren linke Seite ein mit `Surv()` erstelltes Objekt ist (Abschn. 9.2). Die rechte Seite der Modellformel ist entweder der konstante Term 1 für eine globale Anpassung, oder besteht aus (zeitlich konstanten) Faktoren. In diesem Fall resultiert für jede Faktorstufe bzw. Kombination von Faktorstufen g eine separate Schätzung $\hat{S}_g(t)$. Das Argument `type` ist bei der Voreinstellung "kaplan-meier" zu belassen. Mit `conf.type` wird die Transformation für $S(t)$ angegeben, auf deren Basis die nach zugehöriger Rücktransformation gewonnenen Konfidenzintervalle für $S(t)$ konstruiert sind: Mit "plain" erhält man Intervalle auf Basis von $S(t)$ selbst. Geeigneter sind oft die durch "log" erzeugten Intervalle, die auf dem kumulativen hazard $\Lambda(t) = -\ln S(t)$ basieren. Mit "log-log" ergeben sich die Intervalle aus dem logarithmierten kumulativen hazard $\ln \Lambda(t) = \ln(-\ln S(t))$. Auf "none" gesetzt unterbleibt die Berechnung von Konfidenzintervallen.

```
# Schätzung von S(t) ohne Trennung nach Gruppen
> library(survival) # für survfit()
> KM0 <- survfit(Surv(obsT, status) ~ 1, type="kaplan-meier",
```

```

+           conf.type="log", data=dfSurv)

# Schätzung von S(t) getrennt nach Gruppen
> (KM <- survfit(Surv(obsT, status) ~ IV, type="kaplan-meier",
+                   conf.type="log", data=dfSurv))
      records events median 0.95LCL 0.95UCL
IV=A       60     55    13.5      8     31
IV=B       60     49    33.0     17     50
IV=C       60     59     5.5      4     11

```

Die Ausgabe gibt – ggf. getrennt für jede Gruppe – Auskunft über die Anzahl der Beobachtungen (**records**), die Anzahl der Ereignisse (**events**) und unter **median** eine Schätzung für den Median der Überlebenszeit (das 50%-Quantil der geschätzten Survival-Funktion $\hat{S}(t)$) gefolgt von den Grenzen des zugehörigen Konfidenzintervalls (0.95LCL, 0.95UCL). Das zurückgegebene Objekt enthält u. a. in den Komponenten **time** die Ereignis-Zeitpunkte, in **surv** die geschätzte Überlebensfunktion $\hat{S}(t)$ und in **cumhaz** das geschätzte kumulative hazard $\hat{\Lambda}(t) = -\ln \hat{S}(t)$.

Analog zum geschätzten Median der Überlebenszeit ermittelt `quantile(<survfit-Objekt>, probs=<Quantile>)` beliebige Quantile von $\hat{S}(t)$ – also Schätzungen für die Zeitpunkte, zu denen bei einem bestimmten Anteil der Beobachtungsobjekte ein Ereignis aufgetreten ist. In der Voreinstellung `conf.int=TRUE` erhält man zusätzlich die Grenzen des jeweiligen Konfidenzintervalls für ein Quantil.

```

# Quantile der Überlebenszeit mit Grenzen der Konfidenzintervalle
> quantile(KM0, probs=c(0.25, 0.5, 0.75), conf.int=TRUE)
$quantile
25% 50% 75%
4.0 14.5 47.0

$lower
25% 50% 75%
3    10   35

$upper
25% 50% 75%
7    19   68

```

Über `print(<survfit-Objekt>, print.rmean=TRUE)` erhält man auch die geschätzte mittlere Überlebenszeit.

```

> print(KM0, print.rmean=TRUE)
      n  events *rmean *se(rmean) median 0.95LCL 0.95UCL
180.00 163.00 32.64        2.89  14.50   10.00   19.00
* restricted mean with upper limit = 120

```

`summary(<survfit-Objekt>, times=<Vektor>)` gibt detailliert Auskunft über die Werte von $\hat{S}(t)$ zu den unter **times** genannten Zeiten t . So lässt sich etwa das geschätzte 100-Tage Überleben berechnen. Fehlt **times**, umfasst die Ausgabe alle tatsächlich aufgetretenen Ereignis-Zeitpunkte t_i .

```
# Werte der geschätzten Survival-Funktion für 20, 50, 100 Tage
> summary(KM0, times=c(20, 50, 100))      # ohne Trennung nach Gruppen
   time n.risk n.event survival std.err lower 95% CI upper 95% CI
     20     75      105    0.417  0.0367    0.3505    0.495
     50     43       34    0.228  0.0313    0.1741    0.298
    100     22       19    0.122  0.0244    0.0826    0.181
```

Die Ausgabe nennt unter `time` die vorgegebenen Ereignis-Zeitpunkte t_i und unter `n.risk` die Anzahl der Personen, von denen bekannt ist, dass sie jeweils vor t_i noch kein Ereignis hatten aber noch unter Beobachtung sind. Fehlt im Aufruf das Argument `times`, berichtet `n.event` die Anzahl der Ereignisse zu t_i , andernfalls die Anzahl Ereignisse von t_{i-1} bis inkl. t_i . Unter `survival` zeigt die Ausgabe die Schätzung $\hat{S}(t_i)$, unter `std.err` die geschätzte Streuung des Schätzers $\hat{S}(t_i)$ sowie in den letzten beiden Spalten die Grenzen des punktweisen Konfidenzintervalls für $\hat{S}(t_i)$.

9.3.2 Survival, kumulative Inzidenz und kumulatives hazard darstellen

Die grafische Darstellung von $\hat{S}(t)$ mit `plot(<survfit-Objekt>)` (Abb. 9.2) zeigt zusätzlich die Konfidenzintervalle. Die geschätzte kumulative Inzidenz $1 - \hat{S}(t)$ erhält man mit dem Argument `fun=function(x) { 1-x }`. Für das geschätzte kumulative hazard ist beim Aufruf von `plot()` das Argument `fun="cumhaz"` zu verwenden.

```
> plot(KM0, main=expression(paste("KM-Schätzer ", hat(S)(t),
+ " mit CI")), xlab="t", ylab="Survival", lwd=2)

> plot(KM0, main=expression(paste("KM-Schätzer 1-", hat(S)(t),
+ " mit CI")), xlab="t",
+ ylab="kumulative Inzidenz", fun=function(x) {1-x}, lwd=2)

> plot(KM, main=expression(paste("KM-Schätzer ", hat(S)[g](t), lty=1:3,
+ " für Gruppen")), xlab="t", ylab="Survival", lwd=2, col=1:3)

> legend(x="topright", lty=1:3, col=1:3, lwd=2, legend=LETTERS[1:3])

> plot(KM0, main=expression(paste("KM-Schätzer ", hat(Lambda)(t))),
+ xlab="t", ylab="kumulatives hazard", fun="cumhaz", lwd=2)
```

9.3.3 Log-Rank-Test auf gleiche Survival-Funktionen

`survdiff()` aus dem Paket `survival` berechnet den Log-Rank-Test, ob sich die Survival-Funktionen in mehreren Gruppen unterscheiden.⁶

```
survdiff(<Modellformel>, rho=0, data=<Datensatz>)
```

⁶Für eine exakte Alternative vgl. `logrank_test()` aus dem Paket `coin` (Hothorn, Hornik, van de Wiel & Zeileis, 2008).

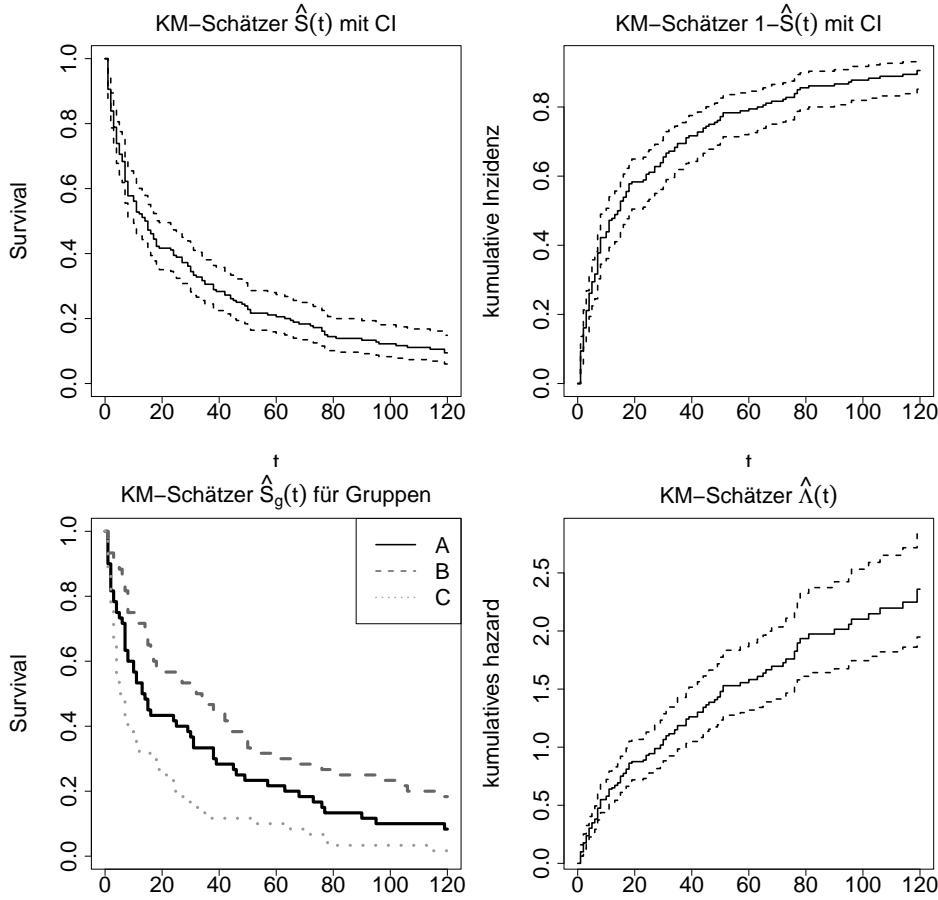


Abbildung 9.2: Kaplan-Meier-Schätzungen: Survival-Funktion $\hat{S}(t)$ sowie kumulative Inzidenz $1 - \hat{S}(t)$ ohne Berücksichtigung der Gruppen (mit Konfidenzintervallen), separate Schätzungen $\hat{S}_g(t)$ getrennt nach Gruppen g sowie die geschätzte kumulative Hazard-Funktion $\hat{\Lambda}(t)$ (mit Konfidenzintervallen)

Als erstes Argument ist eine Modellformel der Form `<Surv-Objekt> ~ <Faktor>` zu übergeben. Stammen die dabei verwendeten Variablen aus einem Datensatz, ist dieser unter `data` zu nennen. Das Argument `rho` kontrolliert, welche Testvariante berechnet wird. Mit der Voreinstellung 0 ist dies der Mantel-Hänszel-Test.

```
> library(survival) # für survdiff()
> survdiff(Surv(obsT, status) ~ IV, data=dfSurv)
   N Observed Expected (0-E)^2/E (0-E)^2/V
IV=A 60      55     54.6    0.00351   0.00552
IV=B 60      49     72.3    7.50042  14.39151
IV=C 60      59     36.2   14.43764  20.10283
```

Chisq= 23.9 on 2 degrees of freedom, p= 7e-06

Die Ausgabe nennt in der Spalte `N` die jeweilige Gruppengröße, unter `Observed` die Anzahl der beobachteten Ereignisse f_g^o pro Gruppe g , unter `Expected` die dort unter der Nullhypothese erwartete Anzahl der Ereignisse f_g^e .

diese erwartete Anzahl von Ereignissen f_g^e , unter $(O-E)^2/E$ den Wert für $\frac{(f_g^o - f_g^e)^2}{f_g^e}$ und unter $(O-E)^2/V \frac{(f_g^o - f_g^e)^2}{\hat{\sigma}_D^2}$, wobei $\hat{\sigma}_D$ die geschätzte Varianz von $D = f_g^o - f_g^e$ ist. Die letzte Zeile liefert den Wert der asymptotisch χ^2 -verteilten Teststatistik sowie die zugehörigen Freiheitsgrade mit dem p -Wert.

Bei geschichteten (stratifizierten) Stichproben lässt sich der Test auch unter Berücksichtigung der Schichtung durchführen. Dazu ist als weiterer Vorhersageterm auf der rechten Seite der Modellformel `strata(<Faktor>)` hinzuzufügen, wobei für dessen Gruppen keine Parameter geschätzt werden.

```
> survdiff(Surv(obsT, status) ~ IV + strata(sex), data=dfSurv)
      N Observed Expected (O-E)^2/E (O-E)^2/V
IV=A 60      55      54.7   0.00185   0.00297
IV=B 60      49      72.1   7.42735  14.28570
IV=C 60      59      36.2  14.41091  20.32224

Chisq= 23.9  on 2 degrees of freedom, p= 6.45e-06
```

9.4 Cox proportional hazards Modell

Ein semi-parametrisches Regressionsmodell für zensierte Survival-Daten ist das Cox proportional hazards (PH) Modell, das den Einfluss von kontinuierlichen Prädiktoren ebenso wie von Gruppierungsfaktoren auf die Überlebenszeit einbeziehen kann. Das Modell macht keine spezifischen Voraussetzungen für die generelle Verteilungsform von Überlebenszeiten, basiert aber auf der Annahme, dass der Zusammenhang der p Prädiktoren X_j mit der logarithmierten Ereignisrate linear ist, sich also mit dem bekannten Regressionsmodell beschreiben lässt. Für die Form des Einflusses der X_j gelten insgesamt folgende Annahmen:

$$\begin{aligned} \ln \lambda(t) &= \ln \lambda_0(t) + \beta_1 X_1 + \cdots + \beta_p X_p &= \ln \lambda_0(t) + \mathbf{X}\boldsymbol{\beta} \\ \lambda(t) &= \lambda_0(t) e^{\beta_1 X_1 + \cdots + \beta_p X_p} &= \lambda_0(t) e^{\mathbf{X}\boldsymbol{\beta}} \\ S(t) &= S_0(t) \exp(\mathbf{X}\boldsymbol{\beta}) &= \exp(-\Lambda_0(t) e^{\mathbf{X}\boldsymbol{\beta}}) \\ \Lambda(t) &= \Lambda_0(t) e^{\mathbf{X}\boldsymbol{\beta}} \end{aligned}$$

Dabei spielt $\ln \lambda_0(t)$ die Rolle des absoluten Terms β_0 , entsprechend schließt die Abkürzung $\mathbf{X}\boldsymbol{\beta}$ keinen absoluten Term ein (Kap. 8). $\lambda_0(t)$ ist das baseline hazard, also die Hazard-Funktion, wenn alle Einflussgrößen X_j gleich 0 bzw. alle Faktoren gleich der Referenzkategorie sind (bei Treatment-Kontrasten). Die Form dieser Funktion – und damit die generelle Verteilung von Überlebenszeiten (etwa Exponential- oder Weibull-Verteilung, s. Abschn. 9.1, 9.5) – bleibt unspezifiziert, weshalb dies der nonparametrische Teil des Modells ist.⁷

Ein exponenziertes Gewicht e^{β_j} ist der multiplikative Faktor, mit dem sich die Ereignisrate verändert, wenn der Prädiktor X_j um eine Einheit wächst. Dies ist das *hazard ratio*, also

⁷Aus diesem Grund ist der Aussagebereich eines angepassten Cox PH-Modells auch auf die in der Stichprobe tatsächlich vorliegenden Überlebenszeiten begrenzt, eine Extrapolation über die maximal beobachtete Überlebenszeit hinaus also unzulässig.

das relative hazard nach Erhöhung von X_j um eine Einheit. Die prozentuale Veränderung der Ereignisrate ist $100 \cdot (e^{\beta_j} - 1)$. Bei $e^{\beta_j} = 1.3$ wäre jede zusätzliche Einheit von X_j mit einer um 30% höheren Ereignisrate assoziiert, bei $e^{\beta_j} = 0.6$ mit einer um 40% niedrigeren Rate. Das Modell impliziert die Annahme, dass das relative hazard zweier Personen i und j mit Prädiktorwerten \mathbf{x}_i und \mathbf{x}_j unabhängig vom baseline hazard $\lambda_0(t)$ sowie unabhängig vom Zeitpunkt t konstant ist.

$$\frac{\lambda_i(t)}{\lambda_j(t)} = \frac{\lambda_0(t) e^{\mathbf{x}_i^\top \boldsymbol{\beta}}}{\lambda_0(t) e^{\mathbf{x}_j^\top \boldsymbol{\beta}}} = \frac{e^{\mathbf{x}_i^\top \boldsymbol{\beta}}}{e^{\mathbf{x}_j^\top \boldsymbol{\beta}}} = e^{(\mathbf{x}_i - \mathbf{x}_j)^\top \boldsymbol{\beta}}$$

Das hazard von Person i ist demnach proportional zum hazard von Person j mit dem über die Zeit konstanten Proportionalitätsfaktor $e^{(\mathbf{x}_i - \mathbf{x}_j)^\top \boldsymbol{\beta}}$. Anders gesagt darf keine Interaktion $t \times \mathbf{X}$ vorliegen. Das Cox PH-Modell wird mit `coxph()` aus dem Paket `survival` angepasst.

```
coxph(<Modellformel>, ties="<Methode>", data=<Datensatz>)
```

Als erstes Argument ist eine Modellformel zu übergeben, deren linke Seite ein mit `Surv()` erstelltes Objekt ist (Abschn. 9.2). Die rechte Seite der Modellformel besteht aus kontinuierlichen Prädiktoren oder Faktoren, die zeitlich konstant oder – bei Daten in Zählprozess-Darstellung (Abschn. 9.2.2) – auch zeitabhängig sein können. Stammen die in der Modellformel verwendeten Variablen aus einem Datensatz, ist dieser unter `data` zu nennen. Zudem sind zwei besondere Vorhersageterme möglich: `strata(<Faktor>)` sorgt dafür, dass ein stratifiziertes Cox PH-Modell angepasst wird – mit einer separaten Baseline-Hazard-Funktion $\lambda_{0g}(t)$ für jede Stufe g des Faktors. Der Term `cluster` ist in Abschn. 9.4.5 beschrieben.

Das Modell berücksichtigt für die Anpassung die in Rangdaten transformierten, ggf. zensierten Ereignis-Zeitpunkte, was es robust gegen Ausreißer macht. Gleichzeitig ist deshalb aber gesondert über das Argument `ties` zu spezifizieren, wie bei Bindungen, also identischen Ereignis-Zeitpunkten und damit uneindeutigen Rängen, vorzugehen ist. Voreinstellung ist "efron"; eine exakte, aber rechenintensive Behandlung von Bindungen erhält man mit "exact". Durch die Rangtransformation hat die Länge der Intervalle zwischen aufgetretenen Ereignissen keinen Einfluss auf die Parameterschätzungen.

Das Cox PH-Modell soll hier für die in Abschn. 9.2.1 simulierten Daten mit zeitlich konstanten Prädiktoren und höchstens einmal auftretenden Ereignissen angepasst werden.

```
> library(survival)                                # für coxph()
> (fitCPH <- coxph(Surv(obsT, status) ~ X + IV, data=dfSurv))
      coef exp(coef) se(coef)    z      p
X     0.49123  1.63433  0.08685  5.656 1.55e-08
IVB   -0.40612  0.66623  0.19675 -2.064  0.03901
IVC    0.57890  1.78407  0.19117  3.028  0.00246

Likelihood ratio test=56.05  on 3 df,  p=4.103e-12
n= 180, number of events= 163

# äquivalenter Aufruf mit Daten in Zählprozess-Darstellung
> coxph(Surv(start, obsT, status) ~ X + IV, data=dfSurvCP)  # ...
```

Die Ausgabe nennt in der Spalte `coef` die geschätzten Koeffizienten $\hat{\beta}_j$, deren exponenzierte Werte $e^{\hat{\beta}_j}$ in der Spalte `exp(coef)` stehen. Für den kontinuierlichen Prädiktor X_j ist $e^{\hat{\beta}_j}$ der Änderungsfaktor für die geschätzte Ereignisrate $\hat{\lambda}(t)$ (also das hazard ratio), wenn X_j um eine Einheit wächst. Die Zeilen `IVB` und `IVC` sind so zu interpretieren, dass `A` als erste Stufe des Faktors `IV` als Referenzgruppe verwendet wurde (Abschn. 2.6.5), so dass Dummy-codierte Variablen für die Stufen `B` und `C` verbleiben (Treatment-Kontraste, Abschn. 12.9.2). Für `IVB` und `IVC` ist $e^{\hat{\beta}_j}$ daher jeweils der multiplikative Änderungsfaktor für $\hat{\lambda}(t)$ verglichen mit Gruppe `A`. Die geschätzten Streuungen $\hat{\sigma}_{\hat{\beta}}$ der $\hat{\beta}_j$ stehen in der Spalte `se(coef)`, die Werte der Wald-Statistik $z = \frac{\hat{\beta}_j}{\hat{\sigma}_{\hat{\beta}}}$ in der Spalte `z` und die zugehörigen p -Werte in der Spalte `p`.

Die letzte Zeile berichtet die Ergebnisse des Likelihood-Quotienten-Tests des Gesamtmodells gegen das Modell ohne Prädiktoren. Teststatistik ist die Devianz-Differenz beider Modelle mit der Differenz ihrer Freiheitsgrade als Anzahl der Freiheitsgrade der asymptotisch gültigen χ^2 -Verteilung. Zusätzliche Informationen liefert `summary(<coxph-Objekt>)`.

```
> summary(fitCPH)
n= 180, number of events= 163

      coef  exp(coef)  se(coef)      z Pr(>|z|)
X     0.49123   1.63433  0.08685  5.656 1.55e-08 ***
IVB -0.40612   0.66623  0.19675 -2.064  0.03901 *
IVC  0.57890   1.78407  0.19117  3.028  0.00246 **

      exp(coef)  exp(-coef) lower .95 upper .95
X       1.6343      0.6119    1.3785    1.9376
IVB     0.6662      1.5010    0.4531    0.9797
IVC     1.7841      0.5605    1.2266    2.5950

Concordance= 0.683  (se = 0.022)
Likelihood ratio test= 56.05  on 3 df,   p=4.103e-12
Wald test            = 54.98  on 3 df,   p=6.945e-12
Score (logrank) test = 54.86  on 3 df,   p=7.365e-12
```

Neben den bereits erwähnten Informationen enthält die Ausgabe zusätzlich die Konfidenzintervalle für die hazard ratios $e^{\hat{\beta}_j}$ in den Spalten `lower .95` und `upper .95`. Die Ergebnisse des Likelihood-Quotienten-, Wald- und Score-Tests beziehen sich alle auf den Test des Gesamtmodells gegen jenes ohne Prädiktoren.

9.4.1 Anpassungsgüte und Modelltests

Ein Maß für die Anpassungsgüte ist die von `summary(<coxph-Objekt>)` unter `Concordance` aufgeführte Konkordanz (Harrels C). Dies ist der Anteil an allen Paaren von Beobachtungsobjekten, bei denen das Beobachtungsobjekt mit empirisch kürzerer Überlebenszeit auch ein höheres vorhergesagtes hazard besitzt.⁸

⁸Details zur Berechnung und zum Zusammenhang mit Somers' d , insbesondere bei Bindungen und zensierten Beobachtungen erläutert `vignette("concordance", package="survival")`.

Aus einem von `coxph()` zurückgegebenen Objekt lassen sich weitere Informationen zur Anpassungsgüte extrahieren, darunter der Wert des Informationskriteriums AIC mit `extractAIC()`.

```
> library(survival)                                # für coxph()
> extractAIC(fitCPH)                             # AIC
[1] 3.000 1399.438
```

Da der hier im Modell berücksichtigte Faktor IV mit mehreren Parametern β_j assoziiert ist, muss seine Signifikanz insgesamt über einen Modellvergleich getestet werden. Dazu dient ein Likelihood-Quotienten-Test, der auf der asymptotisch χ^2 -verteilten Devianz-Differenz zweier hierarchischer Modelle mit demselben Kriterium beruht (Abschn. 8.1.5): Der Prädiktorensatz des eingeschränkten Modells `fitR` ist dabei vollständig im Prädiktorensatz des umfassenderen Modells `fitU` enthalten, das zusätzlich noch den Faktor berücksichtigt. Der Test erfolgt dann mit `anova(fitR, fitU)`.

```
# eingeschränktes Modell ohne Faktor IV
> fitCPH1 <- coxph(Surv(obsT, status) ~ X, data=dfSurv)
> anova(fitCPH1, fitCPH)                         # LQ-Modelltest für IV
Analysis of Deviance Table
Cox model: response is Surv(obsT, status)
Model 1: ~ X
Model 2: ~ X + IV
  loglik  Chisq Df P(>|Chi|)
1 -708.98
2 -696.72 24.52  2 4.738e-06 ***
```

9.4.2 Survival-Funktion, baseline hazard und kumulatives hazard schätzen

Analog zur Kaplan-Meier-Analyse (Abschn. 9.3.1) schätzt `survfit(<coxph-Objekt>)` die Survival-Funktion $\hat{S}_{\bar{x}}(t)$ im Cox PH-Modell für eine pseudo-Beobachtung, das für numerische Prädiktoren den jeweiligen Mittelwert für die gegebene Stichprobe besitzt und für Faktoren die Referenzkategorie (kurz: \bar{x}). Der Median der geschätzten Überlebenszeit findet sich in der Ausgabe unter `median`, beliebige Quantile von $\hat{S}_{\bar{x}}(t)$ erhält man mit `quantile(<coxph-Objekt>, probs=<Quantile>)`.

```
> library(survival)                                # für survfit()
> (CPH <- survfit(fitCPH, conf.type="log"))
records  events  median  0.95LCL  0.95UCL
      180      163       15        11       19

# Quantile der Überlebenszeit ohne Grenzen der Konfidenzintervalle
> quantile(CPH, probs=c(0.25, 0.5, 0.75), conf.int=FALSE)
25% 50% 75%
  5   15   44
```

Oft ist es statt der Angaben für eine pseudo-Beobachtung informativer, an das Argument `newdata` von `survfit()` einen Datensatz zu übergeben, der neue Daten für Variablen mit denselben Namen, und bei Faktoren auch denselben Stufen wie jene der ursprünglichen Prädiktoren

enthält. In diesem Fall berechnet `survfit()` für jede Zeile des Datensatzes die Schätzung $\hat{S}(t)$. Auf diese Weise kann $\hat{S}(t)$ etwa für bestimmte Gruppenzugehörigkeiten oder Werte anderer Prädiktoren ermittelt werden.

In der von `survfit()` zurückgegebenen Liste stehen die Werte für t in der Komponente `time`, für $\hat{S}(t)$ in der Komponente `surv` und für das kumulative hazard $\hat{\Lambda}(t) = -\ln \hat{S}(t)$ in `cumhaz`. Dabei ist `surv` bzw. `cumhaz` eine Matrix mit so vielen Zeilen, wie es Werte für t gibt und so vielen Spalten, wie neue Beobachtungsobjekte (Zeilen von `newdata`) vorhanden sind.

```
# Datensatz: 2 Frauen mit Prädiktor X=-2 in Gruppe A bzw. in C
> dfNew <- data.frame(sex=factor(c("f","f")), levels=levels(dfSurv$sex)),
+                         X=c(-2, -2),
+                         IV=factor(c("A", "C"), levels=levels(dfSurv$IV)))

# wende angepasstes Cox PH-Modell auf neue Daten an
> CPHnew <- survfit(fitCPH, newdata=dfNew)

# Ereignis-Zeitpunkte, Überleben und kumulatives hazard für neue Daten
> with(CPHnew, head(data.frame(t=time, surv=surv, cumhaz=cumhaz), n=3))
   t    surv.1    surv.2    cumhaz.1    cumhaz.2
1 1 0.9717888 0.9502269 0.02861677 0.05105444
2 2 0.9489392 0.9107340 0.05241058 0.09350437
3 3 0.9301547 0.8788203 0.07240439 0.12917483
```

Die grafische Darstellung von $\hat{S}_{\bar{x}}(t)$ bzw. von $\hat{S}(t)$ für die Beobachtungen in `newdata` erfolgt mit `plot(survfit-Objekt)` (Abb. 9.3). Die geschätzte kumulative Inzidenz $1 - \hat{S}(t)$ erhält man mit dem Argument `fun=function(x) { 1-x }`. Für das geschätzte kumulative hazard ist das Argument `fun="cumhaz"` zu verwenden (Abb. 9.3).

```
# Darstellung geschätztes S(t) für pseudo-Beobachtung
> plot(CPH, main=expression(paste("Cox PH-Schätzung ", hat(S)(t),
+ " mit CI"))), xlab="t", ylab="Survival", lwd=2)

# füge geschätztes S(t) für neue Daten hinzu
> lines(CPHnew$time, CPHnew$surv[ , 1], lwd=2, col="blue")
> lines(CPHnew$time, CPHnew$surv[ , 2], lwd=2, col="red")
> legend(x="topright", lwd=2, col=c("black", "blue", "red")),
+         legend=c("pseudo-Beobachtung", "sex=f, X=-2, IV=A",
+                  "sex=f, X=-2, IV=C"))

# kumulatives hazard für neue Beobachtungen
> with(CPHnew, head(data.frame(time, cum_haz=-log(surv)), n=3))
   time    cum_haz.1    cum_haz.2
1     1 0.02861677 0.05105444
2     2 0.05241058 0.09350437
3     3 0.07240439 0.12917483

# Darstellung kumulatives hazard für neue Beobachtungen
```

```
> plot(CPHnew, fun="cumhaz", col=c("blue", "red"),
+       main=expression(paste("Cox PH-Schätzung ",
+                             hat(Lambda)[g](t))),
+       ylab="kumulatives hazard", lwd=2)
> legend(x="bottomright", lwd=2, col=c("blue", "red"),
+         legend=c("sex=f, X=-2, IV=A", "sex=f, X=-2, IV=C"))
```

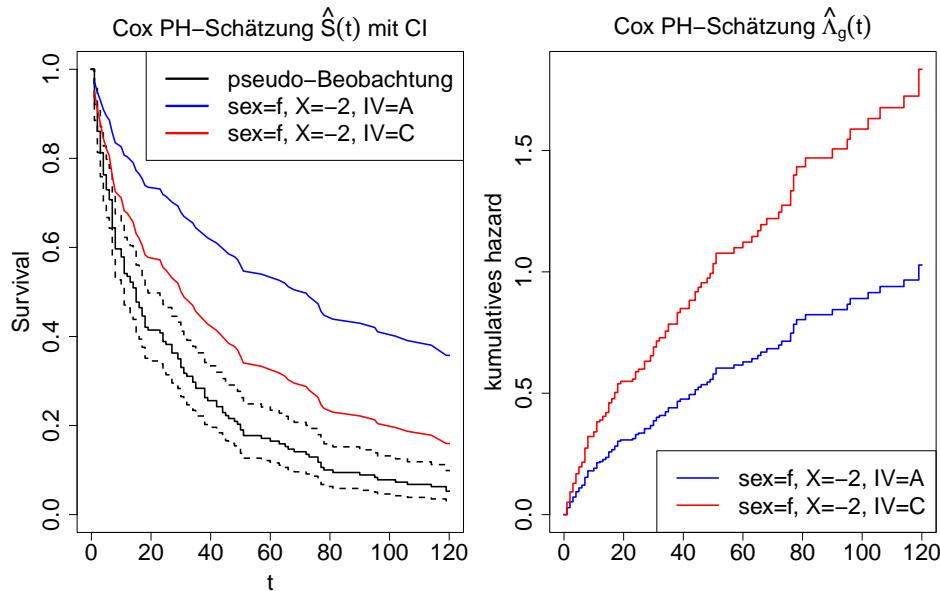


Abbildung 9.3: Cox PH-Schätzungen: Survival-Funktion $\hat{S}_{\bar{x}}(t)$ mit Konfidenzintervallen für ein pseudo-Beobachtungsobjekt, das für numerische Prädiktoren jeweils gleich dem Mittelwert in der Stichprobe ist und für Faktoren gleich der Referenzkategorie. Zusätzlich die kumulative Hazard-Funktion $\hat{\Lambda}_g(t)$ getrennt nach Gruppen g

Speziell für das geschätzte kumulative hazard stellt das Paket **survival** die Funktion **basehaz(<coxph-Objekt>)** als vereinfachte Variante von **survfit()** zur Verfügung. Die Schätzung $\hat{\Lambda}_{\bar{x}}(t)$ kann dabei mit dem Argument **centered=TRUE** für eine pseudo-Beobachtung berechnet werden, die für numerische Prädiktoren jeweils gleich deren Mittelwert in der Stichprobe ist und für Faktoren gleich der Referenzkategorie. Setzt man dagegen **centered=FALSE**, bezieht sich das Ergebnis $\hat{\Lambda}_0(t)$ i. S. einer echten baseline bei Treatment-Kontrasten auf ein Beobachtungsobjekt in der Referenzgruppe jedes Faktors, für das alle Prädiktorwerte gleich 0 sind. Das Ergebnis ist ein Datensatz mit den Variablen **hazard** für $\hat{\Lambda}_0(t)$ und **time** für t .

9.4.3 Modelldiagnostik

Um die Angemessenheit des Cox PH-Modells für die gegebenen Daten beurteilen zu können, sollten drei Aspekte der Modellanpassung geprüft werden: Die Annahme proportionaler hazards, das Vorhandensein besonders einflussreicher Beobachtungen sowie die Linearität von $\ln \lambda(t)$ bzgl. der kontinuierlichen Prädiktoren.

Das logarithmierte kumulative hazard $\ln(-\ln(S(t)))$ muss linear mit $\ln t$ sein. Aus der Annahme proportionaler hazards folgt, dass die separat pro Gruppe geschätzten $\ln(-\ln(S_g(t)))$ par-

allel verlaufen. Diese Voraussetzung lässt sich in einem Diagramm prüfen, das $\ln(-\ln(\hat{S}_g(t_i)))$ gegen $\ln t_i$ aufträgt, wobei $\hat{S}_g(t_i)$ die geschätzten Kaplan-Meier Survival-Funktionen der Überlebenszeit für die Stufen g eines Faktors sind (Abb. 9.4). Dafür ist separat für jeden Prädiktor ein Kaplan-Meier Modell anzupassen und mit `plot(KM-Modell, fun="cloglog")` darzustellen. Damit sich die PH-Annahme bzgl. kontinuierlicher Prädiktoren auf diese Weise prüfen lässt, müssen diese zunächst in Gruppen eingeteilt werden (Abschn. 2.6.7).

Die geschätzten logarithmierten kumulativen hazards sollten im Diagramm linear mit $\ln t_i$ ansteigen und zudem parallel verlaufen. Im Spezialfall des Modells, das für T eine Exponentialverteilung annimmt (Abschn. 9.5), sollte die Steigung jeweils gleich 1 sein. Ist die PH-Annahme offensichtlich für einen Prädiktor verletzt, besteht eine Strategie darin, bzgl. dieses Prädiktors zu stratifizieren – bei kontinuierlichen Variablen nach Einteilung in geeignete Gruppen.

```
> library(survival)                      # für survfit(), cox.zph()
> dfSurv <- transform(dfSurv,    # teile X per Median-Split in 2 Gruppen
+                       Xcut=cut(X, breaks=c(-Inf, median(X), Inf)))

> KMIV   <- survfit(Surv(obsT, status) ~ IV,      # KM-Schätzungen für IV
+                     type="kaplan-meier", data=dfSurv)

> KMXcut <- survfit(Surv(obsT, status) ~ Xcut,    # KM-Schätzungen für X
+                     type="kaplan-meier", data=dfSurv)

# Diagramme ln(-ln(S(t))) gegen ln t
> plot(KMIV, fun="cloglog", main="cloglog-Plot für IV1",
+       xlab="ln t", ylab=expression(ln(-ln(hat(S)[g](t)))), 
+       col=c("black", "blue", "red"), lty=1:3)

> legend(x="topleft", col=c("black", "blue", "red"), lwd=2,
+         lty=1:3, legend=LETTERS[1:3])

> plot(KMXcut, fun="cloglog", main="cloglog-Plot für Xcut",
+       xlab="ln t", ylab=expression(ln(-ln(hat(S)[g](t)))), 
+       col=c("black", "blue"), lty=1:2)

> legend(x="topleft", col=c("black", "blue"), lwd=2,
+         legend=c("lo", "hi"), lty=1:2)
```

Ähnlich wie in der Regressionsdiagnostik (Abschn. 6.5) kann sich die Beurteilung der Voraussetzungen des Cox PH-Modells auch auf die Verteilung von Residuen stützen, insbesondere auf die Schönfeld- und Martingal-Residuen. Beide erhält man mit `residuals(<coxph-Objekt>, type=<Typ>)`, wobei `type` auf "scaledsch" bzw. auf "martingale" zu setzen ist.

`cox.zph(<coxph-Objekt>)` berechnet ausgehend von der Korrelation der Schönfeld-Residuen mit einer Transformation der Überlebenszeit für jeden Prädiktor sowie für das Gesamtmodell einen χ^2 -Test der Nullhypothese, dass die Annahme proportionaler hazards stimmt.

```
> (czph <- cox.zph(fitCPH))
      rho    chisq      p
```

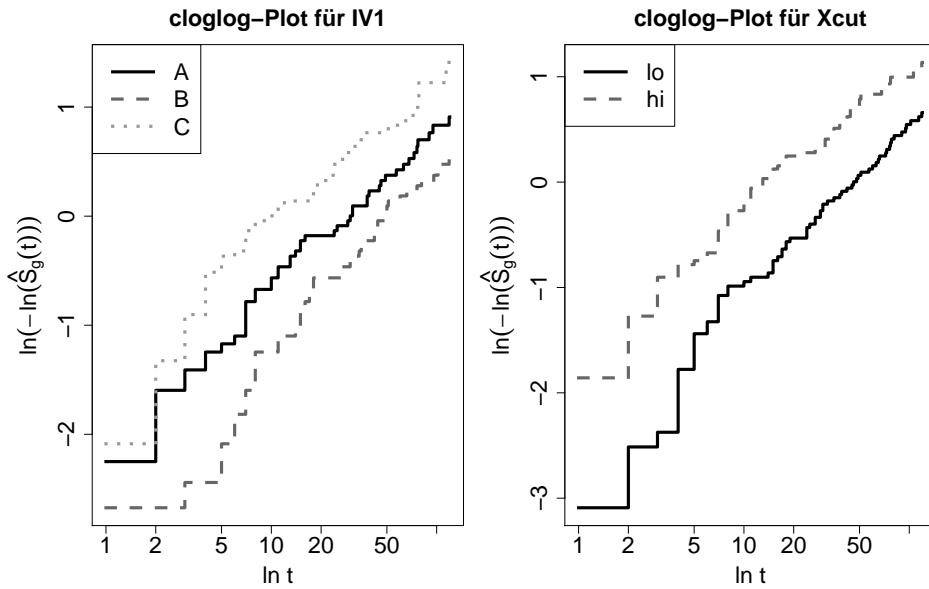


Abbildung 9.4: Beurteilung der Annahme proportionaler hazards bzgl. jedes Prädiktors anhand des Kriteriums linearer paralleler Verläufe von $\ln(-\ln(\hat{S}_g(t_i)))$ mit $\ln t_i$

| | | | |
|--------|---------|--------|-------|
| X | -0.0959 | 1.5316 | 0.216 |
| IVB | 0.1001 | 1.6013 | 0.206 |
| IVC | 0.0216 | 0.0761 | 0.783 |
| GLOBAL | NA | 3.2264 | 0.358 |

Das von `cox.zph()` ausgegebene Objekt lässt sich an `plot()` übergeben, um die Schönfeld-Residuen für jeden Prädiktor gegen eine Transformation der Überlebenszeit darzustellen (Abb. 9.5). Die Diagramme enthalten zur Verdeutlichung des Verlaufs eine Spline-Interpolation (Abschn. 16.1.2) inkl. des zugehörigen Bereichs von ± 2 Standardfehlern. Gibt es eine systematische Variation der Residuen in Abhängigkeit von der Überlebenszeit, ist das ein Hinweis darauf, dass die Annahme proportionaler hazards verletzt ist.

```
> par(mfrow=c(2, 2))                                # Platz für 4 panels
> plot(czph)                                         # Residuen und splines
```

Einflussreiche Beobachtungen können ähnlich wie in der linearen Regression über die von ihnen verursachten Änderungen in den geschätzten Parametern $\hat{\beta}_j$ diagnostiziert werden (Abschn. 6.5.1). Das standardisierte Maß DfBETAS erhält man für jede Beobachtung und jeden Prädiktor, indem man im Aufruf von `residuals()` das Argument `type="dfbetas"` wählt (Abb. 9.6).

```
# Matrix der standardisierten Einflussgrößen DfBETAS
> dfbetas <- residuals(fitCPH, type="dfbetas")
> plot(dfbetas[, 1], type="h", main="DfBETAS für X",
+       ylab="DfBETAS", lwd=2)

> plot(dfbetas[, 2], type="h", main="DfBETAS für IV-B",
+       ylab="DfBETAS", lwd=2)
```

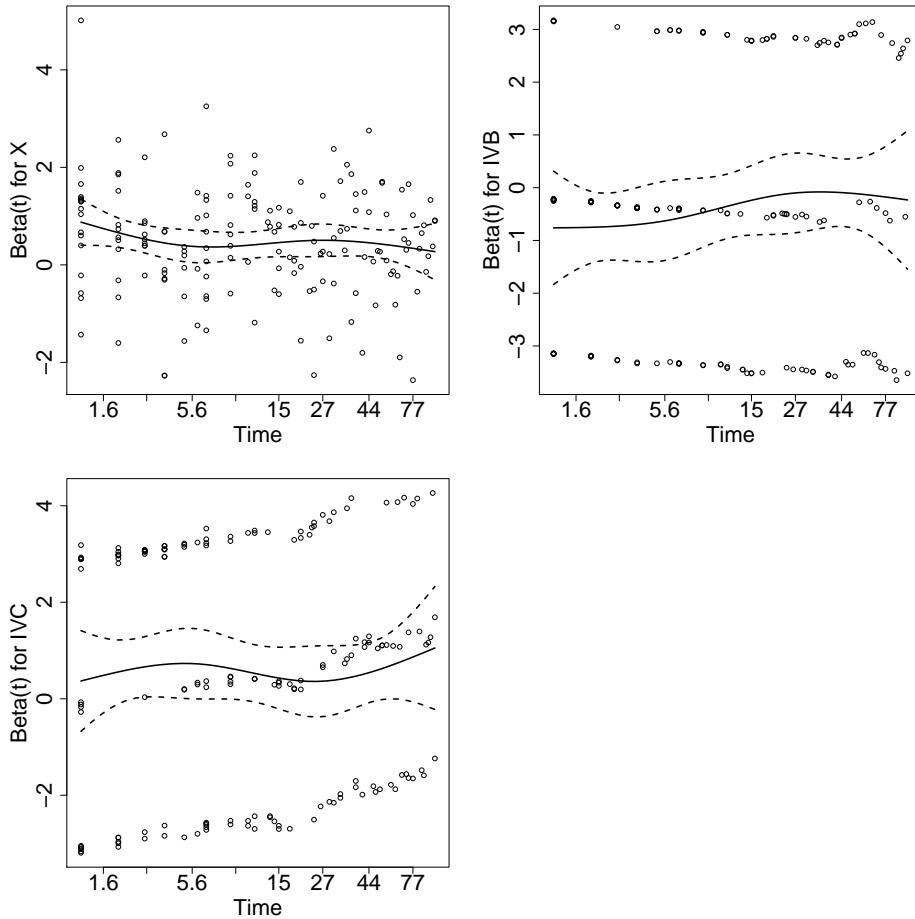


Abbildung 9.5: Beurteilung der Annahme proportionaler hazards bzgl. jedes Prädiktors anhand skalierten Schönfeld-Residuen

```
> plot(dfbetas[, 3], type="h", main="DfBETAS für IV-C",
+       ylab="DfBETAS", lwd=2)
```

Laut Modell sollte der Zusammenhang von $\ln \lambda(t)$ mit den Prädiktoren X_j linear sein. Inwieweit die Daten mit dieser Annahme konsistent sind, lässt sich über den Verlauf der Martingal-Residuen in Abhängigkeit von den Werten der kontinuierlichen X_j einschätzen (Abb. 9.6). Zur grafischen Beurteilung der Verteilung ist es dabei hilfreich, einen nonparametrischen LOESS-Glättter (Abschn. 16.1.3) einzuziehen, der horizontal verlaufen sollte.

```
> resMart <- residuals(fitCPH, type="martingale")
> plot(dfSurv$X, resMart, main="Martingal-Residuen bzgl. X",
+       xlab="X", ylab="Residuen", pch=20)

# zeichne zusätzlich LOESS-Glättter ein
> lines(loess.smooth(dfSurv$X, resMart), lwd=2, col="blue")
> legend(x="bottomleft", col="blue", lwd=2, legend="LOESS fit")
```

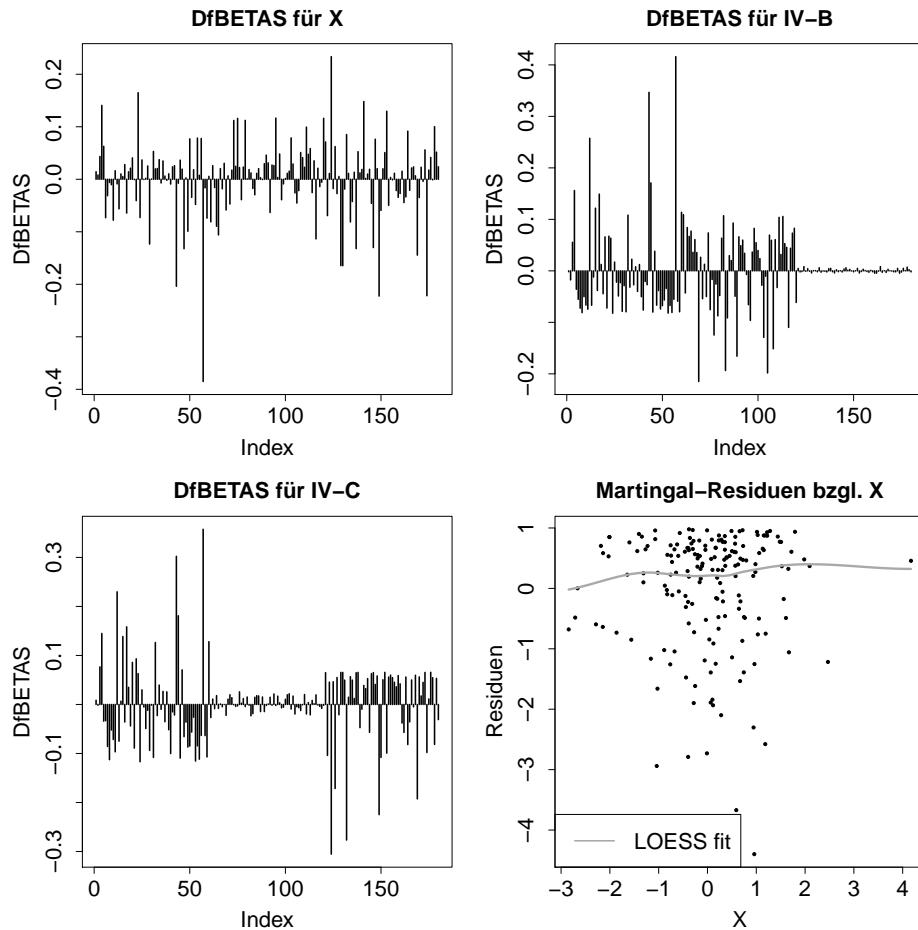


Abbildung 9.6: Diagnostik einflussreicher Beobachtungen anhand der DfBETAS-Werte für jeden Prädiktor sowie Beurteilung der Linearität bzgl. des kontinuierlichen Prädiktors

9.4.4 Vorhersage und Anwendung auf neue Daten

Für jedes Beobachtungsobjekt i liefert `predict(<coxph-Modell>, type="<Typ>")` Schätzungen verschiedener Kennwerte. Mit dem Argument `type="risk"` erhält man die hazard ratios $e^{(x_i - \bar{x})'\hat{\beta}}$ relativ zu einer künstlichen pseudo-Beobachtung, die als Prädiktorwerte den jeweiligen Mittelwert jedes numerischen Prädiktors aus der Stichprobe besitzt und für jeden Faktor die Referenzkategorie (Abschn. 9.4.2). Für stratifizierte Modelle werden diese pseudo-Beobachtungen pro Schicht gebildet, es sei denn man setzt das Argument `reference="sample"`. Für die Werte des linearen Prädiktors $X\hat{\beta}$ selbst ist `type="lp"` zu verwenden.

```
> library(survival) # für predict.coxph()
> pred_risk <- predict(fitCPH, type="risk")
> head(pred_risk, n=4)
[1] 0.5183657 0.9230427 0.8179129 0.5972216
```

Für die manuelle Kontrolle wird zunächst das hazard der pseudo-Beobachtung berechnet, an

dem die hazards der tatsächlichen Beobachtungen relativiert werden.

```
> coef_fit <- coef(fitCPH)

# hazard für künstliche "mittlere" pseudo-Beobachtung
> haz_mean <- exp(coef_fit["X"] *mean(dfSurv$X)
+           + coef_fit["IVB"]*0
+           + coef_fit["IVC"]*0)

# hazard für Beobachtungsobjekte
> haz_obs <- exp(coef_fit["X"] *dfSurv$X
+           + coef_fit["IVB"]*(dfSurv$IV == "B")
+           + coef_fit["IVC"]*(dfSurv$IV == "C"))

> all.equal(haz_obs / haz_mean, pred_risk)
[1] TRUE
```

Zusätzlich akzeptiert das Argument `newdata` von `predict()` einen Datensatz, der neue Daten für Variablen mit denselben Namen, und bei Faktoren auch denselben Stufen wie jene der ursprünglichen Prädiktoren enthält. Als Ergebnis erhält man die vorhergesagten hazard ratios für die neuen Prädiktorwerte relativ zur pseudo-Beobachtung (Abschn. 6.4).

```
> dfNew <- data.frame(X=c(-1, 1),
+                       IV=factor(c("B", "C"), levels=levels(dfSurv$IV)))

> predict(fitCPH, newdata=dfNew, type="risk")
      1          2
0.4027627 2.8808325

# manuelle Kontrolle
> haz_new <- exp(coef_fit["X"] *dfNew$X
+           + coef_fit["IVB"]*(dfNew$IV == "B")
+           + coef_fit["IVC"]*(dfNew$IV == "C"))

> haz_new / haz_mean
[1] 0.4027627 2.8808325
```

9.4.5 Erweiterungen des Cox PH-Modells

Das Cox PH-Modell lässt sich auf verschiedene Situationen erweitern:

- Zeitvariierende Kovariaten X können über die Zählprozess-Darstellung von Daten (Abschn. 9.2.2) repräsentiert und etwa in Interaktionstermen $X \times t$ in die Modellformel von `coxph()` aufgenommen werden. Dafür ist es notwendig, die Gesamt-Beobachtungsintervalle aller Personen mit `survSplit()` an allen vorkommenden Ereignis-Zeitpunkten in Teilintervalle zu zerlegen. Details erläutert `vignette("timedep", package="survival")`.

- Pro Beobachtungsobjekt potentiell mehrfach auftretende Ereignisse lassen sich ebenfalls in Zählprozess-Darstellung speichern. In der Modellformel von `coxph()` kann dann ein Vorhersageterm `cluster(<ID>)` hinzugefügt werden, wobei der Faktor `<ID>` codiert, von welchem Beobachtungsobjekt ein Intervall stammt. Dies bewirkt eine robuste Schätzung der Kovarianzmatrix der Parameterschätzer. Alternativ können wiederkehrende Ereignisse durch Stratifizierung analysiert werden, wobei die Beobachtungen mit jeweils dem ersten, zweiten, dritten, ..., Ereignis ein Stratum bilden.
- Penalisierte Cox-Modelle können mit der Funktion `coxnet()` aus dem Paket `glmnet` (Abschn. 6.6.2) sowie mit dem Paket `coxphf` (Ploner & Heinze, 2018) für die Firth-Korrektur angepasst werden.
- Für Hinweise zur Berücksichtigung von *competing risks* (Beyersmann, Allignol & Schumacher, 2012) vgl. `vignette("compete", package="survival")` und den Abschnitt *Survival Analysis* der CRAN Task Views (Allignol & Latouche, 2020). Dieser Abschnitt geht auch auf *frailty* Modelle ein.

9.5 Parametrische proportional hazards Modelle

Bei spezifischen Vorstellungen über die Verteilung der Überlebenszeit T kommen auch parametrische Regressionsmodelle in Betracht, die sich unter Beibehaltung der Annahme proportionaler hazards als Spezialfälle des Cox PH-Modells ergeben (Abschn. 9.4). Für exponential- oder Weibull-verteilte Überlebenszeiten gibt es dabei zwei äquivalente Möglichkeiten, das lineare Regressionsmodell zu formulieren: Zum einen wie im Cox PH-Modell für das logarithmierte hazard, zum anderen für die logarithmierte Überlebenszeit. Bei der zweiten Darstellung spricht man von einem *accelerated failure time* Modell (AFT). Da in parametrischen Modellen das hazard voll spezifiziert ist, lassen sie sich anders als Cox PH-Modelle auch zur Vorhersage jenseits des letzten beobachteten Ereignisses nutzen.

9.5.1 Darstellung über die Hazard-Funktion

Spezialfälle des Cox PH-Modells ergeben sich, wenn für das baseline hazard $\lambda_0(t)$ eine Verteilung angenommen wird, die mit einer Exponential- oder Weibull-Verteilung von T korrespondiert. Die logarithmierte Ereignisrate soll wie im Cox PH-Modell linear von den Prädiktoren X_j abhängen. Das baseline hazard $\lambda_0(t)$ ist dabei der Verlauf der Ereignisrate für ein Beobachtungsobjekt, für das numerische Prädiktoren X_j gleich 0 bzw. alle Faktoren gleich der Referenzkategorie sind (bei Treatment-Kontrasten). Die exponenzierten Parameter e^{β_j} geben wie im Cox PH-Modell den Änderungsfaktor für die Ereignisrate (also das hazard ratio) an, wenn ein Prädiktor um eine Einheit wächst.

Bei Annahme einer Exponentialverteilung von T mit Erwartungswert $E(T) = b > 0$ und Varianz b^2 ergibt sich die Dichtefunktion $f(t) = \lambda(t) S(t)$ aus der konstanten Hazard-Funktion $\lambda(t) = \lambda = \frac{1}{b}$ und der Survival-Funktion $S(t) = e^{-\frac{t}{b}}$. Die kumulative Hazard-Funktion ist $\Lambda(t) = \frac{t}{b}$. Oft wird die Exponentialverteilung auch mit der Grundrate λ als $f(t) = \lambda e^{-\lambda t}$ bzw. $S(t) = e^{-\lambda t}$ und $\Lambda(t) = \lambda t$ formuliert. In `rexp()` ist mit dem Argument `rate` λ gemeint. Durch den Einfluss der X_j ergibt sich dann als neue Ereignisrate $\lambda' = \lambda e^{X\beta}$. Insgesamt resultieren aus

der Spezialisierung des Cox PH-Modells folgende Modellvorstellungen, wobei die Abkürzung $\mathbf{X}\beta$ keinen absoluten Term β_0 einschließt:

$$\begin{aligned}\lambda(t) &= \frac{1}{b} e^{\mathbf{X}\beta} & = \lambda e^{\mathbf{X}\beta} \\ \ln \lambda(t) &= -\ln b + \mathbf{X}\beta & = \ln \lambda + \mathbf{X}\beta \\ S(t) &= \exp\left(-\frac{t}{b} e^{\mathbf{X}\beta}\right) & = \exp\left(-\lambda t e^{\mathbf{X}\beta}\right) \\ \Lambda(t) &= \frac{t}{b} e^{\mathbf{X}\beta} & = \lambda t e^{\mathbf{X}\beta}\end{aligned}$$

Die Dichtefunktion einer Weibull-Verteilung kann unterschiedlich formuliert werden. `dweibull()` verwendet den Formparameter $a > 0$ für das Argument `shape` und den Skalierungsparameter $b > 0$ für `scale`. Für $a > 1$ steigt das hazard mit t , für $a < 1$ sinkt es, und für $a = 1$ ist es konstant. Die Exponentialverteilung ist also ein Spezialfall der Weibull-Verteilung für $a = 1$. b ist die *charakteristische Lebensdauer*, nach der $1 - \frac{1}{e} \approx 63.2\%$ der Ereignisse aufgetreten sind ($S(b) = \frac{1}{e}$). Die Dichtefunktion $f(t) = \lambda(t) S(t)$ ergibt sich mit dieser Wahl aus der Hazard-Funktion $\lambda(t) = \frac{a}{b} \left(\frac{t}{b}\right)^{a-1}$ und der Survival-Funktion $S(t) = \exp(-(\frac{t}{b})^a)$. Die kumulative Hazard-Funktion ist $\Lambda(t) = (\frac{t}{b})^a$ mit der Umkehrfunktion $\Lambda^{-1}(t) = (bt)^{\frac{1}{a}}$. Der Erwartungswert ist $E(T) = b \Gamma(1 + \frac{1}{a})$.

Analog zur Exponentialverteilung lässt sich die Weibull-Verteilung auch mit $\lambda = \frac{1}{b^a}$ formulieren, so dass $\lambda(t) = \lambda a t^{a-1}$, $S(t) = \exp(-\lambda t^a)$ und $\Lambda(t) = \lambda t^a$ gilt. Durch den Einfluss der X_j ergibt sich dann als neue Ereignisrate $\lambda' = \lambda e^{\mathbf{X}\beta}$. Insgesamt impliziert das Weibull-Modell folgende Zusammenhänge:

$$\begin{aligned}\lambda(t) &= \frac{a}{b} \left(\frac{t}{b}\right)^{a-1} e^{\mathbf{X}\beta} & = \lambda a t^{a-1} e^{\mathbf{X}\beta} \\ \ln \lambda(t) &= \ln \left(\frac{a}{b} \left(\frac{t}{b}\right)^{a-1}\right) + \mathbf{X}\beta & = \ln \lambda + \ln a + (a-1) \ln t + \mathbf{X}\beta \\ S(t) &= \exp\left(-\left(\frac{t}{b}\right)^a e^{\mathbf{X}\beta}\right) & = \exp\left(-\lambda t^a e^{\mathbf{X}\beta}\right) \\ \Lambda(t) &= \left(\frac{t}{b}\right)^a e^{\mathbf{X}\beta} & = \lambda t^a e^{\mathbf{X}\beta}\end{aligned}$$

9.5.2 Darstellung als accelerated failure time Modell

Das betrachtete Exponential- und Weibull-Modell lässt sich äquivalent auch jeweils als lineares Modell der logarithmierten Überlebenszeit formulieren (accelerated failure time Modell, AFT).

$$\ln T = \mathbf{X}\gamma + \mathbf{z} = \boldsymbol{\mu} + \sigma\epsilon$$

Dabei ist ϵ ein Fehlerterm, der im Weibull-Modell einer Typ-I (Gumbel) Extremwertverteilung folgt und durch $\sigma = \frac{1}{a}$ skaliert wird. Sind t_i zufällige Überlebenszeiten aus einer Weibull-Verteilung, sind damit $\ln t_i$ zufällige Beobachtungen einer Extremwertverteilung mit Erwartungswert $\mu = \ln b$. Mit $a = 1$ ergibt sich als Spezialfall das Exponential-Modell.

Ein Parameter γ_j ist im AFT-Modell das über t konstante Verhältnis zweier Quantile von $S(t)$, wenn sich der Prädiktor X_j um eine Einheit erhöht. Ein exponenzielter Parameter e^{γ_j} gibt analog den Änderungsfaktor für die Überlebenszeit bei einer Änderung von X_j um eine Einheit an. Zwischen dem Parameter β_j in der Darstellung als PH-Modell und dem Parameter γ_j in der Formulierung als AFT-Modell besteht für Weibull-verteilte Überlebenszeiten die Beziehung $\beta_j = -\frac{\gamma_j}{a}$, für den Spezialfall exponentialverteilter Überlebenszeiten also $\beta_j = -\gamma_j$.

9.5.3 Anpassung und Modelltests

AFT-Modelle können mit `survreg()` aus dem Paket `survival` angepasst werden.

```
survreg(<Modellformel>, dist=<'Verteilung>", data=<Datensatz>)
```

Als erstes Argument ist eine Modellformel zu übergeben, deren linke Seite ein mit `Surv()` erstelltes Objekt ist (Abschn. 9.2). Dabei ist sicherzustellen, dass alle Ereignis-Zeitpunkte $t_i > 0$ sind und nicht (etwa durch Rundung) Nullen enthalten. Die rechte Seite der Modellformel kann neben – zeitlich konstanten – kontinuierlichen Prädiktoren und Faktoren als besondere Vorhersageterm `strata(<Faktor>)` umfassen. Dieser sorgt dafür, dass ein stratifiziertes Modell angepasst wird, das eine separate Baseline-Hazard-Funktion $\lambda_{0g}(t)$ für jede Stufe g des Faktors beinhaltet. Stammen die in der Modellformel verwendeten Variablen aus einem Datensatz, ist dieser unter `data` zu nennen. Das Argument `dist` bestimmt die für T angenommene Verteilung – mögliche Werte sind unter Annahme proportionaler hazards "`weibull`" oder "`exponential`".⁹

Die Modelle sollen hier für die in Abschn. 9.2.1 simulierten Daten mit zeitlich konstanten Prädiktoren und höchstens einmal auftretenden Ereignissen angepasst werden.

```
> library(survival)                                # für survreg()
> fitWeib <- survreg(Surv(obsT, status) ~ X + IV, dist="weibull",
+                      data=dfSurv)
```

Wald-Tests der Parameter sowie einen Likelihood-Quotienten-Test des Gesamtmodells erhält man mit `summary(<survreg-Objekt>)`.

```
> summary(fitWeib)                                # Parameter- und Modelltests
      Value Std. Error     z      p
(Intercept) 3.423     0.1703 20.10 7.45e-90
X           -0.632    0.1024 -6.17 6.65e-10
IVB          0.504     0.2449  2.06 3.94e-02
IVC          -0.778    0.2327 -3.34 8.29e-04
Log(scale)   0.216     0.0608  3.56 3.73e-04

Scale=1.24

Weibull distribution
Loglik(model)= -695.9  Loglik(intercept only)= -726.9
Chisq= 62.03 on 3 degrees of freedom, p= 2.2e-13
Number of Newton-Raphson Iterations: 5
n=180
```

Die Ausgabe ist weitgehend analog zu jener von `summary(<coxph-Objekt>)` (Abschn. 9.4), wobei in der Spalte `Value` die geschätzten AFT-Parameter $\hat{\gamma}_j = -\hat{\beta}_j \cdot \hat{a}$ genannt werden.¹⁰ Die

⁹Verzichtet man auf die Annahme proportionaler hazards, kommen auch weitere Verteilungen für T in Betracht, über die ?`survreg` Auskunft gibt.

¹⁰Für die $\hat{\beta}_j$ ergibt sich 0.51 (X), -0.41 (IVB) und 0.63 (IVC) – also Schätzungen, die hier denen des Cox PH-Modells sehr ähnlich sind (s. S. 393).

Schätzung \hat{a} des Formparameters der Weibull-Verteilung ist unter `Scale` aufgeführt. Der zugehörige Wald-Test mit der $H_0: \ln a = 0$ steht in der Zeile `Log(scale)`. Eine Alternative hierzu ist der Likelihood-Quotienten-Test des eingeschränkten Modells `<fitR>` mit Exponentialverteilung ($a = 1$, also $\ln a = 0$) gegen das umfassendere Modell `<fitU>` mit Weibull-Verteilung mittels `anova(<fitR>, <fitU>)` (Abschn. 8.1.5).

```
# eingeschränktes Modell mit a=1 -> Exponentialverteilung
> fitExp <- survreg(Surv(obsT, status) ~ X + IV, dist="exponential",
+                      data=dfSurv)

> anova(fitExp, fitWeib)                                # LQ-Modelltest
   Terms Resid. Df    -2*LL Test Df Deviance      Pr(>Chi)
1 X + IV        176 1405.946     NA      NA          NA
2 X + IV        175 1391.839  =  1 14.10752 0.0001726517
```

Da der hier im Modell berücksichtigte Faktor `IV` mit mehreren Parametern γ_j assoziiert ist, muss seine Signifikanz insgesamt über einen Modellvergleich getestet werden. Dazu dient wie beim Vergleich der Modelle mit Exponential- und Weibull-Verteilung ein Likelihood-Quotienten-Test zweier hierarchischer Modelle.

```
# eingeschränktes Modell ohne Faktor IV
> fitR <- survreg(Surv(obsT, status) ~ X, dist="weibull", data=dfSurv)
> anova(fitR, fitWeib)                                # LQ-Modelltest für Faktor IV
   Terms Resid. Df    -2*LL Test Df Deviance      Pr(>Chi)
1      X        177 1418.773
2 X + IV       175 1391.839  +IV  2 26.93433 1.416721e-06
```

9.5.4 Survival-Funktion schätzen

Die geschätzte Verteilungsfunktion $\hat{F}(t)$ für ein mit `survreg()` angepasstes Modell ermittelt `predict()` (Abschn. 6.4). Die meist stattdessen betrachtete geschätzte Survival-Funktion ergibt sich daraus als $\hat{S}(t) = 1 - \hat{F}(t)$.

```
predict(<survreg-Objekt>, newdata=<Datensatz>, type="quantile",
       p=<Quantile>, se=TRUE)
```

Als erstes Argument ist ein `survreg`-Objekt zu übergeben. Das Argument `newdata` erwartet einen Datensatz, der neue Daten für Variablen mit denselben Namen, und bei Faktoren auch denselben Stufen wie jene der ursprünglichen Prädiktoren im `survreg`-Objekt enthält. Mit dem Argument `type="quantile"` liefert `predict()` für jede Zeile in `newdata` für das Quantil $p \in (0, 1)$ den Wert $\hat{F}^{-1}(p)$. Dies ist die Überlebenszeit t_p , für die bei den in `newdata` gegebenen Gruppenzugehörigkeiten und Prädiktorwerten $\hat{F}(t_p) = p$ gilt. Dafür ist an `p` ein Vektor mit Quantilen zu übergeben, deren zugehörige Werte von $\hat{F}(t)$ gewünscht werden. Den geschätzten Median der Überlebenszeit erfährt man etwa mit `p=0.5`, während für einen durchgehenden Funktionsgraphen von $\hat{F}(t)$ bzw. $\hat{S}(t)$ eine fein abgestufte Sequenz im Bereich $(0, 1)$ angegeben werden muss. Setzt man `se=TRUE`, erhält man zusätzlich noch die geschätzte Streuung von $\hat{F}(t)$ (Abb. 9.7).

Mit `se=TRUE` ist das zurückgegebene Objekt eine Liste mit den Werten von $\hat{F}^{-1}(p)$ in der Komponente `fit` und den geschätzten Streuungen in der Komponente `se.fit`. Umfasst `newdata` mehrere Zeilen, sind `fit` und `se.fit` Matrizen mit einer Zeile pro Beobachtungsobjekt und einer Spalte pro Quantil.¹¹

```
# Datensatz: 2 Männer mit Prädiktor X=0 in Gruppe A bzw. in C
> dfNew <- data.frame(sex=factor(c("m", "m"),
+                                 levels=levels(dfSurv$sex)),
+                         X=c(0, 0),
+                         IV=factor(c("A", "C"), levels=levels(dfSurv$IV)))

# geschätzte Werte von F^(-1)
> percs <- (1:99)/100 # Perzentile = Quantile
> FWeib <- predict(fitWeib, newdata=dfNew, type="quantile",
+                    p=percs, se=TRUE)

# stelle geschätzte Survival-Funktion S(t) statt F(t) dar -> 1-percs
# zunächst für Beobachtungsobjekt 1
> matplot(cbind(FWeib$fit[1, ],
+                 FWeib$fit[1, ] - 2*FWeib$se.fit[1, ],
+                 FWeib$fit[1, ] + 2*FWeib$se.fit[1, ]),
+            1-percs,
+            type="l", main=expression(paste("Weibull-Fit ", hat(S)(t),
+            " mit SE")), xlab="t", ylab="Survival", lty=c(1, 2, 2),
+            lwd=2, col="blue")

# für Beobachtungsobjekt 2
> matlines(cbind(FWeib$fit[2, ],
+                  FWeib$fit[2, ] - 2*FWeib$se.fit[2, ],
+                  FWeib$fit[2, ] + 2*FWeib$se.fit[2, ]),
+                 1-percs,
+                 col="red", lwd=2)

> legend(x="topright", lwd=2, lty=c(1, 2, 1, 2),
+          col=c("blue", "blue", "red", "red"),
+          legend=c("sex=m, X=0, IV=A", "+- 2*SE",
+                  "sex=m, X=0, IV=C", "+- 2*SE"))
```

¹¹ Abschnitt 9.4.2 demonstriert die analoge Verwendung von `newdata` in `survfit()`.

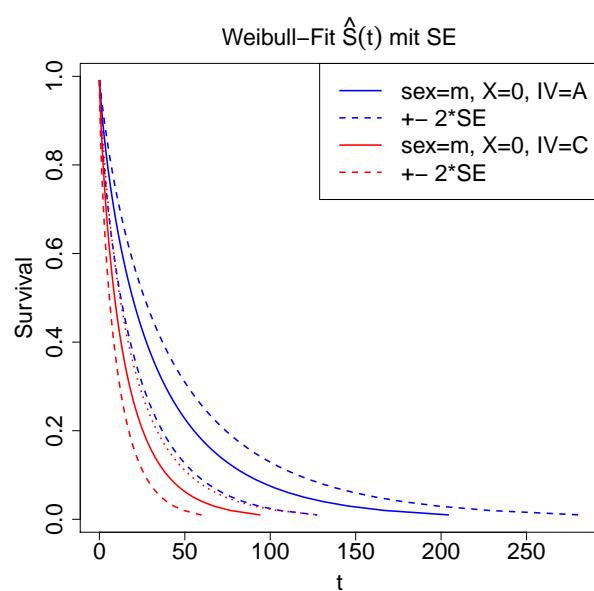


Abbildung 9.7: Schätzung der Survival-Funktion $\hat{S}(t)$ aus Weibull-Modell für zwei männliche Personen mit Prädiktorwert $X = 0$ aus Gruppe A bzw. C

Kapitel 10

Klassische nonparametrische Methoden

Wenn inferenzstatistische Tests zur Datenauswertung herangezogen werden sollen, aber davon ausgegangen werden muss, dass strenge Anforderungen an die Art und Verteilung der erhobenen Daten nicht erfüllt sind, kommen viele konventionelle Verfahren womöglich nicht in Betracht. Dagegen haben nonparametrische Methoden weniger restriktive Voraussetzungen und kommen auch bei kleinen Stichproben in Frage ([Bortz, Lienert & Boehnke, 2010](#); [Büning & Trenkler, 1994](#)). Auch für (gemeinsame) Häufigkeiten kategorialer Variablen sowie ordinale Daten¹ sind viele der folgenden Methoden geeignet und ergänzen damit die in Kap. 8 vorgestellten Modelle.

Einige der klassischen nonparametrischen Methoden approximieren die unbekannte oder nicht praktisch berechenbare Verteilung der verwendeten Teststatistik durch eine mit wachsender Stichprobengröße asymptotisch gültige Verteilung, die analytisch bestimmbar ist. Einen anderen Weg gehen die in Kap. 11 vorgestellten Resampling-Verfahren (bootstrap und Permutationstests), die die Verteilung der Teststatistik allein auf Basis der gezogenen Stichprobe und vieler zufälliger neuer Versionen von ihr schätzen.

10.1 Anpassungstests

Viele Tests setzen voraus, dass die Verteilung der Zielgröße 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.

Solche *goodness-of-fit* Tests werden meist mit der H_0 durchgeführt, dass eine bestimmte Verteilung vorliegt und dieses Vorliegen auch den gewünschten Zustand beschreibt. Da hier die H_1 gegen die fälschliche Nicht-Annahme abzusichern ist, muss der β -Fehler kontrolliert, also eine gewisse power gesichert werden. Angemessene Herangehensweisen beschreibt [Wellek \(2010\)](#).

¹ Auf Rangdaten basierende Tests machen häufig die Voraussetzung, dass die Ränge eindeutig bestimmbar sind, also keine gleichen Werte (*Bindungen, 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.

10.1.1 Binomialtest

Der Binomialtest ist auf Daten von Variablen anzuwenden, die nur zwei Ausprägungen annehmen können. Eine Ausprägung soll dabei als *Treffer* bzw. *Erfolg* bezeichnet werden. Der Test prüft, ob die empirische Auftretenshäufigkeit eines Treffers in einer Stichprobe verträglich mit der H_0 einer bestimmten Trefferwahrscheinlichkeit p_0 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. 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 `xtabs()`. Unter `p` ist p_0 einzutragen. Das Argument `alternative` bestimmt, ob zweiseitig ("two.sided"), links- ("less") oder rechtsseitig ("greater") getestet wird. Die Aussage bezieht sich dabei auf die Reihenfolge p_1 "less" bzw. "greater" p_0 , mit p_1 als Trefferwahrscheinlichkeit unter H_1 . Mit dem Argument `conf.level` wird die Breite des Konfidenzintervalls für p festgelegt.²

Als Beispiel sollen aus einer (unendlich großen) Urne zufällig Lose gezogen werden, wobei $p_0 = 0.25$ ist. Es sei nach der Wahrscheinlichkeit gefragt, bei 7 Ziehungen mindestens die beobachteten 5 Treffer zu ziehen $p_1 > p_0$.

```
> 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 enthält neben einer Zusammenfassung der eingegebenen Daten (`number of successes` und `number of trials`, `probability of success`) den p -Wert (`p-value`). Schließlich wird je nach Fragestellung das zweiseitige, links- oder rechtsseitige Konfidenzintervall für die Trefferwahrscheinlichkeit in der gewünschten Breite genannt. Der ausgegebene p -Wert kann manuell mit Hilfe der Verteilungsfunktion der Binomialverteilung verifiziert werden (Abschn. 5.3.2, Fußnote 10).

```
> (pVal <- pbinom(hits-1, draws, pH0, lower.tail=FALSE))
[1] 0.01287842
```

²Das Intervall ist jenes nach Clopper-Pearson. Für die Berechnung u. a. nach Wilson, Agresti-Coull und Jeffreys vgl. `BinomCI()` aus dem Paket `DescTools`.

Für den zweiseitigen Binomialtest existieren verschiedene Definitionen des p -Wertes als Wahrscheinlichkeit unter H_0 , mindestens so extreme Trefferzahlen wie die beobachtete zu erhalten:

1. R summiert die Wahrscheinlichkeiten unter Gültigkeit der H_0 für alle Ereignisse mit höchstens der Wahrscheinlichkeit des eingetretenen Ereignisses. So ist etwa der von `binom.test(10, 20, p=0.25, alternative="two.sided")` ausgegebene p -Wert gleich `sum(dbinom(10:20, 20, 0.25)) + sum(dbinom(0, 20, 0.25))`. Die „Extremheit“ eines Ereignisses wird hier also i. S. seiner Wahrscheinlichkeit verstanden.
2. Zwei andere Definitionen beziehen sich auf die separaten einseitigen Tests: Für sie muss im Beispiel zum einen die Wahrscheinlichkeit von 5 oder mehr Gewinnen unter H_0 berechnet werden, zum anderen die Wahrscheinlichkeit von 2 (also 7–5) Treffern oder weniger. Der zweiseitige p -Wert kann nun als das Doppelte des kleineren der p -Werte der einseitigen Tests definiert werden.
3. Als weitere Möglichkeit können die p -Werte für jede der beiden einseitigen Fragestellungen addiert werden.

Wenn unter H_0 mit $p = 0.5$ eine symmetrische Binomialverteilung vorliegt, führen die genannten Definitionen zum selben Ergebnis. Nach einer gängigen Konvention ist beim zweiseitigen Test auf jeder Seite $\frac{\alpha}{2}$ einzuhalten, auch wenn bei Überschreitung von $\frac{\alpha}{2}$ auf einer Seite dennoch insgesamt $p < \alpha$ wäre.

```
# 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

# Gesamt-p-Werte
> 2 * min(c(resG$p.value, resL$p.value))      # Variante 2
[1] 0.02575684
```

Hypothesen über die jeweiligen Trefferwahrscheinlichkeiten einer dichotomen Variable in zwei unabhängigen Stichproben lassen sich mit Fishers exaktem Test prüfen (Abschn. 10.2.5), in zwei abhängigen Stichproben mit dem McNemar-Test (Abschn. 10.5.10), in mehr als zwei unabhängigen Stichproben mit einem χ^2 -Test (Abschn. 10.2.3) und in mehr als zwei abhängigen Stichproben mit Cochrancs Q -Test (Abschn. 10.5.8).

10.1.2 Test auf Zufälligkeit (Runs-Test)

Eine zufällige Reihenfolge von N Ausprägungen einer dichotomen Variable sollte sich dann ergeben, wenn die Erfolgswahrscheinlichkeit für jede Realisierung konstant p beträgt und die Realisierungen unabhängig sind. 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*, Abschn. 2.12.2), wobei sowohl eine sehr geringe als auch eine sehr hohe Anzahl gegen die H_0 spricht.

Eine spezielle Funktion für den Test auf Zufälligkeit ist nicht im Basisumfang von R enthalten,³ eine manuelle Durchführung jedoch möglich. Als Beispiel diene jenes aus Bortz et al. (2010, 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
> Nj    <- xtabs(~ 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 <- xtabs(~ runs$values)[1])                            # Iterationen Gruppe 1
f
3

> (rr2 <- xtabs(~ runs$values)[2])                            # Iterationen Gruppe 2
m
2
```

Für den *p*-Wert des beobachteten Wertes von *R* sind die Punktwscheinlichkeiten für alle Fälle aufzuaddieren, die dieses *R* oder extremere Werte ergeben. Die Berechnung der Punktwscheinlichkeiten geschieht hier mit einer eigens erstellten Funktion (Abschn. 17.3). 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. Hier sind 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 Punktwscheinlichkeiten zu addieren sind.

```
# Funktion, um Punktwscheinlichkeit für Anzahl von Iterationen der
# Gruppe 1 (r1), Gruppe 2 (r2), mit Gruppengrößen n1 & n2 zu berechnen
> getP <- function(r1, r2, n1, n2) {
+   # Punktwscheinlichkeit für r1+r2 ungerade
+   p <- (choose(n1-1, r1-1) * choose(n2-1, r2-1)) / choose(n1+n2, n1)
+
+   # Punktwscheinlichkeit für r1+r2 gerade: 2*ungerade
+   ifelse((r1+r2) %% 2 == 0, 2*p, p)
+ }

> n1    <- Nj[1]                                              # Größe Gruppe 1
```

³Vergleiche hierfür `RunsTest()` aus dem `DescTools` Paket.

```

> n2    <- Nj[2]                                # Größe Gruppe 2
> N     <- sum(Nj)                             # Gesamt-N
> rMin <- 2                                    # Untergrenze für Anzahl der Iterationen

# Obergrenze Anzahl Iterationen, Fallunterscheidung: n1 == n2?
> (rMax <- ifelse(n1 == n2, N, 2*min(c(n1, n2)) + 1))
[1] 7

# addiere Punktwscheinlichkeiten für R=beobachtet und größer
> p3.2   <- getP(3, 2, n1, n2)                 # r1=3, r2=2 -> R=5
> p2.3   <- getP(2, 3, n1, n2)                 # r1=2, r2=3 -> R=5
> p3.3   <- getP(3, 3, n1, n2)                 # r1=3, r2=3 -> R=6
> p4.3   <- getP(4, 3, n1, n2)                 # r1=4, r2=3 -> R=7
> (pGrEq <- p3.2 + p2.3 + p3.3 + p4.3)      # p-Wert einseitig
0.5714286

# Punktwscheinlichkeit aller anderen Fälle
> p2.2   <- getP(2, 2, n1, n2)                 # r1=2, r2=2 -> R=4
> p1.2   <- getP(1, 2, n1, n2)                 # r1=1, r2=2 -> R=3
> p2.1   <- getP(2, 1, n1, n2)                 # r1=2, r2=1 -> R=3
> p1.1   <- getP(1, 1, n1, n2)                 # r1=1, r2=1 -> R=2
> (pLess <- p2.2 + p1.2 + p2.1 + p1.1)
0.4285714

> pGrEq + pLess                               # Kontrolle: Summe beider Wkt. = 1
[1] 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*n1*n2) / N)                # Erwartungswert von R
> varR  <- (2*n1*n2*(2*n1*n2 - N)) / (N^2 * (N-1))  # Varianz von R
> rZ    <- (rr-muR) / sqrt(varR)              # z-Transformiertes R
> (pVal <- pnorm(rZ, lower.tail=FALSE))        # p-Wert einseitig
0.4184066

```

In Form des Wald-Wolfowitz-Tests lässt sich auch prüfen, ob zwei Stichproben 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.

10.1.3 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 H_0 , dass die Verteilungsfunktion gleich der angegebenen ist, kann eine ungerichtete wie gerichtete H_1 getestet werden. Der Test lässt sich durch eine visuell-explorative Analyse mittels eines Quantil-Quantil-Diagramms ergänzen (Abschn. 14.6.5).

```
ks.test(x=<Vektor>, y=<Verteilungsfunktion>, ...,
        alternative=c("two.sided", "less", "greater"))
```

Das Argument **x** benötigt den Vektor der Daten. Soll eine Variable aus einem Datensatz geprüft werden, ist alternativ eine Modellformel der Form **<Variable> ~ 1** möglich, wobei der Datensatz unter **data** zu nennen ist. **y** ist die Verteilungsfunktion der Variable unter H_0 . Um durch Komma getrennte Argumente an diese Verteilungsfunktion zu ihrer genauen Spezifikation übergeben zu können, dienen die ... Auslassungspunkte. Mit **alternative** wird die H_1 definiert, wobei sich "less" und "greater" darauf beziehen, ob **y** stochastisch kleiner oder größer als **x** ist.⁴

Im Beispiel soll zum Test die Verteilungsfunktion der Normalverteilung mit Erwartungswert $\mu = 1$ und Streuung $\sigma = 2$ herangezogen werden.

```
> DV <- rnorm(8, mean=1.5, sd=3)                      # Daten
> ks.test(DV, y=pnorm, mean=1, sd=2, alternative="two.sided")
One-sample Kolmogorov-Smirnov test
data: DV
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 gerichtete Tests manuell berechnet.

```
> Fn      <- ecdf(DV)      # Funktion für kumulierte relative Häufigkeiten
> sortDV <- sort(DV)      # sortierte Daten
> emp     <- Fn(sortDV)   # kumulierte relative Häufigkeiten

# theoretische Werte: Verteilungsfkt. Normalverteilung N(mu=1, sigma=2)
> theo   <- pnorm(sortDV, mean=1, sd=2)
> diff1 <- emp - theo      # direkte Differenzen
> diff2 <- c(0, emp[-length(emp)]) - theo      # verschobene Differenzen

# 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))))
```

⁴Bei 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.

```
[1] 0.2325919

# Teststatistik für "greater": Betrag stärkste Abweichung nach oben
> (Dgreat <- abs(max(c(diff1, diff2))))
[1] 0.09828234

# Kontrolle über Ausgabe von ks.test()
> ks.test(DV, y=pnorm, mean=1, sd=2, alternative="less")$statistic
D^-
0.2325919

> ks.test(DV, y=pnorm, mean=1, sd=2, alternative="greater")$statistic
D^+
0.09828234
```

Abbildung 10.1 veranschaulicht die für den Test relevanten Differenzen zwischen empirischer und angenommener Verteilungsfunktion.

```
> plot(Fn, main="Kolmogorov-Smirnov-Anpassungstest", xlab=NA)
> curve(pnorm(x, mean=1, sd=2), n=200, add=TRUE) # Verteilungsfunktion

# direkte Abweichungen
> matlines(rbind(sortDV, sortDV), rbind(emp, theo),
+           col=rgb(0, 0, 1, 0.7), lty=1, lwd=2)

# verschobene Abweichungen
> matlines(rbind(sortDV, sortDV), 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 H_0 vorliegt, die nur die Verteilungsklasse benennt. Für einen korrekten Test dürfen die Parameter nicht auf Basis der Daten in \mathbf{x} geschätzt werden. Abschnitt 7.1 stellt dafür Alternativen vor.

Der Kolmogorov-Smirnov-Test lässt sich auch verwenden, um die Daten einer stetigen Variable aus zwei Stichproben daraufhin zu prüfen, ob sie mit der H_0 verträglich sind, dass die Variable in beiden Bedingungen dieselbe Verteilung besitzt (Abschn. 10.5.1).

10.1.4 χ^2 -Test auf eine feste Verteilung

Der χ^2 -Test auf eine feste Verteilung prüft Daten einer kategorialen Variable daraufhin, ob die empirischen Auftretenshäufigkeiten der einzelnen Kategorien verträglich mit einer theo-

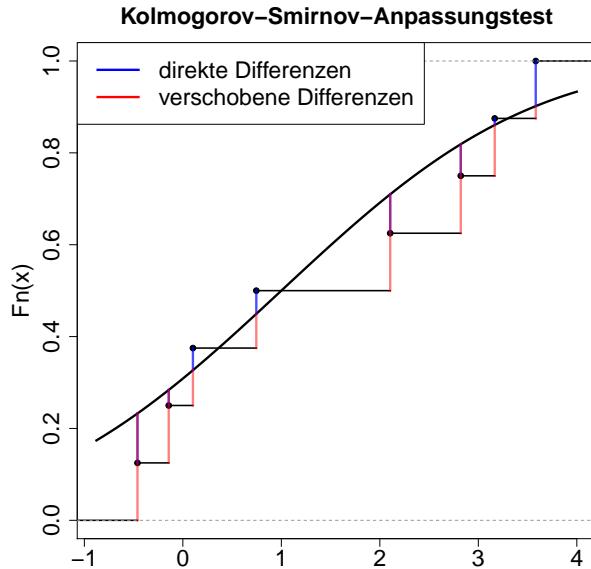


Abbildung 10.1: Kolmogorov-Smirnov-Anpassungstest: Abweichungen zwischen kumulierten relativen Häufigkeiten der Daten und der Verteilungsfunktion der Normalverteilung $\mathcal{N}(\mu = 1, \sigma = 2)$

retischen Verteilung unter H_0 sind, die die Wahrscheinlichkeit jeder Kategorie angibt.⁵ Als ungerichtete H_1 ergibt sich, dass die tatsächliche Verteilung nicht gleich der unter H_0 genannten ist.

```
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 Häufigkeitstabelle als Ergebnis von `xtabs()` sein. Unter p ist ein Vektor einzutragen, der die Wahrscheinlichkeiten für das Auftreten der Kategorien unter H_0 enthält und dieselbe Länge wie x haben muss. Treten erwartete Häufigkeiten von Kategorien (das Produkt der Klassenwahrscheinlichkeiten mit der Stichprobengröße) < 5 auf, sollte das Argument `simulate.p.value` auf `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 Würfel daraufhin getestet werden, ob er fair ist. Die Auftretenswahrscheinlichkeit jeder Augenzahl unter H_0 beträgt $\frac{1}{6}$.

```
> nRolls <- 50                                # Anzahl Ziehungen
> nCateg <- 6                                  # Anzahl Kategorien
> pH0    <- rep(1/nCateg, nCateg)              # Verteilung unter H0
> myData <- sample(1:nCateg, nRolls, replace=TRUE) # simulierte Daten
> (tab   <- xtabs(~ myData))                  # Häufigkeiten
```

⁵Dieser Test ist auch bei quantitativen Variablen durchführbar, wobei zunächst eine Einteilung der Werte in disjunkte Klassen vorzunehmen ist. Die getestete H_0 ist dann in dem Sinne schwächer, dass alle Verteilungen äquivalent sind, die zu gleichen Klassenwahrscheinlichkeiten führen.

```

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` gleich FALSE ist) samt zugehörigem p -Wert (p-value). Das Ergebnis lässt sich manuell prüfen:

```

> expected <- pH0 * nRolls      # erwartete Häufigkeiten unter H0
> (statChisq <- sum((tab-expected)^2 / expected))    # Teststatistik
[1] 12.16

> (pVal <- pchisq(statChisq, nCateg-1, lower.tail=FALSE))    # p-Wert
[1] 0.03265998

```

10.1.5 χ^2 -Test auf eine Verteilungsklasse

Als Spezialfall des χ^2 -Tests auf Gleichheit von Verteilungen (Abschn. 10.2.2) kann die Verträglichkeit der Daten mit der H_0 getestet werden, dass die zugehörige theoretische Verteilung etwa aus der Familie der Normalverteilungen stammt – andere Verteilungsfamilien lassen sich analog testen. Die H_1 ist ungerichtet, der Test asymptotisch korrekt bei wachsender Stichprobengröße und kann durch eine visuell-explorative Begutachtung eines Quantil-Quantil-Diagramms ergänzt werden (Abschn. 14.6.5, 15.2.5).

Die Daten sind zunächst in disjunkte Klassen einzuteilen, deren empirische Auftretenshäufigkeiten mit den unter H_0 erwarteten verglichen werden.⁶ Die Parameter der konkreten Normalverteilung unter H_0 folgen entweder aus theoretischen Erwägungen oder werden auf Basis der Daten geschätzt, häufig anhand des Mittelwertes und der korrigierten Stichprobenstreitung. Durch eine solche Schätzung beider Parameter aus den Daten reduziert sich die Anzahl der Freiheitsgrade beim Test um 2.⁷

Das Paket `DescTools` enthält mit `PearsonTest()` eine Funktion für den χ^2 -Test auf Normalverteiltheit, über die sowohl die Klasseneinteilung wie auch die Korrektur der Freiheitsgrade per Argument festgelegt werden können.

```
PearsonTest(x=<Vektor>, n.classes=<Anzahl>, adjust=TRUE)
```

⁶ Die Klassenbildung führt dazu, dass statt der Verträglichkeit mit einer bestimmten Verteilung die schwächere H_0 einer Verträglichkeit mit allen Verteilungen getestet wird, die zu denselben erwarteten Häufigkeiten führen.

⁷ Für eine korrekte Testkonstruktion wäre eigentlich eine feste Klasseneinteilung mit gruppierten Maximum-Likelihood- oder Minimum- χ^2 -Schätzung von μ und σ notwendig, die in Abschn. 16.4 demonstriert wird.

Der Datenvektor wird der Funktion als Argument `x` übergeben, `n.classes` legt die Zahl der zu bildenden Klassen fest. Die Funktion wählt die genaue Lage der Klassengrenzen so, dass alle Klassen unter H_0 dieselbe Wahrscheinlichkeit besitzen. Über `adjust` wird angegeben, ob die Zahl der Freiheitsgrade in Folge einer Schätzung der Parameter der Normalverteilung unter H_0 aus den Daten um 2 nach unten zu korrigieren ist.

```
> DV    <- rnorm(100, mean=100, sd=15)          # simulierte Daten
> nCls <- 6                                     # Anzahl Klassen
> library(DescTools)                            # für PearsonTest()
> PearsonTest(DV, n.classes=nCls, adjust=TRUE)
Pearson chi-square normality test
data: DV
P = 3.08, p-value = 0.3795
```

Die Ausgabe nennt den empirischen χ^2 -Wert unter `P` zusammen mit dem zugehörigen p -Wert. Für einen manuellen Test sollen ebenfalls unter H_0 gleichwahrscheinliche Intervalle als Basis dienen. Deren innere Grenzen liefert `qnorm()`, wobei Erwartungswert und Streuung aus den Daten geschätzt werden.

```
# innere Intervallgrenzen für gleichwahrscheinliche Intervalle
> limits <- qnorm(seq(1/nCls, (nCls-1)/nCls, length.out=nCls-1),
+                  mean(DV), sd(DV))

> DVcut      <- cut(DV, c(-Inf, limits, Inf))  # +äußere Intervallgrenzen
> (intFreq <- xtabs(~ DVcut))                      # beobachtete Häufigkeiten
DVcut
(-Inf,87.3] (87.3,95.1] (95.1,101] (101,108] (108,115] (115, Inf]
      19        14        14        16        22        15
```

Beim Vergleich von beobachteten und erwarteten Klassenhäufigkeiten ist zu berücksichtigen, dass der von `chisq.test()` ausgegebene p -Wert nicht die Reduktion der Freiheitsgrade widerspiegelt und deshalb für diesen Test nicht der richtige ist. Ist `<Objekt>` das Ergebnis von `chisq.test()`, muss die Bestimmung des korrekten p -Wertes manuell mit dem empirischen Wert der Teststatistik, den richtigen Freiheitsgraden und der Verteilungsfunktion `pchisq(<Quantil>, <Freiheitsgrade>)` erfolgen.

```
# Test für gleiche Klassenwahrscheinlichkeiten, falsche Freiheitsgrade
> (resChisq <- chisq.test(intFreq, p=rep(1/nCls, nCls)))
Chi-squared test for given probabilities
data: intFreq
X-squared = 3.08, df = 5, p-value = 0.6877

# Test mit den korrigierten Freiheitsgraden und p-Wert
> statChisq <- resChisq$statistic           # Teststatistik chi^2
> dfChisq   <- resChisq$parameter - 2       # korrigierte Freiheitsgrade
> (pval     <- pchisq(statChisq, dfChisq, lower.tail=FALSE))
0.3794545
```

10.2 Analyse von gemeinsamen Häufigkeiten kategorialer Variablen

Inwieweit die gemeinsamen empirischen Auftretenshäufigkeiten der Ausprägungen kategorialer Variablen theoretischen Vorstellungen entsprechen, kann durch folgende Tests geprüft werden. Sie ergänzen die in Kap. 8 vorgestellten Regressionsmodelle für bestimmte Spezialfälle. Eine vertiefte Darstellung gibt das Buch von Fagerland, Lydersen und Laake (2017), dessen Methoden und Daten im Paket `contingencytables` (Fagerland, 2024) enthalten sind.

10.2.1 χ^2 -Test auf Unabhängigkeit

Beim χ^2 -Test auf Unabhängigkeit wird die empirische Kontingenztafel von zwei an derselben Stichprobe erhobenen kategorialen Variablen daraufhin geprüft, ob sie verträglich mit der H_0 ist, dass beide Variablen unabhängig sind.⁸ Die H_1 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 `xtabs(~ <Faktor1> + <Faktor2>)`. 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 einer zweiten Variable zu sein hat.

Unter H_0 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 < 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.

```
> N      <- 50                      # Stichprobengröße
> smokes <- factor(sample(c("no", "yes"), N, replace=TRUE))
> siblings <- factor(round(abs(rnorm(N, 1, 0.5))))
> cTab    <- xtabs(~ 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
```

⁸Dieser Test ist auch bei quantitativen Variablen durchführbar, wobei zunächst eine Einteilung der Werte in disjunkte Klassen vorzunehmen ist. Die getestete H_0 ist dann in dem Sinne schwächer, dass nur die Unabhängigkeit bzgl. der vorgenommenen Klasseneinteilung getestet wird.

```

data: cTab
X-squared = 4.6667, df = 2, p-value = 0.09697

Warning message:
In chisq.test(cTab) : Chi-squared approximation may be incorrect

```

Das Ergebnis lässt sich manuell kontrollieren:

```

> P <- nlevels(smokes)                      # Anzahl smokes Kategorien
> Q <- nlevels(siblings)                    # Anzahl siblings Kategorien

# erwartete Häufigkeiten: Summe aller 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

> (pVal <- pchisq(statChisq, (P-1)*(Q-1), lower.tail=FALSE)) # p-Wert
[1] 0.09697197

```

Siehe Abschn. 8.5 für eine Verallgemeinerung zur Analyse höherdimensionaler Kontingenztafeln absoluter Häufigkeiten mit Hilfe log-linearer Modelle.

10.2.2 χ^2 -Test auf Gleichheit von Verteilungen

Beim χ^2 -Test auf Gleichheit von bedingten Verteilungen werden die empirischen eindimensionalen Häufigkeitsverteilungen einer kategorialen Variable in unterschiedlichen Stichproben daraufhin geprüft, ob sie mit der H_0 verträglich sind, dass ihre Stufen in allen Bedingungen jeweils dieselben Auftretenswahrscheinlichkeiten besitzen.⁹ Die H_1 ist ungerichtet, der Test asymptotisch korrekt bei wachsender Stichprobengröße. Getestet wird wie in Abschn. 10.2.1 mit `chisq.test()` auf Basis der Kontingenztafel gemeinsamer Häufigkeiten.

Als Beispiel soll die politische Orientierung von 100 Studenten aus zwei Studiengängen getestet werden. Zielgröße 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     <- xtabs(~ studies + vote)                # Kontingenztafel
> addmargins(cTab)                                    # Randsummen
            vote
studies   A   B   C   D   E   Sum

```

⁹Dieser Test ist auch bei quantitativen Variablen durchführbar, wobei zunächst eine Einteilung der Werte in disjunkte Klassen vorzunehmen ist. Die getestete H_0 ist dann in dem Sinne schwächer, dass nur die Gleichheit der Verteilungen bzgl. der vorgenommenen Klasseneinteilung getestet wird.

| | X | 3 | 8 | 12 | 19 | 8 | 50 |
|-----|----|----|----|----|----|-----|----|
| Y | 8 | 17 | 16 | 7 | 2 | 50 | |
| Sum | 11 | 25 | 28 | 26 | 10 | 100 | |

```
# simulate.p.value=TRUE wegen geringer erwarteter Häufigkeiten
> chisq.test(cTab, simulate.p.value=TRUE)
Pearson's Chi-squared test with simulated p-value
(based on 2000 replicates)
data: cTab
X-squared = 15.2226, df = NA, p-value = 0.005497
```

10.2.3 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 dichotomen Variable in mehreren Bedingungen. Der Test ist asymptotisch korrekt bei wachsender Stichprobengröße, die H_1 ist ungerichtet.

```
prop.test(x=<Erfolge>, n=<Stichprobenumfänge>)
```

Die Argumente `x` und `n` beziehen sich auf die Anzahl der Treffer und die Stichprobengrößen in den Gruppen. Für sie können Vektoren gleicher Länge mit Einträgen für jede Gruppe angegeben werden. Anstelle von `x` und `n` ist auch eine Matrix mit zwei Spalten möglich, die in jeder Zeile die Zahl der Treffer und Nicht-Treffer für jeweils eine Gruppe enthält.

```
> 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 berichtet 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`). Dasselbe 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 `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.

10.2.4 Fishers exakter Test auf Unabhängigkeit

Werden zwei dichotome Variablen in einer Stichprobe erhoben, kann mit Fishers exaktem Test die H_0 geprüft werden, dass beide Variablen unabhängig sind.¹⁰ Anders als beim χ^2 -Test zur selben Hypothese sind hier auch gerichtete Alternativhypthesen über den Zusammenhang möglich.

```
fisher.test(x, y=NULL, 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 des *odds ratio* (OR) in der theoretischen Kontingenztafel. Linksseitiges Testen bedeutet, dass unter H_1 $OR < 1$ ist (negativer Zusammenhang), rechtsseitiges Testen entsprechend, $OR > 1$ (positiver Zusammenhang). Mit dem Argument `conf.level` wird die Breite des Konfidenzintervalls für das OR 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 <- xtabs(~ disease + diagT) # Kontingenztafel
> addmargins(contT1) # Randsummen
diagT
```

¹⁰Die H_0 ist äquivalent zur Hypothese, dass das wahre *odds ratio* der Kontingenztafel beider Variablen gleich 1 ist (Abschn. 10.2.6). Die Verallgemeinerung auf Variablen mit mehr als zwei Stufen erläutert `?fisher.test`. Der Test setzt voraus, dass die Randhäufigkeiten fest sind. Der Boschloo-Test für dieselbe Hypothese ist dagegen nicht bedingt auf die Randhäufigkeiten. Er ist im Paket `exact2x2` (Fay, 2020) implementiert.

| 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 das OR in der gewünschten Breite sowie die bedingte Maximum-Likelihood-Schätzung des OR für die gegebenen Randhäufigkeiten (`sample estimates`).¹¹

Der *p*-Wert lässt sich manuell nachprüfen: Als Variante eines Permutationstests (Abschn. 11.3) müssen hierfür die Punktwahrscheinlichkeiten für die vorliegende sowie für i.S. der H_1 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 <- dhyper(8, 8+2, 1+4, 8+1))
[1] 0.04495504

# Punktwahrscheinlichkeit für extremere Kontingenztafel
> (p2 <- dhyper(9, 9+1, 0+5, 9+0))
[1] 0.001998002

> (pVal <- p1 + p2)      # Summe der Punktwahrscheinlichkeiten -> p-Wert
[1] 0.04695305
```

Die H_0 , dass zwei dichotome Variablen in mehreren, durch eine dritte kategoriale Variable gebildeten Bedingungen jeweils unabhängig sind, prüft der Cochran-Mantel-Hänszel-Test, für den die Funktion `mantelhaen.test()` existiert. Sie ist auch im allgemeineren Fall anwendbar, dass die Unabhängigkeit kategorialer Variablen mit mehr als zwei Stufen zu testen ist.

¹¹In bestimmten Grenzfällen ist der berichtete *p*-Wert nicht konsistent damit, ob das Konfidenzintervall für das OR den Wert 1 enthält. Ein Beispiel dafür ist `fisher.test(matrix(c(6, 12, 12, 5), nrow=2L))`. Hintergründe und Lösungen erläutert das Paket `exact2x2`.

10.2.5 Fishers exakter Test auf Gleichheit von Verteilungen

Für Daten einer dichotomen Variable aus zwei unabhängigen Stichproben prüft Fishers exakter Test die H_0 , dass die Variablen in beiden Bedingungen identische Erfolgswahrscheinlichkeit besitzen. Anders als beim χ^2 -Test zur selben Hypothese sind hier auch gerichtete Alternativhypothesen möglich.

Der Aufruf von `fisher.test()` ist identisch zu jenem beim Test auf Unabhängigkeit (vgl. auch Fußnoten 10, 11). 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.

Im Beispiel sei an je einer Stichprobe aus der Population der Frauen und der Männer erhoben worden, ob die Person raucht. Geprüft wird die H_0 , dass der Anteil der Raucher in beiden Populationen gleich ist, wobei als H_1 hier vermutet wird, dass Frauen generell häufiger rauchen. Der Aufbau der Kontingenztafel verlangt nach einem linksseitigen Test.

```
> N           <- 20                                # Gruppengröße
> smokesFem  <- rbinom(N, size=1, p=0.6)        # Daten Frauen
> smokesMale <- rbinom(N, size=1, p=0.4)        # Daten Männer

# Faktoren, die Rauchverhalten und Geschlecht codieren
> smokes <- factor(c(smokesFem, smokesMale), labels=c("no", "yes"))
> sex     <- factor(rep(c("f", "m"), each=N))      # Faktor Geschlecht
> contT2 <- xtabs(~ sex + smokes)                # Kontingenztafel
> addmargins(contT2)                             # Randsummen
    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
```

10.2.6 Kennwerte von (2×2) -Konfusionsmatrizen

(2×2) -Kontingenztafeln als Ergebnis einer zweifachen dichotomen Klassifikation (*Konfusionsmatrizen*) erlauben die Berechnung mehrerer Kennwerte, um Eigenschaften der Klassifikation zu beschreiben. Hierzu zählen die Sensitivität, Spezifität und Relevanz sowie das odds ratio und das relative Risiko.¹²

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          #             Diagnose
disease  isHealthy  isIll  Sum    # krank?  "gesund"  "krank"
       no        8     2   10    # nein     TN       FP
       yes       1     4    5    # ja       FN       TP
       Sum       9     6   15

> TN <- contT1[1, 1]           # true negative
> TP <- contT1[2, 2]           # true positive / hit
> FP <- contT1[1, 2]           # false positive
> FN <- contT1[2, 1]           # false negative / miss
```

Sensitivität, Spezifität und Relevanz

Der Basisumfang von R stellt zur Ermittlung der folgenden Kennwerte keine eigene Funktion bereit, eine manuelle Berechnung ist jedoch unkompliziert. 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 $TP / (TP + FN)$, 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 die Wahrscheinlichkeit eines Fehlers zweiter Art (β) gleich $1 - \text{Sensitivität}$.

```
> (sensitivity <- recall <- TP / (TP+FN))
[1] 0.8
```

Die Spezifität ist $TN / (TN + FP)$, 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 der Wahrscheinlichkeit eines Fehlers erster Art (α), dem somit $\frac{FP}{TN+FP}$ entspräche.

¹²Das Paket **riskyR** (Neth, Gaisbauer, Gradwohl & Gaissmaier, 2019) stellt eine Vielzahl von Auswertungsfunktionen und Diagrammen zur Analyse sowie zum besseren Verständnis von Konfusionsmatrizen bereit.

```
> (specificity <- TN / (TN+FP))
[1] 0.8
```

Die Relevanz, je nach Kontext auch als Präzision oder positiver Vorhersagewert bezeichnet, ist $TP / (TP + FP)$. 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.

```
> (relevance <- precision <- ppv <- 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.

```
> (CCR <- sum(diag(contT1)) / sum(contT1))
[1] 0.8
```

Das F -Maß als harmonisches Mittel von Präzision und recall wird bisweilen als integriertes Gütemaß für eine Klassifikation herangezogen. Er ist nicht mit Werten einer F -Verteilung zu verwechseln.

```
> (Fval <- 1 / mean(1 / c(precision, recall)))
[1] 0.7272727
```

Odds ratio, Yules Q und relatives Risiko

Das odds ratio (OR, Chancenverhältnis) ist ein Zusammenhangsmaß für zwei dichotome Variablen und ergibt sich aus der (2×2) -Kontingenztafel ihrer gemeinsamen Häufigkeiten: $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$. Das OR berechnet sich durch Division der *Wettquotienten* (Chancen) $\frac{a}{b}$ und $\frac{c}{d}$, also als $\frac{a \cdot d}{b \cdot c}$. Repräsentieren die Zeilen der Kontingenztafel zwei Bedingungen und die Spalten die An- bzw. Abwesenheit eines Merkmals, drückt das OR die multiplikative Änderung der bedingten Verteilung des Merkmals beim Wechsel der Bedingungen aus.

Das OR lässt sich manuell, oder alternativ über `OddsRatio()` aus dem Paket `DescTools` ermitteln. Diese Funktion akzeptiert als Argument die Kontingenztafel absoluter Häufigkeiten und gibt ein über das Argument `conf.level` wählbares Konfidenzintervall für unterschiedliche Berechnungsmethoden aus. Das logarithmierte OR_{ln} ist die Differenz der logits $\ln \frac{\hat{P}_j}{1-\hat{P}_j}$, wenn \hat{P}_j die geschätzte Auftretenswahrscheinlichkeit des Merkmals in der Zeile j ist. Ist eine Zelle der empirischen Kontingenztafel gleich 0, wird vor der OR-Berechnung üblicherweise 0.5 zu allen Zellen addiert.

```
> library(DescTools)                                # für OddsRatio()
> OddsRatio(contT1, conf.level=0.95)               # odds ratio + Wald CI
odds ratio    lwr.ci      upr.ci
16.000000  1.092859  234.247896

> aa <- contT1[1, 1]                               # manuelle Kontrolle
```

```
> bb  <- contT1[1, 2]
> cc  <- contT1[2, 1]
> dd  <- contT1[2, 2]
> (OR <- (aa/bb) / (cc/dd))           # odds ratio
[1] 16
```

Auch Yules Q -Koeffizient bezieht sich auf die (2×2) -Kontingenztafel der gemeinsamen Häufigkeiten zweier dichotomer Variablen und lässt sich als Maß ihres Zusammenhangs interpretieren. Q ist mit `YuleQ()` aus dem Paket `DescTools` oder manuell als $\frac{ad-bc}{a+d+b+c}$ zu berechnen. Wurde das OR bereits ermittelt, gilt alternativ $Q = \frac{OR-1}{OR+1}$.

```
> library(DescTools)                  # für YuleQ()
> YuleQ(contT1)                     # Yules Q
[1] 0.8823529

> (OR-1) / (OR+1)                   # Kontrolle
[1] 0.882353
```

Das ebenfalls aus einer (2×2) -Kontingenztafel berechnete relative Risiko bezeichnet das Verhältnis der bedingten relativen Häufigkeiten $\frac{a}{a+b}$ und $\frac{c}{c+d}$. Im Fall der vorliegenden Daten wäre dies das Verhältnis der (auf die Zeilen) bedingten relativen Häufigkeiten, dass Gesunde bzw. Kranke als gesund diagnostiziert werden. Die Berechnung übernimmt `RelRisk()` aus dem Paket `DescTools`.

```
# "Risiko", dass Gesunde bzw. Kranke als gesund diagnostiziert werden
> (risk <- proportions(contT1, margin="disease"))
  diagT
disease  isHealthy  isIll
  no        0.8      0.2
  yes       0.2      0.8

> library(DescTools)                  # für RelRisk()
> RelRisk(contT1)                   # relatives Risiko
[1] 4

> risk[1, 1] / risk[2, 1]            # Kontrolle
[1] 4
```

10.2.7 ROC-Kurve und AUC

Das Paket `pROC` (Robin et al., 2011) ermöglicht die Berechnung von ROC-Kurven (*receiver operating characteristic*) bei einer dichotomen Klassifikation auf Basis einer logistischen Regression (Abschn. 8.1). ROC-Kurven stellen die Sensitivität gegen 1 – Spezifität der Klassifikation dar, wenn die Klassifikationsschwelle für die vorhergesagte Trefferwahrscheinlichkeit variiert.

```
roc(<glm-Fit>$y ~ <glm-Fit>$fitted.values,
  levels=<Klassen>,
  direction=c("auto", "<", ">"))
```

Im Funktionsaufruf ist `<glm-Fit>` das Ergebnis der Anpassung einer logistischen Regression. Ohne weitere Spezifikation ist es uneindeutig, auf welche Stufe der dichotomen Variable `y` links der `~` sich die vorhergesagte Trefferwahrscheinlichkeit rechts der `~` bezieht. In der Voreinstellung legt `roc()` die Reihenfolge selbst so fest, dass die Stufe mit der höheren medianen Trefferwahrscheinlichkeit als Zielkategorie betrachtet wird. Damit können sich keine Vorhersagen ergeben, die systematisch schlechter als der Zufall sind. Es empfiehlt sich sehr, die Daten mit den Argumenten `levels` und `direction` eindeutig zu beschreiben. Hier legt `levels` als Vektor die Reihenfolge der Stufen in `y` fest. `direction="<"` bedeutet, dass die zweite Stufe jene ist, auf die sich die vorhergesagte Trefferwahrscheinlichkeit bezieht.

Als integriertes Gütemaß der Klassifikation berechnet `roc()` die Fläche unter der ROC-Kurve (AUC). Weitere Optionen und Funktionen erlauben es, den Konfidenzbereich etwa für die Sensitivität aus Bootstrap-Replikationen (Abschn. 11.1) zu ermitteln und grafisch darzustellen (Abb. 10.2).

```

> N      <- 100                      # Stichprobengröße
> height <- rnorm(N, 175, 7)        # Körpergröße
> age     <- rnorm(N, 30, 8)
> weight <- 0.4*height + 0.3*age + rnorm(N, 0, 3) # Gewicht

# Median-Split Gewicht
> wFac <- cut(weight, breaks=c(-Inf, median(weight), Inf),
+               labels=c("lo", "hi"))

> regDf <- data.frame(wFac, height, age)      # Datensatz

# logistische Regression
> fit_glm <- glm(wFac ~ height + age, family=binomial(link="logit"),
+                  data=regDf)

> library(pROC)                         # für roc(), ci.se()
> (rocRes <- roc(fit_glm$y ~ fit_glm$fitted.values,
+                 levels=c(0, 1), direction="<",
+                 plot=TRUE, ci=TRUE, main="ROC-Kurve",
+                 xlab="1-Spezifität (TN / (TN+FP))",
+                 ylab="Sensitivität (TP / (TP+FN)))")
# ... (gekürzte Ausgabe)
Data: fit_glm$fitted.values in 50 controls (fit_glm$y 0)
    < 50 cases (fit_glm$y 1).
Area under the curve: 0.9132
95% CI: 0.8586-0.9678 (DeLong)

# stelle Konfidenzbereich für die Sensitivität dar
> rocCI <- ci.se(rocRes)                # CIs für Sensitivität
> plot(rocCI, type="shape")

```

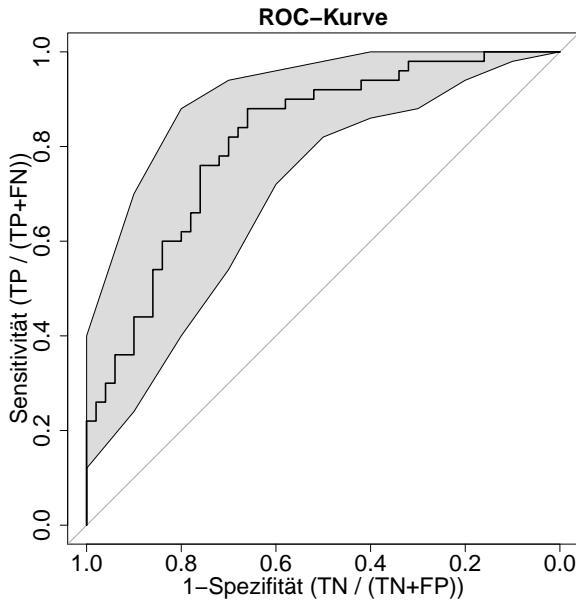


Abbildung 10.2: ROC-Kurve für eine dichotome Klassifikation samt Konfidenzbereich für die Sensitivität

10.3 Maße für Zusammenhang und Übereinstimmung

10.3.1 Zusammenhang stetiger ordinaler Variablen: Spearmans ρ und Kendalls τ

Zur Berechnung der Kovarianz und Korrelation zweier stetiger ordinaler Variablen nach Spearman bzw. nach Kendall dienen die Funktionen `cov()` und `cor()`, für die dann das Argument `method` auf "spearman" bzw. auf "kendall" zu setzen ist. Spearmans ρ ist gleich der gewöhnlichen Pearson-Korrelation beider Variablen, nachdem ihre Werte durch die zugehörigen Ränge ersetzt wurden.

```
> DV1 <- c(100, 76, 56, 99, 50, 62, 36, 69, 55, 17)
> DV2 <- c(42, 74, 22, 99, 73, 44, 10, 68, 19, -34)
> cor.test(~ DV1 + DV2, method="spearman")      # Spearmans rho
[1] 0.6727273

> cor(rank(DV1), rank(DV2))                      # Korrelation der Ränge
[1] 0.6727273
```

Für Kendalls τ_a ist die Differenz der Anzahl konkordanter Paare n_{CP} und diskonkordanter Paare n_{DP} von je zwei abhängigen Messwerten zu bilden und an der Gesamtzahl möglicher Paare $\binom{n}{2} = \frac{n(n-1)}{2}$ zu relativieren. Sind X und Y die Variablen aus den abhängigen Stichproben vom Umfang n , 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 .

Kendalls τ_b unterscheidet sich von τ_a , wenn jeweils innerhalb von X und Y identische Werte (Bindungen) vorliegen können, wobei n_{TX} die Anzahl der Bindungen in X und n_{TY} die Anzahl der Bindungen in Y angibt. Dann relativiert τ_b die Differenz $n_{CP} - n_{DP}$ am geometrischen

Mittel $\sqrt{(n_{CP} + n_{DP} + n_{TX}) \cdot (n_{CP} + n_{DP} + n_{TY})}$. `cor(..., method="kendall")` berechnet bei Bindungen τ_b .

```
> cor.test(~ DV1 + DV2, method="kendall")      # Kendalls tau-b
[1] 0.6

# Matrizen der paarweisen Rangvergleiche jeweils in X und Y
> cmpMat1 <- outer(DV1, DV1, ">")           # Vergleiche in X
> cmpMat2 <- outer(DV2, DV2, ">")           # 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]))
[1] 36

# Anzahl diskonkordanter Paare -> Rangordnung in X und Y verschieden
> nDP    <- sum((cmpMat1 != cmpMat2)[selMat])        # Anzahl Personen
> N       <- length(DV1)                            # Anzahl möglicher Paare
> nPairs <- choose(N, 2)                           # Kendalls tau-a
> (tau   <- (nCP-nDP) / nPairs)
[1] 0.6
```

Spearmans und Kendalls empirische Maße des Zusammenhangs lassen sich mit `cor.test()` inferenzstatistisch daraufhin prüfen, ob sie mit der H_0 verträglich sind, dass der theoretische 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. Alternativ kann auch eine Modellformel `~ <Vektor1> + <Vektor2>` ohne Variable links der `~` angegeben werden. Stammen die Variablen in der Modellformel aus einem Datensatz, ist dieser unter `data` zu nennen. Ob die H_1 zwei- (`"two.sided"`), links- (negativer Zusammenhang, `"less"`) oder rechtsseitig (positiver Zusammenhang, `"greater"`) ist, legt das Argument `alternative` fest. Für den Test ordinaler Daten nach Spearman und Kendall ist das Argument `method` auf `"spearman"` bzw. `"kendall"` zu setzen. Über das Argument `use` können verschiedene Strategien zur Behandlung fehlender Werte ausgewählt werden (Abschn. 2.13.4).

```
> cor.test(DV1, DV2, method="spearman")
Spearman's rank correlation rho
data: DV1 and DV2
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** ergibt sich als Summe der quadrierten Differenzen zwischen den Rängen zugehöriger Messwerte in beiden Stichproben.¹³

```
> sum((rank(DV1) - rank(DV2))^2) # Teststatistik S
[1] 54

> cor.test(DV1, DV2, method="kendall") # Test nach Kendall
Kendall's rank correlation tau
data: DV1 and DV2
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 berichtet den Wert der Teststatistik (**T**) gemeinsam mit dem zugehörigen *p*-Wert (**p-value**) sowie Kendalls τ_b (**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 (Fußnote 13).

Goodman und Kruskals γ basiert ebenfalls auf der Anzahl konkordanter und diskonkordanter Paare. γ stimmt mit Kendalls τ_a und τ_b überein, wenn keine Rangbindungen vorliegen. γ lässt sich über **GoodmanKruskalGamma()** aus dem Paket **DescTools** berechnen. Ebenfalls aus diesem Paket stammt die Funktion **SomersDelta()**, die Somers' d ausgibt – ein weiteres Zusammenhangsmaß kategorialer Variablen, das die Anzahl konkordanter und diskonkordanter Paare berücksichtigt.

10.3.2 Zusammenhang kategorialer Variablen: ϕ , Cramérs V , Kontingenzkoeffizient

Als Maße des Zusammenhangs von zwei ungeordneten kategorialen Variablen mit P bzw. mit Q Kategorien dienen

- der ϕ -Koeffizient (bei dichotomen Variablen gleich deren Korrelation, auch als *Matthews-Korrelationskoeffizient* bezeichnet),
- dessen Verallgemeinerung Cramérs V (für dichotome Variablen identisch mit $|\phi|$),
- der Pearson-Kontingenzkoeffizient CC
- und Tschuprows T (identisch zu Cramérs V für quadratische Kontingenztafeln).

Diese Kennwerte basieren auf dem χ^2 -Wert der Kontingenztafel der gemeinsamen Häufigkeiten beider Variablen. Beruht die Kontingenztafel auf n Beobachtungen, und ist $L = \min(P, Q)$ der kleinere Wert der Anzahl ihrer Zeilen und Spalten, gelten folgende Zusammenhänge: $n(L - 1)$

¹³Die Berechnung des zugehörigen *p*-Wertes ist nur über eine intern definierte Funktion möglich, die Verteilungsfunktion der Teststatistik ist nicht direkt als R-Funktion vorhanden.

ist der größtmögliche χ^2 -Wert. Weiterhin gilt $\phi = \sqrt{\frac{\chi^2}{n}}$, $V = \sqrt{\frac{\chi^2}{n(L-1)}}$, $CC = \sqrt{\frac{\chi^2}{n+\chi^2}}$ und $T = \sqrt{\frac{\phi^2}{\sqrt{(P-1)\cdot(Q-1)}}}$. Die Kennwerte (und einige weitere) lassen sich mit `Assocs()` aus dem Paket `DescTools` ermitteln, Tschuprows T mit `TschuprowT()` aus demselben Paket.

```
> P      <- 2
> Q      <- 3
> DV1   <- cut(c(100, 76, 56, 99, 50, 62, 36, 69, 55, 17), breaks=P)
> DV2   <- cut(c(42, 74, 22, 99, 73, 44, 10, 68, 19, -34), breaks=Q)
> cTab  <- xtabs(~ DV1 + DV2)                      # Kontingenztafel
> N     <- sum(cTab)                                # Anzahl Beobachtungen
> library(DescTools)                               # für Assocs()
> Assocs(cTab)

          estimate lwr.ci upr.ci
Phi Coeff.        0.5477    -     -
Contingency Coeff. 0.4804    -     -
Cramer V          0.5477  0.0000 1.0000
Goodman Kruskal Gamma 0.7778  0.2679 1.0000
Kendall Tau-b      0.4950  0.0499 0.9401
Stuart Tau-c        0.5600  0.0423 1.0000
Somers D C|R       0.5600  0.0423 1.0000
Somers D R|C       0.4375  0.0296 0.8454
Pearson Correlation 0.5345 -0.1433 0.8710
Spearman Correlation 0.5217 -0.1607 0.8667
Lambda C|R         0.1667  0.0000 1.0000
Lambda R|C         0.4000  0.0000 1.0000
Lambda sym          0.2727  0.0000 0.9457
Uncertainty Coeff. C|R 0.1754 -0.0578 0.4087
Uncertainty Coeff. R|C 0.2671 -0.1107 0.6450
Uncertainty Coeff. sym 0.2118 -0.0766 0.5001
Mutual Information  0.2755    -     -

> TschuprowT(cTab)                                # Tschuprows T
[1] 0.4605779

# erwartete Zellhäufigkeiten
> expected <- outer(rowSums(cTab), colSums(cTab)) / N
> chisqVal <- sum((cTab-expected)^2 / expected)    # chi^2-Wert
> (phiVal  <- sqrt(chisqVal / N))                 # phi-Koeffizient
[1] 0.5477226

> L      <- min(dim(cTab))                         # Minimum Zeilen, Spalten
> phisqMax <- L-1                                  # maximaler phi^2-Wert
> chisqMax <- N*phisqMax                           # maximaler chi^2-Wert
> (CrV    <- sqrt(chisqVal / chisqMax)) # Cramérs V
[1] 0.5477226
```

```
> (CC <- sqrt(chisqVal / (N+chisqVal))) # Kontingenzkoeffizient
[1] 0.4803845

> (TschT <- sqrt((chisqVal/N) / sqrt((P-1)*(Q-1)))) # Tschuprows T
[1] 0.4605779
```

Für ordinale kategoriale Variablen wird etwa von [Agresti \(2018\)](#) der *linear-by-linear* Test auf Zusammenhang vorgeschlagen. Das Paket `coin` stellt für seine Umsetzung `tbl_test()` bereit.

10.3.3 Inter-Rater-Übereinstimmung

Wenn mehrere Personen (*Rater*) dieselben Objekte in Kategorien einordnen oder auf einer Skala bewerten, ist häufig die Inter-Rater-Übereinstimmung bzw. -Reliabilität als Grad der Ähnlichkeit der Urteile relevant ([Wirtz & Caspar, 2002](#)). Hierbei lassen sich zum einen Fälle unterscheiden, bei denen entweder nur zwei oder auch mehr Rater vorhanden sind. Zum anderen ist zu differenzieren, ob kontinuierliche Bewertungen oder ungeordnete bzw. geordnete Kategorien zum Einsatz kommen. Das im Folgenden verwendete Paket `DescTools` bringt neben den beschriebenen noch weitere Funktionen zur Analyse der Inter-Rater-Übereinstimmung mit, etwa für Krippendorffs α . Für den Stuart-Maxwell-Test, ob zwei Rater die Kategorien einer mehrstufigen Variable mit denselben Grundwahrscheinlichkeiten verwenden, s. Abschn. [10.5.11](#).

Prozentuale Übereinstimmung

Die relative Übereinstimmung der Urteile zweier oder mehr Personen, die dieselben Beobachtungsobjekte in mehrere Kategorien einteilen, kann mit der Funktion `Agree()` aus dem Paket `DescTools` berechnet werden. Sie erwartet als Argument eine spaltenweise aus den Urteilen der Rater zusammengestellte Matrix. Als Beispiel diene jenes aus [Bortz et al. \(2010, p. 458 ff.\)](#), in dem zwei Rater 100 Beobachtungsobjekte in drei Kategorien einteilen.

```
> categ <- c("V", "N", "P") # Rating-Kategorien
> lvls <- factor(categ, levels=categ) # Kategorien als Faktor
> rtr1 <- rep(lvls, c(60, 30, 10)) # Urteile Rater 1, 2
> rtr2 <- rep(rep(lvls, nlevels(lvls)), c(53,5,2, 11,14,5, 1,6,3))
> cTab <- xtabs(~ rtr1 + rtr2) # Kontingenztafel
> addmargins(cTab) # Randsummen
rtr2
rtr1   V    N    P  Sum
      V 53    5    2   60
      N 11   14    5   30
      P  1    6    3   10
Sum  65   25   10  100

# relative Übereinstimmung
> library(DescTools) # für Agree()
```

```
> Agree(cbind(rtr1, rtr2))                                # Ausgabe gekürzt ...
[1] 0.7
```

Zur manuellen Kontrolle dient die mit `xtabs()` ermittelte Kontingenztafel der gemeinsamen Häufigkeiten der Urteile. Übereinstimmend vergebene Kategorien stehen in der Diagonale, sofern Zeilen und Spalten dieselben Kategorien in derselben Reihenfolge umfassen.

```
> (agree <- sum(diag(proportions(cTab)))) 
[1] 0.7
```

Im Anschluss soll das Beispiel um die Urteile eines dritten Raters erweitert werden.

```
> rtr3 <- rep(rep(lvls, nlevels(lvls)), c(48,8,3, 15,10,7, 3,4,2))
> Agree(cbind(rtr1, rtr2, rtr3))                          # Ausgabe gekürzt ...
[1] 0.6
```

Die manuelle Kontrolle benötigt zunächst die 3D Kontingenztafel der Urteile.

```
> cTab3 <- xtabs(~ rtr1 + rtr2 + rtr3)
> sum(diag(apply(proportions(cTab3), 3, diag)))
[1] 0.6
```

Cohens κ

Der Grad, mit dem die Urteile zweier Rater übereinstimmen, die dieselben Beobachtungsobjekte in mehrere ungeordnete Kategorien einteilen, kann 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 Funktion `CohenKappa()` aus dem Paket `DescTools` berechnen, die als Argument die Kontingenztafel der Urteile beider Rater erwartet. Mit `conf.level` lässt sich zusätzlich die Breite des Konfidenzintervalls für κ festlegen.

```
> cTab <- xtabs(~ rtr1 + rtr2)                            # Kontingenztafel
> library(DescTools)                                       # für CohenKappa()
> CohenKappa(cTab, conf.level=0.95)
  kappa      lwr.ci      upr.ci
0.4285714  0.2574917  0.5996511

# beobachteter Anteil an Übereinstimmungen (Diagonalelemente)
> fObs <- sum(diag(proportions(cTab)))

# zufällig erwartbarer Anteil an Übereinstimmungen
> fExp <- sum(rowSums(proportions(cTab)) * colSums(proportions(cTab)))
> (Ckappa <- (fObs-fExp) / (1-fExp))                      # Cohens kappa
[1] 0.4285714
```

Fleiss' κ

`KappaM()` aus dem Paket `DescTools` ermittelt den κ -Koeffizienten nach Fleiss als Maß der Übereinstimmung von mehr als zwei Ratern, die dieselben Objekte in mehrere ungeordnete Kategorien einteilen. Dafür ist die spaltenweise aus den Urteilen der Rater zusammengestellte Matrix als Argument zu übergeben.

Als Beispiel diene jenes aus [Bortz et al. \(2010\)](#), p. 460 ff.), in dem sechs Rater dieselben 30 Begriffe einer von fünf Kategorien (a–e) zuordnen (für das Erstellen eigener Funktionen s. Abschn. 17.3).

```
> rtr1 <- 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)]
> rtr2 <- 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)]
> rtr3 <- 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)]
> rtr4 <- 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)]
> rtr5 <- 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)]
> rtr6 <- 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(rtr1, rtr2, rtr3, rtr4, rtr5, rtr6)
> library(DescTools)                                # für KappaM
> KappaM(rateMat, conf.level=0.95)
  kappa      lwr.ci      upr.ci
0.4302445  0.3824725  0.4780165

# manuelle Berechnung
> nRtr <- ncol(rateMat)                          # Anzahl Rater
> nObs <- nrow(rateMat)                          # Anzahl Objekte

# Vektor aller Urteile
> ratings <- c(rtr1, rtr2, rtr3, rtr4, rtr5, rtr6)

# zugehöriger Faktor, welches Objekt beurteilt wurde
> obsFac <- factor(rep(1:nObs, nRtr))

# Kontingenztafel der Häufigkeiten, wie oft jedes Objekt (Zeilen)
# einer Kategorie (Spalten) zugeordnet wurde
> cTab <- xtabs(~ obsFac + ratings)

# relative Häufigkeit jeder Beurteilungskategorie
> rateTab <- proportions(xtabs(~ ratings))
```

```
# per Zufall erwartbarer Anteil an Übereinstimmungen
> fExp <- sum(rateTab^2)

# beobachtete paarweise Übereinstimmung pro Objekt
> fObsAll <- apply(cTab, 1, function(x) {
+           sum(x*(x-1)) / (nRtr*(nRtr-1)) } )

# mittlere beobachtete paarweise Übereinstimmung
> fObs     <- mean(fObsAll)
> (Fkappa <- (fObs-fExp) / (1-fExp))          # Fleiss' kappa
[1] 0.4302445
```

Gewichtetes Cohens κ

Liegen geordnete Kategorien vor, kann nicht nur berücksichtigt werden, ob Übereinstimmung vorliegt oder nicht, sondern auch, wie stark ggf. die Diskrepanz der Urteile ist. Für die Situation mit zwei Ratern stehen in `CohenKappa()` aus dem Paket `DescTools` mehrere Gewichtungen zur Verfügung: Wird das Argument `weights="Equal-Spacing"` 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. \(2010, p. 484 ff.\)](#), in dem zwei Rater die jeweils 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 <- xtabs(~ tv1 + tv2)                  # Kontingenztafel 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(DescTools)                         # für CohenKappa()
> CohenKappa(cTab, weights="Equal-Spacing", conf.level=0.95)
```

```
kappa      lwr.ci      upr.ci
0.31566846  0.07905284  0.55228407
```

Für die manuelle Berechnung muss für jede Zelle der Kontingenztafel beider Rater ein Gewicht angegeben werden, das der Ähnlichkeit der von den Ratern verwendeten Kategorien entspricht. Die Gewichte bilden eine Toeplitz-Matrix mit konstanten Diagonalen, wenn sie in die Kontingenztafel eingetragen werden. Übereinstimmende Kategorien (Haupt-Diagonale) haben dabei eine Ähnlichkeit von 1, die minimale Ähnlichkeit ist 0. Bei gleichabständigen Kategorien sind die Gewichte dazwischen linear zu wählen (s. Abschn. 17.3 für selbst erstellte Funktionen).

```
> P <- ncol(cTab)                                # Anzahl Kategorien

# bei Unabhängigkeit erwartete Häufigkeiten
> expected <- outer(rowSums(cTab), colSums(cTab)) / sum(cTab)

# gleichabständige Gewichtungsfaktoren für Ähnlichkeit der Kategorien
> (myWeights <- seq(0, 1, length.out=P))
[1] 0.0 0.2 0.4 0.6 0.8 1.0

# Toeplitz-Matrix der Ähnlichkeits-Gewichte
> (weightsMat <- outer(1:P, 1:P, function(x, y) {
+                               1 - ((abs(x-y)) / (P-1)) } ))
     [,1]  [,2]  [,3]  [,4]  [,5]  [,6]
[1,] 1.0  0.8  0.6  0.4  0.2  0.0
[2,] 0.8  1.0  0.8  0.6  0.4  0.2
[3,] 0.6  0.8  1.0  0.8  0.6  0.4
[4,] 0.4  0.6  0.8  1.0  0.8  0.6
[5,] 0.2  0.4  0.6  0.8  1.0  0.8
[6,] 0.0  0.2  0.4  0.6  0.8  1.0

# Summe der gewichteten beobachteten relativen Häufigkeiten
> wfObs <- sum(cTab * weightsMat) / sum(cTab)

# Summe der gewichteten erwarteten relativen Häufigkeiten
> wfExp <- sum(expected * weightsMat) / sum(cTab)

# gewichtetes Cohens kappa
> (wKappa <- (wfObs-wfExp) / (1-wfExp))
[1] 0.3156685
```

Kendalls W

Für stetige ordinale Urteile von mehr als zwei Ratern kann Kendalls W von der Funktion `KendallW()` aus dem Paket `DescTools` ermittelt werden, die als Argument eine spaltenweise

aus den Urteilen der Rater zusammengesetzte Matrix erwartet.¹⁴ Als Beispiel diene jenes aus Bortz et al. (2010, p. 466 ff.), in dem drei Rater sechs Beobachtungsobjekte entsprechend der Ausprägung eines Merkmals in eine Rangreihe bringen.

```
> rtr1    <- c(1, 6, 3, 2, 5, 4)                      # Urteile Rater 1
> rtr2    <- c(1, 5, 6, 2, 4, 3)                      # Urteile Rater 2
> rtr3    <- c(2, 3, 6, 5, 4, 1)                      # Urteile Rater 3
> ratings <- cbind(rtr1, rtr2, rtr3)                  # Matrix der Urteile
> library(DescTools)                                    # für KendallW()
> KendallW(ratings, test=TRUE)
Kendall's coefficient of concordance W
data: ratings
Kendall chi-squared=8.5238, df=5, subjects=6, raters=3, p-value=0.1296
alternative hypothesis: W is greater 0
sample estimates:
W
0.568254
```

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(Ratings, 2, rank)` zu bilden.

```
> rankMat  <- ratings                                # Matrix der Ränge
> nObj     <- nrow(rankMat)                          # Anzahl Objekte
> nRtr     <- ncol(rankMat)                          # Anzahl Rater
> (rankSum <- rowSums(rankMat))                     # Rangsumme pro Objekt
[1] 4 14 15 9 13 8

# Summe der quadrierten Abweichungen der Rangsummen vom Erwartungswert
> (S <- sum((rankSum - (nRtr*(nObj+1) / 2))^2))
[1] 89.5

> (W <- (12 / (nRtr^2 * nObj*(nObj^2 - 1))) * S)   # Kendalls W
[1] 0.568254

> (chisqVal <- nRtr * (nObj-1) * W)                 # Teststatistik
[1] 8.52381

> (pVal <- pchisq(chisqVal, nObj-1, lower.tail=FALSE)) # p-Wert
[1] 0.1296329
```

¹⁴Der zugehörige Signifikanztest ist äquivalent zum Friedman-Test (Abschn. 10.5.7), wobei den Ratern die Beobachtungsobjekte bzw. Blöcke entsprechen und den Objekten die unterschiedlichen Bedingungen. Mit m Ratern ist $\frac{m \cdot W - 1}{m - 1}$ zudem gleich der durchschnittlichen Spearman-Rangkorrelation alle möglichen Paare von Ratern.

Intra-Klassen-Korrelation

Die Intra-Klassen-Korrelation (ICC) wird von `ICC()` aus dem Paket `DescTools` ermittelt. Als Argument ist die Matrix der Urteile zu übergeben, wobei jede Spalte die Urteile eines Raters umfasst. Die sich auf ein Objekt beziehenden Urteile befinden sich in jeweils einer Zeile.

Die ICC findet in verschiedenen Bereichen der Statistik Anwendung, u. a. eignet sie sich zur Quantifizierung der Übereinstimmung von Ratern, die intervallskalierte Urteile über dieselben Objekte abgeben.¹⁵ Diese Situation lässt sich auch als Fall eines varianzanalytischen SPF- $p \cdot q$ -Designs mit einem Zwischen-Gruppen-Faktor (den Ratern) und einem Intra-Gruppen-Faktor (den Objekten) betrachten (Abschn. 7.8). Unterschiedliche ICC-Varianten resultieren abhängig davon, ob zum einen Rater bzw. Objekte als feste Faktoren oder als Random-Faktoren zu betrachten sind, und ob zum anderen Einzeldaten oder bereits aggregierte Werte vorliegen.

```
> nRtr <- 4                                # Anzahl Rater
> nObs <- 6                                 # Beobachtungen pro Rater
> rtr1 <- c(9, 6, 8, 7, 10, 6)              # Ratings Rater 1
> rtr2 <- c(2, 1, 4, 1, 5, 2)                # " 2
> rtr3 <- c(5, 3, 6, 2, 6, 4)                # " 3
> rtr4 <- c(8, 2, 8, 6, 9, 7)                # " 4
> library(DescTools)                         # für ICC()
> ICC(cbind(rtr1, rtr2, rtr3, rtr4))
Call: ICC(x = cbind(rtr1, rtr2, rtr3, rtr4))
```

Intraclass correlation coefficients

| | type | ICC | F | df1 | df2 | p | lowerBound | upperBound |
|----------------------|-------|------|------|-----|-----|------|------------|------------|
| Single_raters_abs | ICC1 | 0.17 | 1.8 | 5 | 18 | 0.16 | -0.13 | 0.72 |
| Single_random_raters | ICC2 | 0.29 | 11.0 | 5 | 15 | 0.00 | 0.02 | 0.76 |
| Single_fixed_raters | ICC3 | 0.71 | 11.0 | 5 | 15 | 0.00 | 0.34 | 0.95 |
| Avg_raters_absolute | ICC1k | 0.44 | 1.8 | 5 | 18 | 0.16 | -0.88 | 0.91 |
| Avg_random_raters | ICC2k | 0.62 | 11.0 | 5 | 15 | 0.00 | 0.07 | 0.93 |
| Avg_fixed_raters | ICC3k | 0.91 | 11.0 | 5 | 15 | 0.00 | 0.68 | 0.99 |

Die Ausgabe nennt den Wert von sechs ICC-Varianten samt der p -Werte ihres jeweiligen Signifikanztests und der Grenzen des zugehörigen zweiseitigen Vertrauensintervalls. Die Varianten ICC1–ICC3 gehen davon aus, dass jeder Wert ein Einzelurteil darstellt, während die Varianten ICC1k–ICC3k für den Fall gedacht sind, dass jeder Wert seinerseits bereits ein Mittelwert mehrerer Urteile desselben Raters über dasselbe Objekt ist. Welcher ICC-Wert der zu einer konkreten Untersuchung passende ist, hat der Nutzer selbst zu entscheiden, wobei eine nähere Erläuterung über `?ICC` zu finden ist.

Für die manuelle Berechnung sind Effekt- und Residual-Quadratsummen aus zwei Varianzanalysen notwendig: zum einen aus der einfaktoriellen ANOVA (Abschn. 7.4) mit nur den Objekten als Gruppierungsfaktor, zum anderen aus der zweifaktoriellen ANOVA (Abschn. 7.6) mit Objekten und Ratern als Faktoren.

¹⁵Die ICC ist kein nonparametrisches Verfahren, soll aber als Maß der Inter-Rater-Reliabilität dennoch hier aufgeführt werden.

```

> ratings <- c(rtr1, rtr2, rtr3, rtr4)                      # alle Ratings

# zugehöriger Faktor, von welchem Rater ein Rating stammt
> rateFac <- factor(rep(paste("rater", 1:nRtr, sep=""), each=nObs))

# zugehöriger Faktor, zu welchem Objekt ein Rating gehört
> obsFac <- factor(rep(paste("obj", 1:nObs, sep=""), times=nRtr))

# Varianzanalyse nur mit Faktor Beobachtungsobjekt
> (anObs <- anova(lm(ratings ~ obsFac)))                  # ...

# zweifaktorielle Varianzanalyse: Beobachtungsobjekt und Rater
> (anBoth <- anova(lm(ratings ~ obsFac + rateFac)))      # ...

# relevante mittlere Quadratsummen extrahieren
> MSobs   <- anObs["obsFac",    "Mean Sq"] # Effekt Objekt 1fakt. ANOVA
> MSEobs  <- anObs["Residuals", "Mean Sq"] # Residuen 1fakt. ANOVA
> MSrtr   <- anBoth["rateFac",   "Mean Sq"] # Effekt Rater 2fakt. ANOVA
> MSEboth <- anBoth["Residuals", "Mean Sq"] # Residuen 2fakt. ANOVA

> (ICC1 <- (MSobs-MSEobs) / (MSobs + (nRtr-1)*MSEobs))
[1] 0.1657418

> (ICC2 <- (MSobs-MSEboth) / (MSobs + (nRtr-1)*MSEboth
+           + ((nRtr/nObs) * (MSrtr - MSEboth))))
[1] 0.2897638

> (ICC3 <- (MSobs-MSEboth) / (MSobs + (nRtr-1)*MSEboth))
[1] 0.7148407

> (ICC1k <- (MSobs-MSEobs) / (MSobs))
[1] 0.4427971

> (ICC2k <- (MSobs-MSEboth) / (MSobs + ((1/nObs) * (MSrtr-MSEboth))))
[1] 0.6200505

> (ICC3k <- (MSobs-MSEboth) / MSobs)
[1] 0.9093155

```

Bland-Altman Diagramm

Ein Bland-Altman Diagramm (auch: Tukey *mean difference* Diagramm) erlaubt es, die Übereinstimmung von zwei Ratern oder Messmethoden bzgl. einer and denselben Objekten i erhöhenen quantitativen Variable zu beurteilen. Dafür trägt ein Streudiagramm (Abschn. 14.6.8) die paarweise gebildeten Differenzen d_i der Werte auf der y -Achse gegen ihren jeweiligen Mittelwert m_i auf der x -Achse auf (Abb. 10.3). Eine rudimentäre Version kann mit `tmd()` aus

dem im Basisumfang von R enthaltenen Paket **lattice** (Sarkar, 2008) erstellt werden – eine reichhaltigere Variante ist manuell möglich.

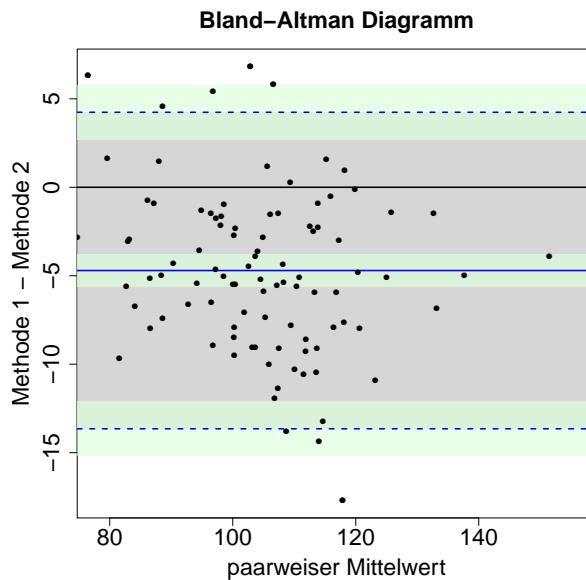


Abbildung 10.3: Bland-Altman Diagramm der paarweisen Differenzen gegen die paarweisen Mittelwerte mit *bias* und *limits of agreement* samt Konfidenzintervallen

```
> N      <- 100                      # Anzahl Messungen
> method1 <- rnorm(N, 100, 15)       # Messwerte Methode 1
> method2 <- method1 + rnorm(N, 5, 5) # Messwerte Methode 2 (bias +5)
> m_i    <- (method1 + method2) / 2   # paarweise Mittelwerte
> d_i    <- method1 - method2        # paarweise Differenzen

# Streudiagramm paarweise Differenzen gegen paarweise Mittelwerte
> xLoUp <- 1.05*range(m_i)          # x-Achse Grenzen
> plot(m_i, d_i, pch=16, main="Bland-Altman Diagramm",
+       xlab="paarweiser Mittelwert", ylab="Methode 1 - Methode 2",
+       xlim=xLoUp, xaxis="i")
```

Eine horizontale Linie auf Höhe des Mittelwerts \bar{d} der paarweisen Differenzen dient als Maß für die systematische Verzerrung (*bias*). Der bias $\bar{d} \pm z_{0.975} \cdot s_d$ markiert den Bereich der *limits of agreement*, wobei $z_{0.975}$ das 97,5%-Quantil der Standardnormalverteilung (1,96) und s_d die Streuung der Differenzen ist. In diesem Bereich werden bei normalverteilten Differenzen 95% der d_i erwartet. Die limits of agreement sollten deshalb innerhalb des aus inhaltlicher Sicht noch akzeptablen Abweichungsbereichs liegen.

```
> m_di   <- mean(d_i)                # Mittelwert der Differenzen
> var_di <- var(d_i)                 # Varianz der Differenzen
> alpha   <- 0.05                   # Signifikanzniveau
> LoA_z  <- qnorm(1-(alpha/2), 0, 1) # 97,5% z-Wert Standard-NV
> LoAlo  <- m_di - LoA_z*sqrt(var_di) # LoA unten
> LoAup  <- m_di + LoA_z*sqrt(var_di) # LoA oben
```

```
# limits of agreement-Bereich, bias und 0-bias Referenz darstellen
> rect(xLo, LoAlo, xUp, LoAup, col="#cccccccc", border=NA) # LoA Bereich
> abline(h=c(LoAlo, LoAup), col="darkgreen", lwd=2, lty=2) # LoA
> abline(h=m_di, col="blue", lwd=2)      # geschätzter bias
> abline(h=0, col="darkgray", lwd=2)      # Referenzlinie 0 = kein bias
```

Als Maß für die Präzision des berechneten bias \bar{d} sowie der limits of agreement lassen sich zudem t -Konfidenzintervalle unter Annahme von Normalverteiltheit der Differenzen berechnen.

```
# t-Konfidenzintervall für Erwartungswert Differenzvariable
> tCrit     <- qt(alpha/2, df=N-1, lower.tail=FALSE) # kritischer t-Wert
> mdi_CIlo <- m_di - tCrit*sqrt(var_di/N)
> mdi_CIup <- m_di + tCrit*sqrt(var_di/N)

# t-Konfidenzintervall für untere LoA-Grenze, zunächst LoA Varianz
> var_LoA    <- (var(d_i) / N) + (LoA_z^2 * var_di / (2*(N-1)))
> LoAlo_CIlo <- LoAlo - tCrit*sqrt(var_LoA)
> LoAlo_CIup <- LoAlo + tCrit*sqrt(var_LoA)

# t-Konfidenzintervall für obere LoA-Grenze
> LoAup_CIlo <- LoAup - tCrit*sqrt(var_LoA)
> LoAup_CIup <- LoAup + tCrit*sqrt(var_LoA)

# Konfidenzbereiche darstellen
> rect(xLo, LoAlo_CIlo, xUp, LoAlo_CIup, col="#ddffddaa", border=NA)
> rect(xLo, mdi_CIlo,   xUp, mdi_CIup,   col="#ddffddaa", border=NA)
> rect(xLo, LoAup_CIlo, xUp, LoAup_CIup, col="#ddffddaa", border=NA)
> points(m_i, d_i, pch=16) # wiederhole Datenpunkte für Sichtbarkeit
```

10.4 Tests auf gleiche Variabilität

Die in Abschn. 10.5.4 und 10.5.6 vorgestellten Tests prüfen, ob eine Variable in zwei oder mehr unabhängigen Bedingungen denselben Median besitzt. Dafür setzen sie u. a. voraus, dass die Variable in allen Bedingungen dieselbe Variabilität hat, die Verteilungen also gleich breit sind. Wird untersucht, ob die erhobenen Werte mit dieser Annahme konsistent sind, ist zu berücksichtigen, dass i. d. R. die H_0 den gewünschten Zustand darstellt.¹⁶ Angemessene Herangehensweisen beschreibt [Wellek \(2010, Kap. 9\)](#).

10.4.1 Mood-Test

Mit dem Mood-Test lässt sich prüfen, ob die empirische Variabilität einer stetigen ordinalen Variable in zwei unabhängigen Stichproben mit der H_0 verträglich ist, dass die Variable in

¹⁶Für die zugehörigen parametrischen Tests und den Fall mit mehr als zwei Gruppen s. Abschn. 7.2.

beiden Bedingungen dieselbe theoretische Variabilität besitzt.¹⁷

```
mood.test(x=<Vektor>, y=<Vektor>,
           alternative=c("two.sided", "less", "greater"))
```

Unter x und y sind die Daten aus beiden Stichproben einzutragen. Alternativ zu x und y kann auch eine Modellformel $\langle AV \rangle \sim \langle UV \rangle$ eingegeben werden. Stammen die Variablen in der Modellformel aus einem Datensatz, ist dieser unter data zu nennen. Das Argument alternative bezieht sich auf die Variabilität von x im Vergleich zu jener von y.

```
> DV1 <- c(12, 13, 29, 30)                      # Daten Gruppe 1
> DV2 <- c(15, 17, 18, 24, 25, 26)            # Daten Gruppe 2
> Nj  <- c(length(DV1), length(DV2))          # Gruppengrößen
> DV  <- c(DV1, DV2)                          # Gesamt-Daten
> IV  <- factor(rep(1:2, Nj), labels=c("A", "B")) # welche Gruppe
> mood.test(DV ~ IV, alternative="greater")
Mood two-sample test of scale
data: DV by IV
Z = 2.6968, p-value = 0.003500
alternative hypothesis: greater
```

Die Ausgabe des Tests umfasst den Wert der z-transformierten Teststatistik (Z), wofür Erwartungswert und Varianz einer asymptotisch gültigen Normalverteilung verwendet werden. Der p-Wert wird unter p-value ausgegeben. Das Ergebnis lässt sich manuell prüfen, wobei auf die enge Verwandtschaft zum Wilcoxon-Rangsummen-Test hingewiesen sei (Abschn. 10.5.4): Lediglich die Wahl der Gewichte ist hier eine andere.

```
# Gewichte: quadrierte Differenzen der Ränge zu ihrem Mittelwert
> gX <- (rank(DV) - mean(rank(DV)))^2
> MN <- sum(gX[IV == "A"])    # Teststat.: Summe Gewichte der 1. Gruppe

# Erwartungswert und theoretische Varianz der Teststatistik
> N      <- sum(Nj)                  # Anzahl Beobachtungsobjekte
> muMN  <- (Nj[1] * (N^2 - 1)) / 12   # Erwartungswert
> varMN <- (Nj[1]*Nj[2] * (N+1) * (N^2 - 4)) / 180   # Varianz
> (MNz <- (MN-muMN) / sqrt(varMN))    # theoretische z-Transformation
[1] 2.696799

> (pVal <- pnorm(MNz, 0, 1, lower.tail=FALSE))  # p-Wert einseitig
[1] 0.003500471
```

10.4.2 Ansari-Bradley-Test

Der Ansari-Bradley-Test stellt eine Alternative zum Mood-Test dar und ist ebenfalls für den Vergleich der Variabilität von stetigen, ordinalen Daten aus zwei unabhängigen Stichproben

¹⁷Voraussetzen ist, dass die Verteilung in beiden Stichproben denselben theoretischen Median hat. Bei Zweifeln daran können die Daten zuvor gruppenweise zentriert werden, indem man von jedem Wert den Gruppenmedian abzieht.

geeignet (Fußnote 17).

```
ansari.test(x=<Vektor>, y=<Vektor>, exact=NULL,
            alternative=c("two.sided", "less", "greater"))
```

Unter x und y sind die Daten aus beiden Stichproben einzutragen. Alternativ zu x und y kann auch eine Modellformel $\langle AV \rangle \sim \langle UV \rangle$ eingegeben werden. Stammen die Variablen in der Modellformel aus einem Datensatz, ist dieser unter data zu nennen. Das Argument alternative bezieht sich auf die Variabilität x im Vergleich zu jener von y. Für die (bei großen Stichproben u. U. etwas zeitaufwendigere) Berechnung auf Basis der exakten anstatt asymptotisch gültigen Normalverteilung ist exact=TRUE zu setzen. Das folgende Beispiel verwendet die Daten aus Abschn. 10.4.1.

```
> ansari.test(DV ~ IV, alternative="greater", exact=FALSE)
Ansari-Bradley test
data: DV by IV
AB = 6, p-value = 0.004687
alternative hypothesis: true ratio of scales is greater than 1
```

Die Ausgabe des Tests umfasst den Wert der Teststatistik (AB) sowie den zugehörigen *p*-Wert (p-value). Das Ergebnis lässt sich manuell bestätigen, wobei sich die Teststatistik lediglich in der Wahl der Gewichte von jener des Mood-Tests unterscheidet.

```
# Gewichtung: mittlere absolute Abweichung der Ränge zu ihrem Mittel
> gX <- mean(rank(DV)) - abs(rank(DV) - mean(rank(DV)))
> (AB <- sum(gX[IV == "A"])) # Teststat.: Summe Gewichte der 1. Gruppe
[1] 6

# Erwartungswert und theoretische Varianz der Teststatistik
# abhängig davon, ob Gesamt-N gerade oder ungerade ist
> muABe <- (Nj[1] * (N+2)) / 4 # gerades N
> muABo <- (Nj[1] * (N+1)^2) / (4*N) # ungerades N
> varABe <- (Nj[1]*Nj[2] * (N^2 - 4)) / (48 * (N-1)) # gerades N
> varABo <- (Nj[1]*Nj[2] * (N+1) * (3 + N^2)) / (48*N^2) # ungerades N
> muAB <- ifelse(N %% 2, muABo, muABe) # hier passender EW
> varAB <- ifelse(N %% 2, varABo, varABe) # hier passende Varianz
> ABz <- (AB-muAB) / sqrt(varAB) # theo. z-Transformation
> pnorm(ABz, 0, 1) # p-Wert linksseitig
[1] 0.004687384
```

10.5 Tests auf Übereinstimmung von Verteilungen

Die im Folgenden aufgeführten Tests lassen sich mit Werten von stetigen ordinalen (aber nicht notwendigerweise metrischen und normalverteilten) Variablen durchführen und testen Hypothesen über die Lage ihrer Verteilungen. Eine Alternative stellen die in Abschn. 11.3 vorgestellten Permutationstests dar.

10.5.1 Kolmogorov-Smirnov-Test für zwei Stichproben

Der Kolmogorov-Smirnov-Test prüft die Daten einer stetigen Variable aus zwei unabhängigen Stichproben daraufhin, ob sie mit der Nullhypothese verträglich sind, dass die Variable in beiden Bedingungen dieselbe Verteilung besitzt (für den Anpassungstest s. Abschn. 10.1.3). Als *Omnibus*-Test prüft er gleichzeitig, ob Lage und Form beider Verteilungen übereinstimmen. Dazu wird für die Argumente `x` und `y` von `ks.test()` jeweils der Vektor der Daten aus den Bedingungen angegeben. Alternativ ist eine Modellformel der Form $\langle AV \rangle \sim \langle UV \rangle$ möglich, wobei $\langle UV \rangle$ die Zugehörigkeit der Werte der Zielgröße $\langle AV \rangle$ zu Gruppen definiert. Stammen die Variablen in der Modellformel aus einem Datensatz, ist dieser unter `data` zu nennen.

Über das Argument `alternative` sind ungerichtete wie gerichtete Alternativhypotesen prüfbar, wobei sich letztere darauf beziehen, ob `y` stochastisch kleiner ("`less`") oder größer ("`greater`") als `x` ist (Abschn. 10.1.3, Fußnote 4).

```
> DV1  <- round(rnorm(8, mean=1, sd=2), 2)      # Daten Stichprobe 1
> DV2  <- round(rnorm(8, mean=3, sd=2), 2)      # Daten Stichprobe 2
> d_DV <- data.frame(DV=c(DV1, DV2),            # Datensatz
+                      IV=rep(c("G1", "G2"), each=8))

> ks.test(DV ~ IV, data=d_DV, alternative="greater")
Two-sample Kolmogorov-Smirnov test
data: DV by IV
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. 10.4).

```
> 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[-length(both1)]) - 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

# grafische Darstellung: lege Wertebereich der x-Achse fest
> xRange <- c(min(sortBoth)-1, max(sortBoth)+1)

# 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)

# füge kumulierte relative Häufigkeiten 2 hinzu
> par(new=TRUE)
> 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 rel. Häufigkeiten 1",
+      "kumulierte rel. Häufigkeiten 2"), col=c("blue", "red"), lwd=2)
```

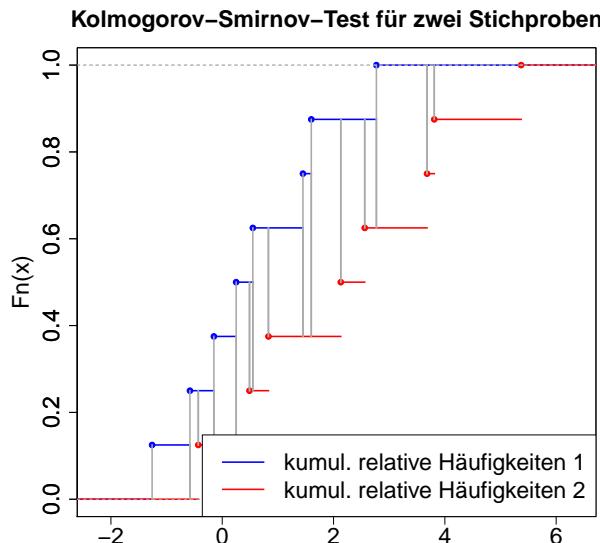


Abbildung 10.4: Kolmogorov-Smirnov-Test für zwei Stichproben: Abweichungen zwischen kumulierten relativen Häufigkeiten der Daten aus beiden Bedingungen.

10.5.2 Vorzeichen-Test

Der Vorzeichen-Test für den Median ist anwendbar, wenn eine Variable eine symmetrische Verteilung unbekannter Form besitzt. Es wird die H_0 getestet, dass der Median der Verteilung gleich einem bestimmten Wert m_0 ist.¹⁸ Sowohl gerichtete wie ungerichtete Alternativhypotesen sind möglich. Eine Umsetzung liefert `SignTest()` aus dem Paket `DescTools`.

```
SignTest(<Vektor>, mu=<Median unter H0>, conf.level=0.95,
         alternative=c("two.sided", "less", "greater"))
```

Neben dem Vektor der Daten ist für das Argument `mu` der Median unter H_0 anzugeben. Die Argumente `alternative` und `conf.level` beziehen sich auf die Größe des Medians unter H_1 im Vergleich zu seiner Größe unter H_0 bzw. auf die Breite seines Vertrauensintervalls.

```
> medH0 <- 30                                     # Median unter H0
> DV      <- sample(0:100, 20, replace=TRUE)       # Daten
> library(DescTools)                             # für SignTest()
> SignTest(DV, mu=medH0)
One-sample Sign-Test
data:  DV
S = 15, number of differences = 20, p-value = 0.04139
alternative hypothesis: true median is not equal to 30
95.9 percent confidence interval:
 34 74
sample estimates:
median of the differences
        45
```

Teststatistik ist die Anzahl der beobachteten Werte $> m_0$.¹⁹ Diese Anzahl besitzt unter H_0 eine Binomialverteilung mit der Trefferwahrscheinlichkeit 0.5. Um den p -Wert der Teststatistik zu berechnen, dient `pbinom()` für die Verteilungsfunktion der Binomialverteilung. Da die Binomialverteilung unter H_0 symmetrisch ist, kann der p -Wert des zweiseitigen Binomialtests als das Doppelte des einseitigen p -Wertes gebildet werden.

```
> N      <- length(DV)                           # Anzahl an Beobachtungen
> DVsel <- DV[DV != medH0]                      # Werte = medH0 eliminieren
> (obs  <- sum(DVsel > medH0))                # Teststatistik
[1] 15

# rechtsseitiger Test
> (pGreater <- pbinom(obs-1, N, 0.5, lower.tail=FALSE))
[1] 0.02069473
```

¹⁸Bei Variablen mit symmetrischer Verteilung und eindeutig bestimmtem Median ist dieser gleich dem Erwartungswert, sofern letzterer existiert.

¹⁹Werte gleich m_0 werden also nicht berücksichtigt. Andere Vorgehensweisen werden diskutiert. So können im Fall geradzahlig vieler Werte gleich m_0 die Hälfte dieser Werte als $< m_0$, die andere Hälfte als $> m_0$ codiert werden. Im Fall ungeradzahlig vieler Werte gleich m_0 wird ein Wert eliminiert und dann wie für geradzahlig viele Werte beschrieben verfahren.

```
> (pTwoSided <- 2 * pGreater)
[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. Hierfür ist ebenfalls `SignTest()` geeignet, wobei die Funktion intern die paarweisen Differenzen bildet und wie beschrieben mit $m_0 = 0$ testet.

10.5.3 Wilcoxon-Vorzeichen-Rang-Test für eine Stichprobe

Der Wilcoxon-Vorzeichen-Rang-Test prüft, ob die in einer Stichprobe ermittelten Werte einer Variable mit symmetrischer Verteilung mit der H_0 verträglich sind, dass der Median der Verteilung gleich einem bestimmten Wert m_0 ist. Anders als beim t -Test für eine Stichprobe (Abschn. 7.3.1) wird nicht vorausgesetzt, dass die Variable normalverteilt ist. Im Vergleich zum Vorzeichen-Test für den Median wird neben der Anzahl der Werte $> m_0$ auch das Ausmaß ihrer Differenz zu m_0 berücksichtigt.

```
wilcox.test(x=<Vektor>, alternative=c("two.sided", "less", "greater"),
            mu=0, conf.int=FALSE)
```

Unter `x` ist der Datenvektor einzutragen. Alternativ zu `x` kann auch eine Modellformel $\text{AV} \sim 1$ angegeben werden. Stammt die Variable AV aus einem Datensatz, ist dieser unter `data` zu nennen. Mit `alternative` wird festgelegt, ob die H_1 bzgl. des Vergleichs mit dem unter H_0 angenommenen Median `mu` gerichtet oder ungerichtet ist. `"less"` und `"greater"` beziehen sich dabei auf die Reihenfolge Median unter H_1 `"less"` bzw. `"greater"` m_0 . Um ein Konfidenzintervall für den Median zu erhalten, ist `conf.int=TRUE` zu setzen.

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 H_0 verträglich sind, dass der theoretische Median 110 beträgt. Unter H_1 sei der Median größer.

```
> IQ      <- c(99, 131, 118, 112, 128, 136, 120, 107, 134, 122)
> medH0 <- 110                                     # Median unter H0
> wilcox.test(IQ, alternative="greater", mu=medH0, conf.int=TRUE)
Wilcoxon signed rank test
data: IQ
V = 48, p-value = 0.01855
alternative hypothesis: true location is greater than 110
95 percent confidence interval:
113.5 Inf
sample estimates:
(pseudo)median
121

# alternativer Aufruf mit Modellformel für Variable aus Datensatz
> IQdat <- data.frame(IQ=IQ)
> wilcox.test(IQ ~ 1, alternative="greater", data=IQdat,
```

```
+ mu=medH0, conf.int=TRUE) # ...
```

Die Ausgabe liefert die Teststatistik (`V`) und den p -Wert (`p-value`). Das Ergebnis lässt sich manuell nachvollziehen. Die diskret verteilte Teststatistik berechnet sich dabei als Summe der Ränge der absoluten Differenzen zu m_0 , wobei nur die Ränge von Werten aufsummiert werden, die $> m_0$ sind. Um den p -Wert der Teststatistik zu erhalten, dient `psignrank()` als Verteilungsfunktion der Teststatistik (Abschn. 5.3.2, Fußnote 10). Wurde `conf.int=TRUE` gesetzt, enthält die Ausgabe neben dem Konfidenzintervall für den Pseudo-Median unter (pseudo)median den zugehörigen Hodges-Lehmann-Schätzer (Abschn. 2.7.4).

```
> (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 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

# p-Wert einseitig
> (pVal <- psignrank(V-1, length(IQ), lower.tail=FALSE))
[1] 0.01855469

# Hodges-Lehmann-Schätzer: Median der Mittelwerte aller Wertepaare
> pairM <- outer(IQ, IQ, FUN="+") / 2 # Mittelwerte aller Paare
> (HLloc <- median(pairM[lower.tri(pairM, diag=TRUE)])) # deren Median
[1] 121
```

10.5.4 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 stetigen ordinalen Variable daraufhin miteinander verglichen, ob sie mit der H_0 verträglich sind, dass die Verteilungen der Variable in den zugehörigen Bedingungen identisch sind. Anders als im analogen t -Test (Abschn. 7.3.2) 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 sind. Es kann sowohl gegen eine ungerichtete wie gerichtete H_1 getestet werden, die sich auf den Lageparameter der Verteilungen (etwa den Median) bezieht.²⁰

²⁰Ohne Annahme gleicher Verteilungsform beziehen sich Null- und Alternativhypothese auf die Wahrscheinlichkeit dafür, dass eine zufällige Beobachtung aus der ersten Gruppe größer bzw. kleiner als eine zufällig

```
wilcox.test(x=<Vektor>, y=<Vektor>, paired=FALSE, conf.int=FALSE,
            alternative=c("two.sided", "less", "greater"))
```

Unter `x` sind die Daten der ersten Stichprobe einzutragen, unter `y` entsprechend die der zweiten. Alternativ zu `x` und `y` kann auch eine Modellformel $\langle \text{AV} \rangle \sim \langle \text{UV} \rangle$ angegeben werden. Dabei ist $\langle \text{UV} \rangle$ ein Gruppierungsfaktor derselben Länge wie $\langle \text{AV} \rangle$ und gibt für jede Beobachtung der Zielvariable $\langle \text{AV} \rangle$ die Gruppenzugehörigkeit an. Stammen die Variablen in der Modellformel aus einem Datensatz, muss dieser unter `data` eingetragen werden. Mit `alternative` wird festgelegt, ob die H_1 gerichtet oder ungerichtet ist. `"less"` und `"greater"` beziehen sich dabei auf den Lageparameter in der Reihenfolge `x` `"less"` bzw. `"greater"` `y`. Das Argument `paired=FALSE` bestimmt, dass es sich um unabhängige Stichproben handelt. Um ein Konfidenzintervall für die Differenz der Lageparameter zu erhalten, ist `conf.int=TRUE` zu setzen.

Als Beispiel diene jenes aus [Bortz et al. \(2010, p. 201 ff.\)](#): Es wird vermutet, dass sich durch die Einnahme eines Medikaments die Reaktionszeit im Vergleich zu einer Kontrollgruppe verkürzen lässt.

```
> rtCtrl <- 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)
> Nj      <- c(length(rtCtrl), length(rtDrug))    # Gruppengrößen

# Faktor der Gruppenzugehörigkeiten
> IV      <- factor(rep(1:2, Nj), labels=c("control", "drug"))
> rtAll <- c(rtCtrl, rtDrug)                      # Gesamtstichprobe
> wilcox.test(rtAll ~ IV, alternative="greater", conf.int=TRUE)

Wilcoxon rank sum test
data: rtAll by IV
W = 94, p-value = 0.4333
alternative hypothesis: true location shift is greater than 0
95 percent confidence interval:
-11 Inf
sample estimates:
difference in location
2
```

Die Ausgabe nennt den Wert der Teststatistik (`W`) gefolgt vom p -Wert (`p-value`). Das Ergebnis lässt sich manuell nachvollziehen. Dazu muss für die Daten 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 $\sum_{i=1}^{n_1} i = \frac{n_1(n_1+1)}{2}$ abgezogen wird. Um den p -Wert zu erhalten, dient `pwilcox()` als Verteilungsfunktion der Teststatistik (Abschn. 5.3.2, Fußnote 10). Wurde `conf.int=TRUE` gesetzt, enthält die Ausgabe neben dem Konfidenzintervall für die Differenz der Lageparameter unter `difference in location` den zugehörigen Hodges-Lehmann-Schätzer (Abschn. 2.7.4).

```
> gX <- rank(rtAll)                                # Gewichte = Ränge
> (W <- sum(gX[IV == "control"]) - sum(1:Nj[1]))   # Teststatistik
[1] 94
```

gezogene Beobachtung aus der zweiten Gruppe ist.

```
# p-Wert einseitig
> (pVal <- pwilcox(W-1, Nj[1], Nj[2], lower.tail=FALSE))
[1] 0.433342

# Hodges-Lehmann-Schätzer der Differenz der Lageparameter
> pairD <- outer(rtCtrl, rtDrug, "-")      # alle paarweisen Differenzen
> (HLdl <- median(pairD))                  # deren Median
[1] 2
```

Die Funktion `pairwise.wilcox.test()` dient dazu, simultan alle paarweisen Vergleiche zwischen jeweils zwei von insgesamt mehreren Gruppen mit dem Wilcoxon-Rangsummen-Test durchzuführen. Sie bietet mit dem Argument `p.adjust.method` verschiedene Möglichkeiten zur Adjustierung des α -Niveaus.

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(rtCtrl, rtDrug, ">=")))
[1] 94
```

10.5.5 Wilcoxon-Test für zwei abhängige Stichproben

Der Wilcoxon-Test für zwei 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. Alternativ kann auch eine Modellformel `Pair(x, y) ~ 1` verwendet werden, wobei dann das Argument `paired=TRUE` entfällt. Sind `x` und `y` aus der Modellformel Teil eines Datensatzes, ist dieser unter `data` zu nennen.

Bei im Long-Format vorliegenden Daten kann analog zum Test für zwei unabhängige Stichproben eine Modellformel $\langle AV \rangle \sim \langle UV \rangle$ angegeben werden – ggf. zusammen mit einem Datensatz `data`, aus dem die Variablen stammen. Die Daten in $\langle AV \rangle$ müssen so geordnet sein, dass innerhalb jeder von $\langle UV \rangle$ definierten Gruppe dieselbe Reihenfolge von Beobachtungsobjekten vorliegt, um die paarweise Zuordnung sicherzustellen. Zudem dürfen keine fehlenden Werte vorhanden sein.

Nach Bildung der paarweisen Differenzen einander zugeordneter Werte wird im Wilcoxon-Vorzeichen-Rang-Test für eine Stichprobe die H_0 getestet, dass der theoretische Median der Differenzwerte gleich 0 ist.

```
> N      <- 20                      # Stichprobengröße
> DVpre <- rnorm(N, mean=90, sd=15)  # Daten prä
> DVpost <- rnorm(N, mean=100, sd=15) # Daten post
> DV     <- c(DVpre, DVpost)        # Gesamt-Daten
```

```
# Faktor, der jedem Wert von DV Messzeitpunkt zuordnet
# dabei Kontrolle der Reihenfolge der Faktorstufen: pre vor post
> IV <- factor(rep(0:1, each=N), labels=c("pre", "post"))
> wilcox.test(DV ~ IV, alternative="less", paired=TRUE, conf.int=TRUE)
Wilcoxon signed rank exact test
data: DV by IV
V = 57, p-value = 0.03793
alternative hypothesis: true location shift is less than 0
95 percent confidence interval:
-Inf -2.236827
sample estimates:
(pseudo)median
-12.85819

# äquivalent: 1-Stichproben Test der Differenzvariable
> DVdiff <- DVpre - DVpost # Differenzvariable
> wilcox.test(DVdiff, alternative="less") # ...
```

10.5.6 Kruskal-Wallis-*H*-Test für unabhängige Stichproben

Der Kruskal-Wallis-*H*-Test verallgemeinert die Fragestellung eines Wilcoxon-Tests auf Situationen, in denen Werte einer Variable in mehr als zwei unabhängigen Stichproben ermittelt wurden. Unter H_0 sind die Verteilungen der Variable in den zugehörigen Bedingungen identisch. Die unspezifische H_1 besagt, dass sich mindestens zwei Lageparameter unterscheiden. Anders als in der einfaktoriellen Varianzanalyse (Abschn. 7.4) 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=<Modellformel>, data=<Datensatz>)
```

Unter **formula** sind Daten und Gruppierungsvariable als Modellformel $\langle AV \rangle \sim \langle UV \rangle$ zu nennen, wobei $\langle UV \rangle$ ein Gruppierungsfaktor derselben Länge wie $\langle AV \rangle$ ist und für jede Beobachtung in der Zielgröße $\langle AV \rangle$ die Gruppenzugehörigkeit angibt. Geschieht dies mit Variablen aus einem Datensatz, muss dieser unter **data** eingetragen werden.

Als Beispiel diene jenes aus [Büning und Trenkler \(1994, p. 183 ff.\)](#): Es wird vermutet, dass der IQ-Wert von Studierenden aus vier Studiengängen einen unterschiedlichen Erwartungswert besitzt.

```
# IQ-Werte von Studierenden 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)
> DV <- c(IQ1, IQ2, IQ3, IQ4) # kombinierte Daten
```

```
# Stichprobengrößen
> Nj <- c(length(IQ1), length(IQ2), length(IQ3), length(IQ4))
> N   <- sum(Nj)                                # Gesamt-N

# Faktor der Gruppenzugehörigkeiten
> IV   <- factor(rep(1:4, Nj), labels=c("I", "II", "III", "IV"))
> KWdf <- data.frame(IV, DV)                   # 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 Teststatistik H gefolgt von den Freiheitsgraden (`df`) und dem p -Wert (`p-value`). Das Ergebnis lässt sich 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.²¹

```
> (rankSumI <- tapply(rank(DV), 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/Nj) - 3*(N+1))  # Teststat.
[1] 1.75531

> (pVal <- pchisq(H, nlevels(IV)-1, lower.tail=FALSE))      # p-Wert
[1] 0.624708
```

Für den Jonckheere-Terpstra-Trend-Test für geordnete Gruppen s. `JonckheereTerpstraTest()` aus dem Paket `DescTools`. Tests nach Dunn bzw. Conover-Iman sind post-hoc Paarvergleiche auf Basis von Rangsummen, deren Testlogik zum Kruskal-Wallis- H -Test passt. Sie lassen sich mit `DunnTest()` bzw. `ConoverTest()` ebenfalls aus dem Paket `DescTools` durchführen.

10.5.7 Friedman-Rangsummen-Test für abhängige Stichproben

Der Rangsummen-Test nach Friedman dient der Analyse von Daten einer 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 (gematchten) Beobachtungsobjekten. Wie im Kruskal-Wallis- H -Test wird die H_0 geprüft, dass die Verteilung der Variable in allen Bedingungen identisch ist. Die unspezifische H_1 besagt, dass sich mindestens zwei Lageparameter unterscheiden. Anders als in der einfaktoriellen Varianzanalyse für abhängige Gruppen (Abschn. 7.5) wird nicht vorausgesetzt, dass die blockweise als Vektor zusammengefasste Variable gemeinsam normalverteilt ist.

²¹Im 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 manuell berechnete Teststatistik und der p -Wert stimmen deshalb nicht exakt mit jenen aus `kruskal.test()` überein.

```
friedman.test(y=<Daten>, groups=<Faktor>, blocks=<Faktor>)
```

Die Daten müssen im Long-Format vorliegen (Abschn. 3.3.11): 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 welchem Beobachtungsobjekt sie stammt, wenn Messwiederholung vorliegt. Alternativ lässt sich auch eine Modellformel der Form `y ~ groups | blocks` mit denselben Bedeutungen nennen. Stammen die Variablen in der Modellformel aus einem Datensatz, muss dieser für das Argument `data` eingetragen werden.

Als Beispiel diene jenes aus Bortz et al. (2010, p. 269 ff.): An Ratten wird die Auswirkung von zentralnervös anregenden Präparaten erhoben, wobei die Anzahl der pro Zeiteinheit gemessenen Umdrehungen in einem Laufrad gemessen wird. Aus jeweils vier hinsichtlich verschiedener Störvariablen homogenen Ratten werden fünf Blöcke gebildet und jeder Block in allen vier Bedingungen beobachtet.

```
# Daten in den einzelnen Bedingungen
> DVcaff <- c(14, 13, 12, 11, 10)          # Koffein
> DVds   <- c(11, 12, 13, 14, 15)          # Medikament einfache Dosis
> DVdd   <- c(16, 15, 14, 13, 12)          # Medikament doppelte Dosis
> DVplac <- c(13, 12, 11, 10, 9)           # Placebo
> DV      <- c(DVcaff, DVds, DVdd, DVplac) # Gesamt-Daten
> nBl    <- length(DVcaff)                  # 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, times=P))      # Faktor Blockzugehörigkeit
> fDf    <- data.frame(IV, DV, blocks)        # Datensatz
> friedman.test(DV ~ IV | blocks, data=fDf)

Friedman rank sum test
data: DV and 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 p -Wert (`p-value`). Das Ergebnis lässt sich manuell nachvollziehen, wobei das zentrale Element der Teststatistik das Quadrat der pro Gruppe gebildeten Summe der Ränge innerhalb jedes Blocks ist.²²

```
> (DVmat <- cbind(DVcaff, DVds, DVdd, DVplac))    # Datenmatrix
   DVcaff  DVds  DVdd  DVplac
[1,]     14     11     16     13
```

²²Im 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.

```
[2,] 13 12 15 12
[3,] 12 13 14 11
[4,] 11 14 13 10
[5,] 10 15 12 9

> (rankMat <- t(apply(DVmat, 1, rank))) # Matrix blockweise Ränge
   DVcaff DVds DVdd DVplac
[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 <- colSums(rankMat))           # gruppenweise Rangsummen
DVcaff DVds DVdd DVplac
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 <- pchisq(S, P-1, lower.tail=FALSE)) # p-Wert
[1] 0.04398959
```

Für den Page-Trend-Test für geordnete abhängige Gruppen s. `PageTest()` aus dem Paket `DescTools`.

10.5.8 Cochran-Q-Test für abhängige Stichproben

Cochrancs *Q*-Test ist analog zum Friedman-Test für den Fall, dass die Messwerte dichotom sind. Bei diesem Test werden die Messwerte nicht zunächst in Ränge umgewandelt, sondern bei der Codierung mit 0 und 1 belassen. Es liegen abhängige Daten aus p Bedingungen vor: Entweder liefert jedes Beobachtungsobjekt Werte aus jeder Bedingung (Messwiederholung), oder aber p homogene (gematchte) Beobachtungsobjekte werden zu einem Block zusammengefasst, von dem dann p abhängige Werte aus den Bedingungen stammen. Der *Q*-Test wird mit `symmetry_test()` aus dem Paket `coin` durchgeführt und prüft die H_0 , dass die Trefferwahrscheinlichkeit in allen Bedingungen identisch ist.

```
symmetry_test(formula=<Modellformel>, data=<Daten>, teststat="quad")
```

Die Daten müssen im Long-Format vorliegen (Abschn. 3.3.11). Unter `formula` ist eine Modellformel der Form $\langle AV \rangle \sim \langle UV \rangle \mid \langle BlockId \rangle$ zu nennen, wobei $\langle UV \rangle$ ein Faktor derselben Länge wie $\langle AV \rangle$ ist und für jede Beobachtung in der Zielgröße $\langle AV \rangle$ die Gruppenzugehörigkeit angibt. $\langle BlockId \rangle$ ist ebenfalls ein Faktor derselben Länge wie $\langle AV \rangle$ und enthält die Blockzugehörigkeit jedes Wertes. Stammen die Variablen in der Modellformel aus einem Datensatz, muss dieser unter `data` eingetragen werden. Für den *Q*-Test ist zudem das Argument `teststat="quad"`

zu setzen, da `symmetry_test()` es 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)

> N      <- 10                      # Anzahl Blöcke
> year   <- factor(rep(1981:1985, times=N))    # Faktor Messzeitpunkt
> P      <- nlevels(year)          # Anzahl Messzeitpunkte
> id     <- factor(rep(1:N, each=P))        # Faktor Blockzugehörigkeit
> library(coin)                   # für symmetry_test()
> 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 p -Wert (p-value). Das Ergebnis lässt sich 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=N, ncol=P, byrow=TRUE)
> rSum     <- rowSums(prefMat)           # Zeilensummen
> cSum     <- colSums(prefMat)          # Spaltensummen

# Teststatistik Q
> (Q <- (P*(P-1)*sum((cSum-mean(cSum))^2)) / (P*sum(rSum)-sum(rSum^2)))
[1] 1.333333

> (pVal <- pchisq(Q, P-1, lower.tail=FALSE))    # p-Wert
[1] 0.8556952
```

10.5.9 Bowker-Test für zwei abhängige Stichproben

Der Bowker-Test ist für Situationen geeignet, in denen eine kategoriale Variable mit mehr als zwei Ausprägungen in zwei abhängigen Stichproben beobachtet wird. Analog zum Friedman-Test besteht die H_0 darin, dass die Verteilung der Variable in beiden Bedingungen identisch ist und damit eine bzgl. der Hauptdiagonale symmetrische Kontingenztafel der gemeinsamen

Häufigkeiten erwartet wird. Die ungerichtete H_1 besagt, dass es eine systematische Abweichung von Übereinstimmung in eine Richtung gibt. `mcnemar.test()` berechnet automatisch den Bowker-Test, wenn eine Kontingenztafel mit jeweils mehr als zwei Zeilen und Spalten als Argument übergeben wird.

```
mcnemar.test(x, y=NULL)
```

Unter `x` kann die quadratische Kontingenztafel der beiden Datenvektoren einer kategorialen 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.

Als Beispiel diene jenes aus [Bortz et al. \(2010, p. 165 ff.\)](#): An denselben Personen soll die empfundene Leistungssteigerung als Wirkung eines Medikaments oder Placebos untersucht werden. Erhoben wird die Einschätzung, ob keine, eine geringe, oder eine starke Wirkung vorliegt.

```
> categ <- factor(1:3, labels=c("lo", "med", "hi")) # Kategorien
> Q      <- nlevels(categ)                           # Anzahl 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))
> cTabBow <- xtabs(~ drug + plac)                  # Kontingenztafel
> addmargins(cTabBow)                               # Randsummen
  plac
drug  lo  med  hi  Sum
  lo  14    7   9   30
  med   5   26  19   50
  hi    1    7  12   20
  Sum  20   40  40  100

> mcnemar.test(cTabBow)
McNemar's Chi-squared test
data:  cTabBow
McNemar's chi-squared = 12.2718, df = 3, p-value = 0.006508
```

Die Ausgabe nennt den Wert der asymptotisch χ^2 -verteilten Teststatistik gefolgt von den Freiheitsgraden (`df`) und dem p -Wert (`p-value`). Da die Symmetrie einer Kontingenztafel zu testen ist, eignet sich hier wie bei Cochran's Q -Test auch die Funktion `symmetry_test()` aus dem `coin` Paket zur Auswertung. Sie akzeptiert die Kontingenztafel der Übereinstimmungen beider Bedingungen als Argument.

```
> library(coin)                                     # für symmetry_test()
> symmetry_test(cTabBow, teststat="quad")          # ...
```

Teststatistik des Bowker-Tests ist die Summe der quadrierten Differenzen von an der Hauptdiagonale gespiegelten Einträgen der Kontingenztafel, die zuvor an der Summe beider Einträge relativiert wurden.

```
# relativierte quadrierte Differenzen Kontingenztafel vs. Transponierte
> sqDiffs <- (cTabBow - t(cTabBow))^2 / (cTabBow + t(cTabBow))

# summiere nur über die Differenzen der oberen Dreiecksmatrix
> (chisqVal <- sum(sqDiffs[upper.tri(cTabBow)]))      # Teststatistik
[1] 12.27179

> (bowDf <- choose(Q, 2))                                # Freiheitsgrade Q*(Q-1)/2
[1] 3

> (pVal <- pchisq(chisqVal, bowDf, lower.tail=FALSE)) # p-Wert
[1] 0.006507799
```

10.5.10 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 H_0 , dass die Verteilung der 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 H_1 besagt, dass es eine systematische Abweichung von Übereinstimmung in eine Richtung gibt. Wie im Bowker-Test kann die Hypothese mit `mcnemar.test()` geprüft werden. Zusätzlich legt hier das Argument `correct` fest, ob eine Stetigkeitskorrektur durchgeführt wird (Voreinstellung: `TRUE`).

Im Beispiel sei an einer Stichprobe jeweils vor und nach einer Informationskampagne die Variable erhoben worden, ob eine Person raucht.

```
> N      <- 20                                     # Anzahl Versuchspersonen
> pre   <- rbinom(N, size=1, prob=0.6)           # Prä-Messung
> post  <- rbinom(N, size=1, prob=0.4)           # Post-Messung

# Konvertierung der numerischen Vektoren in Faktoren
> preFac <- factor(pre, levels=0:1, labels=c("no", "yes"))
> postFac <- factor(post, levels=0:1, labels=c("no", "yes"))
> P       <- nlevels(preFac)                      # Anzahl Kategorien
> cTab    <- xtabs(~ preFac + postFac)          # Kontingenztafel
> addmargins(cTab)                               # Randsummen
  postFac
preFac 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
```

Wie beim Bowker-Test eignet sich auch hier die Funktion `symmetry_test()` aus dem `coin` Paket zur Auswertung, die als Argument die Kontingenztafel der Übereinstimmungen beider Bedingungen erwartet.

```
> library(coin)                                # für symmetry_test()
> 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.

```
# Teststatistik
> (chisqVal <- (cTab[1,2] - cTab[2,1])^2 / (cTab[1,2] + cTab[2,1]))
[1] 0.0909091

> (mcnDf <- choose(P, 2))                  # Freiheitsgrade P*(P-1)/2
[1] 1

> (pVal <- pchisq(chisqVal, mcnDf, lower.tail=FALSE))    # p-Wert
[1] 0.7630246
```

Treten kleine erwartete Zellhäufigkeiten auf (etwa < 5), kann die χ^2 -Approximation der Verteilung der Teststatistik noch ungenau sein. In diesem Fall lässt sich ein *exakter* McNemar-Test mit Hilfe eines ungerichteten Binomialtests durchführen: Er wertet eine Zelle außerhalb der Diagonale der Kontingenztafel als Anzahl der Treffer, die andere Zelle außerhalb der Diagonale als Anzahl der Nicht-Treffer und testet gegen die $H_0: p_0 = 0.5$. Dieser Test gilt jedoch als recht konservativ.

```
# Zellen der Gegendiagonale als Anzahl der Treffer, Nicht-Treffer
> binom.test(c(cTab[1, 2], cTab[2, 1]), p=0.5)
Exact binomial test
data: c(cTab[1, 2], cTab[2, 1])
number of successes = 6, number of trials = 11, p-value = 1
alternative hypothesis: true probability of success not equal to 0.5
95 percent confidence interval:
0.2337936 0.8325119
sample estimates:
probability of success
0.5454545
```

10.5.11 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 (Abschn. 10.3.3). Verglichen mit dem Bowker-Test bezieht er sich auf nur einen Spezialfall, der zu einer asymmetrischen Kontingenztafel der Übereinstimmungen führen kann.

Zur Berechnung des Tests steht aus dem `coin` Paket die Funktion `mh_test()` bereit. Sie erwartet als Argument die Kontingenztafel der Übereinstimmungen der kategorialen Messwerte in beiden Bedingungen. Als Beispiel sei wie beim Bowker-Test jenes aus Bortz et al. (2010, p. 165 ff.) herangezogen.

```
> library(coin)                      # für mh_test()
> 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  lo  med  hi  Sum
lo    14    7    9   30
med    5   26   19   50
hi     1    7   12   20
Sum   20   40   40  100

# Hälften 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))
lo  med  hi
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 <- pchisq(chisqVal, smmhDf, lower.tail=FALSE)) # p-Wert
[1] 0.002312643
```

Im allgemeinen Fall ist es zur Bestimmung der Teststatistik notwendig, die Kovarianzmatrix der Abweichungen beider Randverteilungen unter H_0 zu bestimmen und zu invertieren (Abschn. 12.1.2). 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 Randverteilungen unter H0
> S <- -(cTabBow + t(cTabBow))

# setze Diagonale dieser Kovarianzmatrix
> diag(S) <- rowSums(cTabBow) + colSums(cTabBow) - 2*diag(cTabBow)

# berücksichtige für Teststatistik nur die ersten P-1 Kategorien
> keep      <- seq_len(nrow(cTabBow)-1)
> (chisqVal <- t(d[keep]) %*% solve(S[keep, keep]) %*% d[keep])
[1] 12.13873
```

Eine Alternative zum Stuart-Maxwell-Test auf homogene Randhäufigkeiten ist der Bhapkar-Test, den das Paket **DescTools** umsetzt.

Kapitel 11

Resampling-Verfahren

Resampling-Verfahren kommen für eine Vielzahl von Tests in Frage, können hier aber nur in Grundzügen vorgestellt werden. Ausgangspunkt ist die gesuchte Verteilung einer Teststatistik – etwa eines Schätzers $\hat{\theta}$ für einen theoretischen Parameter θ . Diese Verteilung kann aus verschiedenen Gründen unbekannt sein: So sind etwa die in parametrischen Tests gemachten Annahmen, unter denen ihre Teststatistik eine bekannte Verteilung aufweist, nicht immer zu rechtfertigen. In vielen klassischen nonparametrischen Verfahren ist die Verteilung der Teststatistik zwar im Prinzip exakt zu ermitteln, praktisch aber der Rechenaufwand dafür zu hoch.

Grundidee von Bootstrap-Verfahren und Permutationstests ist es, aus den gegebenen Daten einer festen Basisstichprobe viele neue Zufallsstichproben (*resamples*) zu generieren und die Teststatistik für jedes resample zu ermitteln – die dabei berechneten Werte werden als $\hat{\theta}^*$ bezeichnet. Die empirische Verteilung der $\hat{\theta}^*$ dient der Approximation der theoretischen Verteilung von $\hat{\theta}$. Die Beziehung zwischen resample und Basisstichprobe wird also auf die Beziehung zwischen Basisstichprobe und Population übertragen. Im Vergleich zu klassischen nonparametrischen Tests (Kap. 10) versprechen Resampling-Methoden oft eine höhere Power.

11.1 Nonparametrisches Bootstrapping

Bei Bootstrap-Verfahren (Chihara & Hesterberg, 2022; Davison & Hinkley, 1997) werden die resamples als *Replikationen* bezeichnet. Beim nonparametrischen bootstrap sind dies mit Zurücklegen gezogene Zufallsstichproben aus der Basisstichprobe mit dem Stichprobenumfang. $\hat{\theta}$ ist der auf Grundlage der Basisstichprobe berechnete *plug-in*-Schätzer von θ , wird also auf empirischer Ebene rechnerisch genauso gebildet wie θ auf theoretischer Ebene.¹ Über den – oft weniger interessanten – Bootstrap-Punktschätzer für θ hinaus lassen sich aus der empirischen Verteilung von $\hat{\theta}^*$ vor allem Konfidenzintervalle für θ bestimmen. Dafür approximiert die empirische Verteilung von $\hat{\theta}^* - \hat{\theta}$ in vielen Fällen jene der Pivot-Statistik $\hat{\theta} - \theta$, deren Verteilung unabhängig vom konkreten Wert für θ ist.

Ein vollständiger nonparametrischer bootstrap umfasst alle möglichen Replikationen einer Basisstichprobe vom Umfang n , wofür der Rechenaufwand jedoch meist zu groß ist: Es gibt bereits $2^n - 1$ relevante Teilmengen von Beobachtungsobjekten (die Mächtigkeit der Potenzmenge ohne die leere Menge), wobei die Elemente jeder denkbaren Zusammensetzung noch

¹ θ ist ein *Funktional* der theoretischen Verteilungsfunktion F der ursprünglichen Zufallsvariable, bildet also F auf eine Zahl ab. Analog ist $\hat{\theta}$ dasselbe Funktional der empirischen kumulativen Häufigkeitsverteilung \hat{F}_n der Basisstichprobe vom Umfang n und $\hat{\theta}^*$ dasselbe Funktional der empirischen kumulativen Häufigkeitsverteilung \hat{F}_n^* in einer Replikation.

mit unterschiedlichen Häufigkeiten berücksichtigt werden können. Deswegen wird bootstrapping als *Monte-Carlo-Verfahren* durchgeführt und nur eine zufällige Auswahl von Replikationen berücksichtigt.

11.1.1 Replikationen erstellen

Das Paket `boot` stellt mit `boot()` eine Funktion bereit, die Bootstrap-Replikationen durchführt.²

```
boot(data=<Basissstichprobe>, statistic=<Funktion>,
      R=<Anzahl Replikationen>, strata=<Faktor>)
```

Für `data` ist der Vektor mit den Werten der Basissstichprobe zu übergeben. Basiert $\hat{\theta}$ auf Daten mehrerer Variablen, muss `data` ein Datensatz mit diesen Variablen sein. Das Argument `statistic` erwartet den Namen einer Funktion mit ihrerseits zwei Argumenten zur Berechnung von $\hat{\theta}^*$: Ihr erstes Argument ist ebenfalls der Vektor bzw. Datensatz der Basissstichprobe. Das zweite Argument der für `statistic` genannten Funktion ist ein numerischer Indexvektor, dessen Elemente sich auf die Beobachtungsobjekte beziehen. `boot()` ruft für jede der `R` vielen Replikationen `statistic` auf und übergibt die Basisdaten samt eines zufällig gewählten Indexvektors als Anweisung, wie eine konkrete Replikation aus Fällen der Basissstichprobe zusammengesetzt sein soll.

Das Ergebnis der für `statistic` genannten Funktion muss $\hat{\theta}^*$ sein – sind gleichzeitig mehrere Parameter θ_j zu schätzen, analog ein Vektor mit den $\hat{\theta}_j^*$. Um beim bootstrapping eine vorgegebene Stratifizierung der Stichprobe beizubehalten, kann ein Faktor an `strata` übergeben werden, der eine Gruppeneinteilung definiert. In den Replikationen sind die Gruppengrößen dann gleich jenen der Basissstichprobe.

Als Beispiel diene die Situation eines *t*-Tests für eine Stichprobe auf einen festen Erwartungswert μ_0 (Abschn. 7.3.1). Ziel im folgenden Abschnitt ist die Konstruktion eines Vertrauensintervalls für $\mu (= \theta_1)$. Die Stichprobengröße sei n , der Mittelwert $M (= \hat{\theta}_1)$ und die unkorrigierte Varianz des Mittelwertes $S_M^2 (= \hat{\sigma}_{\hat{\theta}_1}^2 = \hat{\theta}_2)$ als plug-in-Schätzer der theoretischen Varianz $\sigma_M^2 (= \theta_2)$. Dazu ist aus jeder Bootstrap-Replikation der Mittelwert $M^* (= \hat{\theta}_1^*)$ und die unkorrigierte Varianz des Mittelwertes $S_M^{2*} (= \hat{\sigma}_{\hat{\theta}_1^*}^2 = \hat{\theta}_2^*)$ zu berechnen. Für das Erstellen eigener Funktionen s. Abschn. 17.3.

```
> N      <- 200                                # Größe Basissstichprobe
> muH0   <- 100                               # Erwartungswert unter H0
> sdH0   <- 40                                 # Streuung unter H0
> DV     <- rnorm(N, muH0, sdH0)               # Basissstichprobe

# Mittelwert M* & Varianz des Mittelwerts S(M)^2* aus BS-Replikation,
# deren zufällige Zusammensetzung durch Indexvektor idx bestimmt wird
> getM <- function(orgDV, idx) {
+   n      <- length(orgDV[idx])
```

²Das Paket `rsample` (Frick et al., 2024) liefert alternativ einen allgemeinen Ansatz für bootstrapping und Kreuzvalidierung (Abschn. 13.1) als Resampling-Verfahren.

```

+   bsM    <- mean(orgDV[idx])                      # M*
+   bsS2M <- (((n-1)/n) * var(orgDV[idx])) / n      # S(M)^2*
+   c(bsM, bsS2M)
}

> library(boot)                                     # für boot(), boot.ci()
> nR      <- 999                                    # Anzahl BS-Replikationen
> (bsRes <- boot(DV, statistic=getM, R=nR)) # Bootstrap-Replikationen
ORDINARY NONPARAMETRIC BOOTSTRAP                  # Ausgabe gekürzt ...
Bootstrap Statistics :
     original      bias    std. error
t1*  102.655867 -0.09535249    3.057508
t2*   9.162478  -0.02025724   0.881103

```

Die Ausgabe von `boot()` nennt in den Zeilen `t1*` und `t2*` Eigenschaften der von `statistic` zurückgegebenen Bootstrap-Schätzer: In der Spalte `original` stehen die für die Basisstichprobe berechneten Werte. Hier sind dies der Mittelwert und seine unkorrigierte Varianz. In der Spalte `bias` folgt die Verzerrung jedes Bootstrap-Schätzers als Mittelwert der Abweichungen $\hat{\theta}^* - \hat{\theta}$. Für die Punktschätzung von θ kann sie zu einer einfachen Bias-Korrektur eingesetzt werden, indem sie von $\hat{\theta}$ abgezogen wird. Schließlich folgt in der Spalte `std. error` die korrigierte Streuung für jeden Kennwert der pro Replikation von `statistic` berechneten Kennwerte. Diese sind in der von `boot()` zurückgegebenen Liste spaltenweise als Matrix in der Komponente `t` gespeichert.

```

> (M <- mean(DV))                                # Mittelwert Basisstichprobe
[1] 102.6559

> (S2M <- (((N-1)/N) * var(DV)) / N)        # unkorrigierte Varianz von M
[1] 9.162478

> Mstar    <- bsRes$t[, 1]                      # M* jeder Replikation
> S2Mstar <- bsRes$t[, 2]                      # S(M)^2* jeder Replikation
> (biasM <- mean(Mstar) - M)                   # Bias-Schätzung für M
[1] -0.0953525

> mean(S2Mstar) - S2M                          # Bias-Schätzung für S(M)^2
[1] -0.02025724

> c(sd(Mstar), sd(S2Mstar))                  # Streuungen der BS-Kennwerte
[1] 3.057508 0.881103

```

Die Verteilung von $\hat{\theta}^* - \hat{\theta}$ sollte unimodal und symmetrisch sein, nicht unähnlich einer Normalverteilung (Abb. 11.1). Dies lässt sich etwa durch ein Histogramm mit eingezeichneter Dichtefunktion einer passenden Normalverteilung zusammen mit einem nonparametrischen Kerndichteschätzer oder einem Q-Q-Plot prüfen.

```

> hist(Mstar-M, freq=FALSE, breaks="FD")  # Histogramm von M* - M
> rug(jitter(Mstar-M))                 # Einzelwerte der M* - M

```

```
# Dichtefunktion einer Normalverteilung
> curve(dnorm(x, mean(Mstar-M), sd(Mstar-M)),
+         lwd=2, col="blue", add=TRUE)

> lines(density(Mstar-M), lwd=2, col="red", lty=2) # Kerndichteschätzer
> qqnorm(Mstar-M, pch=16)                         # Q-Q-Plot der M* - M
> qqline(Mstar-M, lwd=2, col="blue")                # Normalverteilung-Referenz
```

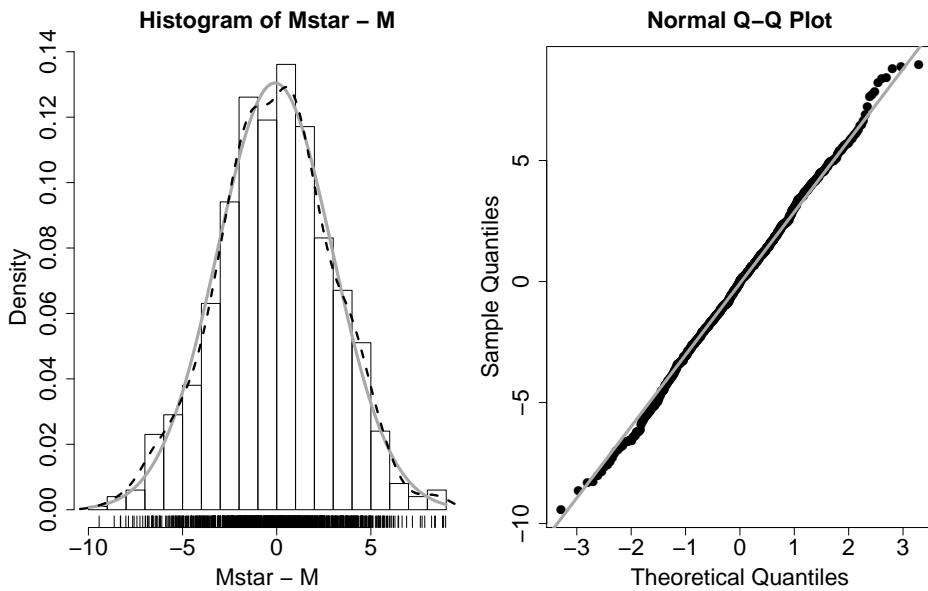


Abbildung 11.1: Histogramm von $M^* - M$ aus Bootstrap-Replikationen mit passender Normalverteilung und nonparametrischem Kerndichteschätzer. Q-Q-Plot von $M^* - M$ mit Vergleich zur Standardnormalverteilung.

Detaillierte Informationen über die Zusammensetzung aller von `boot()` erstellten Replikationen liefert `boot.array(<boot-Objekt>, indices=TRUE)`. Bei einer Basisstichprobe vom Umfang n und R Replikationen ist das Ergebnis mit dem Argument `indices=TRUE` eine $(R \times n)$ -Matrix mit einer Zeile für jede Replikation und einer Spalte pro Beobachtung. Die Zelle (r, i) enthält den Index des ausgewählten Elements der Basisstichprobe. Zusammen mit dem Vektor der Basisstichprobe lassen sich damit alle Replikation rekonstruieren.

```
# Replikationen 1-3: je ausgewählte erste 10 Indizes Basisstichprobe
> bootIdx <- boot.array(bsRes, indices=TRUE)
> bootIdx[1:3, 1:10]
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    78   148   83   72   86   78   33   59   87   141
[2,]    41   116   197   87   183    4   102   129   84   175
[3,]   123   107    60   181   35   133   196   197    15    71

> repl1Idx <- bootIdx[1, ]           # Indizes der ersten Replikation
> repl1DV <- DV[repl1Idx]          # Werte der ersten Replikation
```

```
> head(repl1DV, n=5)
[1] 86.12395 72.41568 135.87073 149.96011 99.03936
```

11.1.2 Bootstrap-Vertrauensintervalle für μ

Ein von `boot()` erzeugtes Objekt ist für das Argument `boot.out` der Funktion `boot.ci()` anzugeben, die das zweiseitige Konfidenzintervall für θ bestimmt.

```
boot.ci(boot.out=<boot-Objekt>, conf=<Breite VI>, index=<Nummer>,
        type=<Methode>)"
```

Gibt die für `boot(..., statistic)` genannte Funktion bei jedem Aufruf J geschätzte Parameter zurück, ist `index` der Reihe nach auf $1, \dots, J$ zu setzen, um das zugehörige Konfidenzintervall zu erhalten. Das Intervall wird mit der Breite `conf` nach einer über `type` festzulegenden Methode gebildet:

- "basic": Das klassische Bootstrap-Intervall um $\hat{\theta}$, dessen Breite durch die $\frac{\alpha}{2}$ - und $1 - \frac{\alpha}{2}$ -Quantile der Werte von $\hat{\theta} - \hat{\theta}^*$ definiert ist.
- "perc": Das Perzentil-Intervall, dessen Grenzen die $\frac{\alpha}{2}$ - und $1 - \frac{\alpha}{2}$ -Quantile der Werte von $\hat{\theta}^*$ sind.
- "norm": Das Normalverteilungs-Intervall mit dem Zentrum in $\hat{\theta} - M(\hat{\theta}^* - \hat{\theta})$ (Bias-Korrektur), dessen Breite definiert ist durch die $\frac{\alpha}{2}$ - und $1 - \frac{\alpha}{2}$ -Quantile der Standardnormalverteilung multipliziert mit der Streuung von $\hat{\theta}^*$.
- "stud": Das t -Vertrauensintervall um $\hat{\theta}$, dessen Breite definiert ist durch die $\frac{\alpha}{2}$ - und $1 - \frac{\alpha}{2}$ -Quantile der Werte von $t^* = \frac{\hat{\theta}^* - \hat{\theta}}{S_{\hat{\theta}}^*}$ multipliziert mit der Streuung von $\hat{\theta}^*$. Voraussetzung ist, dass die Funktion `statistic` als zweites Element den plug-in Schätzer $S_{\hat{\theta}}^{2*}$ der theoretischen Varianz $\sigma_{\hat{\theta}}^2$ zurückliefert. Ist deren geschlossene Form unbekannt oder existiert nicht, lässt sich $S_{\hat{\theta}}^{2*}$ innerhalb jedes Aufrufs von `statistic` im ursprünglichen bootstrapping durch eine eigene (*nested*) Bootstrap-Schätzung ermitteln.
- "bca": Das BC_a -Intervall (*bias-corrected and accelerated*). Für symmetrische Verteilungen und Schätzer mit geringem bias ähnelt es dem Perzentil- und t -Intervall meist stark. Bei schiefen Verteilungen und größerem bias wird das BC_a -Intervall den anderen oft vorgezogen.

Bei den Intervallen gehen zur Erhöhung der Genauigkeit nicht die $\frac{\alpha}{2}$ - und $1 - \frac{\alpha}{2}$ -Quantile selbst von t^* bzw. von $\hat{\theta}^*$ ein, die mit `quantile(..., probs=c(<alpha>/2, 1 - <alpha>/2))` zu ermitteln wären: Bei n_R vielen Replikationen sind dies stattdessen die Elemente mit den Indizes $(n_R + 1) \cdot \frac{\alpha}{2}$ und $(n_R + 1) \cdot (1 - \frac{\alpha}{2})$ der sortierten Werte von t^* bzw. von $\hat{\theta}^*$. Ergibt sich kein ganzzahliges Ergebnis für die Indizes, interpoliert `boot.ci()` jeweils zwischen den angrenzenden Elementen.

```

> alpha <- 0.05                                # alpha-Niveau
> boot.ci(bsRes, conf=1-alpha,
+           type=c("basic", "perc", "norm", "stud", "bca"))
BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS # gekürzte Ausgabe ...
Based on 999 bootstrap replicates
Intervals :
Level      Normal          Basic       Studentized
95%   ( 96.8, 108.7 )   ( 97.0, 109.1 )   ( 96.7, 109.0 )

Level      Percentile        BCa
95%   ( 96.2, 108.3 )   ( 96.2, 108.3 )
Calculations and Intervals on Original Scale

```

Es folgt die manuelle Kontrolle, zunächst für die Indizes $(n_R + 1) \cdot \frac{\alpha}{2}$ und $(n_R + 1) \cdot (1 - \frac{\alpha}{2})$.

```

> (idx <- trunc((nR + 1) * c(alpha/2, 1 - alpha/2)))
[1] 25 975

> tStar <- (Mstar-M) / sqrt(S2Mstar)          # t*
> tCrit <- sort(tStar)[idx]                    # 2.5%, 97.5%-Quantil von t*
> zCrit <- qnorm(c(alpha/2, 1 - alpha/2))    # 2.5%, 97.5%-Quantil N(0,1)
> (ciBasic <- 2*M - sort(Mstar)[idx])         # klassisches VI
[1] 109.11192 96.97443

> (ciPerc <- sort(Mstar)[idx])                # Perzentil-VI
[1] 96.19981 108.33731

> (ciNorm <- M-biasM - zCrit*sd(Mstar))     # N(0, 1)-VI
[1] 108.74383 96.75861

> (ciT <- M - tCrit*sqrt(S2M))              # t-VI
[1] 108.99497 96.70034

```

Bei der manuellen Umsetzung der Bootstrap-Schätzungen sollen als Maß für deren Güte die kumulierten relativen Häufigkeiten von $t^* = \frac{M^*-M}{S_M^*}$ mit der Verteilungsfunktion von $t = \frac{M-\mu_0}{s/\sqrt{n}}$ verglichen werden (mit s als korrigierter Streuung). Im Fall n unabhängiger Realisierungen einer normalverteilten Variable ist dies die t_{n-1} Verteilung (Abb. 11.2).

```

# M*, S(M)* und t* aus Bootstrap-Replikationen
> res <- replicate(nR, getM(DV, sample(seq_along(DV), replace=TRUE)))
> Mstar <- res[1, ]                           # M*
> SMstar <- sqrt(res[2, ])                   # S(M)*
> tStar <- (Mstar-mean(DV)) / SMstar        # t*

# kumulierte relative Häufigkeiten von t*
> plot(tStar, ecdf(tStar)(tStar), col="gray60", pch=1,
+       xlab="t* bzw. t", ylab="P(T <= t)",
+       main="t*: Kumulierte relative Häufigkeiten")

```

```
# theoretische Verteilungsfunktion von t und Legende
> curve(pt(x, N-1), lwd=2, add=TRUE)
> legend(x="topleft", lty=c(NA, 1), pch=c(1, NA),
+         lwd=c(2, 2), col=c("gray60", "black"), legend=c("t*", "t"))
```

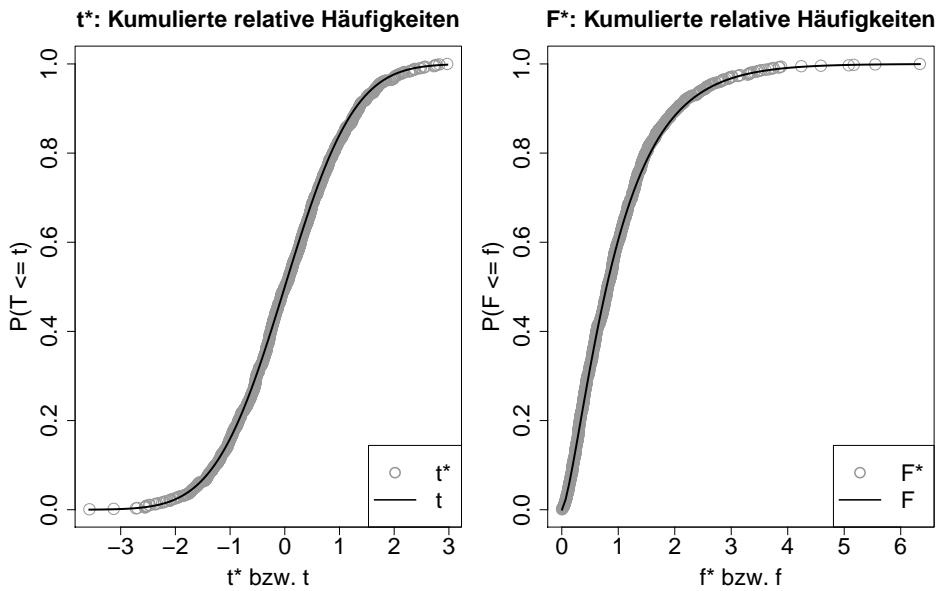


Abbildung 11.2: Kumulierte relative Häufigkeiten von t^* aus Bootstrap-Replikationen mit theoretischer Verteilungsfunktion t_{n-1} . Kumulierte relative Häufigkeiten von F^* aus Bootstrap-Replikationen mit theoretischer Verteilungsfunktion $F_{p-1, n-p}$.

11.1.3 Bootstrap-Vertrauensintervalle für $\mu_2 - \mu_1$

In der Situation eines t -Tests für zwei unabhängige Stichproben (Abschn. 7.3.2) kann bootstrapping eine nonparametrische Schätzung des Konfidenzintervalls für die Differenz der Erwartungswerte $\theta = \mu_2 - \mu_1$ liefern. In der Basisstichprobe ist die Differenz der Gruppenmittelwerte $\hat{\theta} = M_2 - M_1$ ein Schätzer für θ , in jeder Replikation analog $\hat{\theta}^* = M_2^* - M_1^*$. Für die Replikationen ist zu beachten, dass jeweils innerhalb jeder Gruppe mit Zurücklegen aus der Basisstichprobe gezogen wird, damit Gruppenzugehörigkeit und Gruppengrößen erhalten bleiben. Dies lässt sich mit dem Argument `strata` von `boot()` erreichen. Als Beispiel diene jenes aus Abschn. 7.3.2 mit einer bei Frauen und Männern erhobenen Variable.

```
# Datensatz aus Variable bei Männern, Frauen und Gruppierungsfaktor
> n1  <- 18                                # Stichprobenumfang 1
> n2  <- 21                                # Stichprobenumfang 2
> DVm <- rnorm(n1, 180, 10)                 # Stichprobe Männer
> DVf <- rnorm(n2, 175, 6)                  # Stichprobe Frauen
> tDf <- stack(list(m=DVm, f=DVf))        # Gesamtdaten, erste Stufe m
```

```
# Funktion, um Differenz der Mittelwerte von f und m zu berechnen
> getDM <- function(dat, idx) {
+   # Gruppenmittelwerte für durch idx definierte Beobachtungen
+   Mfm <- aggregate(values ~ ind, data=dat, subset=idx, FUN=mean)
+   -diff(Mfm$values) # M-Differenz, Reihenfolge m-f wie in t.test()
+ }

> library(boot)                      # für boot(), boot.ci()
> bsTind <- boot(tDf, statistic=getDM, strata=tDf$ind, R=999)
> boot.ci(bsTind, conf=0.95, type=c("basic", "bca"))
BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS  # gekürzte Ausgabe ...
Based on 999 bootstrap replicates
Intervals :
Level      Basic          BCa
95%  (-0.810,  7.709)  ( 0.096,  8.725)
Calculations and Intervals on Original Scale
```

Als Vergleich diene das parametrische Konfidenzintervall aus dem zugehörigen *t*-Test.

```
> tt <- t.test(values ~ ind, alt="two.sided", var.equal=TRUE, data=tDf)
> tt$conf.int
[1] -0.8963473  8.3796152
```

Die analoge Situation mit zwei abhängigen Stichproben lässt sich auf den Fall einer Stichprobe zurückführen, indem eine Differenzvariable aus der jeweils pro Person berechneten Differenz beider Beobachtungen gebildet wird (Abschn. 7.3.3).

11.1.4 Lineare Modelle: case resampling

Die in Abschn. 6.2.2 und 6.3 vorgestellten Tests der Parameter einer linearen Regression setzen die Gültigkeit verschiedener Annahmen voraus, deren Plausibilität mit Methoden untersucht werden kann, wie sie Abschn. 6.5 erläutert. Sind diese Annahmen verletzt, sind die berechneten Standardfehler der Parameterschätzer womöglich verzerrt und führen zu falschen *p*-Werten. Oft eignen sich in diesem Fall Bootstrap-Verfahren, um angemessenere Vertrauensintervalle für die Regressionsgewichte zu erhalten.

Als Beispiel sei auf die Daten der multiplen linearen Regression in Abschn. 6.3.1 zurückgegriffen, die das Körpergewicht anhand der Prädiktoren Körpergröße, Alter und der für Sport aufgewendeten Zeit vorhersagen soll. Die Daten wurden unter gültigen Modellannahmen simuliert, was es hier erlaubt, die korrekten Standardfehler und Konfidenzintervalle der parametrischen Regressionsanalyse zur Validierung der Bootstrap-Ergebnisse zu verwenden.

```
> sqrt(diag(vcov(fitHAS)))      # parametrische Standardfehler
(Intercept)      height       age       sport
 8.18162620  0.04590320  0.04033012  0.01201414

> confint(fitHAS)              # parametrische Konfidenzintervalle
                           2.5 %    97.5 %

```

| | | |
|-------------|------------|------------|
| (Intercept) | -8.1391666 | 24.3416327 |
| height | 0.4237764 | 0.6060106 |
| age | -0.3667003 | -0.2065910 |
| sport | -0.4398003 | -0.3921046 |

Die erste Methode, bootstrapping auf die Situation eines linearen Modells wie das der Regression anzuwenden, besteht im *case resampling*, auch *Vektor-Sampling* genannt: Hierfür werden die beobachteten Werte aller Variablen als zufällig betrachtet, insbesondere auch jene der Prädiktoren. Für jede Replikation wird nun aus der Menge der beobachteten Personen (*cases*) mit Zurücklegen eine Stichprobe vom ursprünglichen Umfang gezogen. Die Werte der jeweils gezogenen Personen für Prädiktoren und Kriterium liegen der Anpassung des Regressionsmodells für eine Replikation zugrunde. So liefert jede Replikation eine Schätzung für jeden Regressionsparameter, woraus sich deren Bootstrap-Verteilungen – und damit Standardfehler und Konfidenzintervalle bestimmen lassen.

Die Methode gilt als robust, da sich mit jeder Replikation die Design-Matrix des Modells ändert und daher Ausreißer oder übermäßig einflussreiche Beobachtungen die Parameterschätzungen nicht immer verzerren. Case resampling ist auch für verallgemeinerte lineare Modelle geeignet (Kap. 8).

Die im Aufruf von `boot()` für das Argument `statistic` genannte Funktion muss unter `data` einen Datensatz mit den ursprünglichen Werten von Prädiktoren und Kriterium akzeptieren sowie die Parameterschätzungen für die über den Indexvektor `idx` definierte Replikation zurückgeben.

```
# berechne für Daten dat und Replikation idx die Regressionsgewichte
> getRegr <- function(dat, idx) {
+   bsFit <- lm(weight ~ height + age + sport, subset=idx, data=dat)
+   coef(bsFit)                                # Regressionsgewichte Replikation
+ }

> library(boot)                               # für boot(), boot.ci()
> nR <- 999                                    # Anzahl Bootstrap-Replikationen
> (bsRegr <- boot(regrDf, statistic=getRegr, R=nR))
ORDINARY NONPARAMETRIC BOOTSTRAP      # gekürzte Ausgabe ...
Bootstrap Statistics :
      original        bias    std. error
t1*  8.1012331 -0.1025429121  7.86996801
t2*  0.5148935  0.0001543189  0.04557696
t3* -0.2866457  0.0017596402  0.04037238
t4* -0.4159525  0.0004067643  0.01125020
```

Die Zeilen der Ausgabe nennen die Bootstrap-Kennwerte für jeweils einen Koeffizienten in der Reihenfolge des Rückgabewertes der im Aufruf von `boot()` für `statistic` verwendeten Funktion. Für die gewählten Daten stimmen die Bootstrap-Standardfehler weitgehend mit jenen der parametrischen Regressionsanalyse überein. Das Vertrauensintervall für jeden Parameter liefert wieder `boot.ci()`, wobei mit dem Argument `index=⟨Nummer⟩` auszuwählen ist, für welchen Parameter θ_j das Vertrauensintervall benötigt wird.

```
# BCa-Konfidenzintervalle für die Regressionsparameter
> boot.ci(bsRegr, conf=0.95, type="bca", index=1)$bca      # b0
[1,] 0.95 22.45 972.23 -6.975915 24.44877

> boot.ci(bsRegr, conf=0.95, type="bca", index=2)$bca      # height
[1,] 0.95 27.62 977.4 0.4238508 0.5991131

> boot.ci(bsRegr, conf=0.95, type="bca", index=3)$bca      # age
[1,] 0.95 26.18 976.23 -0.3637959 -0.2059037

> boot.ci(bsRegr, conf=0.95, type="bca", index=4)$bca      # sport
[1,] 0.95 22.45 972.22 -0.4382711 -0.3939868
```

Das erste Element des Vektors in der Komponente `bca` der von `boot.ci()` zurückgegebenen Liste nennt die Breite des Intervalls, die beiden folgenden Elemente die zu den $\frac{\alpha}{2}$ - und $1 - \frac{\alpha}{2}$ -Quantilen gehörenden Indizes für den Vektor der sortierten Werte,³ die letzten beiden Elemente sind schließlich die gesuchten Intervallgrenzen. Auch die Konfidenzintervalle gleichen hier jenen aus der parametrischen Regressionsanalyse.

11.1.5 Lineare Modelle: model-based resampling

Varianzanalysen lassen sich wie eine Regression als lineares Modell formulieren, wobei die Rolle der Prädiktoren durch die Gruppenzugehörigkeiten eingenommen wird (Abschn. 12.9). Während die Werte der Prädiktoren in einer Regression nur passiv beobachtet sind, werden Gruppenzugehörigkeiten in einem varianzanalytischen Design experimentell festgelegt. Die Gruppenzugehörigkeiten sollten deshalb für alle resamples konstant gehalten werden. Eine Methode, um dies zu gewährleisten, besteht im *model-based resampling*. Hier werden nur die Werte der Zielgröße als mit zufälligen Fehlern behaftet betrachtet. Dieser Logik folgend berechnet man zunächst für die Basisstichprobe die Modellvorhersage \hat{Y} sowie die zugehörigen Residuen $E = Y - \hat{Y}$.

Da die Residuen überall die bedingte theoretische Streuung σ besitzen, werden hier die standardisierten Residuen verwendet (Abschn. 6.5.2). Um zu gewährleisten, dass E im Mittel 0 ist, sollte das Modell einen absoluten Term β_0 einschließen oder zentrierte Variablen verwenden.

Für jede Replikation wird aus $\frac{E}{\sqrt{1-h}}$ mit Zurücklegen eine Stichprobe E^* vom ursprünglichen Umfang gezogen. Diese Resample-Residuen werden zu \hat{Y} addiert, um die Resample-Zielgröße $Y^* = \hat{Y} + E^*$ zu erhalten. Für jede Bootstrap-Schätzung der Parameter wird dann die Varianzanalyse mit Y^* und dem festen Gruppierungsfaktor der Basisstichprobe berechnet.

Residuen E und Modellvorhersage \hat{Y} für die Basisstichprobe lassen sich sowohl für das H_0 - wie für das H_1 -Modell bilden: Unter der H_0 der einfaktoriellen Varianzanalyse unterscheiden sich die Erwartungswerte in den Gruppen nicht, als Vorhersage \hat{Y} ergibt sich damit für alle Personen der Gesamtmittelwert M . Wird dieses Modell mit der Formel $\langle AV \rangle \sim 1$ für die Basisstichprobe

³Die Indizes sind hier trotz der 999 Replikationen nicht ganzzahlig (25 und 975), da die dem BC_a -Intervall zugrundeliegende Korrektur über die Verschiebung der Intervallgrenzen funktioniert. Vergleiche etwa das Perzentil-Intervall für θ_1 aus `boot.ci(bsRegr, conf=0.95, type="perc", index=1)$percent`.

angepasst, erzeugt das bootstrapping eine Approximation der Verteilung von $\hat{\theta}$ (etwa des F -Bruchs) unter H_0 . Dies erlaubt es, p -Werte direkt als Anteil der resamples zu berechnen, bei denen $\hat{\theta}^*$ i.S. der H_1 mindestens so extrem wie der Wert von $\hat{\theta}$ in der Basisstichprobe ist.⁴

Als Beispiel sei auf die Daten der einfaktoriellen Varianzanalyse in Abschn. 7.4.1 zurückgegriffen. Die Daten wurden unter gültigen Modellannahmen simuliert, was es hier erlaubt, die Verteilung der Bootstrap-Schätzer F^* mit der F -Verteilung zu vergleichen und den Bootstrap- p -Wert mit dem p -Wert der F -Verteilung zu validieren.

```
> anBase <- anova(lm(DV ~ IV))           # ANOVA Basisstichprobe
> Fbase  <- anBase["IV", "F value"]       # F-Wert Basisstichprobe
> (pBase <- anBase["IV", "Pr(>F)"])      # p-Wert Basisstichprobe
[1] 0.002921932

# H0-Modell in Basisstichprobe anpassen
> fit0 <- lm(DV ~ 1)                      # Modellanpassung
> E    <- rstandard(fit0)                  # standardisierte Residuen
> Yhat <- fitted(fit0)                    # ursprüngliche Vorhersage

# ANOVA-Parameter für ursprüngliche Gruppenzugehörigkeiten und Y*
> getAnova <- function(dat, idx) {
+   Ystar <- Yhat + E[idx]                  # Resample-AV Y* = Y^ + E*
+   anBS  <- anova(lm(Ystar ~ IV, data=dat)) # ANOVA Resample-AV
+   anBS["IV", "F value"]                   # F*-Wert Replikation
+ }

> library(boot)                           # für boot(), boot.ci()
> nR      <- 999                         # Anzahl Replikationen
> (bsAnova <- boot(dfCRp, statistic=getAnova, R=nR))
ORDINARY NONPARAMETRIC BOOTSTRAP          # gekürzte Ausgabe ...
Bootstrap Statistics :
      original      bias    std. error
t1*  4.861867 -3.887228   0.8136521
```

Der für den p -Wert notwendige Vergleich $F^* \geq F$ wird hier in zwei Vergleiche aufgeteilt, um robuster gegenüber Problemen der numerischen Repräsentation von Gleitkommazahlen zu sein (Abschn. 1.4.6). Dafür wird nur auf ungefähre, nicht auf exakte Gleichheit von F^* und F geprüft.

```
> Fstar <- bsAnova$t                 # F* aus Bootstrap-Replikationen
> tol   <- .Machine$double.eps^0.5  # numerische Toleranz (Gleitkomma)

# F* größer Basis-F ODER F* ungefähr gleich Basis-F?
> FsIsGEQ <- (Fstar > Fbase) | (abs(Fstar - Fbase) < tol)
```

⁴Der p -Wert kann bei Monte-Carlo-Approximationen zur höheren Genauigkeit nach Hinzufügen eines zusätzlichen extremeren Falles gebildet werden: Ist n_R die Anzahl der generierten resamples und n^* die Anzahl der Fälle, bei denen $\hat{\theta}^*$ mindestens so extrem wie $\hat{\theta}$ ist, setzt man $p = \frac{n^*+1}{n_R+1}$. Auf diese Weise wird vermieden, dass der p -Wert exakt 0 werden kann.

```
# p-Wert: Anteil der mindestens so extremen F* wie Basis-F
> (pValBS <- (sum(FsIsGEQ) + 1) / (length(Fstar) + 1))
[1] 0.004

# Grafik: kumulierte relative Häufigkeiten von F*
> plot(Fstar, ecdf(Fstar)(Fstar), col="gray60", pch=1,
+       xlab="f* bzw. f", ylab="P(F <= f)",
+       main="F*: Kumulierte relative Häufigkeiten")

# theoretische Verteilungsfunktion von t und Legende
> curve(pf(x, P-1, sum(Nj) - P), lwd=2, add=TRUE)
> legend(x="topleft", lty=c(NA, 1), pch=c(1, NA), lwd=c(2, 2),
+         col=c("gray60", "black"), legend=c("F*", "F"))
```

Die Verteilung der F^* ist hier der theoretischen F -Verteilung sehr ähnlich (Abb. 11.2), was auch für die Größenordnung des Bootstrap- p -Wertes verglichen mit dem p -Wert der ursprünglichen Varianzanalyse gilt.

11.1.6 Lineare Modelle: wild bootstrap

Eine Variante des model-based resampling ist der *wild bootstrap* für Situationen, in denen Heteroskedastizität vorliegt. Hier wird E^* aus dem Produkt $\frac{E}{\sqrt{1-h}} \cdot U$ gebildet. Dabei ist U eine unabhängige Zufallsvariable mit $E(U) = 0$ und $E(U^2) = 1$, deren Werte für jede Replikation simuliert werden müssen.

- Eine Wahl für U sind dichotome Variablen mit F_1 -Verteilung, die den Wert $\frac{-(\sqrt{5}-1)}{2}$ mit Wahrscheinlichkeit $p = \frac{\sqrt{5}+1}{2\sqrt{5}}$ annehmen und den Wert $\frac{\sqrt{5}+1}{2}$ mit Wahrscheinlichkeit $1-p = \frac{\sqrt{5}-1}{2\sqrt{5}}$.
- Eine alternative Wahl für U sind ebenfalls dichotome Variablen mit F_2 (Rademacher) Verteilung: Sie nehmen die Werte -1 und 1 jeweils mit Wahrscheinlichkeit $\frac{1}{2}$ an, drehen also das ursprüngliche Vorzeichen jedes Residuums zufällig um.

```
> getAnovaWild <- function(dat, idx) {
+   n <- length(idx)                      # Größe der Replikation
+   # Ur: Rademacher Variablen, Uf: zweite Variante
+   Ur <- sample(c(-1, 1), size=n, replace=TRUE, prob=c(0.5, 0.5))
+   Uf <- sample(c(-(sqrt(5) - 1)/2, (sqrt(5) + 1)/2), size=n,
+                 replace=TRUE, prob=c((sqrt(5) + 1)/(2*sqrt(5)),
+                                         (sqrt(5) - 1)/(2*sqrt(5))))
+
+   Ystar <- Yhat + (Er*Ur)[idx]          # E* mit Rademacher-Variable
+   # Ystar <- Yhat + (Er*Uf)[idx]          # zweite Variante
+   anBS <- anova(lm(Ystar ~ IV, data=dat))    # ANOVA Resample
+   anBS["IV", "F value"]                  # F*-Wert Replikation
}
```

```
# fortfahren wie oben
> bsAnovaW <- boot(dfCRp, statistic=getAnovaWild, R=nR)
> FstarW   <- bsAnova$t                                     # F*-Werte
> tol       <- .Machine$double.eps^0.5                      # numerische Toleranz

# Fstar größer Basis-F ODER Fstar ungefähr gleich Basis-F?
> FsIsGEQ  <- (FstarW > Fbase) | (abs(FstarW - Fbase) < tol)
> (pValBSw <- (sum(FsIsGEQ) + 1) / (length(FstarW) + 1))    # ...
```

Vorhersage und Residuen für die Basisstichprobe können auch nicht wie im Beispiel aus dem H_0 Modell, sondern aus dem H_1 Modell mit der Formel $\langle AV \rangle \sim \langle UV \rangle$ berechnet werden. Dann erzeugt das modellbasierte bootstrapping eine Verteilung von $\hat{\theta}^*$, aus der sich Vertrauensintervalle für θ bestimmen lassen (Abschn. 11.1.2, 11.1.4). Für die Bootstrap-Verteilung von F^* sollte dabei auf das BC_a -Intervall zurückgegriffen werden, da diese Verteilung typischerweise schief ist und der Schätzer einen deutlichen bias aufweist.

11.2 Parametrisches Bootstrapping

Parametrisches bootstrapping setzt ein statistisches Modell dafür voraus, wie Beobachtungen einer Variable Y aus der Kombination theoretischer Parameter θ_j und zufälliger Fehler zu stehen kommen sollen. Jede Replikation ergibt sich dann als neue modellgerechte Simulation eines Datensatzes: Dafür werden die plug-in Schätzer der Basisstichprobe $\hat{\theta}_j$ im Modell anstelle der θ_j eingesetzt und die Fehler zufällig entsprechend der für sie angenommenen Verteilung simuliert. Das weitere Vorgehen ist identisch zum nonparametrischen bootstrap: Für jede Replikation schätzt man zunächst die Parameter $\hat{\theta}_j^*$. Aus deren empirischer Verteilung über alle Replikationen hinweg lassen sich dann Konfidenzintervalle für die θ_j bestimmen.

Auch für den parametrischen bootstrap eignet sich die Funktion `boot()` aus dem gleichnamigen Paket (Abschn. 11.1.1), wenn das Argument `sim="parametric"` gesetzt wird.

```
boot(data=<Basisstichprobe>, statistic=<Funktion>,
      R=<Anzahl Replikationen>, sim="parametric", ran.gen=<Funktion>,
      mle=<Schätzer Basisstichprobe>)
```

Für `data` sind die Daten der Basisstichprobe als Vektor oder Datensatz zu übergeben. Für `statistic` ist eine Funktion mit ihrerseits einem Argument zu nennen – den Daten der Replikation, auf deren Basis sich die Schätzungen $\hat{\theta}^*$ berechnen lassen. `boot()` ruft für jede der `R` vielen Replikationen `statistic` auf und übergibt einen neu simulierten Datensatz mit derselben Form wie jener der Basisstichprobe. Das Ergebnis der für `statistic` genannten Funktion muss ein Vektor der $\hat{\theta}_j^*$ sein.

Die Simulation eines neuen Datensatzes erfolgt in einer eigenen Funktion, die an `ran.gen` zu übergeben ist. Diese muss ihrerseits zwei Argumente besitzen: Das erste für die Daten der Basisstichprobe und das zweite für deren Schätzer $\hat{\theta}_j$. Diese Schätzer sind zunächst für das Argument `mle` (*maximum likelihood estimate*) von `boot()` zu nennen. Allgemein kann `mle` auch ein komplexeres Objekt sein, aus dem sich innerhalb von `ran.gen` die $\hat{\theta}_j$ ableiten lassen,

etwa eine von `lm()` oder `glm()` für die Basisstichprobe angepasste Regression. Das Ergebnis von `ran.gen` ist ein simulierter Datensatz mit derselben Form wie jener der Basisstichprobe, also mit denselben Variablennamen desselben Typs. Er wird pro Replikation an die für `statistic` genannte Funktion übergeben.

11.2.1 Bootstrap-Vertrauensintervalle für $\mu_2 - \mu_1$

Als Beispiel sollen in der Situation eines t -Tests für zwei unabhängige Stichproben (Abschn. 7.3.2, 11.1.3) Konfidenzintervalle für die Differenz der Erwartungswerte $\theta = \mu_2 - \mu_1$ konstruiert werden, wenn nicht von Varianzhomogenität auszugehen ist.

```
# Datensatz aus Variable bei Männern, bei Frauen und Gruppierungsfaktor
> n1 <- 18                                # Stichprobenumfang 1
> n2 <- 21                                # Stichprobenumfang 2
> DVm <- rnorm(n1, 180, 10)                 # Stichprobe Männer
> DVf <- rnorm(n2, 175, 6)                  # Stichprobe Frauen
> tDf <- stack(list(m=DVm, f=DVf))        # Gesamt-Daten, erste Stufe m
```

Für die Simulation neuer Daten sei angenommen, dass sich in jeder Gruppe j die Messwerte y_{ij} als Summe $\mu_j + \epsilon_{ij}$ des Erwartungswerts μ_j und eines normalverteilten Fehlers ϵ_{ij} mit Erwartungswert 0 und gruppenspezifischer Varianz σ_j^2 ergeben. Die Simulations-Funktion nimmt dafür einerseits den Basisdatensatz entgegen, andererseits eine Liste mit zwei Komponenten: In der ersten Komponente wurden alle ursprünglichen Werte y_{ij} durch ihren Gruppenmittelwert M_j ersetzt – dem Schätzer für μ_j in der Basisstichprobe. Analog wurden in der zweiten Komponente alle y_{ij} durch ihre unkorrigierte Gruppenstreuung S_j ersetzt, da die unkorrigierte Gruppenvarianz S_j^2 der plug-in und gleichzeitig maximum likelihood Schätzer in der Basisstichprobe für die gruppenspezifische Varianz σ_j^2 ist.

```
# Funktion für unkorrigierte Varianz S^2 eines Vektors x
> getSDML <- function(x) {
+   sqrt(cov.wt(as.matrix(x), method="ML")$cov[1, 1])
+ }

# ersetze Daten der Basisstichprobe jeweils durch ihren
# Gruppenmittelwert bzw. durch ihre unkorrigierte Gruppenstreuung
> MSD <- list( M=ave(tDf$values, tDf$ind, FUN=mean),
+               SD=ave(tDf$values, tDf$ind, FUN=getSDML))

# Simulation: Argument dat für Daten der Basisstichprobe
# Argument MSD für Liste mit Gruppen-M und -SD der Basisstichprobe
> rGenMD <- function(dat, MSD) {
+   out <- dat
+   # simuliere modellgerecht neue Messwerte
+   out$values <- MSD$M + rnorm(length(MSD$M), mean=0, sd=MSD$SD)
+   return(out)
+ }
```

Die Verteilung der Differenz der Gruppenmittelwerte $M_2^* - M_1^*$ aus jeder Replikation dient schließlich als Grundlage der Konfidenzintervalle für die Differenz der Erwartungswerte $\theta = \mu_2 - \mu_1$.

```
# Funktion für Differenz der Gruppenmittelwerte in einer Replikation
> getMD <- function(dat) {
+   Mfm <- aggregate(values ~ ind, data=dat, FUN=mean)
+   -diff(Mfm$values)
+ }

# parametrischer bootstrap
> library(boot)                                # für boot(), boot.ci()
> nR    <- 999                                  # Anzahl Replikationen
> bsMD <- boot(dat, statistic=getMD, R=nR,
+               sim="parametric", mle=MSD, ran.gen=rGenMD)
```

Das von `boot()` erzeugte Objekt ist das erste Argument für die Funktion `boot.ci()`, die das zweiseitige Konfidenzintervall für θ bestimmt (Abschn. 11.1.2).

```
> boot.ci(bsMD, conf=0.95, type="basic")$basic
[1,] 0.95 975 25 -0.6817411 8.388329
```

Als Vergleich diene das parametrische Konfidenzintervall aus dem zugehörigen *t*-Test ohne Annahme von Varianzhomogenität.

```
> tt <- t.test(values ~ ind, alt="two.sided", var.equal=FALSE, data=tDf)
> tt$conf.int
[1] -1.165688 8.648956
```

11.2.2 Verallgemeinerte lineare Modelle

Für die Anwendung des parametrischen bootstrap bei einem verallgemeinerten linearen Modell (Kap. 8) soll die Poisson-Regression mit den Prädiktoren X_1 und X_2 aus Abschn. 8.4.1 als Beispiel dienen. Für (verallgemeinerte) lineare Modelle ist es besonders einfach, auf Basis einer bereits mit `lm()` oder `glm()` angepassten Regression modellgerecht neue Werte der vorhergesagten Variable zu simulieren, da R hierfür die Funktion `simulate()` bereitstellt. Sie akzeptiert ein von `lm()` oder `glm()` erzeugtes Objekt und liefert in der ersten Komponente der zurückgegebenen Liste die neu simulierten Werte der vorhergesagten Variable. Dafür leitet `simulate()` das statistische Modell mit Schätzern $\hat{\theta}_j$ der Basisstichprobe sowie die angenommene Verteilung der Fehler aus dem übergebenen Objekt selbst ab.

```
# Simulation: Argument dat erhält Daten der Basisstichprobe
# Argument mle erhält Ergebnis von glm() für Basisstichprobe
> rGen <- function(dat, mle) {
+   out   <- dat
+   out$Y <- simulate(mle)[[1]] # 1. Komponente: simulierte Zielgröße
+   return(out)
+ }
```

In jeder Replikation ist eine Poisson-Regression für die übergebenen simulierten Daten anzupassen, aus der sich dann die Regressionskoeffizienten b_j^* als Schätzer $\hat{\theta}_j^*$ ergeben.

```
> getPois <- function(dat) {
+   glmFit <- glm(Y ~ X1 + X2, family=poisson(link="log"), data=dat)
+   coef(glmFit)
+ }
```

`glmFitP` aus Abschn. 8.4.1 speichert das Ergebnis der mit `glm()` für die Basisstichprobe angepassten Poisson-Regression, die hier als Basis für den parametrischen bootstrap dient.

```
> library(boot)                      # für boot(), boot.ci()
> nR <- 999                          # Anzahl Replikationen
> bsPois <- boot(dfCount, statistic=getPois, R=nR,
+                 sim="parametric", mle=glmFitP, ran.gen=rGen)
```

Das von `boot()` erzeugte Objekt ist als erstes Argument für die Funktion `boot.ci()` anzugeben, die das zweiseitige Konfidenzintervall für die θ_j bestimmt (Abschn. 11.1.2).

```
> boot.ci(bsPois, conf=0.95, type="basic", index=1)$basic
[1,] 0.95 975 25 -0.04471856 0.3480377
```

```
> boot.ci(bsPois, conf=0.95, type="basic", index=2)$basic
[1,] 0.95 975 25 -0.3028855 -0.220306
```

```
> boot.ci(bsPois, conf=0.95, type="basic", index=3)$basic
[1,] 0.95 975 25 -0.0004523571 0.04146108
```

Als Vergleich dienen die parametrischen Konfidenzintervalle mit der profile likelihood Methode für die in der Basisstichprobe angepasste Poisson-Regression:

```
> confint(glmFitP)
              2.5 %      97.5 %
(Intercept) -0.0534587597  0.33602778
X1          -0.3036802852 -0.22116851
X2          -0.0007271649  0.04155718
```

11.3 Permutationstests

Permutationstests (Chihara & Hesterberg, 2022; Good, 2004) verwenden analog zu Bootstrap-Verfahren Permutationen einer festen Basisstichprobe als resamples, um ausgehend von der empirischen Verteilung der für diese resamples berechneten Schätzer $\hat{\theta}^*$ Aussagen über die theoretische Verteilung einer Teststatistik $\hat{\theta}$ abzuleiten. Jede Permutation wird dabei so gebildet, dass sie dieselben Werte der Basisstichprobe umfasst, die Reihenfolge der Beobachtungen i. S. der Zusammensetzung der Untersuchungsbedingungen jedoch im Einklang mit der H_0 des Tests im gegebenen Untersuchungsdesign variiert.

Stimmen etwa unter H_0 die Verteilungen einer Zielgröße (AV) in zwei Bedingungen überein, ist die Zugehörigkeit der Beobachtungen zu einer Bedingung für die Ausprägung der AV unwe sentlich und kann deshalb permutiert werden – jede Beobachtung hätte genauso gut aus jeder Bedingung stammen können.⁵ Im Fall von abhängigen Stichproben ist dies separat innerhalb jedes Beobachtungsobjekts zu tun, bei unabhängigen Stichproben entsprechend über Beobachtungsobjekte hinweg. Sind zwei Variablen unter H_0 unabhängig, ist es für die Ausprägung der zweiten AV irrelevant, welchen Wert die erste AV besitzt – die Zuordnung von Werten der zweiten AV zu jenen der ersten kann also permutiert werden.

Die empirische Verteilung von $\hat{\theta}^*$ schätzt die Verteilung von $\hat{\theta}$ unter H_0 , was es erlaubt, p -Werte direkt zu berechnen: Dies ist der Anteil der Permutationen, bei denen $\hat{\theta}^*$ i. S. der H_1 mindestens so extrem wie $\hat{\theta}$ ist. Wählt man für $\hat{\theta}$ die Teststatistik des zugehörigen parametrischen Tests, wird dieses Vorgehen auch als Fisher-Pitman Test bezeichnet. Da die Anzahl möglicher Permutationen (und damit der Rechenaufwand) sehr schnell mit der Stichprobengröße wächst, ist es nur in Spezialfällen oder bei kleinen Stichproben möglich, die empirische Verteilung von $\hat{\theta}^*$ exakt zu bestimmen. Ist die praktische Berechenbarkeit aller $\hat{\theta}^*$ nicht gegeben, lassen sich Permutationstests als Monte-Carlo-Verfahren durchführen, die $\hat{\theta}^*$ nur für eine zufällige Auswahl von Permutationen berechnen (Abschn. 11.1.5, Fußnote 4).

Das bereits in Abschn. 10.5.8, 10.5.10 und 10.5.11 verwendete Paket `coin` stellt für viele Hypothesen exakte, oder aber durch Monte-Carlo-Verfahren approximierte Permutationstests bereit, für deren konventionelle Prüfung sonst auf parametrische Tests oder nonparametrische Verfahren mit einer asymptotisch gültigen Verteilung der Teststatistik zurückgegriffen werden muss – für eine Übersicht s. `vignette("coin")`.⁶

11.3.1 Test auf gleiche Lageparameter in unabhängigen Stichproben

Im Beispiel soll linksseitig getestet werden, ob die Verteilung einer quantitativen Zielgröße in (hier zwei) unabhängigen Stichproben übereinstimmt. Bei Verteilungen gleicher Form ist dies der Fall, wenn ihre Lageparameter identisch sind. Anders als etwa beim Wilcoxon-Rangsummen-Test (Abschn. 10.5.4) oder dem Kruskal-Wallis-Test (Abschn. 10.5.6) sollen hier im Sinne eines Fisher-Pitman Tests die ursprünglichen Werte der Zielgröße ohne Rangtransformation Verwendung finden. Die Umsetzung erfolgt mit `oneway_test()` aus dem Paket `coin`.⁷

```
oneway_test(formula=<Modellformel>, data=<Datensatz>,
            alternative=c("two.sided", "less", "greater"),
            distribution=<Verteilungstyp>)
```

Unter `formula` sind Daten und Gruppierungsvariable als Modellformel $\langle \text{AV} \rangle \sim \langle \text{UV} \rangle$ einzugeben, wobei $\langle \text{UV} \rangle$ ein Faktor derselben Länge wie $\langle \text{AV} \rangle$ ist und für jede Beobachtung der Zielgröße in $\langle \text{AV} \rangle$ die Gruppenzugehörigkeit angibt. Geschieht dies mit Variablen aus einem Datensatz,

⁵Formal muss das Kriterium der *Austauschbarkeit* erfüllt sein (Good, 2004).

⁶Auch das Paket `permute` (Simpson, 2019) bietet flexible Möglichkeiten, um Permutationstests für verschiedenen Untersuchungs-Designs umzusetzen.

⁷Eine weitere Alternative ist der van der Waerden-Test, für den die an $n + 1$ normierten Ränge durch die zugehörigen Quantile aus der Standardnormalverteilung ersetzt werden. Dieser Test lässt sich mit `normal_test()` aus dem Paket `coin` umsetzen.

muss dieser unter `data` eingetragen werden. Mit `alternative` wird festgelegt, ob die H_1 gerichtet ("less" bzw. "greater") oder ungerichtet ("two.sided") ist. Die Anzahl der zufälligen Permutationen, auf deren Basis die Verteilung der Teststatistik approximiert wird, lässt sich über `approximate(nresample=(Anzahl))` für das Argument `distribution` übergeben. Mit der in bestimmten Situationen wählbaren Option "exact" für dieses Argument basiert der p -Wert auf der exakten Verteilung aller möglichen $\hat{\theta}^*$.

```
> Nj <- c(7, 8)                                # Gruppengrößen
> DVa <- round(rnorm(Nj[1], 100, 20))          # Daten Gruppe A
> DVb <- round(rnorm(Nj[2], 110, 20))          # Daten Gruppe B
> dat <- stack(list(a=DVa, b=DVb))            # Gesamt-Daten
> library(coin)                               # für oneway_test()
> (ot <- oneway_test(values ~ indx, alternative="less",
+                      distribution="exact", data=dat))
Exact 2-Sample Permutation Test
data: values by ind (a, b)
Z = -2.0981, p-value = 0.01756
alternative hypothesis: true mu is less than 0
```

Das Ergebnis nennt den Wert der intern verwendeten Teststatistik unter Z ,⁸ gefolgt vom p -Wert unter `p-value`.

`?support` erläutert die Verwendung von `dperm()`, `qperm()` und `rperm()`, um Dichteverteilung und Quantile der Permutations-Teststatistik von `oneway_test()` zu ermitteln sowie aus ihr Zufallszahlen zu erzeugen. Die Permutationsverteilung sollte unimodal und symmetrisch sein, nicht unähnlich einer Normalverteilung (Abb. 11.3).

```
# Dichteverteilung der Teststatistik von oneway_test()
> supp <- support(ot)
> dens <- sapply(supp, dperm, object=ot)          # Dichte
> plot(supp, dens, xlab="Support", ylab=NA, pch=20,
+       main="Dichte Permutationsverteilung")

# Q-Q-Plot der Teststatistik von oneway_test() gegen Standard-NV
> qEmp <- sapply(ppoints(supp), qperm, object=ot) # Quantile Teststat.
> qqnorm(qEmp, xlab="Quantile Normalverteilung",
+         ylab="Permutations-Quantile",
+         main="Permutations- vs. theoretische NV-Quantile")

> abline(a=0, b=1, lwd=2, col="blue")
```

Zum Vergleich mit dem Permutationstest soll zunächst der p -Wert des analogen parametrischen t -Tests ermittelt werden (Abschn. 7.3.2). Bei der folgenden manuellen Kontrolle dient die Mittelwertsdifferenz zwischen beiden Gruppen als Teststatistik. Dabei ist zu beachten, dass $\hat{\theta}^*$ hier nicht für alle $N!$ möglichen Permutationen der Gesamtstichprobe vom Umfang N bestimmt werden muss. Dies ist nur für jene Permutationen notwendig, die auch zu unterschiedlichen

⁸Für deren Wahl s. `vignette("coin_implementation")`.

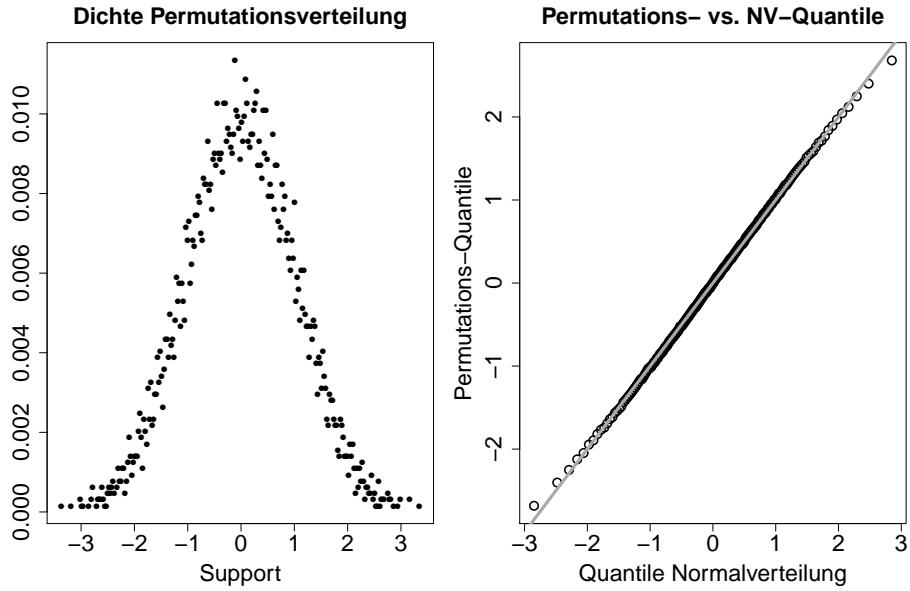


Abbildung 11.3: Dichteverteilung der Teststatistik des Permutationstests. Q-Q-Plot der Teststatistik des Permutationstests im Vergleich zur Standardnormalverteilung.

Gruppenzusammensetzungen führen, also nicht lediglich innerhalb jeder Gruppe die Reihenfolge vertauschen. Es gibt $\binom{N}{n_1}$ viele Möglichkeiten (Kombinationen, Abschn. 2.3.3), zwei Gruppen der Größe n_1 und n_2 zu bilden, dabei ist jede Kombination gleich wahrscheinlich.

```
# Vergleich: p-Wert aus analogem parametrischen t-Test
> tt <- t.test(values ~ ind, alt="less", var.equal=TRUE, data=dat)
> tt$p.value
[1] 0.01483469

# Indizes aller unterschiedlichen Zusammensetzungen Gruppe A*
> idx <- seq_len(sum(Nj)) # Indizes Gesamt-Daten
> idxA <- combn(idx, Nj[1]) # alle n1-Kombinationen

# Mittelwertsdifferenz für gegebene Indizes x der Gruppe A*
> getDM <- function(x) {
+   mean(dat[x, "values"]) - mean(dat[-x, "values"])
+ }

> DMstar <- apply(idxA, 2, getDM) # M-Differenz aller Kombinationen
> DMbase <- mean(DVa) - mean(DVb) # beobachtete M-Differenz
```

Der für den p -Wert notwendige Vergleich $\hat{\theta}^* \leq \hat{\theta}$ wird hier in zwei Vergleiche aufgeteilt, um robuster gegenüber Problemen der numerischen Repräsentation von Gleitkommazahlen zu sein (Abschn. 1.4.6). Dafür wird nur auf ungefähre, nicht auf exakte Gleichheit von $\hat{\theta}^*$ und $\hat{\theta}$ geprüft.

```
# DMstar kleiner Basis-DM ODER DMstar ungefähr gleich Basis-DM?
> tol <- .Machine$double.eps^0.5 # numerische Toleranz
```

```
> DMsIsLEQ <- (DMstar < DMbase) | (abs(DMstar - DMbase) < tol)

# p-Wert: Anteil der mindestens so extremen Mittelwertsdifferenzen
> (pVal <- sum(DMsIsLEQ) / length(DMstar))
[1] 0.01756022
```

11.3.2 Test auf gleiche Lageparameter in abhängigen Stichproben

Beim Test auf Übereinstimmung von Verteilungen aus (hier zwei) abhängigen Stichproben hat die für `oneway_test()` anzugebende Modellformel die Form $\langle \text{AV} \rangle \sim \langle \text{UV} \rangle \mid \langle \text{BlockId} \rangle$. Dabei codiert $\langle \text{UV} \rangle$ als Faktor derselben Länge wie $\langle \text{AV} \rangle$ den Messzeitpunkt. $\langle \text{BlockId} \rangle$ ist ebenfalls ein solcher Faktor und gibt im Fall von Messwiederholung die Zugehörigkeit jedes Wertes zu einem Beobachtungsobjekt (bei gematchten Personen: zu einem Block) an.

```
> N      <- 12                      # Anzahl Personen
> DVpre  <- round(rnorm(N, 100, 20))    # Daten Zeitpunkt prä
> DVpost <- round(rnorm(N, 110, 20))    # Daten Zeitpunkt post
> datPP  <- stack(list(pre=DVpre, post=DVpost))  # Gesamt-Daten
> datPP$id <- factor(rep(1:N, times=2))    # Faktor Personen-ID

# Faktor ind = Messzeitpunkt
> library(coin)                      # für oneway_test()
> oneway_test(values ~ ind | id, alternative="less",
+               distribution=approximate(nresample=9999), data=datPP)
Approximative 2-Sample Permutation Test
data: values by ind (pre, post) stratified by id
Z = -0.6056, p-value = 0.2748
alternative hypothesis: true mu is less than 0
```

Das Ergebnis des Fisher-Pitman Randomisierungstests soll zunächst mit dem p -Wert des analogen parametrischen t -Tests verglichen werden (Abschn. 7.3.3). Bei der anschließenden manuellen Kontrolle dient die mittlere paarweise Differenz zwischen den Werten der zwei abhängigen Bedingungen als Teststatistik. Dafür ist separat für jede Person die Zuordnung ihrer beiden Werte zu einem Testzeitpunkt zu permutieren. Dies ist äquivalent zur Permutation des Vorzeichens der personenweisen Messwertdifferenz. Bei n Personen führt dies zu 2^n verschiedenen Gesamt-Permutationen.

```
> tt <- t.test(values ~ ind, alt="less", paired=TRUE, data=datPP)
> tt$p.value
[1] 0.2839126

# alle  $2^N$  Möglichk., pro Person Vorzeichen Differenz zu permutieren
> DVd    <- DVpre - DVpost        # personenweise Messwertdifferenzen
> ordLst <- lapply(numeric(N), function(x) { c(-1, 1) } )
> ordMat <- data.matrix(expand.grid(ordLst))  # alle  $2^N$  Permutationen

# für Gesamt-Permutation x der Vorzeichen: mittlere personenweise Diff.
```

```
> getMD <- function(x) { mean(abs(DVd) * x) }
> MDstar <- apply(ordMat, 1, getMD) # mittlere Differenzen alle Permut.
> MDbase <- mean(DVd)           # mittl. Differenz Basis-Stichprobe
```

Der für den p -Wert notwendige Vergleich $\hat{\theta}^* \leq \hat{\theta}$ wird hier in zwei Vergleiche aufgeteilt, um robuster gegenüber Problemen der numerischen Repräsentation von Gleitkommazahlen zu sein (Abschn. 1.4.6). Dafür wird nur auf ungefähre, nicht auf exakte Gleichheit von $\hat{\theta}^*$ und $\hat{\theta}$ geprüft.

```
# MDstar kleiner Basis-MD ODER MDstar ungefähr gleich Basis-MD?
> tol <- .Machine$double.eps^0.5      # numerische Toleranz
> MDsIsLEQ <- (MDstar < MDbase) | (abs(MDstar - MDbase) < tol)

# p-Wert: Anteil der mind. so extremen personenweisen Differenzen
> (pVal <- sum(MDsIsLEQ) / length(MDstar))
[1] 0.2800293
```

11.3.3 Test auf Unabhängigkeit von zwei Variablen

Für den Test auf Unabhängigkeit zweier an denselben Personen erhobener Variablen sollen hier dichotome Daten dienen, um das Ergebnis der manuellen Umsetzung mit jenem von Fishers exaktem Test vergleichen zu können – dem passenden Permutationstest (Abschn. 10.2.4).

```
> Nf <- 8                                # Anzahl Personen
> DV1 <- rbinom(Nf, size=1, prob=0.5)      # Daten Zielgröße 1
> DV2 <- rbinom(Nf, size=1, prob=0.5)      # Daten Zielgröße 2

# p-Wert rechtsseitiger Test
> fisher.test(DV1, DV2, alternative="greater")$p.value
[1] 0.7857143
```

Die manuelle Kontrolle verwendet `Permn(<Vektor>)` aus dem Paket `DescTools`, um die Zuordnung von Werten der zweiten Zielgröße zu jenen der ersten zu permutieren. Die Funktion erzeugt eine Matrix mit allen $n!$ vielen Permutationen des übergebenen Vektors in den Zeilen. Mit den Indizes für den Vektor der zweiten Zielgröße liefern diese Permutationen die gewünschten Zuordnungen und führen so zu allen möglichen Kontingenztafeln der gemeinsamen Häufigkeiten mit denselben Randhäufigkeiten wie in der Basisstichprobe. Teststatistik ist die Anzahl der in der Diagonale einer Kontingenztafel stehenden Fälle, also der Übereinstimmungen beider Variablen.

```
# alle Nf! Zuordnungen Zielgröße 1 <-> Zielgröße 2
# Matrix der Permutationen der Indizes 1:Nf
> library(DescTools)                      # für Permn()
> permIdx <- Permn(seq_len(Nf))

# Anzahl der Übereinstimmungen für gegebene Permutation x der ZG 2
> getAgree <- function(x) { sum(diag(table(DV1, DV2[x]))) }
```

Kapitel 11 Resampling-Verfahren

```
# Anzahl der Übereinstimmungen für jede der Nf! Permutationen
> resAgree <- apply(permIdx, 1, getAgree)
> agree12  <- sum(diag(table(DV1, DV2)))      # beobachtete Übereinst.

# p-Wert: Anteil der Permut. mit mindestens so großer Übereinstimmung
> (pVal <- sum(resAgree >= agree12) / length(resAgree))
[1] 0.7857143
```

Kapitel 12

Multivariate Verfahren

Liegen von Beobachtungsobjekten Werte mehrerer Variablen vor, kann sich die Datenanalyse nicht nur auf jede Variable einzeln, sondern auch auf die gemeinsame Verteilung der Variablen beziehen. Solche Fragestellungen sind mit multivariaten Verfahren zu bearbeiten ([Backhaus, Erichson, Plinke & Weiber, 2018](#); [Backhaus, Erichson & Weiber, 2015](#)), deren Anwendung in R [Zelterman \(2015\)](#) vertiefend behandelt. Abschnitte [14.6.8](#), [14.7](#) und [15.3](#) thematisieren Möglichkeiten, multivariate Daten in Diagrammen zu veranschaulichen.

12.1 Lineare Algebra

Vielen statistischen Auswertungen – insbesondere im Bereich multivariater Verfahren – liegt im Kern das Rechnen mit Matrizen zugrunde. Deshalb soll an dieser Stelle auf einige grundlegende Funktionen zur Matrix-Algebra eingegangen werden, Details beschreibt [Fieller \(2017\)](#). Die gezeigten Zusammenhänge sind nur für manuelle Kontrollrechnungen relevant, nicht aber für die reine Anwendung multivariater Tests mit R-eigenen Funktionen.¹ Die Rechentechniken und zugehörigen Konzepte der linearen Algebra seien als bekannt vorausgesetzt ([Fischer & Springborn, 2020](#); [Strang, 2003](#)).

12.1.1 Matrix-Algebra

Eine Matrix \mathbf{X} wird mit $\text{t}(\mathbf{X})$ transponiert. Hierdurch werden die Zeilen von \mathbf{X} zu den Spalten der Transponierten \mathbf{X}^\top und die Spalten von \mathbf{X} zu Zeilen von \mathbf{X}^\top . Es gilt $(\mathbf{X}^\top)^\top = \mathbf{X}$.

```
> N  <- 4                      # Anzahl Zeilen
> p  <- 2                      # Anzahl Spalten
> (X <- matrix(c(20, 26, 10, 19, 29, 27, 20, 12), nrow=N, ncol=p))
 [,1]  [,2]
[1,]    20    29
[2,]    26    27
[3,]    10    20
[4,]    19    12
```

¹Die hier vorgestellten Rechenwege setzen die mathematischen Formeln direkt um. Tatsächlich gibt es häufig numerisch effizientere und stabilere Möglichkeiten, um dieselben Ergebnisse zu erhalten. So entspricht die Implementierung von R-eigenen Funktionen auch meist nicht den hier vorgestellten Rechnungen ([Bates, 2004](#)). Abschnitt [17.5](#) erläutert Wege, die Effizienz der Berechnungen zu steigern. Das Paket **Matrix** ([Bates & Maechler, 2019](#)) enthält fortgeschrittene Methoden der Matrix-Algebra – etwa zu schwachbesetzten Matrizen, die u. a. als Designmatrizen linearer Modelle auftauchen (Abschn. [12.9.1](#)).

```

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

diag(<Vektor>) erzeugt eine Diagonalmatrix, deren Diagonale aus den Elementen von <Vektor> besteht. Besitzt der übergebene Vektor allerdings nur ein Element, liefert die Variante diag(<Anzahl>) als besondere Diagonalmatrix eine Einheitsmatrix mit <Anzahl> vielen Zeilen und Spalten.2

# 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 \mathbf{X} und \mathbf{Y} gleicher Dimensionierung der `+` Operator mit $\mathbf{X} + \mathbf{Y}$ genauso wie `*` für die Multiplikation von Matrizen mit einem Skalar mittels `<Matrix> * <\rightarrow Zahl>`. Auch $\mathbf{X} * \mathbf{Y}$ ist als elementweises Hadamard-Produkt $\mathbf{X} \circ \mathbf{Y}$ von gleich dimensionierten Matrizen zu verstehen. Die Matrix-Multiplikation einer $(p \times q)$ -Matrix \mathbf{X} mit einer $(q \times r)$ -Matrix \mathbf{Y} lässt sich dagegen mit $\mathbf{X} \%*\% \mathbf{Y}$ formulieren.³ Das Ergebnis ist die $(p \times r)$ -Matrix der Skalarprodukte der Zeilen von \mathbf{X} mit den Spalten von \mathbf{Y} . Als Rechenregel ist in vielen Situationen $(\mathbf{XY})^\top = \mathbf{Y}^\top \mathbf{X}^\top$ relevant.

Für den häufig genutzten Spezialfall $\mathbf{X}^\top \mathbf{Y}$ existiert die Funktion `crossprod(X, Y)`, die alle paarweisen Skalarprodukte der Spalten von \mathbf{X} und \mathbf{Y} berechnet.⁴ `crossprod(X)` ist die Kurzform für $\mathbf{X}^\top \mathbf{X}$. Das Kronecker-Produkt $\mathbf{X} \otimes \mathbf{Y}$ erhält man mit der Funktion `kronecker()`, deren Operator-Schreibweise $\mathbf{X} \%x\% \mathbf{Y}$ lautet.

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

²Eine Dezimalzahl wird dabei tranchiert. Eine (1×1) -Diagonalmatrix \mathbf{X} kann mit einem weiteren Argument als `diag(x, nrow=1)` erzeugt werden.

³Mit dem Operator `<Matrix> \%~% <Zahl>` aus dem Paket `expm` (Goulet et al., 2020) ist das Exponenzieren von quadratischen Matrizen möglich, mit `logm()` aus demselben Paket das Logarithmieren.

⁴Sie ist zudem numerisch effizienter als die Verwendung von `t(X) \%*\% Y`. Die Benennung erscheint unglücklich, da Verwechslungen mit dem Vektor-Kreuzprodukt naheliegen, das man stattdessen mit `cross()` aus dem Paket `pracma` (Borchers, 2019b) erhält.

werden in den meisten Rechnungen automatisch so in Zeilen- oder Spaltenvektoren konvertiert, dass die Rechenregeln erfüllt sind.

```
# c(1, 2) wie Zeilenvektor behandelt: (1x2) Vektor %*% (2x3) Matrix
> c(1, 2) %*% rbind(c(1, 2, 3), c(4, 5, 6))
      [,1] [,2] [,3]
[1,]    9   12   15

# c(7, 8, 9) wie Spaltenvektor behandelt: (2x3) Matrix %*% (3x1) Vektor
> rbind(c(1, 2, 3), c(4, 5, 6)) %*% c(7, 8, 9)
      [,1]
[1,] 50
[2,] 122
```

Das Skalarprodukt zweier Vektoren x und y ist mit `crossprod(x, y)` oder mit `sum(x*y)` berechenbar. Das Ergebnis von `crossprod()` ist dabei immer eine Matrix, auch wenn diese nur aus einer Spalte oder einzelnen Zahl besteht.

Als Beispiel sollen einige wichtige Matrizen zu einer gegebenen $(n \times p)$ -Datenmatrix \mathbf{X} mit Variablen in den Spalten berechnet werden: Die spaltenweise zentrierte Datenmatrix $\dot{\mathbf{X}}$ führt zur SSP-Matrix $\dot{\mathbf{X}}^\top \dot{\mathbf{X}}$ (auch SSCP-Matrix genannt, *sum of squares and cross products*).⁵ Sie ist das $(n - 1)$ -fache der korrigierten Kovarianzmatrix S .

```
> (Xc <- diag(N) - matrix(rep(1/N, N^2), nrow=N))      # 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.equal(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

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

> crossprod(Xdot)                                    # äquivalent: SSP-Matrix ...
```

⁵Im Anwendungsfall sollte man statt einer Zentriermatrix die Funktion `scale(Matrix, center=TRUE, scale=FALSE)` verwenden, um eine Matrix spaltenweise zu zentrieren.

```
> (1/(N-1)) * SSP # korrigierte Kovarianzmatrix
      [,1]      [,2]
[1,] 43.58333 20.00000
[2,] 20.00000 59.33333

> (S <- cov(X)) # korrigierte Kovarianzmatrix
      [,1]      [,2]
[1,] 43.58333 20.00000
[2,] 20.00000 59.33333
```

Mit Hilfe der aus den Kehrwerten der korrigierten Streuungen gebildeten Diagonalmatrix $D_S^{-1/2}$ erhält man die zu einer Kovarianzmatrix S gehörende Korrelationsmatrix als $R = D_S^{-1/2} S D_S^{-1/2}$. Hierfür eignet sich `cov2cor(<K>)`.

```
> Dsi <- diag(1/sqrt(diag(S))) # Diagonalmatrix: 1 / Streuungen
> Dsi %*% S %*% Dsi # Korrelationsmatrix
      [,1]      [,2]
[1,] 1.0000000 0.3932968
[2,] 0.3932968 1.0000000

> cov2cor(S) # Kontrolle ...
```

Für Anwendungen, in denen die Zeilen oder Spalten einer Matrix X mit einem Vektor a verrechnet werden müssen, eignet sich `sweep()` (Abschn. 2.8.9). So könnte a etwa zu jeder Zeile von X addiert oder X spaltenweise normiert werden. Alternativ könnte letzteres auch mit XD geschehen, wenn D die Diagonalmatrix aus den Kehrwerten der Spaltenlängen ist.

```
> b <- 2 # Skalar
> a <- c(-2, 1) # Vektor
> sweep(b * X, 2, a, "+") # bX + a
      [,1]      [,2]
[1,] 38      59
[2,] 50      55
[3,] 18      41
[4,] 36      25

> colLens <- sqrt(colSums(X^2)) # euklidische Längen der Spalten
> sweep(X, 2, colLens, "/") # spaltenweise normierte Daten
      [,1]      [,2]
[1,] 0.5101443 0.6307329
[2,] 0.6631876 0.5872341
[3,] 0.2550722 0.4349882
[4,] 0.4846371 0.2609929

# äquivalent: Mult. mit Diag.-Matrix aus Kehrwerten der Spaltenlängen
> X %*% diag(1,colLens) # ...
```

12.1.2 Lineare Gleichungssysteme lösen

Für die Berechnung der Inversen \mathbf{A}^{-1} einer quadratischen Matrix \mathbf{A} mit vollem Rang ist die Funktion `solve(a=Matrix, b=Matrix)` zu nutzen, die die Lösung \mathbf{x} des linearen Gleichungssystems $\mathbf{Ax} = \mathbf{b}$ liefert. Dabei ist für das Argument `a` eine invertierbare quadratische Matrix und für `b` ein Vektor oder eine Matrix passender Dimensionierung anzugeben. Fehlt das Argument `b`, wird als rechte Seite des Gleichungssystems die passende Einheitsmatrix \mathbf{I} angenommen, womit sich als Lösung für \mathbf{x} in $\mathbf{Ax} = \mathbf{I}$ die Inverse \mathbf{A}^{-1} ergibt.⁶

Für \mathbf{A}^{-1} gilt etwa $(\mathbf{A}^{-1})^{-1} = \mathbf{A}$, $(\mathbf{A}^{-1})^\top = (\mathbf{A}^\top)^{-1}$ und $(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}$, wenn \mathbf{B} ebenfalls eine invertierbare Matrix passender Dimensionierung ist.

```
> Y <- matrix(c(1, 1, 1, -1), nrow=2)    # Matrix Y
> (Yinv <- solve(Y))                      # Inverse von Y
      [,1]   [,2]
[1,]  0.5   0.5
[2,]  0.5  -0.5

> Y %*% Yinv                                # Kontrolle: Einheitsmatrix
      [,1]   [,2]
[1,]     1     0
[2,]     0     1

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

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

12.1.3 Norm und Abstand von Vektoren und Matrizen

Das mit `crossprod(Vektor)` ermittelte Skalarprodukt $\mathbf{x}^\top \mathbf{x}$ eines Vektors \mathbf{x} mit sich selbst liefert ebenso wie `sum(Vektor)^2` dessen quadrierte euklidische Länge $\|\mathbf{x}\|^2$ als quadrierte Norm. Genauso eignen sich `colSums(Matrix)^2` und `crossprod(Matrix)`, um die quadrierten Spaltennormen einer Matrix zu ermitteln, die dort im Ergebnis in der Diagonale stehen.

```
> a1 <- c(3, 4, 1, 8, 2)                  # Vektor
> sqrt(crossprod(a1))                     # Norm (euklidische Länge)
[1] 9.69536
```

⁶Für die Pseudoinverse \mathbf{A}^+ einer nicht invertierbaren Matrix \mathbf{A} vgl. `ginv()` aus dem **MASS** Paket. Für solche Matrizen ermittelt `Null()` aus demselben Paket eine Basis des Kerns von \mathbf{A}^\top (*null space*).

```
> sqrt(sum(a1^2))                                # Kontrolle ...
> a2 <- c(6, 9, 10, 8, 7)                      # weiterer Vektor
> A <- cbind(a1, a2)                            # verbinde Vektoren zu Matrix
> sqrt(diag(crossprod(A)))                      # Spaltennormen der Matrix
      a1          a2
9.69536  18.16590
```

```
> sqrt(colSums(A^2))                            # Kontrolle ...
```

`norm(<Matrix>, type=<'Norm'>)` berechnet verschiedene Matrixnormen. Über das Argument `type` lässt sich der genaue Typ auswählen, die Frobenius-Norm erhält man etwa über `type="F"`. Fasst man eine $(p \times q)$ -Matrix A als $(p \cdot q)$ -Vektor a auf, ist die Frobenius-Norm von A gleich der Minkowski-2-Norm von a , also gleich dessen euklidischer Länge.

```
> norm(A, type="F")                             # Frobenius-Norm
[1] 20.59126
```

```
# Kontrolle: 2-Norm des zugehörigen Vektors
> sqrt(crossprod(c(A)))                      # ...
```

Der Abstand zwischen zwei Vektoren ist gleich der Norm des Differenzvektors. Mit `dist()` ist eine Funktion verfügbar, die unterschiedliche Abstandsmaße direkt berechnen kann.

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

Das erste Argument erwartet eine zeilenweise aus Koordinatenvektoren zusammengestellte Matrix. In der Voreinstellung werden alle paarweisen euklidischen Abstände zwischen ihren Zeilen berechnet. Über `method` lassen sich 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.

```
> B <- matrix(sample(-20:20, 12, replace=TRUE), ncol=3)
> dist(B, 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
> sqrt(crossprod(B[1, ] - B[2, ]))
[1,] 26.09598
```

12.1.4 Mahalanobistransformation und Mahalanobisdistanz

Die Mahalanobistransformation ist eine affine Transformation einer multivariaten Variable, deren Zentroid in den Ursprung des Koordinatensystems verschoben und deren Kovarianzmatrix die zugehörige Einheitsmatrix wird. In diesem Sinne handelt es sich um eine Verallgemeinerung der z -Transformation univariater Variablen. Ist \mathbf{S} die Kovarianzmatrix einer multivariaten Variable X mit Zentroid $\bar{\mathbf{x}}$, ergibt $\mathbf{S}^{-\frac{1}{2}}(\mathbf{x} - \bar{\mathbf{x}})$ die Mahalanobistransformierte eines konkreten Datenvektors \mathbf{x} .⁷ Entsprechend ist $(\mathbf{S}^{-\frac{1}{2}}\dot{\mathbf{X}}^\top)^\top$ die Mahalanobistransformation mehrerer Datenvektoren, die zeilenweise die Matrix \mathbf{X} bilden, deren spaltenweise Zentrierte $\dot{\mathbf{X}}$ ist.

Im Beispiel seien an mehreren Bewerbern für eine Arbeitsstelle drei Eigenschaften erhoben worden. Zunächst wird `rmvnorm()` aus dem Paket `mvtnorm` verwendet, um Zufallsvektoren einer multinormalverteilten dreidimensionalen Variable zu simulieren. 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 (Abschn. 12.1.6 zeigt die Berechnung von $\mathbf{S}^{-\frac{1}{2}}$ durch Diagonalisieren von \mathbf{S} mittels Spektralzerlegung durch `eigen()`).

```
# 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
> N      <- 100                                # Anzahl Bewerber
> library(mvtnorm)                            # für rmvnorm()
> X      <- round(rmvnorm(N, 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  $\mathbf{S}^{-\frac{1}{2}}$  durch Diagonalisieren
> SsqrtInv <- Seig$vectors %*% diag(1/sqrtD) %*% t(Seig$vectors)

# Differenzvektoren zwischen Daten und Zentroid
> Xdot <- scale(X, center=ctr, scale=FALSE)

# Mahalanobistransformation der Daten
> Xmt <- t(SsqrtInv %*% t(Xdot))

# Kontrolle: Kovarianzmatrix der transformierten Daten: Einheitsmatrix
> cov(Xmt)                                    # ...

# Kontrolle: Zentroid der transformierten Daten: Null-Vektor
> colMeans(Xmt)                             # ...
```

⁷Jede Transformation der Form $\mathbf{G}\mathbf{S}^{-\frac{1}{2}}(\mathbf{x} - \bar{\mathbf{x}})$ mit \mathbf{G} als Orthogonalmatrix ($\mathbf{G}^\top = \mathbf{G}^{-1}$) würde ebenfalls eine multivariate z -Transformation liefern.

Kapitel 12 Multivariate Verfahren

Die quadrierte Mahalanobisdistanz zwischen zwei Vektoren \mathbf{x} und \mathbf{y} bzgl. einer Kovarianzmatrix \mathbf{S} ist definiert als $(\mathbf{x} - \mathbf{y})^\top \mathbf{S}^{-1}(\mathbf{x} - \mathbf{y})$ und 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, 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 Mahalanobis-transformation 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 i.S. 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 zeilenweise als Matrix
> mahalanobis(mat, ideal, S)   # Quadrat Maha.-Distanz zum Idealprofil
                                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,] 11.28615

> t(y-ideal) %*% Sinv %*% (y-ideal)  # quadrierte Distanz 2
[1,] 1.529668
```

Um den Bewerber mit der geringsten Mahalanobisdistanz zum Idealprofil unter allen Bewerbern zu identifizieren, kann auf `min()` und `which.min()` zurückgegriffen werden.

```
> mDist <- mahalanobis(X, ideal, S)  # quadrierte Distanz alle Bewerber
> min(mDist)                           # geringste quadrierte Distanz
[1] 0.2095796

> (idxMin <- which.min(mDist))        # zugehöriger Bewerber
[1] 16

> X[idxMin, ]                          # bestes 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 euklid. Distanz transformierte Bewerber-Profile - Idealprofil
> crossprod(Xmt[1, ] - t(idealM))      # erster Bewerber
[1,] 11.28615

> crossprod(Xmt[2, ] - t(idealM))      # zweiter Bewerber
[1,] 1.529668
```

12.1.5 Kennwerte von Matrizen

Die Spur (*trace*) einer $(p \times p)$ -Matrix \mathbf{A} ist $\text{tr}(\mathbf{A}) = \sum_{i=1}^p a_{ii}$. Man erhält sie mit `sum(diag(A))`. Zudem sei an $\text{tr}(\mathbf{A}^\top \mathbf{A}) = \text{tr}(\mathbf{A}\mathbf{A}^\top)$ ebenso erinnert⁸ wie an $\text{tr}(\mathbf{AB}) = \text{tr}(\mathbf{BA})$ und $\text{tr}(\mathbf{A}+\mathbf{B}) = \text{tr}(\mathbf{B}+\mathbf{A})$, wenn auch \mathbf{B} eine $(p \times p)$ -Matrix ist.

```
> (A <- matrix(c(9, 1, 1, 4), nrow=2)) # Matrix A
     [,1]   [,2]
[1,]    9     1
[2,]    1     4

> sum(diag(A))                      # Spur von A
[1] 13
```

Die Determinante $\det(\mathbf{A})$ einer quadratischen Matrix \mathbf{A} berechnet `det(A)`. Genau dann, wenn \mathbf{A} invertierbar ist, gilt $\det(\mathbf{A}) \neq 0$. Zudem ist $\det(\mathbf{A}) = \det(\mathbf{A}^\top)$. Für die Determinante zweier $(p \times p)$ -Matrizen \mathbf{A} und \mathbf{B} gilt $\det(\mathbf{AB}) = \det(\mathbf{BA}) = \det(\mathbf{A}) \cdot \det(\mathbf{B})$. Für eine obere $(p \times p)$ -Dreiecksmatrix \mathbf{D} (insbesondere also für Diagonalmatrizen) ist $\det(\mathbf{D}) = \prod_{i=1}^p d_{ii}$, das Produkt der Diagonalelemente. Zusammengenommen folgt für invertierbare Matrizen $\det(\mathbf{A}^{-1}) = \frac{1}{\det(\mathbf{A})}$.⁹

```
> det(A)                           # Determinante von A
[1] 35
```

Während der Rang einer Matrix als Maximum der Anzahl linear unabhängiger Zeilen bzw. Spalten klar definiert ist, wirft seine numerische Berechnung für beliebige Matrizen Probleme auf, weil Computer Zahlen nur mit endlicher Genauigkeit darstellen können (Abschn. 1.4.6). Über die *QR*-Zerlegung mit `qr(<Matrix>)` (Abschn. 12.1.6) lässt sich der Rang aber meist zuverlässig ermitteln. Er befindet sich in der ausgegebenen Liste in der Komponente `rank`, sofern die Voreinstellung `LAPACK=FALSE` beibehalten wurde. Eine $(p \times p)$ -Matrix \mathbf{X} ist genau dann invertierbar, wenn $\text{Rang}(\mathbf{X}) = p$ gilt.

⁸Es sei $\mathbf{a}_{\cdot i}$ die i -te Zeile und $\mathbf{a}_{\cdot j}$ die j -te Spalte von \mathbf{A} . Dann gilt $\text{tr}(\mathbf{A}^\top \mathbf{A}) = \sum_j \mathbf{a}_{\cdot j}^\top \mathbf{a}_{\cdot j} = \sum_j \sum_i a_{ij}^2 =$

⁹ $\det(\mathbf{A}) \cdot \det(\mathbf{A}^{-1}) = \det(\mathbf{AA}^{-1}) = \det(\mathbf{I}) = \prod_{i=1}^p 1 = 1^p = 1$.

```
> qrA <- qr(A)                      # QR-Zerlegung
> qrA$rank                           # Rang
[1] 2
```

Eigenwerte und -vektoren einer quadratischen Matrix berechnet `eigen(Matrix)`. Beide werden als Komponenten einer Liste ausgegeben – die Eigenwerte als Vektor in der Komponente `values`,¹⁰ die normierten Eigenvektoren als Spalten einer Matrix in der Komponente `vectors`. Die Summe der Eigenwerte einer Matrix ist gleich ihrer Spur, das Produkt der Eigenwerte gleich der Determinante.

```
> (eigA <- eigen(A))                # Eigenwerte und -vektoren von A
$values
[1] 9.192582 3.807418

$vectors
[,1]      [,2]
[1,] -0.9819564 0.1891075
[2,] -0.1891075 -0.9819564

# Matrix ist symmetrisch -> Matrix aus Eigenvektoren ist orthogonal
> zapsmall(eigA$vectors %*% t(eigA$vectors))
[,1]  [,2]
[1,]    1    0
[2,]    0    1

> sum(eigA$values)                 # Spur = Summe der Eigenwerte
[1] 13

> prod(eigA$values)                # Determinante = Produkt der Eigenwerte
[1] 35
```

Die Kondition κ einer reellen Matrix \mathbf{X} ist für invertierbare \mathbf{X} gleich dem Produkt $\|\mathbf{X}\| \cdot \|\mathbf{X}^{-1}\|$ der Normen von \mathbf{X} und ihrer Inversen \mathbf{X}^{-1} , andernfalls gleich $\|\mathbf{X}\| \cdot \|\mathbf{X}^+\|$ mit \mathbf{X}^+ als Pseudoinverser. Das Ergebnis hängt damit von der Wahl der Matrixnorm ab, wobei die 2-Norm üblich ist. Die 2-Norm von \mathbf{X} ist gleich der Wurzel aus dem größten Eigenwert von $\mathbf{X}^\top \mathbf{X}$, im Spezialfall symmetrischer Matrizen damit gleich dem betragsmäßig größten Eigenwert von \mathbf{X} . Zur Berechnung von κ dient `kappa(Matrix)`, `exact=FALSE`, wobei für eine numerisch aufwendigere, aber präzisere Bestimmung von κ das Argument `exact=TRUE` zu setzen ist.

```
> X <- matrix(c(20,26,10,19, 29,27,20,12, 17,23,27,25), nrow=4)
> kappa(X, exact=TRUE)                  # Kondition kappa
[1] 7.931935

# Kontrolle: Produkt der 2-Norm jeweils von X und X+
```

¹⁰Eigenwerte werden entsprechend ihrer algebraischen Vielfachheit 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 Fall reeller symmetrischer Matrizen. Ihre Eigenwerte sind alle reell, zudem stimmen algebraische und geometrische Vielfachheiten überein.

```
> Xplus <- solve(t(X) %*% X) %*% t(X) # Pseudoinverse X+
> norm(X, type="2") * norm(Xplus, type="2") # Kondition kappa
[1] 7.931935
```

Bei Verwendung der 2-Norm gilt zudem $\kappa = \sqrt{\frac{\lambda_{\max}}{\lambda_{\min}}}$ mit λ_{\max} als größtem und λ_{\min} als kleinstem Eigenwert $\neq 0$ von $\mathbf{X}^\top \mathbf{X}$.

```
> evX <- eigen(t(X) %*% X)$values # Eigenwerte X^\top X
> sqrt(max(evX) / min(evX[evX >= .Machine$double.eps]))
[1] 7.931935
```

Der Konditionsindex gibt für jeden Eigenwert λ_i den Quotienten $\sqrt{\frac{\lambda_i}{\lambda_{\min}}}$ an.

```
> sqrt(evX / min(evX[evX >= .Machine$double.eps]))
[1] 7.931935 1.502511 1.000000
```

12.1.6 Zerlegungen von Matrizen

Matrizen lassen sich auf unterschiedliche Weise *zerlegen*, also als Produkt anderer Matrizen darstellen. Diese Eigenschaft lässt sich in manchen Berechnungen ausnutzen, um eine höhere numerische Genauigkeit zu erreichen als bei der direkten Umsetzung mathematischer Formeln – etwa in der Regression (Bates, 2004).

Eine reelle symmetrische Matrix \mathbf{S} ist mit dem Spektralsatz diagonalisierbar, d. h. als *Spektralzerlegung* $\mathbf{S} = \mathbf{G}\mathbf{D}\mathbf{G}^\top$ darstellbar. Dabei ist \mathbf{G} die Orthogonalmatrix ($\mathbf{G}^\top = \mathbf{G}^{-1}$) mit den normierten Eigenvektoren von \mathbf{S} in den Spalten und \mathbf{D} die aus den Eigenwerten von \mathbf{S} in zugehöriger Reihenfolge gebildete reelle Diagonalmatrix.

```
> X      <- matrix(c(20,26,10, 19,29,27, 20,12,17, 23,27,25), nrow=4)
> S      <- cov(X)           # Kovarianzmatrix S von X
> eigS   <- eigen(S)         # Eigenwerte und -vektoren von S
> G      <- eigS$vectors    # Matrix: normierte Eigenvektoren
> D      <- diag(eigS$values) # Diagonalmatrix aus Eigenwerten
> all.equal(S, G %*% D %*% t(G)) # Spektralzerlegung S = G D G^\top
[1] TRUE
```

Ist eine Matrix \mathbf{S} diagonalisierbar, lassen sich Matrizen finden, durch die \mathbf{S} ebenfalls berechnet werden kann, was in verschiedenen Anwendungsfällen nützlich ist. So lässt sich \mathbf{S} dann als Quadrat einer Matrix $\mathbf{A} = \mathbf{G}\mathbf{D}^{\frac{1}{2}}\mathbf{G}^\top$ darstellen ($\mathbf{S} = \mathbf{A}^2 = \mathbf{A}\mathbf{A}$),¹¹ oder als Produkt einer Matrix \mathbf{N} mit deren Transponierter ($\mathbf{S} = \mathbf{N}\mathbf{N}^\top$). Dabei ist $\mathbf{N} = \mathbf{G}\mathbf{D}^{\frac{1}{2}}$ die spaltenweise aus den Eigenvektoren von \mathbf{S} zusammengestellte Matrix, deren Spaltenlängen jeweils gleich der Wurzel aus den zugehörigen Eigenwerten sind.¹²

¹¹ $\mathbf{AA} = \mathbf{GD}^{\frac{1}{2}}\mathbf{G}^\top \mathbf{GD}^{\frac{1}{2}}\mathbf{G}^\top = \mathbf{GD}^{\frac{1}{2}}\mathbf{D}^{\frac{1}{2}}\mathbf{G}^\top = \mathbf{GDG}^\top = \mathbf{S}$. Siehe auch `sqrtm()` aus dem Paket `expm`.

¹² $\mathbf{NN}^\top = \mathbf{GD}^{\frac{1}{2}}(\mathbf{GD}^{\frac{1}{2}})^\top = \mathbf{GD}^{\frac{1}{2}}(\mathbf{D}^{\frac{1}{2}})^\top \mathbf{G}^\top = \mathbf{GDG}^\top = \mathbf{S}$.

```
# stelle S als Quadrat A^2 einer Matrix A dar
> sqrtD <- diag(sqrt(eigS$values)) # Diagonalmatrix Wurzel Eigenwerte
> A      <- G %*% sqrtD %*% t(G)      # Matrix A = G D^(1/2) G^t
> all.equal(S, A %*% A)             # Zerlegung S = A^2
[1] TRUE

# stelle S als Produkt N N^t dar
> N <- eigS$vectors %*% sqrt(diag(eigS$values))           # Matrix N
> all.equal(S, N %*% t(N))          # Zerlegung S = N N^t
[1] TRUE
```

Die Singulärwertzerlegung einer beliebigen Matrix \mathbf{X} liefert `svd(X)` in Form einer Liste. Deren Komponente \mathbf{d} ist der Vektor der Singulärwerte, \mathbf{u} die spaltenweise aus den Links-Singulärvektoren und \mathbf{v} die spaltenweise aus den Rechts-Singulärvektoren zusammengesetzte Orthogonalmatrix. Insgesamt gilt damit $\mathbf{X} = \mathbf{u}\mathbf{D}\mathbf{v}^\top$, wenn \mathbf{D} die aus den Singulärwerten gebildete Diagonalmatrix ist. Die Spalten von \mathbf{u} sind Eigenvektoren von $\mathbf{X}\mathbf{X}^\top$, die von \mathbf{v} analog Eigenvektoren von $\mathbf{X}^\top\mathbf{X}$. Die Singulärwerte von \mathbf{X} sind gleich der Wurzel aus den Eigenwerten von $\mathbf{X}^\top\mathbf{X}$. Die Kondition κ (Abschn. 12.1.5) ist also auch gleich dem Quotienten vom größten zum kleinsten Singulärwert $\neq 0$ von \mathbf{X} .

```
> svdX <- svd(X)                      # Singulärwertzerlegung von X
> all.equal(X, svdX$u %*% diag(svdX$d) %*% t(svdX$v))   # X = u D v^t
[1] TRUE

# Wurzel aus Eigenwerten von X^t X = Singulärwerte von X
> all.equal(sqrt(eigen(t(X) %*% X)$values), svdX$d)
[1] TRUE
```

Die mit `chol(X)` durchgeführte Cholesky-Zerlegung einer symmetrischen, positiv-definiten Matrix \mathbf{X} berechnet eine obere Dreiecksmatrix \mathbf{R} , so dass $\mathbf{X} = \mathbf{R}^\top\mathbf{R}$ gilt.

```
> R <- chol(S)                      # Cholesky-Zerlegung von S
> all.equal(S, t(R) %*% R)          # Zerlegung S = R^t R
[1] TRUE
```

Die QR -Zerlegung einer beliebigen Matrix \mathbf{X} erhält man mit der `qr(X)` Funktion. Sie berechnet Matrizen \mathbf{Q} und \mathbf{R} , so dass $\mathbf{X} = \mathbf{Q}\mathbf{R}$ gilt. Dabei ist \mathbf{Q} eine Orthogonalmatrix derselben Dimensionierung wie \mathbf{X} ($\mathbf{Q}^\top = \mathbf{Q}^{-1}$, ggf. über die Pseudoinverse definiert) und \mathbf{R} eine obere Dreiecksmatrix. Die Ausgabe ist eine Liste, die den Rang von \mathbf{X} in der Komponente `rank` enthält, sofern die Voreinstellung `LAPACK=FALSE` beibehalten wurde. Um die Matrizen \mathbf{Q} und \mathbf{R} zu erhalten, müssen die Hilfsfunktionen `qr.Q(QR-Liste)` bzw. `qr.R(QR-Liste)` auf die von `qr()` ausgegebene Liste angewendet werden.

```
> qrX <- qr(X)                      # QR-Zerlegung von X
> Q   <- qr.Q(qrX)                  # extrahiere Q
> R   <- qr.R(qrX)                  # extrahiere R
> all.equal(X, Q %*% R)              # Zerlegung X = Q R
[1] TRUE
```

12.1.7 Orthogonale Projektion

In vielen statistischen Verfahren spielt die orthogonale Projektion von Daten im durch n Beobachtungsobjekte aufgespannten n -dimensionalen Personenraum U auf bestimmte Unterräume eine entscheidende Rolle (Wickens, 2015). Hier sei V ein solcher p -dimensionaler Unterraum ($n > p$) und die $(n \times p)$ -Matrix \mathbf{A} spaltenweise eine Basis von V ($\text{Rang}(\mathbf{A}) = p$). Für jeden Datenpunkt \mathbf{x} aus U liefert die orthogonale Projektion den Punkt \mathbf{v} aus V mit dem geringsten euklidischen Abstand zu \mathbf{x} (Abb. 12.1). \mathbf{v} ist dadurch eindeutig bestimmt, dass $(\mathbf{x} - \mathbf{v}) \perp V$ gilt, $\mathbf{x} - \mathbf{v}$ also senkrecht auf V und damit senkrecht auf den Spalten von \mathbf{A} steht ($\mathbf{A}^\top(\mathbf{x} - \mathbf{v}) = \mathbf{0}$). Ist \mathbf{y} der Koordinatenvektor von \mathbf{v} bzgl. der Basis \mathbf{A} , erfüllt $\mathbf{v} = \mathbf{Ay}$ somit die Bedingung $\mathbf{A}^\top(\mathbf{x} - \mathbf{Ay}) = \mathbf{0}$, was sich zu $\mathbf{A}^\top \mathbf{Ay} = \mathbf{A}^\top \mathbf{x}$ umformen lässt – den *Normalengleichungen*.

Da \mathbf{A} als Basis vollen Spaltenrang besitzt, existiert die Inverse $(\mathbf{A}^\top \mathbf{A})^{-1}$, weshalb der Koordinatenvektor \mathbf{y} des orthogonal auf V projizierten Vektors \mathbf{x} bzgl. der Basis \mathbf{A} durch $(\mathbf{A}^\top \mathbf{A})^{-1} \mathbf{A}^\top \mathbf{x}$ gegeben ist.¹³ Die Koordinaten bzgl. der Standardbasis berechnen sich entsprechend durch $\mathbf{A}(\mathbf{A}^\top \mathbf{A})^{-1} \mathbf{A}^\top \mathbf{x}$. Die Matrix $\mathbf{A}(\mathbf{A}^\top \mathbf{A})^{-1} \mathbf{A}^\top$ wird als orthogonale Projektion \mathbf{P} auf V bezeichnet.¹⁴ Ist \mathbf{C} eine Orthogonalbasis von V ($\mathbf{C}^\top \mathbf{C} = \mathbf{I}$ mit \mathbf{I} als $(p \times p)$ -Einheitsmatrix), gilt $\mathbf{P} = \mathbf{C}(\mathbf{C}^\top \mathbf{C})^{-1} \mathbf{C}^\top = \mathbf{CC}^\top$.

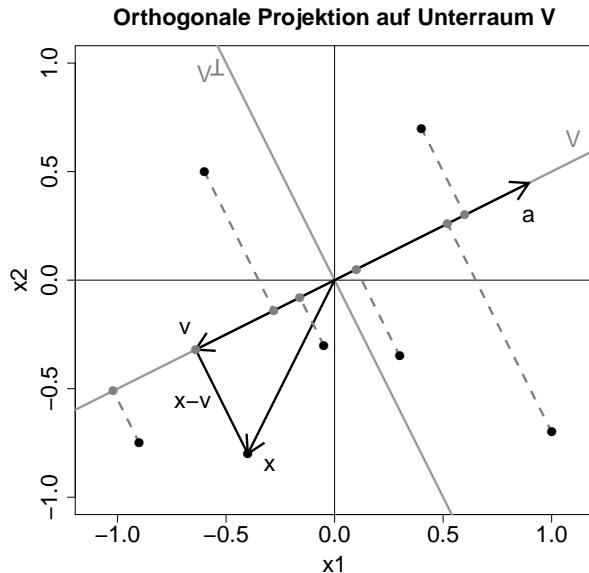


Abbildung 12.1: Orthogonale Projektion von Vektoren \mathbf{x} auf Unterraum V mit Basis \mathbf{a}

Eigenschaften

Im Fall eines eindimensionalen Unterraums V , dessen Basisvektor \mathbf{a} bereits normiert ist ($\|\mathbf{a}\|^2 = \mathbf{a}^\top \mathbf{a} = 1$), vereinfacht sich die Projektion zu $\mathbf{aa}^\top \mathbf{x}$, die Koordinaten des projizierten Vektors bzgl. \mathbf{a} liefert entsprechend $\mathbf{a}^\top \mathbf{x}$ – das Skalarprodukt von \mathbf{a} und \mathbf{x} .

¹³ $(\mathbf{A}^\top \mathbf{A})^{-1} \mathbf{A}^\top$ ist die Pseudoinverse \mathbf{A}^+ von \mathbf{A} .

¹⁴ Im Kontext linearer Modelle ist \mathbf{P} die *Hat-Matrix* (Abschn. 6.3.1).

Vektoren, die bereits aus V stammen, bleiben durch die Projektion unverändert, insbesondere ist also $\mathbf{P}\mathbf{A} = \mathbf{A}$.¹⁵ Orthogonale Projektionen sind idempotent ($\mathbf{P}^2 = \mathbf{P}$) und symmetrisch ($\mathbf{P}^\top = \mathbf{P}$),¹⁶ was sie auch bereits vollständig charakterisiert. Ferner gilt $\text{Rang}(\mathbf{P}) = \text{Rang}(\mathbf{A}) = p$, der Dimension von V . Zudem ist $\text{Rang}(\mathbf{P}) = \text{tr}(\mathbf{P})$.¹⁷ \mathbf{P} besitzt nur die Eigenwerte 1 mit Vielfachheit p und 0 mit Vielfachheit $n - p$. Die Eigenvektoren zum Eigenwert 1 bilden dabei eine Orthogonalbasis von V , da Vektoren \mathbf{v} aus V durch \mathbf{P} unverändert bleiben. Ist für zwei orthogonale Projektionen \mathbf{P}_1 und \mathbf{P}_2 ihr Produkt $\mathbf{P}_1\mathbf{P}_2$ symmetrisch, gilt $\mathbf{P}_1\mathbf{P}_2 = \mathbf{P}_2\mathbf{P}_1$.¹⁸

Die Projektion auf das orthogonale Komplement V^\perp von V berechnet sich mit der $(n \times n)$ -Einheitsmatrix \mathbf{I} als $\mathbf{I} - \mathbf{P}$.¹⁹ Der Rang von $\mathbf{I} - \mathbf{P}$ ist gleich $n - \text{Rang}(\mathbf{A})$, der Dimension von V^\perp . Für \mathbf{v} aus V gilt $(\mathbf{I} - \mathbf{P})\mathbf{v} = \mathbf{0}$,²⁰ insbesondere ist also $(\mathbf{I} - \mathbf{P})\mathbf{A} = \mathbf{0}$.

Beispiele

Als Beispiel sollen die im vorangehenden Abschnitt simulierten Daten mit der Projektion \mathbf{P}_1 im durch die Beobachtungsobjekte aufgespannten n -dimensionalen Personenraum auf den eindimensionalen Unterraum projiziert werden, dessen Basisvektor $\mathbf{1}$ aus lauter 1-Einträgen besteht. Dies bewirkt, dass alle Werte jeweils durch den Mittelwert der zugehörigen Variable ersetzt werden.²¹ Die Projektion auf das orthogonale Komplement liefert die Differenzen zum zugehörigen Spaltenmittelwert. Damit ist $\mathbf{I} - \mathbf{P}_1$ gleich der Zentriermatrix (Abschn. 12.1.1).

```
> X <- matrix(c(20,26,10,19, 29,27,20,12, 17,23,27,25), nrow=4)

# Basisvektor eines eindimensionalen Unterraums: lauter 1-Einträge
> ones <- rep(1, nrow(X))
> P1 <- ones %*% solve(t(ones) %*% ones) %*% t(ones)      # Projektion
> P1x <- P1 %*% X           # Koordinaten der Projektion
> head(P1x, n=3)            # Werte ersetzt durch Variablen-Mittelwerte
[1,] [2,] [3,]
[1,] 18.75   22   23
[2,] 18.75   22   23
[3,] 18.75   22   23

> colMeans(X)               # Kontrolle: Spaltenmittel der Daten
[1] 18.75 22.00 23.00

# orthogonale Projektion auf normierten Vektor als Basis des Unterraums
```

¹⁵Ist \mathbf{v} aus V , lässt sich $\mathbf{v} = \mathbf{C}\mathbf{y}$ schreiben, wobei \mathbf{y} der Koordinatenvektor von \mathbf{v} bzgl. einer Orthogonalbasis \mathbf{C} von V ist. Damit folgt $\mathbf{P}\mathbf{v} = \mathbf{C}\mathbf{C}^\top\mathbf{v} = \mathbf{C}\mathbf{C}^\top\mathbf{C}\mathbf{y} = \mathbf{C}\mathbf{y} = \mathbf{v}$.

¹⁶Zunächst gilt $\mathbf{P}^2 = (\mathbf{C}\mathbf{C}^\top)(\mathbf{C}\mathbf{C}^\top) = \mathbf{C}\mathbf{C}^\top = \mathbf{P}$. Weiter gilt $\mathbf{P}^\top = (\mathbf{C}\mathbf{C}^\top)^\top = \mathbf{C}\mathbf{C}^\top = \mathbf{P}$.

¹⁷ $\text{tr}(\mathbf{P}) = \text{tr}(\mathbf{C}\mathbf{C}^\top) = \text{tr}(\mathbf{C}^\top\mathbf{C}) = \text{tr}(\mathbf{I}) = p$.

¹⁸ $\mathbf{P}_1\mathbf{P}_2 = (\mathbf{P}_1\mathbf{P}_2)^\top = \mathbf{P}_2^\top\mathbf{P}_1^\top = \mathbf{P}_2\mathbf{P}_1$.

¹⁹Ist $\mathbf{w} = (\mathbf{I} - \mathbf{P})\mathbf{x}$ aus V^\perp und \mathbf{v} aus V , muss $\mathbf{w}^\top\mathbf{v} = \mathbf{0}$ gelten. \mathbf{v} lässt sich als $\mathbf{v} = \mathbf{C}\mathbf{y}$ schreiben, wobei \mathbf{y} der Koordinatenvektor von \mathbf{v} bzgl. einer Orthogonalbasis \mathbf{C} von V ist. Nun gilt $\mathbf{w}^\top\mathbf{v} = ((\mathbf{I} - \mathbf{P})\mathbf{x})^\top\mathbf{C}\mathbf{y} = \mathbf{x}^\top(\mathbf{I} - \mathbf{P})\mathbf{C}\mathbf{y} = (\mathbf{x}^\top - \mathbf{x}^\top\mathbf{C}\mathbf{C}^\top)\mathbf{C}\mathbf{y} = \mathbf{x}^\top\mathbf{C}\mathbf{y} - \mathbf{x}^\top\mathbf{C}\mathbf{C}^\top\mathbf{C}\mathbf{y} = \mathbf{x}^\top\mathbf{C}\mathbf{y} - \mathbf{x}^\top\mathbf{C}\mathbf{y} = \mathbf{0}$.

²⁰ $(\mathbf{I} - \mathbf{P})\mathbf{v} = \mathbf{I}\mathbf{v} - \mathbf{P}\mathbf{v} = \mathbf{v} - \mathbf{v} = \mathbf{0}$, da \mathbf{v} durch \mathbf{P} unverändert bleibt.

²¹ $\mathbf{P}_1 = \mathbf{1}_n(\mathbf{1}_n^\top\mathbf{1}_n)^{-1}\mathbf{1}_n^\top = \frac{1}{n}\mathbf{1}_n\mathbf{1}_n^\top = \frac{1}{n}\mathbf{1}_{n \times n}$.

```

> a <- ones / sqrt(crossprod(ones))           # normiere Basisvektor
> P2 <- a %*% t(a)                         # orthogonale Projektion
> all.equal(P1, P2)                          # Kontrolle: Vergleich mit erster Projektion
[1] TRUE

# Projektion auf orthogonales Komplement von P1
> IP1 <- diag(nrow(X)) - P1
> IP1x <- IP1 %*% X                         # zentrierte Daten

# Kontrolle
> all.equal(IP1x, scale(X, center=colMeans(X), scale=FALSE),
+            check.attributes=FALSE)
[1] TRUE

```

Als weiteres 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 2D 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
> head(Px3, n=3)                           # 3. Komponente auf 0 gesetzt
      [,1]  [,2]  [,3]
[1,]    20    29     0
[2,]    26    27     0
[3,]    10    20     0

```

Mit Hilfe der QR -Zerlegung lassen sich orthogonale Projektionen numerisch stabiler berechnen als mit der direkten Umsetzung der mathematischen Formel: Mit $\mathbf{X} = \mathbf{Q}\mathbf{R}$ folgt für die Pseudoinverse $\mathbf{X}^+ = \mathbf{R}^{-1}\mathbf{Q}^\top$ ²² und für die Projektion (Hat-Matrix \mathbf{H}) $\mathbf{P} = \mathbf{Q}\mathbf{Q}^\top$.²³

```

> qrX <- qr(X)                            # QR-Zerlegung
> Q <- qr.Q(qrX)                         # extrahiere Q
> R <- qr.R(qrX)                         # extrahiere R
> Xplus <- solve(t(X) %*% X) %*% t(X)   # Pseudoinverse X+
> all.equal(Xplus, solve(R) %*% t(Q))     # X+ = R^(-1) Q^t
[1] TRUE

> all.equal(X %*% Xplus, Q %*% t(Q))      # P = Q Q^t
[1] TRUE

```

²²Hier sei voller Spaltenrang von \mathbf{X} vorausgesetzt. Dann ist $\mathbf{X}^+ = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top = ((\mathbf{Q}\mathbf{R})^\top \mathbf{Q}\mathbf{R})^{-1}(\mathbf{Q}\mathbf{R})^\top =$ $(\mathbf{R}^\top \mathbf{Q}^\top \mathbf{Q}\mathbf{R})^{-1} \mathbf{R}^\top \mathbf{Q}^\top = (\mathbf{R}^\top \mathbf{R})^{-1} \mathbf{R}^\top \mathbf{Q}^\top = \mathbf{R}^{-1} \mathbf{R}^{t-1} \mathbf{R}^\top \mathbf{Q}^\top = \mathbf{R}^{-1} \mathbf{Q}^\top$.

²³ $\mathbf{X}\mathbf{X}^+ = \mathbf{Q}\mathbf{R}\mathbf{R}^{-1}\mathbf{Q}^\top = \mathbf{Q}\mathbf{Q}^\top$.

12.2 Hauptkomponentenanalyse

Die Hauptkomponentenanalyse dient dazu, die Hauptstreuungsrichtungen multivariater Daten im durch die Variablen aufgespannten Raum zu identifizieren. Hauptkomponenten sind neue Variablen, die als Linearkombinationen der beobachteten Variablen gebildet werden und folgende Eigenschaften besitzen:

- Die Linearkombinationen sind standardisiert, d. h. die Koeffizientenvektoren haben jeweils die Länge 1. Die Koeffizientenvektoren sind die normierten Eigenvektoren der Kovarianzmatrix der Daten (bis auf potentiell unterschiedliche Vorzeichen i. S. von Rotationen um 180°) und damit paarweise orthogonal.²⁴
- Die Hauptkomponenten sind zentriert und paarweise unkorreliert.²⁵
- Der euklidische Abstand zwischen zwei Punkten im Raum der Hauptkomponenten ist derselbe wie im Raum der beobachteten Variablen.²⁶
- 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 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.

12.2.1 Berechnung

Für die Berechnung der Hauptkomponenten samt ihrer jeweiligen Streuung stehen `prcomp()` und `princomp()` zur Verfügung.

```
prcomp(<Matrix oder Modellformel>, scale=FALSE, data=<Datensatz>)
princomp(<Matrix oder Modellformel>, data=<Datensatz>,
         cor=FALSE, covmat)
```

In der ersten Variante akzeptieren bei Funktionen die spaltenweise aus den Variablen zusammengestellte Datenmatrix. Damit die Hauptkomponenten aus den standardisierten Variablen berechnet werden, ist `prcomp(..., scale=TRUE)` bzw. `princomp(..., cor=TRUE)` zu setzen. Alternativ lassen sich die Variablen in beiden Funktionen als rechte Seite einer Modellformel nennen. Stammen sie aus einem Datensatz, muss dieser unter `data` übergeben werden. `princomp()` erlaubt es zusätzlich, die Kovarianzmatrix der Daten über das Argument `covmat`

²⁴Es sei $\bar{\mathbf{x}}$ das Zentroid der spaltenweise aus den Variablen zusammengestellten Datenmatrix \mathbf{X} , \mathbf{x} ein Datenvektor und \mathbf{G} die Matrix der spaltenweise zusammengestellten normierten Eigenvektoren der Kovarianzmatrix \mathbf{S} von \mathbf{X} (Abschn. 12.1.5). Mit dem Spektralsatz ist \mathbf{G} eine Orthogonalmatrix ($\mathbf{G}^\top = \mathbf{G}^{-1}$, s. Abschn. 12.1.6). Dann berechnet sich der zugehörige Vektor \mathbf{y} der Hauptkomponenten als $\mathbf{y} = \mathbf{G}^\top(\mathbf{x} - \bar{\mathbf{x}})$.

²⁵Genauer gesagt ist ihre Kovarianz 0 – da ihre Varianz auch 0 sein kann, ist die Korrelation nicht immer definiert. \mathbf{S} lässt sich mit $\mathbf{S} = \mathbf{G}\mathbf{D}\mathbf{G}^\top$ diagonalisieren (Abschn. 12.1.6). Dabei ist \mathbf{D} die zu \mathbf{G} gehörende, aus den Eigenwerten von \mathbf{S} gebildete Diagonalmatrix. Damit gilt $V(\mathbf{y}) = V(\mathbf{G}^\top(\mathbf{x} - \bar{\mathbf{x}})) = V(\mathbf{G}^\top\mathbf{x} - \mathbf{G}^\top\bar{\mathbf{x}}) = V(\mathbf{G}^\top\mathbf{x}) = \mathbf{G}^\top V(\mathbf{x})\mathbf{G} = \mathbf{G}^\top \mathbf{S} \mathbf{G} = \mathbf{G}^\top \mathbf{G} \mathbf{D} \mathbf{G}^\top \mathbf{G} = \mathbf{D}$. Die Varianzen der Hauptkomponenten sind also gleich den Eigenwerten der Kovarianzmatrix der Daten, die Kovarianzen sind 0.

²⁶Es seien \mathbf{x}_1 und \mathbf{x}_2 zwei Datenvektoren mit zugehörigen Hauptkomponenten-Vektoren \mathbf{y}_1 und \mathbf{y}_2 . Dann gilt $\|\mathbf{x}_2 - \mathbf{x}_1\|^2 = (\mathbf{x}_2 - \mathbf{x}_1)^\top(\mathbf{x}_2 - \mathbf{x}_1) = (\mathbf{x}_2 - \mathbf{x}_1)^\top \mathbf{G} \mathbf{G}^\top (\mathbf{x}_2 - \mathbf{x}_1) = (\mathbf{G}^\top(\mathbf{x}_2 - \mathbf{x}_1))^\top \mathbf{G}^\top (\mathbf{x}_2 - \mathbf{x}_1) = \|\mathbf{G}^\top(\mathbf{x}_2 - \mathbf{x}_1)\|^2 = \|\mathbf{G}^\top \mathbf{x}_2 - \mathbf{G}^\top \mathbf{x}_1\|^2 = \|\mathbf{G}^\top(\mathbf{x}_2 - \bar{\mathbf{x}}) - \mathbf{G}^\top(\mathbf{x}_1 - \bar{\mathbf{x}})\|^2 = \|\mathbf{y}_2 - \mathbf{y}_1\|^2$.

separat zu spezifizieren, etwa um direkt eine robuste Schätzung der theoretischen Kovarianzmatrix zu verwenden (Abschn. 2.7.9).²⁷

```
> sigma <- matrix(c(4,2, 2,3), ncol=2) # 2x2 Kovarianzmatrix
> mu     <- c(1, 2)                      # Zentroid für Zufallsvektoren
> N      <- 50                           # Anzahl Personen
> library(mvtnorm)                      # für rmvnorm()
> X      <- rmvnorm(N, mean=mu, sigma=sigma) # Multinormalverteilung
> (pca  <- prcomp(X))                  # Hauptkomponentenanalyse
Standard deviations:
[1] 2.436560 1.270408
```

Rotation:

| | PC1 | PC2 |
|------|------------|------------|
| [1,] | -0.7335078 | 0.6796810 |
| [2,] | -0.6796810 | -0.7335078 |

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 Eigenwerten der korrigierten Kovarianzmatrix der Daten (Abschn. 12.1.5). Die zweite Komponente beinhaltet die als Spalten einer Matrix \mathbf{G} zusammengestellten Koeffizienten der Linearkombinationen zur Bildung der Hauptkomponenten (Überschrift `Rotation`). Dies sind gleichzeitig die normierten Eigenvektoren der korrigierten Kovarianzmatrix.

```
> eig    <- eigen(cov(X))        # Eigenwerte, -vektoren Kovarianzmatrix
> eigVal <- eig$values          # Eigenwerte
> sqrt(eigVal)                 # deren Wurzel = Streuungen der HK
[1] 2.436560 1.270408

# normierte Eigenvektoren = Koeffizientenvektoren Linearkombinationen
> (G <- eig$vectors)
[,1]      [,2]
[1,] -0.7335078  0.6796810
[2,] -0.6796810 -0.7335078
```

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. 12.2). Die Hauptkomponenten sind die orthogonale Projektion der zentrierten Daten auf die Geraden in Richtung der Eigenvektoren der Kovarianzmatrix (Abschn. 12.1.7). Die Streuung der projizierten Daten entlang der Geraden ist daher gleich der Streuung der zugehörigen Hauptkomponente. Sie ist zudem gleich der Wurzel aus dem entsprechenden Eigenwert der Kovarianzmatrix.

```
# normierte Eigenvektoren * Wurzel aus zugehörigem Eigenwert
> B    <- G %*% diag(sqrt(eigVal))
> ctr  <- colMeans(X)                # Zentroid
```

²⁷Für die robuste Hauptkomponentenanalyse s. das Paket `pcaPP` ([Filzmoser, Fritz & Kalcher, 2018](#)).

```

> xMat <- rbind(ctr[1] - B[1, ], ctr[1])
> yMat <- rbind(ctr[2] - B[2, ], ctr[2])

# Punkt-Steigungsform der durch Eigenvektoren definierten Achsen
> ab1 <- solve(cbind(1, xMat[, 1]), yMat[, 1])      # Achse 1
> ab2 <- solve(cbind(1, xMat[, 2]), yMat[, 2])      # Achse 2

# Daten im Streudiagramm darstellen
> plot(X, xlab="x", ylab="y", pch=20, asp=1,
+       main="Datenwolke und Hauptkomponenten")

> abline(coef=ab1, lwd=2, col="gray")                  # Achse 1
> abline(coef=ab2, lwd=2, col="gray")                  # 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(20, NA, NA, 16),
+                               lty=c(NA, 1, 1, NA), col=c("black", "gray", "blue", "red"))

```

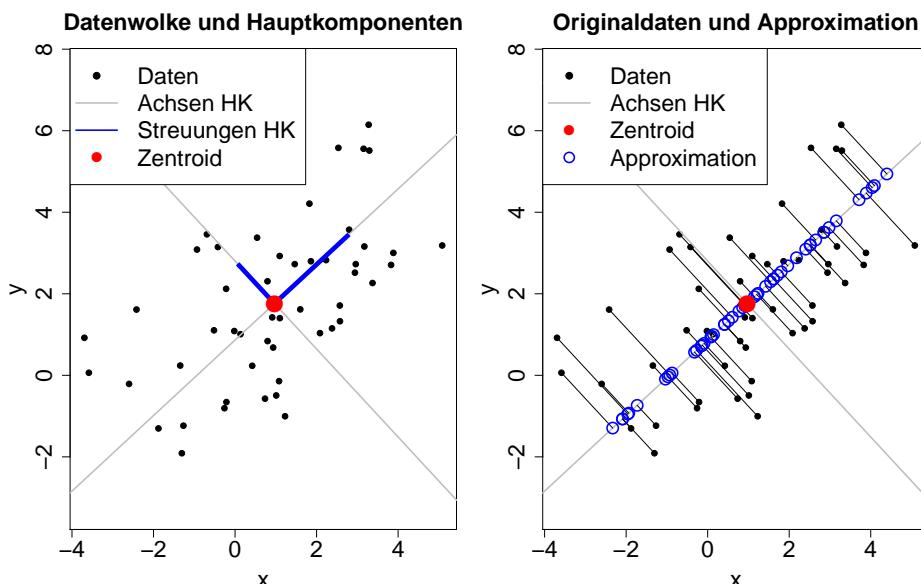


Abbildung 12.2: Hauptkomponentenanalyse: Koordinatensystem mit Achsen in Richtung der ins Zentroid verschobenen Eigenvektoren sowie Streuungen der Hauptkomponenten. Originaldaten und Approximation durch erste Hauptkomponente

Aus der Matrix \mathbf{G} der Koeffizienten der standardisierten Linearkombinationen und der spaltenweise zentrierten Datenmatrix $\dot{\mathbf{X}}$ berechnen sich die Hauptkomponenten als neue Variablen durch $\dot{\mathbf{X}}\mathbf{G}$. Man erhält sie mit `predict(<PCA>)`.²⁸ Dabei ist `<PCA>` das Ergebnis einer Haupt-

²⁸Das Ergebnis ist – bis auf mögliche Spiegelung der Achsen – identisch mit dem der metrischen multidimensionalen Skalierung (Abschn. 12.4) mit `cmdscale(dist(X))`.

Kapitel 12 Multivariate Verfahren

komponentenanalyse mit `prcomp()` oder `princomp()`.

```
> Y <- predict(pca)                      # Hauptkomponenten
> head(Y, n=3)
   PC1      PC2
[1,] -4.257878 -1.171098
[2,] -1.021898 -0.370420
[3,]  1.500310  1.683688

> (Ysd <- apply(Y, 2, sd))            # deren Streuungen
   PC1      PC2
2.436560  1.270408

# Kontrolle
> Xdot <- scale(X, center=TRUE, scale=FALSE)    # zentrierte Daten
> Y_XdotG <- Xdot %*% G                         # Hauptkomponenten
> head(Y_XdotG, n=3)
   PC1      PC2
[1,] -4.257878 -1.171098
[2,] -1.021898 -0.370420
[3,]  1.500310  1.683688
```

Mit der Singulärwertzerlegung von $\hat{\mathbf{X}}$ (Abschn. 12.1.6) ergeben sich die Hauptkomponenten auch als spaltenweise aus den Links-Singulärvektoren zusammengestellte Matrix \mathbf{u} , deren Spalten mit den Singulärwerten skaliert wurden.

```
> svd_Xdot <- svd(Xdot)                      # Singulärwertzerlegung
> Y_svd <- svd_Xdot$u %*% diag(svd_Xdot$d)  # Hauptkomponenten
> head(Y_svd, n=3)                           # ...
```

Bei `predict(<PCA>)` lässt sich für das Argument `newdata` eine Matrix mit neuen Werten der ursprünglichen Variablen angeben, die dann in Werte auf den Hauptkomponenten umgerechnet werden.

```
# Hauptkomponenten für neue Daten berechnen
> Xnew <- matrix(1:4, ncol=2)
> predict(pca, newdata=Xnew)
   PC1      PC2
[1,] -0.8672799 -0.8858317
[2,] -2.2804688 -0.9396585
```

Die Streuungen der Hauptkomponenten, den Anteil ihrer Varianz an der Gesamtvarianz (i. S. der Spur der Kovarianzmatrix der Daten) sowie den kumulativen Anteil der Varianz der Hauptkomponenten an der Gesamtvarianz gibt `summary(<PCA>)` aus.

```
> summary(pca)
Importance of components:
   PC1      PC2
Standard deviation     2.4366  1.2704
```

```
Proportion of Variance 0.7863 0.2137
Cumulative Proportion 0.7863 1.0000

# Kontrolle: Anteil der Varianz der HK an Gesamtvarianz
> Ysd^2 / sum(diag(cov(X)))
[1] 0.7862549 0.2137451
```

12.2.2 Dimensionsreduktion

Häufig dient die Hauptkomponentenanalyse der Datenreduktion: Wenn von Beobachtungsobjekten Werte von sehr vielen Variablen vorliegen, ist es oft wünschenswert, diese Daten durch Werte von weniger Variablen möglichst gut zu approximieren. Als Kriterium dient dabei die Summe der quadrierten Abstände zwischen Originaldaten und Approximation. Eine perfekte Reproduktion der Originaldaten lässt sich zunächst wie folgt erreichen: Sei die *Ladungs-Matrix* \mathbf{B} spaltenweise aus den Koeffizientenvektoren der Hauptkomponenten zusammengestellt. Zusätzlich sollen die Spalten von \mathbf{B} als Länge die Streuung der jeweils zugehörigen Hauptkomponente besitzen, was mit $\mathbf{B} = \mathbf{G}\mathbf{D}^{\frac{1}{2}}$ erreicht wird. Weiter sei \mathbf{H} die *Score-Matrix* der standardisierten Hauptkomponenten, die jeweils Mittelwert 0 und Varianz 1 besitzen. Ferner sei \bar{x} das Zentroid der Originaldaten. Für die Datenmatrix \mathbf{X} gilt dann $\mathbf{X} = \mathbf{HB}^T + \bar{x}$.²⁹

```
> B <- G %*% diag(sqrt(eigVal))
> H <- scale(Y)           # Score-Matrix = standard. Hauptkomponenten
> HB <- H %*% t(B)       # H * B^t

# Reproduktion der Originaldaten: addiere Zentroid c: H * B^t + c
> repr <- sweep(HB, 2, ctr, "+")
> all.equal(X, repr)      # Kontrolle: Übereinstimmung mit X
[1] TRUE
```

Sollen die ursprünglichen Daten nun im obigen Sinne optimal durch weniger Variablen repräsentiert werden, können die Spalten von \mathbf{B} von rechts kommend nacheinander gestrichen werden, wodurch sich etwa eine Matrix \mathbf{B}' ergibt. Ebenso sind die zugehörigen standardisierten Hauptkomponenten, also die Spalten von \mathbf{H} , zu streichen, wodurch die Matrix \mathbf{H}' entsteht. Die approximierte Daten $\mathbf{X}' = \mathbf{H}'\mathbf{B}'^T + \bar{x}$ liegen dann in einem affinen Unterraum mit niedrigerer Dimension als die ursprünglichen Daten (Abb. 12.2).³⁰

```
# approximiere Originaldaten durch niedrig-dimensionale Daten
> HB1 <- H[, 1, drop=FALSE] %*% t(B[, 1, drop=FALSE]) # H' * B'^t
> repr1 <- sweep(HB1, 2, ctr, "+") # H' * B'^t + c
> sum((X-repr1)^2)               # Summe der quadrierten Abweichungen
[1] 79.08294
```

²⁹Zunächst gilt $\mathbf{H} = \mathbf{YD}^{-\frac{1}{2}}$, wobei $\mathbf{Y} = \dot{\mathbf{X}}\mathbf{G} = \mathbf{ZXG}$ die Matrix der (zentrierten) Hauptkomponenten und \mathbf{Z} die Zentriermatrix zu \mathbf{X} ist (Abschn. 12.1.1). Wegen $\mathbf{G}^T = \mathbf{G}^{-1}$ folgt $\mathbf{HB}^T + \bar{x} = \mathbf{YD}^{-\frac{1}{2}}(\mathbf{GD}^{\frac{1}{2}})^T + \bar{x} = \mathbf{ZXGD}^{-\frac{1}{2}}\mathbf{D}^{\frac{1}{2}}\mathbf{G}^T + \bar{x} = \dot{\mathbf{X}}\mathbf{GG}^T + \bar{x} = \mathbf{X}$.

³⁰Die mittlere Summe der quadrierten Abstände von \mathbf{X} und \mathbf{X}' ist gleich der Summe der q kleinsten Eigenwerte der Kovarianzmatrix \mathbf{S} , wenn q Spalten von \mathbf{B} gestrichen wurden.

```
# approximierte Daten liegen in eindimensionalem Unterraum
> qr(scale(repr1, center=TRUE, scale=FALSE))$rank
[1] 1

# grafische Darstellung der Originaldaten und der Approximation
> plot(X, xlab="x", ylab="y", pch=20, asp=1,
+       main="Originaldaten und Approximation")

> abline(coef=ab1, lwd=2, col="gray")                      # Achse 1
> abline(coef=ab2, lwd=2, col="gray")                      # Achse 2
> segments(X[, 1], X[, 2], repr1[, 1], repr1[, 2])        # Approximation
> points(repr1, pch=1, lwd=2, col="blue", cex=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(20, NA, 16, 1),
+                               lty=c(NA, 1, NA, NA), col=c("black", "gray", "red", "blue"))
```

Die Kovarianzmatrix der Daten lässt sich als $\mathbf{S} = \mathbf{B}\mathbf{B}^\top$ zerlegen³¹ (Abschn. 12.1.6). Die Reproduktion von \mathbf{S} durch die um rechte Spalten gestrichene Matrix \mathbf{B}' mit $\mathbf{B}'\mathbf{B}'^\top$ wird schlechter, bleibt aber optimal im Vergleich zu allen anderen Matrizen gleicher Dimensionierung.

```
> cov(X)                                              # Kovarianzmatrix von X
      [,1]      [,2]
[1,] 3.939795  2.155180
[2,] 2.155180  3.610964

> B %*% t(B)                                         # reproduziert Kovarianzmatrix ...
> B[, 1] %*% t(B[, 1])                                # approximiere Kovarianzmatrix mit B' * B'^t
      [,1]      [,2]
[1,] 3.194211  2.959811
[2,] 2.959811  2.742612
```

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. Die Streuungen sind gleich den zugehörigen Eigenwerten der unkorrigierten Kovarianzmatrix der Daten.³² In der Ausgabe von `princomp()` sind die Koeffizienten der standardisierten Linearkombination zunächst nicht mit aufgeführt, befinden sich jedoch in der Komponente `loadings` der zurückgelieferten Liste.

```
> (pcaPrin <- princomp(X))
Call:
```

³¹ $\mathbf{B}\mathbf{B}^\top = \mathbf{G}\mathbf{D}^{\frac{1}{2}}(\mathbf{G}\mathbf{D}^{\frac{1}{2}})^\top = \mathbf{G}\mathbf{D}^{\frac{1}{2}}\mathbf{D}^{\frac{1}{2}}\mathbf{G}^\top = \mathbf{G}\mathbf{D}\mathbf{G}^\top = \mathbf{S}$.

³² Weiterhin basiert `prcomp()` intern auf der Singulärwertzerlegung der Datenmatrix \mathbf{X} , `princomp()` hingegen auf der Spektralzerlegung der Kovarianzmatrix \mathbf{S} der Daten (Abschn. 12.1.6). Die Singulärwertzerlegung der Datenmatrix gilt als numerisch stabiler bei schlecht konditionierten Matrizen i. S. der Kondition κ (Abschn. 12.1.5), kann aber langsamer sein. `prcomp()` sollte meist vorgezogen werden.

```

princomp(x = X)
Standard deviations:
    Comp.1      Comp.2
2.412071  1.257640
2 variables and 50 observations.

# Kontrolle: unkorrigierte Streuungen der Hauptkomponenten
> Sml <- cov.wt(pc, method="ML")$cov
> sqrt(diag(Sml))
    PC1      PC2
2.412071  1.257640

> pcaPrin$loadings                         # Koeffizientenmatrix ...

```

12.2.3 Visualisierung

Die Varianzen der Hauptkomponenten werden häufig in einem Liniendiagramm (Abschn. 14.2) als Scree-Plot dargestellt, der mit `plot(<PCA>, type="b")` aufzurufen ist (Abb. 12.3). Dabei ist `<PCA>` das Ergebnis einer Hauptkomponentenanalyse mit `prcomp()` oder `princomp()`. Für eine grafische Veranschaulichung der Hauptkomponenten selbst über die Hauptachsen des die gemeinsame Verteilung der Daten charakterisierenden Streuungsellipsoids s. Abschn. 14.6.8, Abb. 14.24. Ein mit `biplot(<PCA>)` erzeugter Biplot stellt die ersten beiden Hauptkomponenten und die Ladungs-Matrix gleichzeitig in einem Diagramm dar, das eine inhaltliche Interpretation der Hauptkomponentenanalyse fördern soll (Abb. 12.3). Die Definition eines Biplots ist nicht ganz einheitlich und kann zwischen je nach Software leicht unterschiedlich ausfallen, s. `?biplot`.

```

> plot(pca, type="b")                      # Scree-Plot
> biplot(pca)                            # stelle Biplot dar

```

Abbildung 12.3: Scree-Plot und Biplot zur Visualisierung der Ergebnisse einer Hauptkomponentenanalyse

Für die Darstellung im Biplot werden die Spalten der Hauptkomponenten \mathbf{Y} mit dem Vektor $1/\mathbf{f}$ skaliert. Dabei ist \mathbf{f} gleich den Singulärwerten von \mathbf{X} , also gleich der Wurzel aus den Eigenwerten der Kovarianzmatrix von \mathbf{X} multipliziert mit der Wurzel aus der Anzahl an Beobachtungen. Zur Darstellung der Variablen werden die zugehörigen normierten Eigenvektoren, gleich den Rechts-Singulärvektoren von \mathbf{X} , mit \mathbf{f} skaliert.

```

> (f <- svd_Xdot$d)                      # Skalierungsfaktor
[1] 17.229078  8.983144

> sqrt(eigVal)*sqrt(N)                   # Kontrolle
[1] 17.229078  8.983144

> head(Y %*% diag(1/f), n=3)            # Koordinaten HK für Biplot

```

```
[,1] [,2]
[1,] -0.24713328 -0.13036620
[2,] -0.05931238 -0.04123501
[3,]  0.08708010  0.18742743

> G %*% diag(f)                      # Koord. Variablen für Biplot
[,1] [,2]
[1,] -12.51065  6.044307
[2,] -11.59258 -6.522982
```

12.3 Faktorenanalyse

Der Faktorenanalyse liegt die Vorstellung zugrunde, dass sich die korrelativen Zusammenhänge zwischen vielen beobachtbaren Merkmalen X_j (mit $j = 1, \dots, p$) aus wenigen latenten Variablen F_k (den *Faktoren* mit $k = 1, \dots, q$) speisen, die ursächlich auf die Ausprägung der X_j wirken.³³ Die Wirkungsweise wird dabei als linear angenommen, d. h. die X_j sollen sich als Linearkombination der F_k zzgl. eines zufälligen Fehlers ergeben.³⁴ Weiterhin beruht die Faktorenanalyse auf der Annahme, dass sich bei unterschiedlichen Beobachtungsobjekten zwar die Ausprägungen der F_k unterscheiden, die den Einfluss der Faktoren auf die X_j charakterisierenden Koeffizienten der Linearkombinationen dagegen fest, also für alle Beobachtungsobjekte gleich sind.

Zur Vereinbarung der Terminologie sei \boldsymbol{x} der p -Vektor der beobachtbaren Merkmalsausprägungen einer Person, \boldsymbol{f} der q -Vektor ihrer Faktorausprägungen, $\boldsymbol{\epsilon}$ der p -Vektor der Fehler und $\boldsymbol{\Lambda}$ die $(p \times q)$ -*Ladungsmatrix* der zeilenweise zusammengestellten Koeffizienten der Linearkombinationen für jedes X_j . Das Modell geht von standardisierten X_j und F_k 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 $\boldsymbol{x} = \boldsymbol{\Lambda}\boldsymbol{f} + \boldsymbol{\epsilon}$ formulieren.³⁵ Die X_j abzüglich des Fehlervektors seien als *reduzierte* Variablen \hat{X}_j bezeichnet, für deren Werte einer Person $\hat{\boldsymbol{x}} = \boldsymbol{\Lambda}\boldsymbol{f}$ gilt.

Es sind nun zwei Varianten denkbar: Zum einen kann das Modell unkorrelierter Faktoren angenommen werden, deren Korrelationsmatrix \boldsymbol{K}_f dann 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 $\boldsymbol{\Lambda}$ gleichzeitig die auch als *Faktorstruktur* bezeichnete Korrelationsmatrix von \boldsymbol{x} und \boldsymbol{f} . Zum anderen ist das Modell potentiell korrelierter Faktoren mit Einträgen von \boldsymbol{K}_f ungleich 0 außerhalb der Hauptdiagonale

³³Für weitere Verfahren, die die Beziehungen latenter und beobachtbarer Variablen modellieren, vgl. den Abschnitt *Psychometric Models* der CRAN Task Views (Mair, 20120). Lineare Strukturgleichungsmodelle werden durch die Pakete `lavaan` (Rosseel, 2012), `OpenMx` (Boker et al., 2011) und `sem` (Fox, Nie & Byrnes, 2020) – mit ergänzenden Funktionen in `psych` – unterstützt.

³⁴In diesem Sinne ist die Faktorenanalyse das Gegenteil der Hauptkomponentenanalyse, in der die Hauptkomponenten Linearkombinationen der beobachtbaren Variablen sind.

³⁵Für n Personen seien die Werte auf den beobachtbaren Variablen zeilenweise in einer $(n \times p)$ -Matrix \boldsymbol{X} zusammengefasst, analog die Faktorwerte in einer $(n \times q)$ -Matrix \boldsymbol{F} und die Fehler in einer $(n \times p)$ -Matrix \boldsymbol{E} . Dann lautet das Modell $\boldsymbol{X} = \boldsymbol{F}\boldsymbol{\Lambda}^\top + \boldsymbol{E}$.

möglich. Hier bilden die Faktoren in einer geeigneten Darstellung keinen rechten Winkel, weshalb das Modell auch als schiefwinklig (*oblique*) bezeichnet wird. Die Faktorstruktur berechnet sich zu ΛK_f , zur Unterscheidung wird Λ selbst als *Faktormuster* bezeichnet.

Die Korrelationsmatrix der beobachtbaren Variablen \mathbf{K}_x ergibt sich im Modell korrelierter Faktoren als $\mathbf{K}_x = \Lambda K_f \Lambda^\top + \mathbf{D}_\epsilon$, wenn die Diagonalmatrix \mathbf{D}_ϵ die Kovarianzmatrix der Fehler ist. Im Modell unkorrelierter Faktoren vereinfacht sich die Gleichung zu $\mathbf{K}_x = \Lambda \Lambda^\top + \mathbf{D}_\epsilon$. Analog zum Vektor der reduzierten Variablen $\hat{\mathbf{x}}$ sei $\mathbf{K}_{\hat{\mathbf{x}}} = \Lambda K_f \Lambda^\top$ bzw. $\mathbf{K}_{\hat{\mathbf{x}}} = \Lambda \Lambda^\top$ als reduzierte Korrelationsmatrix bezeichnet. Dabei ist zu beachten, dass $\mathbf{K}_{\hat{\mathbf{x}}}$ nicht die Korrelationsmatrix von $\hat{\mathbf{x}}$ ist – die Diagonalelemente sind nicht 1, sondern ergänzen die Diagonalelemente von \mathbf{D}_ϵ (die Fehlervarianzen) zu 1. Die Diagonalelemente von $\mathbf{K}_{\hat{\mathbf{x}}}$ heißen Kommunalitäten. Sie sind gleichzeitig die Varianzen von Λf und damit ≥ 0 .

Bei der *exploratorischen* Faktorenanalyse besteht der Wunsch, bei vorausgesetzter Gültigkeit eines der beiden Modelle auf Basis vieler Messwerte der X_j eine Schätzung der Ladungsmatrix $\hat{\Lambda}$ sowie ggf. der Korrelationsmatrix der Faktoren \hat{K}_f zu erhalten.³⁶ Die Anzahl der latenten Faktoren sei dabei vorgegeben. Praktisch immer lassen sich jedoch durch Rotation viele Ladungs- und ggf. Korrelationsmatrizen finden, die zum selben Resultat i. S. von \mathbf{K}_x führen (s. u.).

Die Aufgabe kann analog zur Hauptkomponentenanalyse auch so verstanden werden, dass es die empirisch gegebene korrelative Struktur der X_j i. S. ihrer Korrelationsmatrix \mathbf{K}_x möglichst gut durch die Matrix $\hat{\Lambda}$ und ggf. \hat{K}_f zu reproduzieren gilt: $\hat{\Lambda} \hat{\Lambda}^\top$ bzw. $\hat{\Lambda} \hat{K}_f \hat{\Lambda}^\top$ sollte also höchstens auf der Diagonale nennenswert von \mathbf{K}_x abweichen und dort positive Einträge ≤ 1 besitzen. Die mit $\hat{\Lambda}$ geschätzten Einflüsse der Faktoren auf die beobachtbaren Variablen sollen gleichzeitig dazu dienen, die Faktoren inhaltlich mit Bedeutung zu versehen.

Die Faktorenanalyse wird mit `factanal()` durchgeführt, weitere Varianten enthält das Paket `psych`.

```
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 "varimax" für das Argument `rotation` liegt der Rechnung das Modell unkorrelierter Faktoren zugrunde.³⁷

Das folgende Beispiel behandelt den Fall, dass unkorrelierte Faktoren angenommen werden.

³⁶Die *konfirmatorische* Faktorenanalyse, bei der theoretische Erwägungen ein bestimmtes, auf Konsistenz mit den Daten zu testendes Λ vorgeben, ist mit Hilfe linearer Strukturgleichungsmodelle durchzuführen (Fußnote 33).

³⁷Weitere Rotationsarten, etwa für das Modell korrelierter Faktoren, stellt das Paket `GPArotation` (Bernaards & 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 geladen wurde.

Kapitel 12 Multivariate Verfahren

```

> N <- 200                                # Anzahl Beobachtungsobjekte
> P <- 6                                    # Anzahl beobachteter Variablen
> Q <- 2                                    # simulierte Anzahl von Faktoren

# 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=P, ncol=Q, 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(Q)                            # Modell unkorrelierter Faktoren
> mu <- c(5, 15)                           # hypothetisches Zentroid der Faktoren
> library(mvtnorm)                         # für rmvnorm()
> FF <- rmvnorm(N, mean=mu, sigma=Kf)       # simulierte Faktorwerte
> E <- rmvnorm(N, rep(0, P), diag(P))        # simulierte Fehler

# simulierte beobachtete Variablen im Modell der Faktorenanalyse
> X <- FF %*% t(Lambda) + E                # Datenmatrix
> (fa <- factanal(X, factors=2, scores="regression"))

Uniquenesses:
[1] 0.492 0.701 0.595 0.589 0.582 0.542

```

Loadings:

| | Factor1 | Factor2 |
|------|---------|---------|
| [1,] | 0.564 | -0.437 |
| [2,] | 0.546 | |
| [3,] | -0.145 | 0.619 |
| [4,] | -0.334 | 0.548 |
| [5,] | 0.325 | 0.559 |
| [6,] | -0.674 | |

| | Factor1 | Factor2 |
|----------------|---------|---------|
| SS loadings | 1.308 | 1.191 |
| Proportion Var | 0.218 | 0.199 |
| Cumulative Var | 0.218 | 0.417 |

Test of the hypothesis that 2 factors are sufficient.
The chi square statistic is 4.74 on 4 degrees of freedom.
The p-value is 0.315

Die Ausgabe von `factanal()` liefert unter der Überschrift `Uniqueness` die geschätzten Fehler-

varianzen der X_j , also die Einträge von \hat{D}_ϵ , 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. 12.4). Die Faktoren sind dabei entsprechend der durch sie aufgeklärten Varianz geordnet. Die berechnete Ladungsmatrix weicht leicht von der zur Simulation verwendeten 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 an, also die durch ihn bei allen Variablen aufgeklärte Varianz. Als Gesamtvarianz wird hier die Spur der Kovarianzmatrix der Variablen verstanden. Da die Variablen standardisiert sind, ist dies gleichzeitig die Spur ihrer Korrelationsmatrix, also die Anzahl der Variablen.³⁸ `Proportion Var` ist der vom Faktor aufgeklärte Anteil an der Gesamtvarianz und `Cumulative Var` der kumulative Anteil der durch die Faktoren aufgeklärten Varianz. Die Komponente `scores` der von `factanal()` erzeugten Liste enthält die hier über die Regressionsmethode geschätzten Faktorwerte. Die von `factanal()` ggf. verwendete Rotationsmatrix findet sich in der Komponente `rotmat`.

```
# Spaltensumme der quadrierten Ladungen -> aufgeklärte Varianz
> Lhat <- fa$loadings # geschätzte Ladungsmatrix
> colSums(Lhat^2)
  Factor1   Factor2
1.307778  1.191397

# Spaltensummen der quadrierten Ladungen geteilt durch Spur der
# Korrelationsmatrix -> Anteil der aufgeklärten Varianz
> colSums(Lhat^2) / sum(diag(cor(X)))
  Factor1   Factor2
0.2179629  0.1985661

> head(fa$scores, n=3) # geschätzte Faktorwerte
  Factor1   Factor2
[1,] -1.1439721  0.4063908
[2,] -1.2996309 -0.4458015
[3,]  0.9340950 -0.4548785
```

Schließlich folgt in der Ausgabe von `factanal()` die χ^2 -Teststatistik für den Test der H_0 , dass das Modell mit der angegebenen Zahl an Faktoren tatsächlich gilt. Ein kleiner p -Wert deutet daraufhin, dass die Daten nicht mit einer perfekten Passung konsistent sind. Weitere Hilfestellung zur Frage, wie viele Faktoren zu wählen sind, liefert etwa das Kaiser-Guttman-Kriterium, dem zufolge nur so viele Faktoren zu berücksichtigen sind, wie es Eigenwerte von \mathbf{K}_x größer 1 gibt, dem Mittelwert dieser Eigenwerte (Abb. 12.4).³⁹ Eine andere Herangehensweise besteht

³⁸Setzt man `rotation="none"`, ist dies ist bei der durch `factanal()` verwendeten Methode, um ein $\hat{\Lambda}$ zu erzeugen, gleichzeitig der zugehörige Eigenwert der geschätzten reduzierten Korrelationsmatrix $\hat{K}_{\hat{x}} = \hat{\Lambda}\hat{\Lambda}^\top$. Bei der Methode handelt es sich um die iterative Maximum-Likelihood Kommunalitätenschätzung. Bei Rotation oder anderen Schätzmethoden gilt diese Gleichheit dagegen nicht.

³⁹Die Summe der Eigenwerte einer Matrix ist gleich deren Spur (Abschn. 12.1.5) – im Fall der Korrelationsmatrix \mathbf{K}_x also gleich der Anzahl der Variablen p , da in der Diagonale überall 1 steht. Der Mittelwert der Eigenwerte

darin, den abfallenden Verlauf der Eigenwerte von \mathbf{K}_x zu betrachten, um einen besonders markanten Sprung zu identifizieren (Scree-Test). Hierbei hilft die grafische Darstellung dieser Eigenwerte als Liniendiagramm im Scree-Plot (Abb. 12.4).⁴⁰

Mit dem von `factanal()` zurückgegebenen Objekt sollen nun die Kommunalitäten sowie die geschätzte Korrelationsmatrix der beobachtbaren Variablen ermittelt werden. Schließlich folgt die grafische Darstellung der Faktorladungen der Variablen (Abb. 12.4).

```
> 1 - fa$uniquenesses          # Kommunalitäten: 1 - Fehlervarianzen
[1] 0.5084807 0.2985865 0.4045587 0.4113273 0.4180855 0.4581350

> rowSums(Lhat^2)             # Zeilensummen der quadrierten Ladungen
[1] 0.5084808 0.2985873 0.4045598 0.4113266 0.4180855 0.4581344

# geschätzte Korrelationsmatrix der Variablen: Lambda * Lambda^t + D_e
> KxEst <- Lhat %*% t(Lhat) + diag(fa$uniquenesses)

# grafische Darstellung der Faktorladungen
> plot(Lhat, xlab="Faktor 1", ylab="Faktor 2",
+       xlim=c(-1.1, 1.1), ylim=c(-1.1, 1.1), pch=20, asp=1,
+       main="Faktorenanalyse: Ladung & Faktoren")

> abline(h=0, v=0)            # Achsen mit Ursprung 0
> arrows(0, 0, c(1, 0), c(0, 1), col="blue", lwd=2) # zeichne Faktoren
> angles <- seq(0, 2*pi, length.out=200)      # Winkel Einheitskreis
> circ   <- cbind(cos(angles), sin(angles)) # Koordinaten Einheitskreis
> lines(circ)                         # Einheitskreis
```

Durch orthogonale oder schiefwinklige Rotation der Faktoren lassen sich aus der von `factanal()` geschätzten Ladungsmatrix und den zugehörigen Faktorwerten weitere Ladungsmatrizen berechnen, die eine ebenso gute Reproduktion der beobachteten Korrelationsmatrix \mathbf{K}_x liefern. Ist allgemein die Rotationsmatrix \mathbf{G} eine Orthogonalmatrix ($\mathbf{G}^\top = \mathbf{G}^{-1}$), ergibt sich die neue geschätzte Ladungsmatrix $\tilde{\Lambda}$ im Modell unkorrelierter Faktoren aus der alten durch $\tilde{\Lambda} = \hat{\Lambda}\mathbf{G}$ (Abb. 12.4). Die neue geschätzte Korrelationsmatrix $\tilde{\Lambda}\tilde{\Lambda}^\top + \hat{D}_\epsilon$ stimmt dann mit $\hat{\Lambda}\hat{\Lambda}^\top + \hat{D}_\epsilon$ überein.

```
> ang <- pi/3                  # Rotationswinkel

# Matrix für orthogonale Rotation der Faktoren
> G <- matrix(c(cos(ang), sin(ang), -sin(ang), cos(ang)), nrow=2)
> (Lrot <- Lhat %*% G)         # neue Ladungsmatrix
     [,1]      [,2]
[1,] -0.09662051 -0.706502158
[2,]  0.25977831 -0.480731207
```

ist damit $\frac{1}{p}p = 1$.

⁴⁰Weitere Verfahren, die der Klärung der geeigneten Anzahl von Faktoren dienen sollen, sind die Parallelanalyse mit `fa.parallel()` aus dem Paket `psych` sowie das *very simple structure* Verfahren mit `VSS()` aus demselben Paket.

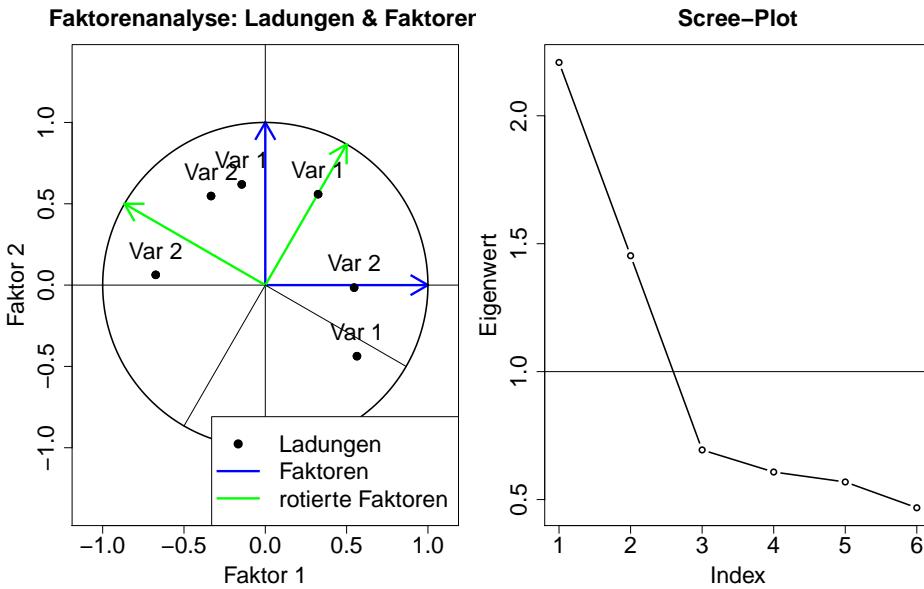


Abbildung 12.4: Faktorenanalyse: Faktorladungen der Variablen sowie rotierte Faktoren. Scree-Plot der Eigenwerte der Korrelationsmatrix der Daten

```
[3,] 0.46393903 0.435109654
[4,] 0.30755796 0.562791872
[5,] 0.64659330 -0.001618208
[6,] -0.28224873 0.615199187
```

```
# 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, ])

# Variablen beschriften und Legende einfügen
> text(Lhat[, 1], Lhat[, 2]+0.06, labels=paste("Var", 1:Q))
> 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 \mathbf{G} eine schiefwinklige Rotationsmatrix, ergibt sich die neue geschätzte Ladungsmatrix $\tilde{\Lambda}$ aus der alten durch $\tilde{\Lambda} = \hat{\Lambda}(\mathbf{G}^\top)^{-1}$. Die neue geschätzte Korrelationsmatrix der Faktoren ist $\tilde{\mathbf{K}}_f = \mathbf{G}^\top \hat{\mathbf{K}}_f \mathbf{G}$. Die geschätzte Faktorstruktur, also die Matrix der Korrelationen zwischen beobachtbaren Variablen und Faktoren, berechnet sich hier als $\hat{\Lambda} \hat{\mathbf{K}}_f \mathbf{G}$. Die neue geschätzte Korrelationsmatrix $\tilde{\Lambda} \tilde{\mathbf{K}}_f \tilde{\Lambda}^\top + \hat{\mathbf{D}}_\epsilon$ stimmt dann mit $\hat{\Lambda} \hat{\mathbf{K}}_f \hat{\Lambda}^\top + \hat{\mathbf{D}}_\epsilon$ überein.

```
# Matrix für schiefwinklige Rotation der Faktoren
> G <- matrix(c(0.7071, 0.7071, -0.866, 0.5), nrow=2)

# neue Korrelationsmatrix der Faktoren (bisher: Einheitsmatrix)
> KfRot <- t(G) %*% diag(Q) %*% G

# neue geschätzte Ladungsmatrix
> Lrot <- Lhat %*% solve(t(G))

# neue geschätzte Faktorstruktur
> facStruct <- Lhat %*% diag(Q) %*% 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
```

12.4 Multidimensionale Skalierung

Die multidimensionale Skalierung ist ein weiteres Verfahren zur Dimensionsreduktion mit dem Ziel, Beobachtungsobjekte in einem Raum so anzurordnen, dass ihre räumliche Lage zueinander jeweils ihre globale paarweise Unähnlichkeit widerspiegelt. Der Raum soll von möglichst wenigen Merkmalsdimensionen aufgespannt sein. Eine gewählte Anordnung wird als *Konfiguration* bezeichnet. Die inhaltliche Bedeutung der resultierenden Merkmalsdimensionen muss aus den Ergebnissen indirekt erschlossen werden.

Die Ausgangssituation unterscheidet sich von jener in der Hauptkomponentenanalyse⁴¹ und der Faktorenanalyse dahingehend, dass zunächst unbekannt ist, bzgl. welcher Merkmale Objekte beurteilt werden, wenn eine Aussage über ihre generelle Ähnlichkeit zu anderen Objekten getroffen wird. Anders formuliert sind Anzahl und Bedeutung der Variablen, für die Objekte Werte besitzen, nicht gegeben. Dementsprechend werden in einer empirischen Erhebung oft auch nicht separat bestimmte Eigenschaften von einzelnen Objekten gemessen – vielmehr gilt es, in einem Paarvergleich das Ausmaß der Unähnlichkeit von je zwei Objekte zu beurteilen.

Die *metrische* multidimensionale Skalierung sucht nach einer Konfiguration, so dass die paarweise Unähnlichkeit möglichst gut mit dem jeweiligen euklidischen Abstand zwischen den Punkten übereinstimmt.⁴² Sie wird mit `cmdscale()` durchgeführt.

⁴¹Tatsächlich sind beide Verfahren eng verwandt, s. Abschn. 12.2.1, Fußnote 28 sowie Abschn. 12.2.2.

⁴²Andere Metriken als der euklidische Abstand sind auch möglich. Die nichtmetrische multidimensionale Skalierung wird durch `monoMDS()` aus dem `vegan` Paket (Oksanen et al., 2019) bereit gestellt.

```
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 i. S. 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 (Abschn. 12.1.3). Für `k` ist anzugeben, durch wie viele Variablen der Raum aufgespannt werden soll, in dem `cmdscale()` die Objekte anordnet und ihre euklidischen Distanzen berechnet. Liegen Unähnlichkeitswerte zwischen n Objekten vor, kann die gewünschte Dimension höchstens $n - 1$ sein, Voreinstellung ist 2.

Die Ausgabe umfasst eine $(n \times k)$ -Matrix mit den n Koordinaten der Objekte im k -dimensionalen Variablenraum, wobei das Zentroid im Ursprung des Koordinatensystems liegt. Mit `x.ret=TRUE` enthält das Ergebnis zwei zusätzliche Informationen: 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. Das Ergebnis ist insofern uneindeutig, als Rotation, Spiegelung und gleichförmige Verschiebung der Punkte ihre Distanzen zueinander nicht ändern.

Als Beispiel liegen die Straßen-Entfernungen zwischen deutschen Städten vor, die die multidimensionale Skalierung auf zwei Dimensionen räumlich anordnen soll.

```
# Städte, deren Entfernungen berücksichtigt werden
> cities <- c("Augsburg", "Berlin", "Dresden", "Hamburg", "Hannover",
+           "Karlsruhe", "Kiel", "München", "Rostock", "Stuttgart")

# erstelle Distanzmatrix aus Vektor der Entfernungen
> n      <- length(cities)                      # Anzahl der Städte
> dstMat <- matrix(numeric(n^2), nrow=n)        # zunächst leere Matrix
> 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[, n:1])[, n: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 grafische Darstellung der ermittelten Koordinaten erlaubt es, das Ergebnis mit den tatsächlichen Positionen zu vergleichen (Abb. 12.5). Hier ist zu erkennen, dass das Ergebnis für die meisten Städte gut mit der realen Topografie übereinstimmt, wenn man die Möglichkeit einer Drehung im Uhrzeigersinn um 90° berücksichtigt und dann die West-Ost-Richtung spiegelt.

```
> xLims <- range(mds[, 1]) + c(0, 250)      # x-Achsenbereich
> plot(mds, xlim=xLims, xlab="Nord-Süd", ylab="Ost-West", pch=16,
+       main="Anordnung der Städte nach MDS")
# füge Städtenamen hinzu
> text(mds[, 1]+50, mds[, 2], adj=0, labels=cities)
```

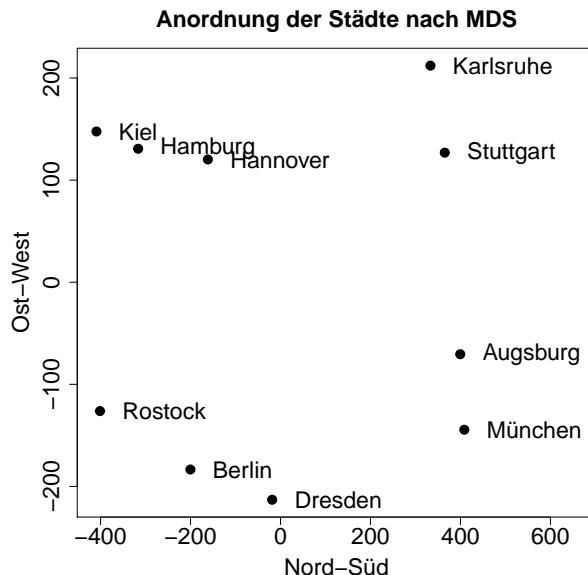


Abbildung 12.5: Ergebnis der multidimensionalen Skalierung auf Basis der Distanzen zwischen deutschen Städten

12.5 Multivariate multiple Regression

Die univariante multiple Regression (Abschn. 6.3) lässt sich zur multivariaten multiplen Regression verallgemeinern, bei der durch p Prädiktoren X_j (mit $j = 1, \dots, p$) nicht nur ein Kriterium Y vorhergesagt werden soll, sondern r Kriteriumsvariablen Y_l (mit $l = 1, \dots, r$) gleichzeitig. Die Parameterschätzung im multivariaten Fall ist auf den univariaten zurückführbar: Die i. S. der

geringsten Quadratsumme der Residuen optimale Parameterwahl geht aus der Zusammenstellung der r unabhängig voneinander durchgeführten Regressionen von jeweils einem Kriterium Y_l auf alle Prädiktoren X_j hervor.

In der Berechnung der multivariaten multiplen Regression mittels `lm()` ist die Modellformel zunächst wie im univariaten Fall aufzubauen. Im Unterschied dazu ist das Kriterium auf der linken Seite der Formel 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 (i. S. eines geeigneten quantitativen Maßes) vorhergesagt werden.

```
> N      <- 100                      # Anzahl Versuchspersonen
> height <- rnorm(N, 175, 7)        # Prädiktor 1
> age    <- rnorm(N, 30, 8)         # Prädiktor 2
> sport   <- abs(rnorm(N, 60, 30)) # Prädiktor 3

# modellgerechte Simulation beider Kriterien
> weight <- 0.5*height - 0.3*age - 0.4*sport + 10 + rnorm(N, 0, 3)
> health <- -0.3*age + 0.6*sport + rnorm(N, 4)
> Y      <- cbind(weight, health)    # Matrix der Kriterien

# multivariat formulierte Modellformel für lm()
> (fitM <- 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: Koeffizienten beider univariater Regressionen separat
> coef(lm(weight ~ height + age + sport))
(Intercept) height      age      sport
10.4124    0.5072   -0.3473  -0.4089

> coef(lm(health ~ height + age + sport))
Coefficients:
(Intercept) height      age      sport
2.44980    0.01031   -0.29678  0.59826
```

Die multivariate Regressionsanalyse als inferenzstatistischer Test liefert i. d. R. andere Ergebnisse als die separaten univariaten Regressionsanalysen (Abschn. 12.9.6, 12.9.7). Das von `lm()` erzeugte Objekt muss dazu an die Funktion `summary(manova())` übergeben werden, die `aov()`

(Abschn. 7.4.2) auf den multivariaten Fall verallgemeinert und ebenso wie diese aufzurufen ist. Mit dem Argument `test` von `summary()` lassen sich verschiedene multivariate Teststatistiken wählen, etwa die Hotelling-Lawley-Spur mit "Hotelling-Lawley" (für weitere vgl. `?summary.manova` und Abschn. 12.9.7). Anders als bei den univariaten Tests ist beim multivariaten Test der Parameter die Reihenfolge der Prädiktoren relevant (für die manuelle Kontrolle s. Abschn. 12.9.8).

```
> summary(manova(fitM), test="Hotelling-Lawley")
      Df Hotelling-Lawley approx F num Df den Df    Pr(>F)
height     1          8.056    382.7      2      95 < 2.2e-16 ***
age       1          6.897    327.6      2      95 < 2.2e-16 ***
sport      1         257.924   12251.4      2      95 < 2.2e-16 ***
Residuals 96
```

12.6 Hotellings T^2

12.6.1 Test für eine Stichprobe

Hotellings T^2 -Test für eine Stichprobe prüft die Datenvektoren von r gemeinsam normalverteilten Variablen daraufhin, ob sie mit der H_0 konsistent sind, dass ihr theoretisches Zentroid μ mit einem bestimmten Vektor μ_0 übereinstimmt. Der zweiseitige univariate t -Test (Abschn. 7.3.1) ist zu Hotellings T^2 -Test äquivalent, wenn nur Daten einer Zielgröße vorliegen. Der Test lässt sich mit der aus der univariaten Varianzanalyse (Abschn. 7.4.3) bekannten Funktion

```
anova(lm(<AV> ~ 1, data=<Datensatz>), test="Hotelling-Lawley")
```

durchführen. In der für `lm()` anzugebenden Formel $\langle AV \rangle \sim 1$ sind die $\langle AV \rangle$ Werte dabei multivariat, d. h. in Form einer spaltenweise aus den einzelnen Variablen zusammengestellten Matrix zu übergeben. Von den Zeilenvektoren der Daten muss zunächst das Zentroid unter H_0 abgezogen werden, damit die Formel zum Test der $H_0 : \mu - \mu_0 = \mathbf{0}$ führt, die äquivalent zu $\mu = \mu_0$ ist. Stammen die Variablen aus einem Datensatz, ist dieser unter `data` zu nennen. Weiter ist das Argument `test` von `anova()` zu verwenden, um eine multivariate Teststatistik auszuwählen.⁴³ Dies kann "Hotelling-Lawley" für die Hotelling-Lawley-Spur sein, wobei die Wahl hier nicht relevant ist: Wenn nur eine Bedingung vorliegt, sind alle auswählbaren Teststatistiken zu Hotellings T^2 -Statistik äquivalent.

Im Beispiel sollen die Werte zweier Variablen betrachtet werden. Zunächst sind nur die Daten aus einer Stichprobe (von später dreien) relevant.

```
> muH0 <- c(-3, 2)                                # Zentroid unter H0
# theoretische 2x2 Kovarianzmatrix für Zufallsvektoren
> sigma <- matrix(c(16,-2, -2,9), byrow=TRUE, ncol=2)
> mu11 <- c(-4, 4)                                # theoretisches Zentroid
> Nj    <- c(15, 25, 20)                           # Gruppengrößen
```

⁴³Bei der multivariaten Formulierung des Modells wird intern aufgrund der generischen `anova()` Funktion automatisch `anova.mlm()` verwendet, ohne dass dies explizit angegeben werden muss (Abschn. 17.3.7).

```
# Datenmatrix für 1. Gruppe mit Variablen in den Spalten
> library(mvtnorm)                                # für rmvnorm()
> Y11 <- round(rmvnorm(Nj[1], mean=mu11, sigma=sigma))

# ziehe Zentroid unter H0 von allen Zeilenvektoren ab
> Y11ctr <- scale(Y11, center=muH0, scale=FALSE)
> (anRes <- anova(lm(Y11ctr ~ 1), test="Hotelling-Lawley"))
   Df Hotelling-Lawley approx F num Df den Df Pr(>F)
(Intercept) 1           1.3987   9.0917      2     13 0.003390 **
Residuals   14


```

Die Ausgabe nennt die Hotelling-Lawley-Spur tr_{HL} in der Spalte **Hotelling-Lawley**, die F -verteilte Teststatistik $\frac{n-r}{r}\text{tr}_{\text{HL}}$ in der Spalte **approx F**. Weiterhin sind die Freiheitsgrade der zugehörigen F-Verteilung (**num Df**, **den Df**) und der entsprechende p -Wert (**Pr(>F)**) aufgeführt.

Das Ergebnis lässt sich auch manuell prüfen. Hotellings T^2 ist gleich dem n -fachen der quadrierten Mahalanobisdistanz zwischen dem Zentroid der Daten und dem Zentroid unter H_0 bzgl. der korrigierten Kovarianzmatrix der Daten (Abschn. 12.1.4). Außerdem gilt $T^2 = (n - 1)\text{tr}_{\text{HL}}$.

```
> n    <- nrow(Y11)                               # Stichprobengröße
> ctr <- colMeans(Y11)                            # Zentroid

# Hotellings T^2 Teststatistik
> (T2 <- n * (t(ctr-muH0) %*% solve(cov(Y11)) %*% (ctr-muH0)))
[1,] 19.58203

# Kontrolle: n-faches der quadrierten Mahalanobisdistanz
> n * mahalanobis(ctr, muH0, cov(Y11))
[1] 19.58203

# Hotelling-Lawley-Spur
> Tr_HL <- anRes[1, "Hotelling-Lawley"]
> (n-1) * Tr_HL
[1] 19.58203

> r      <- ncol(Y11)                            # Anzahl Variablen
> (F_HL <- ((n-r) / r) * Tr_HL)                 # Teststatistik
[1] 9.091655

> (pVal <- pf(F_HL, r, n-r, lower.tail=FALSE))   # p-Wert
[1] 0.003389513
```

12.6.2 Test für zwei unabhängige Stichproben

Hotellings T^2 -Test für zwei unabhängige Stichproben prüft die in zwei Bedingungen erhobenen Datenvektoren von r gemeinsam normalverteilten Variablen mit identischen Kovarianzmatrizen daraufhin, ob sie mit der H_0 konsistent sind, dass ihre theoretischen Zentroide übereinstimmen. Der Test ist äquivalent zum univariaten t -Test für zwei unabhängige Stichproben, wenn nur Werte einer Zielgröße vorliegen (Abschn. 7.3.2).

Hotellings T^2 -Test lässt sich wie der Test für eine Stichprobe in Abschn. 12.6.1 durchführen. Dabei sind in der Modellformel $\langle AV \rangle \sim \langle UV \rangle$ für $\text{lm}()$ die $\langle AV \rangle$ Werte beider Gruppen in Form einer spaltenweise aus den einzelnen Variablen zusammengestellten Matrix zu nennen, 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, wobei auch bei zwei Gruppen alle multivariaten Teststatistiken äquivalent sind.

Das im vorangehenden Abschnitt begonnene Beispiel soll nun um die in einer zweiten Bedingung erhobenen Daten der betrachteten Variablen erweitert werden. Als Teststatistik wird hier wieder die Hotelling-Lawley-Spur trHL gewählt.

```
> mu21 <- c(3, 3)                      # Zentroid Zufallsvektoren 2. Bedingung

# Datenmatrix aus 2. Bedingung mit Variablen in den Spalten
> library(mvtnorm)                     # für rmvnorm()
> Y21 <- round(rmvnorm(Nj[2], mean=mu21, sigma=sigma))
> Yht <- rbind(Y11, Y21)                # Gesamt-Datenmatrix beider Bedingungen

# codiere für jede Zeile der Datenmatrix die zugehörige Bedingung
> IVht <- factor(rep(1:2, Nj[1:2]))
> anova(lm(Yht ~ IVht), test="Hotelling-Lawley")
Analysis of Variance Table
  Df Hotelling-Lawley approx F num Df den Df    Pr(>F)
(Intercept)  1          2.4627   45.560      2     37 1.049e-10 ***
IVht         1          0.6073   11.235      2     37 0.0001539 ***
Residuals   38


```

Die Ausgabe ist analog zu Abschn. 12.6.1, die Kennwerte des Tests des Faktors stehen in der mit seinem Namen bezeichneten Zeile. Für die Teststatistik gilt $F = \frac{n_1+n_2-r-1}{r} \text{trHL}$.

Schließlich existiert mit `manova()` eine Funktion, die `aov()` (Abschn. 7.4.2) auf den multivariaten Fall verallgemeinert und ebenso wie diese aufzurufen ist. Die Modellformel ist dieselbe wie beim Aufruf von `lm()`.

```
> (sumRes <- summary(manova(Yht ~ IVht), test="Hotelling-Lawley"))
  Df Hotelling-Lawley approx F num Df den Df    Pr(>F)
IVht         1          0.6073   11.235      2     37 0.0001539 ***
Residuals   38
```

Das Ergebnis lässt sich auch manuell prüfen. Hotellings T^2 ist bis auf einen Faktor gleich der quadrierten Mahalanobisdistanz beider Zentroide bzgl. einer geeigneten Schätzung der Kovarianzmatrix der Differenzvektoren. Außerdem gilt $T^2 = (n_1 + n_2 - 2)\text{tr}_{\text{HL}}$.

```

> n1    <- nrow(Y11)                      # Gruppengröße 1
> n2    <- nrow(Y21)                      # Gruppengröße 2
> ctr1 <- colMeans(Y11)                   # Zentroid 1. Bedingung
> ctr2 <- colMeans(Y21)                   # Zentroid 2. Bedingung

# unkorrigierte Kovarianzmatrizen aus beiden Bedingungen
> S1 <- cov.wt(Y11, method="ML")$cov
> S2 <- cov.wt(Y21, method="ML")$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))
> T2
[1] 23.07731

# Kontrolle: quadrierte Mahalanobisdistanz beider Zentroide bzgl. Su
> ((n1*n2) / (n1+n2)) * mahalanobis(ctr1, ctr2, Su)
[1] 23.07731

# Hotelling-Lawley-Spur
> Tr_HL <- sumRes$stats["IVht", "Hotelling-Lawley"]

# Kontrolle: T^2 = (n1+n2-2) * Hotelling-Lawley-Spur
> (n1+n2-2) * Tr_HL
[1] 23.07731

> r      <- ncol(Y11)                      # Anzahl Variablen
> (F_HL <- ((n1+n2-r-1) / r) * Tr_HL)      # Teststatistik
[1] 11.23501

> (pVal <- pf(F_HL, r, n1+n2-r-1, lower.tail=FALSE))      # p-Wert
[1] 0.0001538927

```

12.6.3 Test für zwei abhängige Stichproben

Analog zum univariaten t -Test (Abschn. 7.3.3) kann der multivariate T^2 -Test für zwei abhängige Stichproben auf die Situation einer Stichprobe zurückgeführt werden: Dazu bildet man variablenweise pro Beobachtungsobjekt die Differenz der Daten aus beiden Bedingungen und stellt die Differenzvariablen spaltenweise zu einer neuen Matrix zusammen. Hotellings T^2 -Test für eine Stichprobe ist mit diesen Differenzdaten dann mit der H_0 durchzuführen, dass ihr

theoretisches Zentroid $\mu_0 = \mathbf{0}$ ist. Anders als in Abschn. 12.6.1 entfällt hier wegen $\mu_0 = \mathbf{0}$ die Notwendigkeit, μ_0 von den Zeilenvektoren der Datenmatrix der Differenzvariablen abzuziehen, bevor `anova()` aufgerufen wird.

```

> N      <- 20                                # Anzahl Personen
> P      <- 2                                 # Anzahl Messzeitpunkte
> Y1t0  <- rnorm(N, mean=90,  sd=15)        # AV 1 Zeitpunkt t0
> Y1t1  <- rnorm(N, mean=100, sd=15)         # AV 1 Zeitpunkt t1
> Y2t0  <- rnorm(N, mean=85,  sd=15)          # AV 2 Zeitpunkt t0
> Y2t1  <- rnorm(N, mean=105, sd=15)         # AV 2 Zeitpunkt t1

# Faktor, der im Long-Format den Messzeitpunkt codiert
> IV <- factor(rep(1:P, each=N), labels=c("t0", "t1"))
> id <- factor(rep(1:N, times=P))           # Faktor Personen-ID

# Datensatz im Long-Format
> Ydf <- data.frame(id, Y1=c(Y1t0, Y1t1), Y2=c(Y2t0, Y2t1), IV)

# bilde pro Variable personenweise Differenz zwischen t0 und t1
> dfDiff <- aggregate(cbind(Y1, Y2) ~ id, data=Ydf, FUN=diff)
> DVdiff <- data.matrix(dfDiff[, -1])          # als Matrix
> anova(lm(DVdiff ~ 1), test="Hotelling-Lawley")
Analysis of Variance Table
      Df Hotelling-Lawley approx F num Df den Df   Pr(>F)
(Intercept) 1            0.87529    7.8777     2      18 0.003486 ***
Residuals   19

```

12.6.4 Univariate Varianzanalyse mit abhängigen Gruppen (RB- p)

Daten einer eigentlich univariaten Varianzanalyse mit p abhängigen Gruppen (RB- p Design, Abschn. 7.5) können auch multivariat ausgewertet werden, wodurch die Voraussetzung der Zirkularität entfällt. Hierfür sind zunächst blockweise alle $\binom{p}{2} = \frac{p(p-1)}{2}$ Differenzvariablen zwischen je zwei Gruppen zu bilden und spaltenweise zu einer Matrix zusammenzufassen. Deren Spalten sind für $p > 2$ jedoch linear abhängig, da es höchstens $p - 1$ linear unabhängige Differenzvariablen gibt. Um diese Redundanz zu beseitigen, müssen $p - 1$ Spalten der Matrix ausgewählt werden, in deren Differenzen insgesamt alle p Gruppen eingeflossen sind. Die übrigen Spalten der ursprünglichen Matrix der Differenzvariablen werden gestrichen. Für die beibehaltenen Differenzvariablen ist Hotellings T^2 -Test für eine Stichprobe mit der H_0 durchzuführen, dass ihr Zentroid der Vektor $\mathbf{0}$ ist. Das Vorgehen ist damit analog zu jenem bei Hotellings T^2 -Test für zwei abhängige Stichproben.

Als Beispiel diene jenes aus Abschn. 7.5 mit einer zu vier Messzeitpunkten erhobenen Zielgröße.

```

> P      <- 4                                # Anzahl Messzeitpunkte
> DVw <- cbind(DV_t1, DV_t2, DV_t3, DV_t4)  # Datenmatrix Wide-Format

```

```
# alle paarweisen Differenzen der Spalten der Datenmatrix
> diffMat <- combn(1:P, 2, function(x) { DVw[ , x[1]] - DVw[ , x[2]] })

# wähle P-1 Differenzvariablen, die nicht redundant sein dürfen
> DVdiff <- diffMat[ , 1:(P-1), drop=FALSE]
> anova(lm(DVdiff ~ 1), test="Hotelling-Lawley")
Analysis of Variance Table
      Df Hotelling-Lawley approx F num Df den Df Pr(>F)
(Intercept) 1           1.2474    2.9106     3      7 0.1104
Residuals    9


```

Alternativ eignet sich die in Abschn. 7.5.3 vorgestellte Funktion `Anova()` aus dem `car` Paket, für die Daten im Wide-Format benötigt werden. Für den multivariaten Test ist hier bei der Anwendung von `summary()` das Argument `multivariate=TRUE` zu setzen. Alle ausgegebenen Teststatistiken sind hier äquivalent zum T^2 -Test, weil letztlich ein multivariater Test für nur eine Gruppe vorliegt.

```
> fitRBp <- lm(DVw ~ 1)                                # Zwischen-Gruppen Design
> intraRBp <- data.frame(IV=gl(P, 1))                 # Intra-Gruppen Design
> library(car)                                         # für Anova()
> AnovaRBp <- Anova(fitRBp, idata=intraRBp, idesign=~IV)
> summary(AnovaRBp, multivariate=TRUE, univariate=FALSE) # ...


```

12.7 Multivariate Varianzanalyse (MANOVA)

12.7.1 Einfaktorielle MANOVA

Die einfaktorielle multivariate Varianzanalyse prüft die in mehreren Bedingungen einer Gruppierungsvariable erhobenen Datenvektoren von r gemeinsam normalverteilten Variablen mit identischen Kovarianzmatrizen daraufhin, ob sie mit der H_0 konsistent sind, dass ihre theoretischen Zentroide übereinstimmen. Der Test ist äquivalent zur univariaten einfaktoriellen Varianzanalyse, wenn nur die Daten einer Zielgröße vorliegen (Abschn. 7.4).

Zur Durchführung eignet sich wie bei Hotellings T^2 -Test für zwei Stichproben die `manova()` Funktion als Verallgemeinerung von `aov()`. Dabei ist die Modellformel links der \sim multivariat zu formulieren, also eine spaltenweise aus den Variablen zusammengestellte Datenmatrix mit den Beobachtungsobjekten in den Zeilen zu übergeben. Als Gruppierungsvariable auf der rechten Seite der \sim ist ein Faktor zu nennen, der für jede Zeile der Datenmatrix 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 (Abschn. 12.9.7). Bei zwei Gruppen sind alle Teststatistiken äquivalent.

Das Beispiel aus Abschnitt 12.6.1 und 12.6.2 soll nun um die in einer dritten Bedingung erhobenen Daten der betrachteten beiden Variablen erweitert werden (für die manuelle Kontrolle s. Abschn. 12.9.9).

```
> mu31 <- c(1, -1)                      # Zentroid Zufallsvektoren 3. Bedingung

# Datenmatrix aus 3. Bedingung mit Variablen in den Spalten
> library(mvtnorm)                      # für rmvnorm()
> Y31    <- round(rmvnorm(Nj[3], mean=mu31, sigma=sigma))
> Ym1    <- rbind(Y11, Y21, Y31)        # Matrix der Daten aus allen Gruppen
> IVman <- factor(rep(1:3, Nj))         # Bedingung für jede Zeile Datenmatrix
> manRes1 <- manova(Ym1 ~ IVman)        # Anpassung multivar. Varianzanalyse
> summary(manRes1, test="Wilks")          # Wilks' Lambda
   Df      Wilks  approx F  num Df  den Df  Pr(>F)
IVman       2  0.42011     15.199      4      112  5.89e-10 ***
Residuals  57


```

12.7.2 Zweifaktorielle MANOVA

Die zweifaktorielle multivariate Varianzanalyse ist wie die einfaktorielle mit `manova()` durchzuführen, lediglich die Spezifikation der Modellformel rechts der `~` erweitert sich um die zusätzlich zu berücksichtigende Gruppierungsvariable. Links der `~` steht in der Modellformel weiterhin eine spaltenweise aus den Variablen zusammengesetzte Datenmatrix mit den Beobachtungsobjekten in den Zeilen. Als Gruppierungsvariablen können nun die Faktoren genannt werden, deren Stufen die Bedingungskombinationen festlegen, in denen die Daten erhoben wurden. Jeder Faktor gibt für jede Zeile der Datenmatrix an, aus welcher Bedingung bzgl. der durch ihn codierten Gruppierungsvariable der Datenvektor stammt (für die manuelle Kontrolle s. Abschn. 12.9.10).

```
# Daten aus den 3 Bedingungen der UV 1 in 2. Stufe der UV 2
> mu12 <- c(-1, 4)                      # Zentroid Zufallsvektoren 1. Bedingung
> mu22 <- c( 4, 8)                      # Zentroid Zufallsvektoren 2. Bedingung
> mu32 <- c( 4, 0)                      # Zentroid Zufallsvektoren 3. Bedingung
> library(mvtnorm)                      # für rmvnorm()
> Y12 <- round(rmvnorm(Nj[1], mu12, sigma))
> Y22 <- round(rmvnorm(Nj[2], mu22, sigma))
> Y32 <- round(rmvnorm(Nj[3], mu32, sigma))

# vollständige Datenmatrix aus den 3x2 Bedingungen (Variablen=Spalten)
> Ym2 <- rbind(Ym1, Y12, Y22, Y32)

# codiere: jede Zeile Datenmatrix: zugehörige Bedingung beider Faktoren
> IV1      <- rep(IVman, times=2)
> IV2      <- factor(rep(1:2, each=sum(Nj)))
> manRes2 <- manova(Ym2 ~ IV1*IV2)        # Anpassung MANOVA
> summary(manRes2, test="Pillai")           # Pillai-Bartlett-Spur
   Df      Pillai  approx F  num Df  den Df  Pr(>F)
```

| | | | | | | |
|-----------|-----|---------|--------|---|-----|---------------|
| IV1 | 2 | 0.72468 | 32.389 | 4 | 228 | < 2.2e-16 *** |
| IV2 | 1 | 0.18107 | 12.493 | 2 | 113 | 1.254e-05 *** |
| IV1:IV2 | 2 | 0.15762 | 4.876 | 4 | 228 | 0.0008615 *** |
| Residuals | 114 | | | | | |

Auch bei der multivariaten zweifaktoriellen Varianzanalyse ist im Fall ungleicher Zellbesetzungen zu beachten, dass R in der Voreinstellung Quadratsummen vom Typ I berechnet (Abschn. 7.6.2, 12.9.6). Manova() aus dem car Paket erlaubt es, analog zur Verwendung von Anova() (Abschn. 7.5.3), Quadratsummen vom Typ II und III zu berechnen. Zuvor ist das contrasts Argument für lm() notwendig, um von der Dummy-Codierung (Treatment-Kontraste) zur Effektcodierung der Faktoren zu wechseln (Abschn. 12.9.2).

```
# wechsle von Dummy-Codierung der Faktoren zur Effektcodierung
> fitIII <- lm(Ym2 ~ IV1*IV2,
+                 contrasts=list(IV1=contr.sum, IV2=contr.sum))

> library(car)                                     # für Manova()
> Manova(fitIII, type="III")                      # Quadratsummen Typ III
Type III MANOVA Tests: Pillai test statistic
      Df test stat approx F num Df den Df   Pr(>F)
(Intercept) 1  0.37696  34.184     2    113 2.456e-12 ***
IV1         2  0.43976  16.066     4    228 1.325e-11 ***
IV2         1  0.05937   3.566     2    113 0.0314819 *
IV1:IV2     2  0.15762   4.876     4    228 0.0008615 ***
```

12.8 Diskriminanzanalyse

Die Diskriminanzanalyse bezieht sich auf dieselbe Erhebungssituation wie die einfaktorielle MANOVA und teilt deren Voraussetzungen (Abschn. 12.7.1): Beobachtungsobjekte aus p Gruppen liefern Werte auf r quantitativen Zielgrößen Y_l . Diese Variablen seien in jeder Gruppe multinormalverteilt mit derselben invertierbaren Kovarianzmatrix, aber u. U. abweichenden Erwartungswertvektoren. Anlass zur Anwendung kann eine zuvor durchgeführte signifikante MANOVA sein, deren unspezifische Alternativhypothese offen lässt, wie genau sich die Gruppenzentroide unterscheiden.

Die Diskriminanzanalyse erzeugt $\min(p - 1, r)$ viele Linearkombinationen $LD = b_0 + b_1 Y_1 + \dots + b_l Y_l + \dots + b_r Y_r$ der ursprünglichen Variablen, auf denen sich die Gruppenunterschiede im folgenden Sinne besonders deutlich zeigen.⁴⁴ Die als *Diskriminanzfunktionen*, oder auch als *Fishers lineare Diskriminanten* bezeichneten LD sind unkorrelierte Variablen, deren jeweiliger F -Bruch aus der einfaktoriellen Varianzanalyse mit der Diskriminanten als Zielgröße sukzessive maximal ist. Hier zeigt sich eine Ähnlichkeit zur Hauptkomponentenanalyse (Abschn. 12.2), die die Gruppenzugehörigkeit der Objekte jedoch nicht berücksichtigt und neue unkorrelierte Variablen mit schrittweise maximaler Varianz bildet.

⁴⁴Lässt man auch quadratische Funktionen der ursprünglichen Variablen zu, ergibt sich die quadratische Diskriminanzanalyse. Sie wird mit qda() aus dem MASS Paket berechnet.

Die Diskriminanzanalyse lässt sich auch mit der Zielsetzung durchführen, Objekte anhand mehrerer Merkmale möglichst gut hinsichtlich eines bestimmten Kriteriums klassifizieren zu können. Hierfür wird zunächst ein Trainingsdatensatz benötigt, von dessen Objekten sowohl die Werte der diagnostischen Variablen als auch ihre Gruppenzugehörigkeit bekannt sind. Mit diesem Datensatz werden die Koeffizienten der Diskriminanzfunktionen bestimmt, die zur späteren Klassifikation anderer Objekte ohne bekannte Gruppenzugehörigkeit dienen.⁴⁵ Die lineare Diskriminanzanalyse lässt sich mit `lda()` aus dem MASS Paket durchführen.

```
lda(formula=<Modellformel>, data=<Datensatz>,
    CV=<Kreuzvalidierung>, prior=<Basiswahrscheinlichkeiten>,
    method=<Kovarianzschätzung>)"
```

Das erste Argument ist eine Modellformel der Form $\langle UV \rangle \sim \langle AV \rangle$. Bei ihr ist abweichend von den bisher betrachteten linearen Modellen die links von der \sim stehende, vorherzusagende Variable ein Faktor der Gruppenzugehörigkeiten, während die quantitativen Zielgrößen die Rolle der Prädiktoren auf der rechten Seite einnehmen. Stammen die in der Modellformel verwendeten Variablen aus einem Datensatz, ist dieser unter `data` zu nennen.

In der Voreinstellung verwendet `lda()` die relativen Häufigkeiten der Gruppen als Maß für ihre Auftretenswahrscheinlichkeiten in der Population. Über das Argument `prior` lassen sich letztere auch explizit in Form eines Vektors in der Reihenfolge der Stufen von $\langle UV \rangle$ vorgeben. Setzt man das Argument `method="mve"`, verwendet `lda()` eine robuste Schätzung von Mittelwerten und Kovarianzmatrix.

Das Beispiel verwendet dieselben Daten wie die einfaktorielle MANOVA (Abschn. 12.7.1), wobei die ungleichen Gruppenhäufigkeiten zunächst als Indikator für ihre Wahrscheinlichkeiten dienen sollen.

```
> Ydf1 <- data.frame(IVman, DV1=Ym1[ , 1], DV2=Ym1[ , 2])
> library(MASS)                                         # für lda()
> (ldaRes <- lda(IVman ~ DV1 + DV2, data=Ydf1))
Call:
lda(IVman ~ DV1 + DV2, data = Ydf1)

Prior probabilities of groups:
      1       2       3 
0.2500000 0.4166667 0.3333333 

Group means:
      DV1     DV2
1 -3.266667 5.60
2  2.360000 4.04
3 -0.350000 -0.40
```

⁴⁵Für weitere Klassifikationsverfahren wie Varianten der Clusteranalyse, CART-Modelle oder *support vector machines* vgl. die Abschnitte *Cluster Analysis* (Leisch & Gruen, 2020), *Multivariate Statistics* (Hewson, 2020) und *Machine Learning & Statistical Learning* (Hothorn, 2020) der CRAN Task Views. Die logistische und multinomiale Regression (Abschn. 8.1, 8.3) lassen sich ebenfalls zur Klassifikation verwenden und besitzen weniger Verteilungsvoraussetzungen als die Diskriminanzanalyse.

Coefficients of linear discriminants:

| | LD1 | LD2 |
|-----|-------------|-------------|
| DV1 | 0.06710397 | -0.27380306 |
| DV2 | -0.31861364 | -0.07850291 |

Proportion of trace:

| LD1 | LD2 |
|--------|--------|
| 0.6327 | 0.3673 |

Die Ausgabe nennt unter der Überschrift **Prior probabilities of groups** die angenommenen Gruppenwahrscheinlichkeiten. Unter **Group means** folgt eine zeilenweise aus den Vektoren der Gruppenzentroide zusammengestellte Matrix, die in der von **lda()** zurückgegebenen Liste in der Komponente **means** enthalten ist. Die Koeffizienten b_l der Diskriminanzfunktionen finden sich spaltenweise unter **Coefficients of linear discriminants**, das Ergebnis speichert diese Matrix in der Komponente **scaling**. In der Ausgabe fehlen die für eine Interpretation der Ergebnisse meist nicht interessanten absoluten Terme b_0 der Linearkombinationen. Unter **Proportion of trace** lässt sich der Anteil der von jeder Diskriminanzfunktion aufgeklärten Varianz an der Gesamtvarianz zwischen den Gruppen im unten näher erläuterten Sinn ableSEN.

Das Argument **CV=TRUE** (Voreinstellung ist **FALSE**) bewirkt eine Kreuzvalidierung, wobei gleichzeitig für jede Beobachtung und jede Gruppe die a-posteriori Wahrscheinlichkeit i. S. von Bayes berechnet wird, dass die Beobachtung zu einer Gruppe gehört. Die Matrix dieser Wahrscheinlichkeiten findet sich dann in der Komponente **posterior** der ausgegebenen Liste. Dagegen unterbleibt in diesem Fall die Berechnung der Koeffizienten für die Diskriminanzfunktionen.

```
> ldaP <- lda(IVman ~ DV1 + DV2, CV=TRUE, data=Ydf1)
> ldaP$posterior # Wahrscheinlichkeiten ...
```

Die aus der Regression bekannte **predict(⟨lda-Objekt⟩, ⟨Datensatz⟩)** Funktion (Abschn. 6.4) dient zur Vorhersage der Gruppenzugehörigkeiten auf Basis eines von **lda()** ausgegebenen Objekts. Dazu ist als zweites Argument ein Datensatz mit Variablen zu nennen, die dieselben Namen wie die AVn aus der ursprünglichen Analyse tragen. Die von **predict()** ausgegebene Liste enthält in der Komponente **x** die Diskriminanten selbst und in der Komponente **class** die vorhergesagte Kategorie. Die Güte der Vorhersage lässt sich für die Trainingsstichprobe etwa an der Konfusionsmatrix gemeinsamer Häufigkeiten von tatsächlichen und vorhergesagten Gruppenzugehörigkeiten ablesen (für weitere Maße der Übereinstimmung kategorialer Variablen s. Abschn. 10.2.6, 10.3.3).

```
> ldaPred <- predict(ldaRes, Ydf1) # Vorhersage für ursprüngliche Daten
> head(ldaPred$x, n=3) # Diskriminanten
   LD1          LD2
1 0.4166295  0.98818001
2 -2.6352990 -0.34445522
3 -2.7193092  1.37686604

> head(ldaPred$class) # Klassifikation
[1] 3 1 1 1 1 3
```

Levels: 1 2 3

```
# Kontingenztafel tatsächlicher und vorhergesagter Kategorien
> cTab <- table(IVman, ldaPred$class, dnn=c("IVman", "ldaPred"))
> addmargins(cTab)
      ldaPred
IVman   1   2   3  Sum
  1    9   3   3  15
  2    3  19   3  25
  3    1   6  13  20
Sum   13  28  19  60

> sum(diag(cTab)) / sum(cTab)      # Rate der korrekten Klassifikation
[1] 0.6833333
```

Die manuelle Kontrolle beruht auf den in Abschn. 12.9.9 berechneten Matrizen \mathbf{B} (SSP-Matrix der durch das zugehörige Gruppenzentroid ersetzen Daten) und \mathbf{W} (SSP-Matrix der Residuen). Die Koeffizientenvektoren der Diskriminanzfunktionen erhält man aus den Eigenvektoren von $\mathbf{W}^{-1}\mathbf{B}$. Diese werden dafür zum einen so umskaliert, dass die Residual-Quadratsumme der univariaten Varianzanalysen mit je einer Diskriminante als Zielgröße gleich $N - p$, die mittlere Residual-Quadratsumme also gleich 1 ist. Die F -Brüche dieser Varianzanalysen sind gleich den Eigenwerten von $\mathbf{W}^{-1}\mathbf{B}$, die mit dem Quotient der Freiheitsgrade der Quadratsummen innerhalb ($N - p$) und zwischen den Gruppen ($p - 1$) multipliziert wurden. Zum anderen werden die Diskriminanten so verschoben, dass ihr Mittelwert jeweils 0 beträgt. Der Anteil der Eigenwerte von $\mathbf{W}^{-1}\mathbf{B}$ an ihrer Summe, also an der Spur von $\mathbf{W}^{-1}\mathbf{B}$, wird von `lda()` unter `Proportion of trace` genannt.

```
> eigWinvB <- eigen(solve(WW) %*% BB) # Eigenwerte, -vektoren W^-1 * B
> eigVec  <- eigWinvB$vectors          # Eigenvektoren
> eigVal  <- eigWinvB$values          # Eigenwerte
> p       <- nlevels(IVman)           # Anzahl Gruppen
> N       <- sum(Nj)                  # gesamt-N
> My     <- colMeans(Ym1)             # Mittelwerte Variablen

# Proportion of trace, alternativ: eigVal / sum(eigVal)
> eigVal / sum(diag(solve(WW) %*% BB))
[1] 0.6327184 0.3672816

# Skalierungsfaktoren für Eigenvektoren
> scl <- sqrt((N-p) / diag(t(eigVec) %*% WW %*% eigVec))
> b0  <- -scl * t(eigVec) %*% My      # absolute Terme b_0

# Skalierung der Eigenvektoren -> Matrix mit Koeffizienten b_k
> (bk <- eigVec %*% diag(scl))
      [,1]      [,2]
[1,]  0.06710397 -0.27380306
[2,] -0.31861364 -0.07850291
```

Kapitel 12 Multivariate Verfahren

```
# prüfe, ob Diskriminanten mit Ergebnis von predict() übereinstimmen
> ld <- sweep(Ym1 %*% bk, 2, b0, "+") # Diskriminanten
> all.equal(ld, ldaPred$x, check.attributes=FALSE)
[1] TRUE

# univ. ANOVAs mit je einer Diskriminante als Zielgröße: SSw=N-p, MSw=1
> anova(lm(ld[, 1] ~ IVman)) # ANOVA mit 1. Diskriminante
Analysis of Variance Table
Response: ld1
Df Sum Sq Mean Sq F value Pr(>F)
IVman 2 39.651 19.826 19.826 2.911e-07 ***
Residuals 57 57.000 1.000

> anova(lm(ld[, 2] ~ IVman)) # ANOVA mit 2. Diskriminante
Analysis of Variance Table
Response: ld2
Df Sum Sq Mean Sq F value Pr(>F)
IVman 2 23.017 11.508 11.508 6.335e-05 ***
Residuals 57 57.000 1.0000

# F-Werte der ANOVAs aus Eigenwerten von W^-1 * B
> ((N-p) / (p-1)) * eigVal
[1] 19.82570 11.50846
```

Wurden der Diskriminanzanalyse gleiche Gruppenwahrscheinlichkeiten zugrunde gelegt, ergibt sich die vorhergesagte Gruppenzugehörigkeit für eine Beobachtung aus dem minimalen euklidischen Abstand zu den Gruppenzentroiden im durch die Diskriminanzfunktionen gebildeten Koordinatensystem: Dazu sind die Diskriminanten für alle Beobachtungen zu berechnen und die Gruppenmittelwerte der Trainingsstichprobe auf jeder Diskriminante zu bilden. Für die zu klassifizierende Beobachtung wird jene Gruppe als Vorhersage ausgewählt, deren Zentroid am nächsten an der Beobachtung liegt.

```
# Diskriminanzanalyse mit Annahme gleicher Gruppenwahrscheinlichkeiten
> priorP <- rep(1/nlevels(IVman), nlevels(IVman)) # gleiche W'keit
> ldaEq <- lda(IVman ~ DV1 + DV2, prior=priorP, data=Ydf1)
> predEq <- predict(ldaEq, Ydf1) # Diskrim., Vorhersage
> LDmat <- predEq$x # Diskriminanten

# Datensatz der Gruppenzentroide
> ctrDf <- aggregate(cbind(LD1, LD2) ~ IVman, FUN=mean, data=LDmat)
> ctrLD <- data.matrix(ctrDf[, -1]) # Matrix Gruppenzentroide

# Matrizen der Differenzvektoren der Beobachtungen zu jedem Zentroid
> diffMat1 <- scale(LDmat, center=ctrLD[1, ], scale=FALSE) # zu Z1
> diffMat2 <- scale(LDmat, center=ctrLD[2, ], scale=FALSE) # zu Z2
> diffMat3 <- scale(LDmat, center=ctrLD[3, ], scale=FALSE) # zu Z3
```

```
# euklidische Distanzen als jeweilige Länge des Differenzvektors
> dst1  <- sqrt(diag(tcrossprod(diffMat1))) # zu Zentroid 1
> dst2  <- sqrt(diag(tcrossprod(diffMat2))) # zu Zentroid 2
> dst3  <- sqrt(diag(tcrossprod(diffMat3))) # zu Zentroid 3
> dstMat <- cbind(dst1, dst2, dst3)           # Matrix: alle Distanzen

# jede Zeile: identifizierte Spalte mit minimaler Distanz -> Vorhersage
> dstPred <- apply(dstMat, 1, which.min)
> head(dstPred)                                # Klassifikation
1 2 3 4 5 6
3 1 1 1 1 3

# prüfe auf Übereinstimmung mit Ergebnis von predict()
> all(dstPred == unclass(predEq$class))
[1] TRUE
```

12.9 Das allgemeine lineare Modell

Das allgemeine lineare Modell (ALM) liefert einen formalen Rahmen, mit dessen Hilfe sich lineare Regression (Abschn. 6.3, 12.5), Varianzanalyse (Abschn. 7.4, 12.7) und Kovarianzanalyse (Abschn. 7.9) auf dieselbe Weise formulieren und ihre Hypothesen testen lassen. Es integriert dabei die uni- wie multivariaten Varianten dieser Verfahren. R verwendet das ALM implizit beim Anpassen dieser Modelle, was bisweilen bei der Anwendung bedeutsam wird. Daher soll die Grundidee des ALM hier skizziert werden. Ausführliche Darstellungen finden sich bei Agresti (2015) und Christensen (2020). Wickens (2015) fokussiert auf die oft sehr hilfreiche geometrische Interpretation der Verfahren. Abschnitt 12.1 erläutert die verwendeten Konzepte der linearen Algebra.

12.9.1 Modell der multiplen linearen Regression

Das Modell der univariaten multiplen Regression geht von Beobachtungsobjekten $i = 1, \dots, n$ aus, die Daten y_i eines Kriteriums Y und Werte x_{ij} von p Prädiktoren X_j liefern, die jeweils in Vektoren \mathbf{y} bzw. \mathbf{x}_j zusammengefasst werden (Abschn. 6.3).

$$E(y_i) = \beta_0 + \beta_1 x_{i1} + \dots + \beta_j x_{ij} + \dots + \beta_p x_{ip}$$

$$E(\mathbf{y}) = \beta_0 + \beta_1 \mathbf{x}_1 + \dots + \beta_j \mathbf{x}_j + \dots + \beta_p \mathbf{x}_p$$

Dabei ist $E(\mathbf{y})$ der n -Vektor $(E(y_1), \dots, E(y_n))^T$ der Erwartungswerte von y_i . Von den skalaren Parametern β_0 (additive Konstante, absoluter Term) und β_j (theoretische Regressionsgewichte) wird angenommen, dass sie für alle Beobachtungsobjekte identisch sind. Ferner sei

vorausgesetzt, dass mehr Beobachtungen als Parameter vorhanden sind, hier also $n > p + 1$ gilt. In Matrix-Schreibweise lässt sich das Modell so formulieren:

$$E(\mathbf{y}) = \mathbf{1}\beta_0 + \mathbf{X}_p\beta_p = [\mathbf{1}|\mathbf{X}_p] \begin{pmatrix} \beta_0 \\ \beta_p \end{pmatrix} = \mathbf{X}\beta$$

Hier ist $\mathbf{1}$ der n -Vektor $(1, \dots, 1)^\top$, β_p der p -Vektor $(\beta_1, \dots, \beta_p)^\top$, β der $(p+1)$ -Vektor $(\beta_0, \beta_p^\top)^\top$ und \mathbf{X}_p die $(n \times p)$ -Matrix der spaltenweise zusammengestellten Vektoren \mathbf{x}_j . Die Prädiktoren sollen linear unabhängig sein, womit \mathbf{X}_p vollen Spaltenrang p besitzt. $\mathbf{X} = [\mathbf{1}|\mathbf{X}_p]$ ist die $(n \times (p+1))$ -Designmatrix, deren Spalten den Unterraum V mit Dimension $\text{Rang}(\mathbf{X}) = p+1$ aufspannen.⁴⁶ Das Modell lässt sich als Behauptung verstehen, dass $E(\mathbf{y})$ in V liegt. Für einen solchen modellverträglichen Vektor von Erwartungswerten existiert ein Parametervektor β , mit dem $E(\mathbf{y}) = \mathbf{X}\beta$ gilt.⁴⁷

Im ALM ergeben sich die personenweisen Erwartungswerte eines Kriteriums als lineare Funktion der Prädiktoren. Der Zusammenhang zwischen einer Zielgröße Y und einem Prädiktor X selbst muss im ALM dagegen nicht linear sein. Ein quadratischer Zusammenhang zwischen Y und X könnte etwa als Modell $E(\mathbf{y}) = \beta_0 + \beta_1 \mathbf{x} + \beta_2 \mathbf{x}^2$ formuliert werden – hier gäbe es bei einer inhaltlichen Vorhersagevariable X zwei Prädiktoren, nämlich X und X^2 .

Im Fall der multivariaten multiplen linearen Regression mit r Kriterien Y_l stellt man die Vektoren $E(\mathbf{y}_l)$ der Erwartungswerte der einzelnen Kriterien spaltenweise zu einer $(n \times r)$ -Matrix $E(\mathbf{Y})$ zusammen. Genauso werden die r vielen p -Vektoren $\beta_{p,l} = (\beta_{p,1l}, \dots, \beta_{p,jl}, \dots, \beta_{p,pl})^\top$ der theoretischen Regressionsgewichte jeweils aus der Regression mit den Prädiktoren X_j und dem Kriterium Y_l spaltenweise zu einer $(p \times r)$ -Matrix \mathbf{B}_p zusammengefasst. Die r absoluten Terme $\beta_{0,l}$ bilden einen r -Vektor β_0 . Das Modell lautet dann:

$$E(\mathbf{Y}) = \beta_0 + \mathbf{X}_p \mathbf{B}_p = [\mathbf{1}|\mathbf{X}_p] \begin{pmatrix} \beta_0^\top \\ \mathbf{B}_p \end{pmatrix} = \mathbf{X}\mathbf{B}$$

Die Designmatrix im multivariaten Fall stimmt mit jener im univariaten Fall überein. Die Modellparameter sind im univariaten Fall bei Kenntnis von \mathbf{X} und $E(\mathbf{y})$ als $\beta = \mathbf{X}^+ E(\mathbf{y})$ identifizierbar – der Lösung der *Normalengleichungen* (Abschn. 12.1.7). Dies gilt analog auch im multivariaten Fall mit $\mathbf{B} = \mathbf{X}^+ E(\mathbf{Y})$. Dabei ist $\mathbf{X}^+ = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top$ die Pseudoinverse von \mathbf{X} , also $\mathbf{X}^+ \mathbf{X} = \mathbf{I}$ mit der $(n \times n)$ -Einheitsmatrix \mathbf{I} . Die Parameter sind gleich den Koordinaten des Vektors der Erwartungswerte, der orthogonal auf V projiziert wurde, bzgl. der durch \mathbf{X} definierten Basis (Abschn. 12.1.7).

⁴⁶Die Designmatrix erhält man mit der Funktion `model.matrix()`, die als Argument ein mit `lm()` erstelltes Modell, oder auch nur die rechte Seite einer Modellformel akzeptiert (Abschn. 5.2; Venables & Ripley, 2002, p. 144 ff.).

⁴⁷Die \mathbf{x}_j sind feste Realisierungen eines Zufallsvektors, also *stochastische Prädiktoren*. Sie enthalten damit nicht alle möglichen Prädiktorwerte, sondern nur jeweils n viele. Man könnte daher auch vom Vektor $E(\mathbf{y}|\mathbf{X})$ der auf eine konkrete Designmatrix \mathbf{X} bedingten Erwartungswerte von \mathbf{y} sprechen, worauf hier aber verzichtet wird. Die \mathbf{x}_j müssen fehlerfrei sein, bei den x_{ij} muss es sich also um die wahren Prädiktorwerte handeln. Ohne diese Annahme kommen lineare Strukturgleichungsmodelle zur Auswertung in Betracht (Abschn. 12.3, Fußnote 33).

Für eine univariate multiple Regression mit zwei Prädiktoren X_j und drei Beobachtungsobjekten, die Prädiktorwerte x_{ij} besitzen, ergibt sich folgendes Modell:

$$\begin{aligned} E(\mathbf{y}) &= \begin{pmatrix} E(y_1) \\ E(y_2) \\ E(y_3) \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}_{\mathbf{1}} \beta_0 + \begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \\ x_{31} & x_{32} \end{pmatrix}_{\mathbf{X}_p} \begin{pmatrix} \beta_1 \\ \beta_2 \end{pmatrix}_{\boldsymbol{\beta}_p} \\ &= \begin{pmatrix} 1 & x_{11} & x_{12} \\ 1 & x_{21} & x_{22} \\ 1 & x_{31} & x_{32} \end{pmatrix}_{\mathbf{X}} \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{pmatrix}_{\boldsymbol{\beta}} \\ &= \begin{pmatrix} \beta_0 + \beta_1 x_{11} + \beta_2 x_{12} \\ \beta_0 + \beta_1 x_{21} + \beta_2 x_{22} \\ \beta_0 + \beta_1 x_{31} + \beta_2 x_{32} \end{pmatrix} \end{aligned}$$

Im Fall einer moderierten Regression wird das Modell um zusätzliche Prädiktoren erweitert, die als Interaktionsterm gleich dem Produkt von Einzelprädiktoren sind (Abschn. 6.3.4). Hierfür kann man aus Produkten der Spalten von \mathbf{X}_p eine weitere Matrix \mathbf{X}_{pI} mit den neuen Prädiktoren erstellen, so dass die neue Designmatrix $\mathbf{X} = [\mathbf{1} | \mathbf{X}_p | \mathbf{X}_{pI}]$ ist. Der Parametervektor $\boldsymbol{\beta}$ ist entsprechend um passend viele Interaktionsparameter zu ergänzen.

12.9.2 Modell der einfaktoriellen Varianzanalyse

Analog zur Regression wird in der univariaten einfaktoriellen Varianzanalyse mit p unabhängigen Bedingungen (CR- p Design, Abschn. 7.4) zunächst die Zugehörigkeit zu jeder der p Faktorstufen zu einem eigenen dichotomen Prädiktor X_j^* , der als *Indikatorvariable* bezeichnet wird. Beobachtungsobjekte erhalten für ein X_j^* den Wert 1, wenn sie sich in der zugehörigen Bedingung j befinden, sonst 0.⁴⁸ Jedes Beobachtungsobjekt erhält also auf einem X_j^* die 1 und auf den verbleibenden $p - 1$ Indikatorvariablen die 0. Die spaltenweise aus den X_j^* zusammengestellte $(n \times p)$ -Matrix \mathbf{X}_p^* heißt *Inzidenzmatrix* und hat vollen Spaltenrang p . Die Komponenten des Vektors \mathbf{y} und die Zeilen der Inzidenzmatrix \mathbf{X}_p^* seien gemeinsam entsprechend der Gruppenzugehörigkeit geordnet.⁴⁹

Für die einfaktorielle Varianzanalyse mit drei Gruppen, zwei Personen pro Gruppe sowie den Parametern β_0 und β_j^* ergibt sich folgendes Modell. Beobachtungen aus unterschiedlichen Gruppen sind durch waagerechte Linien getrennt.

⁴⁸ Anders als in der Regression werden die Werte der Indikatorvariablen hier durch die Zuordnung von Beobachtungen zu Gruppen systematisch hergestellt, sind also keine stochastischen Prädiktoren.

⁴⁹ Eine der folgenden Zusammenfassung ähnliche Exposition enthält die Vignette des Pakets `codingMatrices` (Venables, 2018). Dabei entsprechen sich folgende Bezeichnungen: Die Inzidenzmatrix \mathbf{X}_p^* hier ist dort \mathbf{X} . Die reduzierte Designmatrix \mathbf{X} hier ist dort $\tilde{\mathbf{X}}$, die Codiermatrix \mathbf{C} hier ist dort \mathbf{B}_* , die Matrix $[\mathbf{1} | \mathbf{C}]$ hier ist dort $\mathbf{B} = [\mathbf{1} | \mathbf{B}_*] = \mathbf{C}^{-1}$, der Parametervektor $\boldsymbol{\beta}_{p-1}$ hier ist dort $\boldsymbol{\beta}_*$.

$$\begin{aligned}
 E(\mathbf{y}) &= \begin{pmatrix} \mu_1 \\ \mu_1 \\ \mu_2 \\ \mu_2 \\ \mu_3 \\ \mu_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}_1 \beta_0 + \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}_{\mathbf{X}_p^*} \begin{pmatrix} \beta_1^* \\ \beta_2^* \\ \beta_3^* \end{pmatrix}_{\beta_p^*} \\
 &= \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \end{pmatrix}_{\mathbf{X}^*} \begin{pmatrix} \beta_0 \\ \beta_1^* \\ \beta_2^* \\ \beta_3^* \end{pmatrix}_{\beta^*} = \begin{pmatrix} \beta_0 + \beta_1^* \\ \beta_0 + \beta_1^* \\ \beta_0 + \beta_2^* \\ \beta_0 + \beta_2^* \\ \beta_0 + \beta_3^* \\ \beta_0 + \beta_3^* \end{pmatrix}
 \end{aligned}$$

Der Vektor $\mathbf{1}$ ist die Summe der Spalten von \mathbf{X}_p^* , liegt also in ihrem Erzeugnis, weshalb die $(n \times (p+1))$ -Designmatrix $\mathbf{X}^* = [\mathbf{1} | \mathbf{X}_p^*]$ wie \mathbf{X}_p^* selbst nur Rang p besitzt. $(\mathbf{X}^*)^+$ ist nun nicht eindeutig bestimmt, da $(\mathbf{X}^*)^\top \mathbf{X}^*$ nicht invertierbar ist. Die Parameter sind hier deswegen anders als in der Regression zunächst nicht identifizierbar. Aus diesem Grund verändert man das Modell, indem man eine Nebenbedingung $\mathbf{v}^\top \beta_p^* = 0$ über die Parameter einführt und so deren frei variierbare Anzahl um 1 reduziert. Dabei ist \mathbf{v} ein p -Vektor, der eine Linearkombination der Parameter β_j^* festlegt, die 0 ist.⁵⁰

Es sei nun $\beta_{p-1} = (\beta_1, \dots, \beta_{p-1})^\top$ der $(p-1)$ -Vektor der neuen, *reduzierten* Parameter, den *Kontrasten*. Weiter sei \mathbf{C} eine $(p \times (p-1))$ -Matrix, so dass die $(p \times p)$ -Matrix $[\mathbf{1} | \mathbf{C}]$ vollen Rang p besitzt und unter Erfüllung der Nebenbedingung $\mathbf{v}^\top \beta_p^* = 0$ gilt.⁵¹

$$\begin{aligned}
 \beta_p^* &= \mathbf{C} \beta_{p-1} \\
 \beta^* &= [\mathbf{1} | \mathbf{C}] \begin{pmatrix} \beta_0 \\ \beta_{p-1} \end{pmatrix} = [\mathbf{1} | \mathbf{C}] \beta
 \end{aligned}$$

Die *Codiermatrix* \mathbf{C} definiert für jede in den p Zeilen stehende Gruppenzugehörigkeit die zugehörigen Werte auf den neuen, nun $p-1$ Prädiktoren X_j , die mit den Spalten von \mathbf{C} korrespondieren. Die Nebenbedingung $\mathbf{v}^\top \beta_p^* = 0$ ist erfüllt, wenn \mathbf{C} so gewählt wird, dass $\mathbf{v}^\top \mathbf{C} = \mathbf{0}^\top$ ist.⁵² Dabei können für ein bestimmtes \mathbf{v} mehrere Wahlmöglichkeiten für \mathbf{C} bestehen. Da $[\mathbf{1} | \mathbf{C}]$ vollen Rang p besitzt, existiert $[\mathbf{1} | \mathbf{C}]^{-1}$, und es gilt:

$$\beta = [\mathbf{1} | \mathbf{C}]^{-1} \beta^*$$

⁵⁰Alternativ kann auch der Parameter $\beta_0 = 0$ gesetzt werden, womit $\mathbf{X}^* = \mathbf{X}_p^*$ ist. Diese Möglichkeit zur Parametrisierung soll hier nicht weiter verfolgt werden, um das Modell wie jenes der Regression formulieren zu können (Fußnote 55).

⁵¹Für die Parameterschätzungen gilt dies analog (Abschn. 12.9.4, Fußnote 61).

⁵²Denn dann gilt $\mathbf{v}^\top \beta_p^* = \mathbf{v}^\top \mathbf{C} \beta_{p-1} = \mathbf{0}^\top \beta_{p-1} = 0$. \mathbf{v} steht senkrecht auf den Spalten von \mathbf{C} , ist also eine Basis des orthogonalen Komplements des von den Spalten von \mathbf{C} aufgespannten Unterraums. Mit anderen Worten ist \mathbf{v} wegen $\mathbf{0} = (\mathbf{v}^\top \mathbf{C})^\top = \mathbf{C}^\top \mathbf{v}$ eine Basis des Kerns von \mathbf{C}^\top .

Die *Kontrastmatrix* $[\mathbf{1}|\mathbf{C}]^{-1}$ bildet den Vektor β^* aller ursprünglichen Parameter auf den Vektor aller neuen Parameter β im Unterraum ab, in dem β^* mit der gewählten Nebenbedingung frei variieren kann. Anders gesagt ergeben sich die Komponenten von β als Linearkombinationen der ursprünglichen Parameter β^* , wobei $[\mathbf{1}|\mathbf{C}]^{-1}$ zeilenweise aus den Koeffizientenvektoren zusammengestellt ist. $[\mathbf{1}|\mathbf{C}]^{-1}$ induziert so die inhaltliche Bedeutung der neuen Parameter.⁵³ Das reduzierte Modell lautet insgesamt:

$$E(\mathbf{y}) = \mathbf{1}\beta_0 + \mathbf{X}_p^* \beta_p^* = \mathbf{1}\beta_0 + \mathbf{X}_p^* \mathbf{C} \beta_{p-1} = [\mathbf{1}|\mathbf{X}_{p-1}] \begin{pmatrix} \beta_0 \\ \beta_{p-1} \end{pmatrix} = \mathbf{X}\beta$$

Hier ist $\mathbf{X}_{p-1} = \mathbf{X}_p^* \mathbf{C}$ die reduzierte $(n \times (p-1))$ -Inzidenzmatrix und $\mathbf{X} = [\mathbf{1}|\mathbf{X}_{p-1}]$ die reduzierte $(n \times p)$ -Designmatrix mit vollem Spaltenrang p , wobei $n > p$ vorausgesetzt wird. Der p -Vektor β der neuen Parameter ist so wie im Fall der Regression über $\mathbf{X}^+ E(\mathbf{y})$ identifizierbar.

Die inhaltliche Bedeutung der Parameter in β hängt von der Wahl der Nebenbedingung \mathbf{v} und der Codiermatrix \mathbf{C} ab. R verwendet in der Voreinstellung die *Dummy-Codierung (Treatment-Kontraste)*, bei denen die Indikatorvariablen X_j^* zunächst erhalten bleiben. In der Voreinstellung wird dann der ursprüngliche, zur ersten Gruppe gehörende Parameter $\beta_1^* = 0$ gesetzt.⁵⁴ Hier ist die Nebenbedingung also $\beta_1^* = (1, 0, \dots, 0) \beta_p^* = 0$, d. h. $\mathbf{v} = (1, 0, \dots, 0)^\top$. `contr.treatment(<Anzahl>, base=<Referenzgruppe>)` gibt die Matrix \mathbf{C}_t für Treatment-Kontraste aus. Als Argument ist dabei die Anzahl der Gruppen p sowie optional die Nummer der Referenzstufe zu nennen – in der Voreinstellung die Stufe 1. Die Spalten von \mathbf{C}_t sind paarweise orthogonal und besitzen die Länge 1 ($\mathbf{C}_t^\top \mathbf{C}_t = \mathbf{I}$).

```
> contr.treatment(4)          # Treatment-Kontraste für 4 Gruppen
  2 3 4
1 0 0 0
2 1 0 0
3 0 1 0
4 0 0 1
```

Treatment-Kontraste bewirken, dass die reduzierte Inzidenzmatrix \mathbf{X}_{p-1} durch Streichen der ersten Spalte von \mathbf{X}_p^* entsteht. Die Bezeichnung dieses Codierschemas leitet sich aus der Situation ab, dass die erste Faktorstufe eine Kontrollgruppe darstellt, während die übrigen zu Treatment-Gruppen gehören. Die Parameter können dann als Wirkung der Stufe j i. S. der Differenz zur Kontrollgruppe verstanden werden: Für die Mitglieder der ersten Gruppe ist $E(y_i) = \mu_1 = \beta_0$, da für diese Gruppe alle verbleibenden $X_j = 0$ sind (mit $j = 2, \dots, p$). Für Mitglieder jeder übrigen Gruppe j erhält man $E(y_i) = \mu_j = \beta_0 + \beta_j = \mu_1 + \beta_j$, da dann $X_j = 1$ ist. Damit ist $\beta_j = \mu_j - \mu_1$ gleich der Differenz des Erwartungswertes der Gruppe j zum Erwartungswert der Referenzgruppe.⁵⁵

⁵³Das Paket `hypr` (Rabe, Vasishth, Hohenstein, Kliegl & Schad, 2023) hilft dabei, den Zusammenhang zwischen Kontrastmatrix und getesteten Hypothesen zu verstehen – s. insbesondere `vignette("hypr-contrasts")`.

⁵⁴Die in einem linearen Modell mit kategorialen Variablen von R weggelassene Gruppe ist die erste Stufe von `levels(<Faktor>)`.

⁵⁵Die in Fußnote 50 erwähnte Möglichkeit der Parametrisierung führt zum *cell means* Modell, bei dem $\mathbf{X} = \mathbf{X}^* = \mathbf{X}_p^*$ gilt und die Parameter β_j^* direkt die Bedeutung der Gruppenerwartungswerte μ_j erhalten.

Für $p = 3$, zwei Personen pro Gruppe und Treatment-Kontraste ergibt sich folgendes Modell mit den reduzierten Parametern β_0 und β_j :

$$E(\mathbf{y}) = \begin{pmatrix} \mu_1 \\ \mu_1 \\ \mu_2 \\ \mu_2 \\ \mu_3 \\ \mu_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}_1 \beta_0 + \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}_{\mathbf{X}_p^*} \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}_{\mathbf{C}_t} \begin{pmatrix} \beta_2 \\ \beta_3 \end{pmatrix}_{\beta_{p-1}}$$

$$= \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 1 \end{pmatrix}_{\mathbf{X}} \begin{pmatrix} \beta_0 \\ \beta_2 \\ \beta_3 \end{pmatrix}_{\beta} = \begin{pmatrix} \beta_0 \\ \beta_0 \\ \frac{\beta_0 + \beta_2}{\beta_0 + \beta_2} \\ \frac{\beta_0 + \beta_2}{\beta_0 + \beta_3} \\ \frac{\beta_0}{\beta_0 + \beta_3} \end{pmatrix}$$

Die von \mathbf{C}_t induzierte Bedeutung der Parameter in β lässt sich direkt an $[\mathbf{1}|\mathbf{C}_t]^{-1}$ ablesen. Die erste Zeile definiert das Verhältnis von β_0 zu den Gruppenerwartungswerten μ_j und ist allgemein ein Vektor, dessen Koeffizienten sich zu 1 summieren – hier $\beta_0 = \mu_1$. Die weiteren Zeilen definieren, wie sich die Kontraste in β_{p-1} als Differenz jedes Gruppenerwartungswerts zum Erwartungswert der Referenzstufe ergeben – etwa $\beta_1 = -\mu_1 + \mu_2$. In diesen Zeilen summieren sich die Koeffizienten zu 0.

```
# Ausgabe 1. Zeile = Spaltennamen, 1. Spalte = Zeilennamen
> solve(cbind(1, contr.treatment(4)))
 1 2 3 4
 1 0 0 0
 2 -1 1 0 0
 3 -1 0 1 0
 4 -1 0 0 1
```

In der mit `getOption("contrasts")` einsehbaren Voreinstellung verwendet R Treatment-Kontraste, wenn ungeordnete kategoriale Prädiktoren in einem linearen Modell vorhanden sind, und polynomiale Kontraste für ordinale Prädiktoren (für andere Möglichkeiten vgl. `?contrasts`). Deren Stufen werden dabei als gleichabständig vorausgesetzt.

```
> getOption("contrasts")          # eingestelltes Codierschema
      unordered      ordered
"contr.treatment"  "contr.poly"
```

In der einfaktoriellen Varianzanalyse ist die Konzeption der Parameter jedoch oft eine andere. Hier sind dies die Effektgrößen $\alpha_j = \mu_j - \mu$, also die jeweilige Differenz der Gruppenerwartungswerte zum (ungewichteten, also gleichgewichteten) mittleren Erwartungswert $\mu = \frac{1}{p} \sum_j \mu_j$. Die Summe der so definierten Effektgrößen α_j in der Rolle der Parameter β_j^* ist 0. Die Nebenbedingung lautet hier also $\sum_j \beta_j^* = \mathbf{1}^\top \beta_p^* = 0$, d. h. $\mathbf{v} = \mathbf{1}$.

Die genannte Parametrisierung lässt sich über die ungewichtete *Effektcodierung* ausdrücken:⁵⁶ Hierfür wird zunächst die Zugehörigkeit zu jeder Faktorstufe zu einem separaten Prädiktor X_j , der die Werte $-1, 0$ und 1 annehmen kann. Beobachtungsobjekte aus der Gruppe j (mit $j = 1, \dots, p-1$) erhalten für X_j den Wert 1 , sonst 0 . Beobachtungsobjekte aus der Gruppe p erhalten auf allen X_j den Wert -1 . Zur Beseitigung der Redundanz wird dann der ursprüngliche, zur letzten Stufe gehörende Parameter β_p^* aus dem Modell gestrichen.⁵⁷ `contr.sum(<Anzahl →Gruppen>)` gibt die Matrix C_e für die Effektcodierung aus, wobei als Argument die Anzahl der Gruppen p zu nennen ist.

```
> 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
```

Mit der ungewichteten Effektcodierung erhält der Parameter β_0 die Bedeutung des ungewichteten mittleren Erwartungswertes $\mu = \frac{1}{p} \sum_j \mu_j$ und die β_j die Bedeutung der ersten $p-1$ Effektgrößen α_j . Für Mitglieder der ersten $p-1$ Gruppen ist nämlich $E(y_i) = \mu_j = \beta_0 + \beta_j$, für Mitglieder der Gruppe p gilt $E(y_i) = \mu_p = \beta_0 - (\beta_1 + \dots + \beta_{p-1})$. Dabei stellt $\beta_1 + \dots + \beta_{p-1}$ die Abweichung $\alpha_p = \mu_p - \mu$ dar, weil sich die Abweichungen der Gruppenerwartungswerte vom mittleren Erwartungswert über alle Gruppen zu 0 summieren müssen. In jeder Komponente von $E(\mathbf{y})$ ist somit $E(y_i) = \mu + \alpha_j$.

Für $p = 3$, zwei Personen pro Gruppe und Effektcodierung ergibt sich folgendes Modell mit den reduzierten Parametern β_0 und β_j :

⁵⁶Ein andere Wahl für C unter der Nebenbedingung $\mathbf{v} = \mathbf{1}$ ist die Helmert-Codierung mit paarweise orthogonalen Spalten von $[1|C]$ (vgl. `?contr.helmert`). Die Parameter haben dann jedoch eine andere Bedeutung.

⁵⁷Alternativ ließe sich $\mu = \sum_j \frac{n_j}{n} \mu_j$ als mit den anteiligen Zellbesetzungen $\frac{n_j}{n}$ gewichtetes Mittel der μ_j definieren. Die zugehörige Nebenbedingung für die β_j^* i.S. der α_j lautet dann $\sum_j \frac{n_j}{n} \beta_j^* = 0$, d.h. $\mathbf{v} = (\frac{n_1}{n}, \dots, \frac{n_p}{n})^\top$. Diese Parametrisierung lässt sich mit der gewichteten Effektcodierung umsetzen, die zunächst der ungewichteten gleicht. In der letzten Zeile der Matrix C erhalten hier jedoch Mitglieder der Gruppe p für die X_j nicht den Wert -1 , sondern $-\frac{n_j}{n_p}$. Für die gewichtete Effektcodierung stellt das Paket `wec` (Nieuwenhuis, te Grotenhuis & Pelzer, 2017) die Funktion `contr.wec()` bereit.

$$\begin{aligned}
 E(\mathbf{y}) &= \begin{pmatrix} \mu_1 \\ \mu_1 \\ \hline \mu_2 \\ \mu_2 \\ \hline \mu_3 \\ \mu_3 \end{pmatrix} = \begin{pmatrix} 1 \\ \frac{1}{1} \\ 1 \\ \frac{1}{1} \\ 1 \\ \frac{1}{1} \end{pmatrix}_1 \beta_0 + \begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{0} & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ \hline 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}_{\mathbf{X}_p^*} \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & -1 \end{pmatrix}_{C_e} \begin{pmatrix} \beta_1 \\ \beta_2 \end{pmatrix}_{\beta_{p-1}} \\
 &= \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ \hline 1 & 0 & 1 \\ 1 & 0 & 1 \\ \hline 1 & -1 & -1 \\ 1 & -1 & -1 \end{pmatrix}_{\mathbf{X}} \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{pmatrix}_{\beta} = \begin{pmatrix} \beta_0 + \beta_1 \\ \beta_0 + \beta_1 \\ \hline \beta_0 & + & \beta_2 \\ \beta_0 & + & \beta_2 \\ \hline \beta_0 - (\beta_1 & + & \beta_2) \\ \beta_0 - (\beta_1 & + & \beta_2) \end{pmatrix}
 \end{aligned}$$

Die von \mathbf{C}_e induzierte Bedeutung der Parameter in β lässt sich direkt an $[\mathbf{1}|\mathbf{C}_e]^{-1}$ ablesen. Die erste Zeile definiert das Verhältnis von β_0 zu den Gruppenerwartungswerten μ_j – hier ist dies das arithmetische Mittel $\beta_0 = \frac{1}{4}\mu_1 + \frac{1}{4}\mu_2 + \frac{1}{4}\mu_3 + \frac{1}{4}\mu_4$. Die weiteren Zeilen definieren, wie sich die Kontraste in β_{p-1} als Vergleich zwischen Gruppenerwartungswert und mittlerem Erwartungswert ergeben, etwa:

$$\begin{aligned}
 \beta_1 &= \frac{3}{4}\mu_1 - \frac{1}{4}\mu_2 - \frac{1}{4}\mu_3 - \frac{1}{4}\mu_4 \\
 &= \frac{3}{4}\mu_1 + \frac{1}{4}\mu_1 - \frac{1}{4}\mu_1 - \frac{1}{4}\mu_2 - \frac{1}{4}\mu_3 - \frac{1}{4}\mu_4 \\
 &= \mu_1 - \frac{1}{4}(\mu_1 + \mu_2 + \mu_3 + \mu_4) \\
 &= \mu_1 - \mu = \alpha_j
 \end{aligned}$$

```

> solve(cbind(1, contr.sum(4)))
      1     2     3     4
[1,]  0.25  0.25  0.25  0.25
[2,]  0.75 -0.25 -0.25 -0.25
[3,] -0.25  0.75 -0.25 -0.25
[4,] -0.25 -0.25  0.75 -0.25

```

Soll R bei ungeordneten kategorialen Prädiktoren in diesem Sinne jede Gruppe nicht wie bei Treatment-Kontrasten mit der Referenzstufe, sondern durch die Effektcodierung generell mit dem Gesamtmittel verglichen, ist dies mit `options()` einzustellen.

```

# gehe zur Effektcodierung für ungeordnete Faktoren über
> options(contrasts=c("contr.sum", "contr.poly"))

# wechsle zurück zu Treatment-Kontrasten
> options(contrasts=c("contr.treatment", "contr.poly"))

```

Neben dieser Grundeinstellung existiert auch die Möglichkeit, das Codierschema für einen Faktor mit `C()` direkt festzulegen.

```
C(<Faktor>, contr=<Codiermatrix>)
```

Das erste Argument von `C()` ist ein Faktor. Für das Argument `contr` ist entweder eine selbst erstellte Codiermatrix C oder eine Funktion wie etwa `contr.sum` zu übergeben, die eine passende Codiermatrix erzeugt. C wird als Attribut "contrasts" des von `C()` zurückgegebenen Faktors gespeichert und automatisch von Funktionen wie `lm()` oder `aov()` verwendet, wenn der neue Faktor Teil der Modellformel ist.

```
> IV <- gl(3, 5) # Faktor: 3 Gruppen à 5 Personen
> (IVe <- C(IV, contr.sum)) # Effektcodierung für Faktor IVe
[1] 1 1 1 1 1 2 2 2 2 3 3 3 3 3
attr(,"contrasts")
 [,1] [,2]
1     1     0
2     0     1
3    -1    -1
Levels: 1 2 3
```

`contrasts(<Faktor>)` gibt die passende Codiermatrix C für die Stufen des übergebenen Faktors aus. Ist mit dem Faktor keine Codiermatrix in Form eines Attributs fest verbunden, wird hierfür die Grundeinstellung verwendet, die mit `getOption("contrasts")` einsehbar ist.

```
> contrasts(IV) # Codiermatrix C nach Grundeinstellung
2 3
1 0 0
2 1 0
3 0 1
```

Schließlich lässt sich das Codierschema temporär auch direkt beim Aufruf der `lm()` Funktion angeben, indem ihr Argument `contrasts` verwendet wird. Das Argument erwartet eine Liste, deren Komponenten als Namen die Bezeichnungen der Faktoren besitzen, die in der Modellformel auftauchen. Die Komponente selbst ist dann eine Kontrastfunktion wie etwa `contr.sum`.

```
> DV <- rnorm(15)
> lm(DV ~ IV, contrasts=list(IV=contr.treatment)) # ...
```

12.9.3 Modell der zweifaktoriellen Varianzanalyse

In der univariaten zweifaktoriellen Varianzanalyse mit p Stufen des ersten und q Stufen des zweiten Gruppierungsfaktors (CRF- pq Design, Abschn. 7.6) ist zunächst für jeden der beiden Haupteffekte eine Inzidenzmatrix analog zum einfaktoriellen Fall zu bilden. Dies sollen hier die $(n \times p)$ -Matrix \mathbf{X}_1^* für den ersten Gruppierungsfaktor und die $(n \times q)$ -Matrix \mathbf{X}_2^* für den zweiten Gruppierungsfaktor sein. Der zugehörige p -Vektor der Parameter für den ersten Gruppierungsfaktor sei $\boldsymbol{\beta}_1^* = (\beta_{1,1}^*, \dots, \beta_{1,p}^*)^\top$, der q -Vektor der Parameter für den zweiten Gruppierungsfaktor sei $\boldsymbol{\beta}_2^* = (\beta_{2,1}^*, \dots, \beta_{2,q}^*)^\top$.

Die $(n \times (p \cdot q))$ -Inzidenzmatrix $\mathbf{X}_{1 \times 2}^*$ für den Interaktionseffekt wird als spaltenweise Zusammensetzung aller $p \cdot q$ möglichen Produkte der Spalten von \mathbf{X}_1^* und \mathbf{X}_2^* gebildet (Abschn. 6.3.4). Der zugehörige $(p \cdot q)$ -Vektor der passend geordneten Parameter sei $\beta_{1 \times 2}^* = (\beta_{1 \times 2,11}^*, \dots, \beta_{1 \times 2,pq}^*)^\top$. Die $(n \times (p + q + p \cdot q))$ -Gesamt-Inzidenzmatrix ist dann $[\mathbf{X}_1^* | \mathbf{X}_2^* | \mathbf{X}_{1 \times 2}^*]$ und der zugehörige $(p + q + p \cdot q)$ -Vektor aller genannten Parameter $(\beta_1^{*\top}, \beta_2^{*\top}, \beta_{1 \times 2}^{*\top})^\top$. Entsprechend ist die ursprüngliche $(n \times (p + q + p \cdot q + 1))$ -Designmatrix $\mathbf{X}^* = [\mathbf{1} | \mathbf{X}_1^* | \mathbf{X}_2^* | \mathbf{X}_{1 \times 2}^*]$ und das Modell analog zur einfaktoriellen Situation:

$$E(\mathbf{y}) = [\mathbf{1} | \mathbf{X}_1^* | \mathbf{X}_2^* | \mathbf{X}_{1 \times 2}^*] \begin{pmatrix} \beta_0 \\ \beta_1^* \\ \beta_2^* \\ \beta_{1 \times 2}^* \end{pmatrix} = \mathbf{X}^* \beta^*$$

Für die zweifaktorielle Varianzanalyse mit $p = 3$, $q = 2$ und einer Person pro Gruppe ergibt sich folgendes Modell:

$$\begin{aligned} E(\mathbf{y}) &= \begin{pmatrix} \mu_{11} \\ \mu_{21} \\ \mu_{31} \\ \mu_{12} \\ \mu_{22} \\ \mu_{32} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}_{\mathbf{X}^*} \begin{pmatrix} \beta_0 \\ \beta_{1,1}^* \\ \beta_{1,2}^* \\ \beta_{1,3}^* \\ \beta_{2,1}^* \\ \beta_{2,2}^* \\ \beta_{1 \times 2,11}^* \\ \beta_{1 \times 2,21}^* \\ \beta_{1 \times 2,31}^* \\ \beta_{1 \times 2,12}^* \\ \beta_{1 \times 2,22}^* \\ \beta_{1 \times 2,32}^* \end{pmatrix}_{\beta^*} \\ &= \begin{pmatrix} \beta_0 + \beta_{1,1}^* + \beta_{2,1}^* + \beta_{1 \times 2,11}^* \\ \beta_0 + \beta_{1,2}^* + \beta_{2,1}^* + \beta_{1 \times 2,21}^* \\ \beta_0 + \beta_{1,3}^* + \beta_{2,1}^* + \beta_{1 \times 2,31}^* \\ \beta_0 + \beta_{1,1}^* + \beta_{2,2}^* + \beta_{1 \times 2,12}^* \\ \beta_0 + \beta_{1,2}^* + \beta_{2,2}^* + \beta_{1 \times 2,22}^* \\ \beta_0 + \beta_{1,3}^* + \beta_{2,2}^* + \beta_{1 \times 2,32}^* \end{pmatrix} \end{aligned}$$

Wie in der einfaktoriellen Varianzanalyse besitzt \mathbf{X}^* nicht vollen Spaltenrang (sondern Rang $(p-1)+(q-1)+(p-1) \cdot (q-1)+1 = p \cdot q$), weshalb die Parameter nicht identifizierbar sind. Das Modell muss deshalb durch separate Nebenbedingungen an die Parameter der Haupteffekte und Interaktion wieder in eines mit weniger Parametern überführt werden. Zunächst seien dafür $\mathbf{v}_1^\top \beta_1^* = 0$ und $\mathbf{v}_2^\top \beta_2^* = 0$ die Nebenbedingungen für die Parameter der Haupteffekte, deren frei variierbare Anzahl sich dadurch auf $p - 1$ bzw. $q - 1$ reduziert. Weiter bezeichne $\mathbf{B}_{1 \times 2}^*$ die in einer $(p \times q)$ -Matrix angeordneten ursprünglichen Parameter $\beta_{1 \times 2}^*$ der Interaktion. Die

Nebenbedingung für diese Parameter lässt sich dann als $\mathbf{v}_1^\top \mathbf{B}_{1\times 2}^* = \mathbf{0}^\top$ und $\mathbf{B}_{1\times 2}^* \mathbf{v}_2 = \mathbf{0}$ schreiben, wodurch deren frei variierbare Anzahl nunmehr $(p-1) \cdot (q-1)$ beträgt.

Analog zum einfaktoriellen Fall ist nun zunächst die $(p \times (p-1))$ -Codiermatrix \mathbf{C}_1 für die $p-1$ Parameter des ersten Gruppierungsfaktors im Vektor $\boldsymbol{\beta}_1$ nach demselben Schema zu bilden wie die $(q \times (q-1))$ -Codiermatrix \mathbf{C}_2 für die $q-1$ Parameter des zweiten Gruppierungsfaktors im Vektor $\boldsymbol{\beta}_2$. Der Rang von $[\mathbf{1} | \mathbf{C}_1]$ sei also p , der Rang von $[\mathbf{1} | \mathbf{C}_2]$ sei q , und es gelte $\mathbf{v}_1^\top \mathbf{C}_1 = \mathbf{0}$ ebenso wie $\mathbf{v}_2^\top \mathbf{C}_2 = \mathbf{0}$. Die $((p \cdot q) \times (p-1) \cdot (q-1))$ -Codiermatrix $\mathbf{C}_{1\times 2}$ für die $(p-1) \cdot (q-1)$ Parameter der Interaktion im Vektor $\boldsymbol{\beta}_{1\times 2}$ erhält man als Kronecker-Produkt \otimes von \mathbf{C}_1 und \mathbf{C}_2 .⁵⁸ Der gemeinsame Vektor der reduzierten Parameter sei schließlich $\boldsymbol{\beta} = (\beta_0, \boldsymbol{\beta}_1^\top, \boldsymbol{\beta}_2^\top, \boldsymbol{\beta}_{1\times 2}^\top)^\top$.

Der Zusammenhang zwischen den neuen Parametern und den ursprünglichen Parametern, die den genannten Nebenbedingungen genügen, ist nun für jeden der drei Effekte wie im einfaktoriellen Fall. Der entsprechende Zusammenhang für die Parameter der Interaktion lässt sich dabei auf zwei verschiedene Arten formulieren – einmal mit $\boldsymbol{\beta}_{1\times 2}^*$ und einmal mit $\mathbf{B}_{1\times 2}^*$.

$$\begin{aligned}\boldsymbol{\beta}_1^* &= \mathbf{C}_1 \boldsymbol{\beta}_1 \\ \boldsymbol{\beta}_2^* &= \mathbf{C}_2 \boldsymbol{\beta}_2 \\ \boldsymbol{\beta}_{1\times 2}^* &= \mathbf{C}_{1\times 2} \boldsymbol{\beta}_{1\times 2} = (\mathbf{C}_2 \otimes \mathbf{C}_1) \boldsymbol{\beta}_{1\times 2} \\ \mathbf{B}_{1\times 2}^* &= \mathbf{C}_1 \boldsymbol{\beta}_{1\times 2} \mathbf{C}_2^\top\end{aligned}$$

Aus dem Produkt jeweils einer ursprünglichen Inzidenzmatrix mit der zugehörigen Codiermatrix berechnen sich die zugehörigen Inzidenzmatrizen für die neuen Parameter: $\mathbf{X}_1 = \mathbf{X}_1^* \mathbf{C}_1$ für $\boldsymbol{\beta}_1$, $\mathbf{X}_2 = \mathbf{X}_2^* \mathbf{C}_2$ für $\boldsymbol{\beta}_2$ und $\mathbf{X}_{1\times 2} = \mathbf{X}_{1\times 2}^* \mathbf{C}_{1\times 2}$ für $\boldsymbol{\beta}_{1\times 2}$. Dabei kann $\mathbf{X}_{1\times 2}$ äquivalent auch nach demselben Prinzip wie $\mathbf{X}_{1\times 2}^*$ gebildet werden: Die $(p-1) \cdot (q-1)$ paarweisen Produkte der Spalten von \mathbf{X}_1 und \mathbf{X}_2 werden dazu spaltenweise zu $\mathbf{X}_{1\times 2}$ zusammengestellt. Die reduzierte $(n \times p \cdot q)$ -Designmatrix mit vollem Spaltenrang ist $\mathbf{X} = [\mathbf{1} | \mathbf{X}_1 | \mathbf{X}_2 | \mathbf{X}_{1\times 2}]$, wobei $n > p \cdot q$ vorausgesetzt wird. Das Modell mit identifizierbaren Parametern lautet damit wieder analog zur einfaktoriellen Situation:

$$E(\mathbf{y}) = [\mathbf{1} | \mathbf{X}_1 | \mathbf{X}_2 | \mathbf{X}_{1\times 2}] \begin{pmatrix} \beta_0 \\ \boldsymbol{\beta}_1 \\ \boldsymbol{\beta}_2 \\ \boldsymbol{\beta}_{1\times 2} \end{pmatrix} = \mathbf{X} \boldsymbol{\beta}$$

Der $p \cdot q$ -Vektor $\boldsymbol{\beta}$ der Parameter ist so wie im einfaktoriellen Fall über $\mathbf{X}^+ E(\mathbf{y})$ identifizierbar.

Die inhaltliche Bedeutung der Parameter in $\boldsymbol{\beta}$ hängt wie im einfaktoriellen Fall von der Wahl der Nebenbedingungen und Codiermatrizen ab. Voreinstellung in R sind Treatment-Kontraste:

⁵⁸Hierbei ist die Reihenfolge relevant, mit denen man die Spalten von $\mathbf{X}_{1\times 2}^*$ und entsprechend die Parameter in $\boldsymbol{\beta}_{1\times 2}^*$ ordnet, die zu den Kombinationen jk der Faktorstufen der Gruppierungsfaktoren gehören: Variiert (wie hier) der erste Index j schnell und der zweite Index k langsam ($\boldsymbol{\beta}_{1\times 2}^* = (\beta_{11}^*, \beta_{21}^*, \beta_{31}^*, \beta_{12}^*, \beta_{22}^*, \beta_{32}^*)^\top$), gilt $\mathbf{C}_{1\times 2} = \mathbf{C}_2 \otimes \mathbf{C}_1$. Dies ist die Voreinstellung in R, die sich etwa an der Ausgabe von `interaction(⟨UV1⟩, ⟨UV2⟩)` zeigt (Abschn. 2.6.2). Variiert dagegen j langsam und k schnell ($\boldsymbol{\beta}_{1\times 2}^* = (\beta_{11}^*, \beta_{12}^*, \beta_{21}^*, \beta_{22}^*, \beta_{31}^*, \beta_{32}^*)^\top$), ist $\mathbf{C}_{1\times 2} = \mathbf{C}_1 \otimes \mathbf{C}_2$ zu setzen.

Für sie ergibt sich hier $\mathbf{v}_1 = (1, 0, \dots, 0)^\top$ und $\mathbf{v}_2 = (0, 1, \dots, 0)^\top$, d. h. es gilt $\beta_{1,1}^* = 0, \beta_{2,1}^* = 0$ sowie $\beta_{1\times 2,1k}^* = 0$ für alle k wie auch $\beta_{1\times 2,j1}^* = 0$ für alle j . In der ersten Zeile und Spalte von $\mathbf{B}_{1\times 2}^*$ sind also alle Einträge 0. Mit Treatment-Kontrasten erhält man für die zweifaktorielle Varianzanalyse mit $p = 3, q = 2$ und einer Person pro Gruppe folgende Codiermatrizen \mathbf{C}_t :

$$\mathbf{C}_{t1} = \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} \quad \mathbf{C}_{t2} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \quad \mathbf{C}_{t1\times 2} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Die ursprünglichen Parameter erhalten mit Treatment-Kontrasten folgende Bedeutung: $\beta_0 = \mu_{11}, \beta_{1,j}^* = \mu_{j1} - \mu_{11}, \beta_{2,k}^* = \mu_{1k} - \mu_{11}, \beta_{1\times 2,jk}^* = (\mu_{jk} - \mu_{j1}) - (\mu_{1k} - \mu_{11})$. Das zugehörige Modell mit reduzierter Designmatrix \mathbf{X} und dem reduzierten Parametervektor $\boldsymbol{\beta}$ lautet:

$$E(\mathbf{y}) = \begin{pmatrix} \mu_{11} \\ \mu_{21} \\ \mu_{31} \\ \mu_{12} \\ \mu_{22} \\ \mu_{32} \end{pmatrix} = \left(\begin{array}{c|cc|cc|cc} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 \end{array} \right)_{\mathbf{X}} \begin{pmatrix} \beta_0 \\ \beta_{1,2} \\ \beta_{1,3} \\ \beta_{2,2} \\ \beta_{1\times 2,22} \\ \beta_{1\times 2,32} \end{pmatrix}_{\boldsymbol{\beta}}$$

$$= \begin{pmatrix} \beta_0 \\ \beta_0 + \beta_{1,2} \\ \beta_0 + \beta_{1,3} \\ \beta_0 + \beta_{2,2} \\ \beta_0 + \beta_{1,2} + \beta_{2,2} + \beta_{1\times 2,22} \\ \beta_0 + \beta_{1,3} + \beta_{2,2} + \beta_{1\times 2,32} \end{pmatrix}$$

Mit der ungewichteten Effektcodierung ergibt sich hier für die Nebenbedingungen $\mathbf{v}_1 = \mathbf{1}$ und $\mathbf{v}_2 = \mathbf{1}$, d. h. es gilt $\sum_j \beta_{1,j}^* = 0, \sum_k \beta_{2,k}^* = 0$ sowie $\sum_j \beta_{1\times 2,jk}^* = 0$ für alle k wie auch $\sum_k \beta_{1\times 2,jk}^* = 0$ für alle j (alle Zeilen- und Spaltensummen von $\mathbf{B}_{1\times 2}^*$ sind 0). Für die zweifaktorielle Varianzanalyse mit $p = 3, q = 2$ und einer Person pro Gruppe erhält man mit Effektcodierung folgende Codiermatrizen \mathbf{C}_e :

$$\mathbf{C}_{e1} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & -1 \end{pmatrix} \quad \mathbf{C}_{e2} = \begin{pmatrix} 1 \\ -1 \end{pmatrix} \quad \mathbf{C}_{e1\times 2} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & -1 \\ -1 & 0 \\ 0 & -1 \\ 1 & 1 \end{pmatrix}$$

Die ursprünglichen Parameter $\beta_0, \beta_{1,jk}^*, \beta_{2,jk}^*$ und $\beta_{1\times 2,jk}^*$ erhalten durch die Nebenbedingungen die Bedeutung der Parameter μ, α_j, β_k und $(\alpha\beta)_{jk}$ der häufig gewählten Formulierung des Modells der zweifaktoriellen Varianzanalyse als $\mu_{jk} = \mu + \alpha_j + \beta_k + (\alpha\beta)_{jk}$.⁵⁹ Mit Effektcodierung lautet das Modell mit reduzierter Designmatrix \mathbf{X} und dem reduzierten Parametervektor $\boldsymbol{\beta}$:

$$E(\mathbf{y}) = \begin{pmatrix} \mu_{11} \\ \mu_{21} \\ \mu_{31} \\ \mu_{12} \\ \mu_{22} \\ \mu_{32} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & -1 & -1 & 1 & -1 & -1 \\ 1 & 1 & 0 & -1 & -1 & 0 \\ 1 & 0 & 1 & -1 & 0 & -1 \\ 1 & -1 & -1 & -1 & 1 & 1 \end{pmatrix}_{\mathbf{1} \quad \mathbf{X}_1 \quad \mathbf{X}_2 \quad \mathbf{X}_{1\times 2}} \begin{pmatrix} \beta_0 \\ \beta_{1,1} \\ \beta_{1,2} \\ \beta_{2,1} \\ \beta_{1\times 2,11} \\ \beta_{1\times 2,21} \end{pmatrix}_{\boldsymbol{\beta}}$$

$$= \begin{pmatrix} \beta_0 + \beta_{1,1} & & & + \beta_{2,1} & + \beta_{1\times 2,11} & \\ \beta_0 & + \beta_{1,2} & + \beta_{2,1} & & & + \beta_{1\times 2,21} \\ \beta_0 - (\beta_{1,1} + \beta_{1,2}) & + \beta_{2,1} & - (\beta_{1\times 2,11} + \beta_{1\times 2,21}) & & & \\ \beta_0 + \beta_{1,1} & & - \beta_{2,1} & - \beta_{1\times 2,11} & & \\ \beta_0 & + \beta_{1,2} & - \beta_{2,1} & & & - \beta_{1\times 2,21} \\ \beta_0 - (\beta_{1,1} + \beta_{1,2}) & - \beta_{2,1} & + \beta_{1\times 2,11} & + \beta_{1\times 2,21} & & \end{pmatrix}$$

12.9.4 Parameterschätzungen, Vorhersage und Residuen

Der n -Vektor der Beobachtungen $\mathbf{y} = (y_1, \dots, y_n)^\top$ ergibt sich im Modell des ALM als Summe von $E(\mathbf{y})$ und einem n -Vektor zufälliger Fehler $\boldsymbol{\epsilon} = (\epsilon_1, \dots, \epsilon_n)^\top$.

$$\begin{aligned} y_i &= E(y_i) + \epsilon_i \\ \mathbf{y} &= \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon} \end{aligned}$$

Die Fehler sollen dabei gemeinsam unabhängig und auch unabhängig von \mathbf{X} sein. Ferner soll für alle $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$ gelten, womit $\boldsymbol{\epsilon}$ multivariat normalverteilt ist mit $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \sigma^2 \mathbf{I})$. Die Voraussetzung gleicher Fehlervarianzen wird im Fall der Regression als Homoskedastizität bezeichnet, im Fall der Varianzanalyse als Varianzhomogenität (Abb. 12.6). Die Beobachtungen sind dann ihrerseits multivariat normalverteilt mit $\mathbf{y} \sim \mathcal{N}(\mathbf{X}\boldsymbol{\beta}, \sigma^2 \mathbf{I})$.⁶⁰

Der Vektor $\hat{\boldsymbol{\beta}}$ der i.S. der geringsten Quadratsumme der Residuen optimalen Parameterschätzungen ist gleich $\mathbf{X}^+ \mathbf{y} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$.⁶¹ Man erhält $\hat{\boldsymbol{\beta}}$ also als Koordinaten des Kriteriums,

⁵⁹Hierbei seien $\alpha_j = \mu_{j.} - \mu$ die Effektgrößen des Haupteffekts des ersten Faktors, $\beta_k = \mu_{.k} - \mu$ die des Haupteffekts des zweiten Faktors und $(\alpha\beta)_{jk} = \mu_{jk} - (\mu + \alpha_j + \beta_k) = \mu_{jk} - \mu_{j.} - \mu_{.k} + \mu$ die der Interaktion. Dabei seien $\mu_{j.} = \frac{1}{q} \sum_k \mu_{jk}$, $\mu_{.k} = \frac{1}{p} \sum_j \mu_{jk}$ und $\mu = \frac{1}{p \cdot q} \sum_j \sum_k \mu_{jk}$ ungewichtete mittlere Erwartungswerte. Liegen gleiche, oder zumindest proportional ungleiche Zellbesetzungen vor ($\frac{n_{jk}}{n_{jk'}} = \frac{n_{j'k}}{n_{j'k'}}$ sowie $\frac{n_{jk}}{n_{j'k}} = \frac{n_{jk'}}{n_{j'k'}}$ für alle j, j', k, k'), lässt sich die Parametrisierung mit gewichteten mittleren Erwartungswerten durch die gewichtete Effektcodierung umsetzen (Fußnote 57).

⁶⁰Zunächst ist $E(\mathbf{y}) = E(\mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}) = \mathbf{X}\boldsymbol{\beta} + E(\boldsymbol{\epsilon}) = \mathbf{X}\boldsymbol{\beta}$. Weiter gilt $V(\mathbf{y}) = V(\mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}) = V(\boldsymbol{\epsilon}) = \sigma^2 \mathbf{I}$.

⁶¹In der Varianzanalyse werden die reduzierten Parameter geschätzt. Für die Beziehung zwischen geschätzten ursprünglichen Parametern und geschätzten reduzierten Parametern gilt $\hat{\boldsymbol{\beta}}^* = [\mathbf{1} | \mathbf{C}] \hat{\boldsymbol{\beta}}$. Auf Basis eines mit `aov()` oder `lm()` angepassten linearen Modells erhält man $\hat{\boldsymbol{\beta}}$ mit `coef()` und $\hat{\boldsymbol{\beta}}^*$ mit `dummpy.coef()`.

Verteilungsvoraussetzung lineare Regression

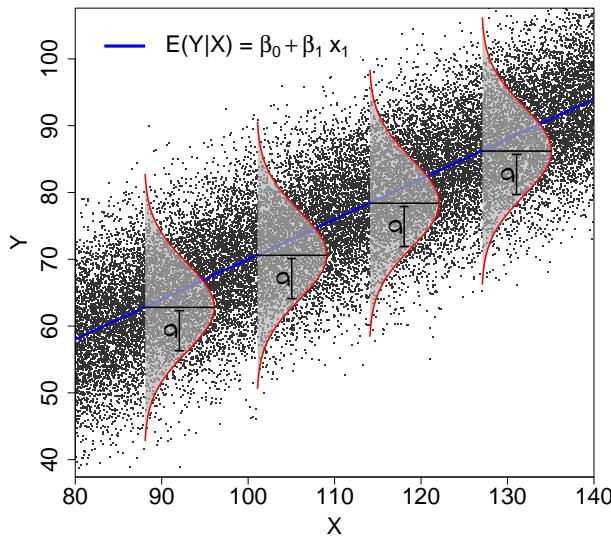


Abbildung 12.6: Verteilungsvoraussetzungen im allgemeinen linearen Modell am Beispiel der einfachen linearen Regression

das orthogonal auf V projiziert wurde, bzgl. der durch \mathbf{X} definierten Basis (Abschn. 12.1.7). Es gilt $\hat{\beta} \sim \mathcal{N}(\beta, \sigma^2(\mathbf{X}^\top \mathbf{X})^{-1})$, $\hat{\beta}$ ist damit erwartungstreuer Schätzer für β .⁶²

Der n -Vektor $\hat{\mathbf{y}} = (\hat{y}_1, \dots, \hat{y}_n)^\top$ der vorhergesagten Werte berechnet sich durch $\mathbf{X}\mathbf{X}^+ \mathbf{y} = \mathbf{P}\mathbf{y}$, wobei die orthogonale Projektion $\mathbf{P} = \mathbf{X}(\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top$ auch als *Hat-Matrix* bezeichnet wird. Man erhält $\hat{\mathbf{y}}$ also als Koordinaten des Kriteriums, das orthogonal auf V projiziert wurde, bzgl. der Standardbasis. Die Vorhersage $\hat{\mathbf{y}}$ liegt damit in V .

Für den n -Vektor der Residuen $\mathbf{e} = (e_1, \dots, e_n)^\top$ gilt $\mathbf{e} = \mathbf{y} - \hat{\mathbf{y}} = \mathbf{I}\mathbf{y} - \mathbf{P}\mathbf{y} = (\mathbf{I} - \mathbf{P})\mathbf{y}$. Die Residuen ergeben sich also als Projektion des Kriteriums auf das orthogonale Komplement von V . Die Residuen \mathbf{e} liegen damit in V^\perp und sind senkrecht zur Vorhersage. Der *Fehlerraum* V^\perp besitzt die Dimension $\text{Rang}(\mathbf{I} - \mathbf{P}) = n - \text{Rang}(\mathbf{X})$. Weiter gilt $E(\mathbf{e}) = \mathbf{0}$ und $V(\mathbf{e}) = \sigma^2(\mathbf{I} - \mathbf{P})$.⁶³

Erwartungstreuer Schätzer für σ^2 ist die Quadratsumme der Residuen $SS_e = \|\mathbf{e}\|^2 = \mathbf{y}^\top (\mathbf{I} - \mathbf{P})\mathbf{y}$ geteilt durch die Anzahl der Fehler-Freiheitsgrade, die gleich der Dimension von V^\perp ist.⁶⁴

In der Regression mit p Prädiktoren und absolutem Term ist $\hat{\sigma}^2 = \frac{\|\mathbf{e}\|^2}{n-(p+1)}$ der quadrierte Standardschätzfehler, in der einfaktoriellen Varianzanalyse mit p Gruppen ist $\hat{\sigma}^2 = \frac{\|\mathbf{e}\|^2}{n-p}$ die mittlere Quadratsumme der Residuen.

⁶²Zunächst ist $E(\hat{\beta}) = E(\mathbf{X}^+ \mathbf{y}) = \mathbf{X}^+ E(\mathbf{y}) = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{X} \beta = \beta$. Weiter gilt $V(\hat{\beta}) = V(\mathbf{X}^+ \mathbf{y}) = \mathbf{X}^+ V(\mathbf{y})(\mathbf{X}^+)^T = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \sigma^2 \mathbf{I} \mathbf{X} (\mathbf{X}^\top \mathbf{X})^{-1} = \sigma^2 (\mathbf{X}^\top \mathbf{X})^{-1}$.

⁶³Zunächst ist $E(\mathbf{e}) = E((\mathbf{I} - \mathbf{P})\mathbf{y}) = E(\mathbf{y}) - E(\mathbf{P}\mathbf{y}) = \mathbf{X}\beta - \mathbf{P}\mathbf{X}\beta$. Da die Spalten von \mathbf{X} in V liegen, bleiben sie durch \mathbf{P} unverändert, es folgt also $E(\mathbf{e}) = \mathbf{X}\beta - \mathbf{P}\mathbf{X}\beta = \mathbf{0}$. Weiter gilt $V(\mathbf{e}) = V((\mathbf{I} - \mathbf{P})\mathbf{y}) = (\mathbf{I} - \mathbf{P})V(\mathbf{y})(\mathbf{I} - \mathbf{P})^\top = \sigma^2(\mathbf{I} - \mathbf{P})(\mathbf{I} - \mathbf{P})^\top$. Als orthogonale Projektion ist $\mathbf{I} - \mathbf{P}$ symmetrisch und idempotent, es gilt also $V(\mathbf{e}) = \sigma^2(\mathbf{I} - \mathbf{P})(\mathbf{I} - \mathbf{P})^\top = \sigma^2(\mathbf{I} - \mathbf{P})$.

⁶⁴ $SS_e = \sum_i e_i^2 = \|\mathbf{e}\|^2 = \mathbf{e}^\top \mathbf{e} = (\mathbf{y} - \hat{\mathbf{y}})^\top (\mathbf{y} - \hat{\mathbf{y}}) = \mathbf{y}^\top \mathbf{y} - \mathbf{y}^\top \hat{\mathbf{y}} - \hat{\mathbf{y}}^\top \mathbf{y} + \hat{\mathbf{y}}^\top \hat{\mathbf{y}} = \mathbf{y}^\top \mathbf{y} - \mathbf{y}^\top \mathbf{P}\mathbf{y} - (\mathbf{P}\mathbf{y})^\top \mathbf{y} + (\mathbf{P}\mathbf{y})^\top (\mathbf{P}\mathbf{y}) = \mathbf{y}^\top \mathbf{y} - \mathbf{y}^\top \mathbf{P}\mathbf{y} - \mathbf{y}^\top \mathbf{P}\mathbf{y} + \mathbf{y}^\top \mathbf{P}\mathbf{P}\mathbf{y} = \mathbf{y}^\top \mathbf{y} - \mathbf{y}^\top \mathbf{P}\mathbf{y} = \mathbf{y}^\top (\mathbf{I} - \mathbf{P})\mathbf{y}$.

Im multivariaten Fall werden mehrere Datenvektoren von Zielgrößen \mathbf{y}_l spaltenweise zu einer Matrix \mathbf{Y} zusammengestellt. Setzt man in den genannten Formeln \mathbf{Y} für \mathbf{y} ein, berechnen sich Parameterschätzungen, die spaltenweise aus den Vektoren der Vorhersagen $\hat{\mathbf{y}}_l$ zusammengestellte Matrix $\hat{\mathbf{Y}}$ und die spaltenweise aus den Vektoren der Residuen \mathbf{e}_l zusammengestellte Matrix \mathbf{E} wie in der univariaten Formulierung.

12.9.5 Hypothesen über parametrische Funktionen testen

Im ALM können verschiedenartige Hypothesen über die Modellparameter β_j getestet werden. Eine Gruppe solcher Hypothesen ist jene über den Wert einer *parametrischen Funktion*. Im univariaten Fall ist eine parametrische Funktion $\psi = \sum_j c_j \beta_j = \mathbf{c}^\top \boldsymbol{\beta}$ eine Linearkombination der Parameter, wobei die Koeffizienten c_j zu einem Vektor \mathbf{c} zusammengestellt werden. Beispiele für parametrische Funktionen sind etwa a-priori Kontraste aus der Varianzanalyse (Abschn. 7.4.6). Auch ein Parameter β_j selbst (z.B. ein Gewicht in der Regression) ist eine parametrische Funktion, bei der \mathbf{c} der j -te Einheitsvektor ist. Die H_0 lässt sich dann als $\psi = \psi_0$ mit einem festen Wert ψ_0 formulieren.

Eine parametrische Funktion wird mit Hilfe der Parameterschätzungen als $\hat{\psi} = \mathbf{c}^\top \hat{\boldsymbol{\beta}}$ erwartungstreu geschätzt, und es gilt $\hat{\psi} \sim \mathcal{N}(\psi, \sigma^2 \mathbf{c}^\top (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{c})$.⁶⁵ $\hat{\psi}$ lässt sich auch direkt als Linearkombination der Beobachtungen y_i im Vektor \mathbf{y} formulieren:

$$\hat{\psi} = \mathbf{c}^\top \hat{\boldsymbol{\beta}} = \mathbf{c}^\top (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y} = (\mathbf{X} (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{c})^\top \mathbf{y} = \mathbf{a}^\top \mathbf{y}$$

Dabei ist $\mathbf{a} = \mathbf{X} (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{c}$ der zu \mathbf{c} gehörende n -Vektor der *Schätzerkoeffizienten*, mit dem auch $\hat{\psi} = \mathbf{a}^\top \hat{\mathbf{y}}$ ⁶⁶ sowie $\sigma_{\hat{\psi}}^2 = \mathbf{a}^\top \mathbf{a} \sigma^2 = \|\mathbf{a}\|^2 \sigma^2$ gilt.⁶⁷ Damit lässt sich die t -Teststatistik in der üblichen Form $\frac{\hat{\psi} - \psi_0}{\hat{\sigma}_{\hat{\psi}}}$ definieren:

$$t = \frac{\hat{\psi} - \psi_0}{\hat{\sigma} \sqrt{\mathbf{c}^\top (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{c}}} = \frac{\hat{\psi} - \psi_0}{\|\mathbf{a}\| \sqrt{\|\mathbf{e}\|^2 / (n - \text{Rang}(\mathbf{X}))}}$$

Unter H_0 ist t zentral t -verteilt mit $n - \text{Rang}(\mathbf{X})$ Freiheitsgraden (im Fall der Regression $n - (p + 1)$, im Fall der einfaktoriellen Varianzanalyse $(n - p)$).

12.9.6 Lineare Hypothesen als Modellvergleiche formulieren

Analog zur einzelnen parametrischen Funktion können Hypothesen über einen Vektor ψ von m parametrischen Funktionen $\psi_j = \mathbf{c}_j^\top \boldsymbol{\beta}$ gleichzeitig formuliert werden, wobei die ψ_j linear unabhängig sein sollen. Diese *linearen* Hypothesen besitzen die Form $\psi = \mathbf{L} \boldsymbol{\beta}$. Dabei ist \mathbf{L} eine

⁶⁵Zunächst ist $E(\hat{\psi}) = E(\mathbf{c}^\top \hat{\boldsymbol{\beta}}) = \mathbf{c}^\top E(\hat{\boldsymbol{\beta}}) = \mathbf{c}^\top \boldsymbol{\beta} = \psi$. Weiter gilt $V(\hat{\psi}) = V(\mathbf{c}^\top \hat{\boldsymbol{\beta}}) = \mathbf{c}^\top V(\hat{\boldsymbol{\beta}}) \mathbf{c} = \mathbf{c}^\top \sigma^2 (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{c}$. Bei $\hat{\psi}$ handelt es sich um einen *Gauß-Markoff-Schätzer*, also den linearen erwartungstreuen Schätzer mit der geringsten Varianz.

⁶⁶Zunächst ist $\mathbf{X}^\top \mathbf{a} = \mathbf{X}^\top \mathbf{X} (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{c} = \mathbf{c}$, also $\mathbf{c}^\top = \mathbf{a}^\top \mathbf{X}$. Damit gilt $\mathbf{a}^\top \mathbf{y} = \mathbf{c}^\top (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y} = \mathbf{a}^\top \mathbf{X} (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y} = \mathbf{a}^\top \mathbf{P} \mathbf{y} = \mathbf{a}^\top \hat{\mathbf{y}}$.

⁶⁷ $\|\mathbf{a}\|^2 = \mathbf{a}^\top \mathbf{a} = (\mathbf{X} (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{c})^\top \mathbf{X} (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{c} = \mathbf{c}^\top (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{X} (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{c} = \mathbf{c}^\top (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{c}$.

Matrix aus den zeilenweise zusammengestellten Koeffizientenvektoren \mathbf{c}_j für die Linearkombinationen der allgemein u Parameter im Vektor $\boldsymbol{\beta}$ ($u > m$). \mathbf{L} ist dann eine $(m \times u)$ -Matrix mit Rang m . Der unter H_0 für $\boldsymbol{\psi}$ angenommene Vektor sei $\boldsymbol{\psi}_0$.

Im multivariaten Fall mit r Variablen Y_i und der $(u \times r)$ -Matrix der Parameter \mathbf{B} hat eine lineare Hypothese die Form $\boldsymbol{\Psi} = \mathbf{LB}$, wobei die unter H_0 für $\boldsymbol{\Psi}$ angenommene Matrix $\boldsymbol{\Psi}_0$ sei.

Die hier vorgestellten linearen Hypothesen aus der Varianzanalyse und Regression haben die Form $\boldsymbol{\psi}_0 = \mathbf{0}$ (bzw. $\boldsymbol{\Psi}_0 = \mathbf{0}$) und beziehen sich auf den Vergleich zweier nested Modelle (Abschn. 6.3.3, 7.4.3): Das umfassendere (*unrestricted*) H_1 -Modell mit Designmatrix \mathbf{X}_u besitzt dabei im univariaten Fall allgemein u freie Parameter. Für das eingeschränkte (*restricted*) H_0 -Modell mit Designmatrix \mathbf{X}_r nimmt man an, dass m der Parameter des umfassenderen Modells 0 sind, wodurch es noch $r = u - m$ freie Parameter besitzt. Man erhält \mathbf{X}_r , indem man die zu den auf 0 gesetzten Parametern gehörenden m Spalten von \mathbf{X}_u streicht.

Die freien Parameter des eingeschränkten Modells bilden eine echte Teilmenge jener des umfassenderen Modells. Daher liegt der von den Spalten von \mathbf{X}_r aufgespannte Unterraum V_r der Vorhersage des eingeschränkten Modells vollständig in jenem des umfassenderen Modells V_u , dem Erzeugnis der Spalten von \mathbf{X}_u . Umgekehrt liegt der Fehlerraum des umfassenderen Modells V_u^\perp vollständig in jenem des eingeschränkten Modells V_r^\perp . Die H_0 lässt sich auch so formulieren, dass $E(\mathbf{y})$ in V_r liegt.

Im multivariaten Fall besitzt \mathbf{B} allgemein u Zeilen mit freien Parametern. Für das eingeschränkte H_0 -Modell nimmt man an, dass m Zeilen von \mathbf{B} gleich 0 sind. Dadurch besitzt \mathbf{B} unter H_0 noch $r = u - m$ Zeilen mit freien Parametern. \mathbf{X}_u und \mathbf{X}_r stehen wie im univariaten Fall zueinander, unter H_0 soll also jede Spalte von $E(\mathbf{Y})$ in V_r liegen. Für die inferenzstatistische Prüfung s. Abschn. 12.9.7.

Univariate einfaktorielle Varianzanalyse

In der univariaten einfaktoriellen Varianzanalyse sind unter H_0 alle Gruppenerwartungswerte gleich. Dies ist äquivalent zur Hypothese, dass $\beta_{p-1} = \mathbf{0}$, also $\boldsymbol{\beta} = (\beta_0, \beta_{p-1}^\top)^\top = (\beta_0, \mathbf{0}^\top)^\top$ gilt. In $H_0 : \mathbf{L}\boldsymbol{\beta} = \mathbf{0}$ hat \mathbf{L} hier damit die Form $[\mathbf{0} | \mathbf{I}]$, wobei $\mathbf{0}$ der $(p-1)$ -Vektor $(0, \dots, 0)^\top$ und \mathbf{I} die $((p-1) \times (p-1))$ -Einheitsmatrix ist. Das H_0 -Modell mit einem Parameter β_0 lässt sich als $E(\mathbf{y}) = \mathbf{1}\beta_0$ formulieren. Im umfassenderen H_1 -Modell mit p Parametern gibt es keine Einschränkung für β_{p-1} , es ist damit identisch zum vollständigen Modell der einfaktoriellen Varianzanalyse $E(\mathbf{y}) = \mathbf{X}\boldsymbol{\beta}$: Hier ist also $u = p$, $\mathbf{X}_u = \mathbf{X}$, $r = 1$, $\mathbf{X}_r = \mathbf{1}$ und $m = p - 1$ (Abschn. 12.9.2).

Univariate zweifaktorielle Varianzanalyse: Quadratsummen vom Typ I

Wie in Abschn. 7.6.2 erläutert, existieren in der zweifaktoriellen Varianzanalyse (CRF- pq Design) Typen von Quadratsummen, die bei ungleichen Zellbesetzungen zu unterschiedlichen Tests führen können. Proportional ungleiche Zellbesetzungen sind dabei gegeben, wenn $\frac{n_{jk}}{n_{jk'}} = \frac{n_{j'k}}{n_{j'k'}}$ sowie $\frac{n_{jk}}{n_{j'k}} = \frac{n_{jk'}}{n_{j'k'}}$ für alle j, j', k, k' gilt. Ist dies nicht der Fall, spricht man von unbalancierten Zellbesetzungen.

Zunächst sind für den Test jedes Effekts (beide Haupteffekte und Interaktionseffekt) das eingeschränkte H_0 -Modell sowie das umfassendere H_1 -Modell zu definieren. Die H_0 des Haupteffekts des ersten Faktors lässt sich mit Hilfe des zugehörigen Parametervektors als $\beta_1 = \mathbf{0}$ formulieren. Bei Quadratsummen vom Typ I wählt man für den Test des Haupteffekts des ersten Faktors als H_0 -Modell das Gesamt-Nullmodell, bei dem alle Parametervektoren $\beta_1, \beta_2, \beta_{1 \times 2}$ gleich $\mathbf{0}$ sind und damit $E(\mathbf{y}) = \mathbf{1}\beta_0$ gilt. Hier gilt also für die Designmatrix des eingeschränkten Modells $\mathbf{X}_r = \mathbf{X}_0 = \mathbf{1}$. Das sequentiell folgende H_1 -Modell für den Test des ersten Faktors ist jenes, bei dem man die zugehörigen $p - 1$ Parameter in β_1 dem eingeschränkten Modell hinzufügt und die Designmatrix entsprechend erweitert, d. h. $\mathbf{X}_u = [\mathbf{1}|\mathbf{X}_1]$:

$$E(\mathbf{y}) = [\mathbf{1}|\mathbf{X}_1] \begin{pmatrix} \beta_0 \\ \beta_1 \end{pmatrix}$$

Die H_0 des Haupteffekts des zweiten Faktors lautet mit Hilfe des zugehörigen Parametervektors $\beta_2 = \mathbf{0}$. Bei Quadratsummen vom Typ I fällt die Wahl für das H_0 -Modell des zweiten Faktors auf das H_1 -Modell des ersten Faktors ($\mathbf{X}_r = [\mathbf{1}|\mathbf{X}_1]$). Das H_1 -Modell für den Test des zweiten Faktors ist jenes mit der sequentiell erweiterten Designmatrix $\mathbf{X}_u = [\mathbf{1}|\mathbf{X}_1|\mathbf{X}_2]$:

$$E(\mathbf{y}) = [\mathbf{1}|\mathbf{X}_1|\mathbf{X}_2] \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{pmatrix}$$

Durch den sequentiellen Aufbau der Modellvergleiche ist die Reihenfolge der Gruppierungsfaktoren beim Test der Haupteffekte in Fällen mit unbalancierten Zellbesetzungen rechnerisch bedeutsam.

Die H_0 des Interaktionseffekts lautet mit Hilfe des zugehörigen Parametervektors $\beta_{1 \times 2} = \mathbf{0}$. Die Wahl für das H_0 -Modell der Interaktion ist das sequentielle H_1 -Modell des zweiten Faktors ($\mathbf{X}_r = [\mathbf{1}|\mathbf{X}_1|\mathbf{X}_2]$). Das H_1 -Modell der Interaktion ist das in Abschn. 12.9.3 vorgestellte vollständige Modell der zweifaktoriellen Varianzanalyse ($\mathbf{X}_u = \mathbf{X}$):

$$E(\mathbf{y}) = [\mathbf{1}|\mathbf{X}_1|\mathbf{X}_2|\mathbf{X}_{1 \times 2}] \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \beta_{1 \times 2} \end{pmatrix} = \mathbf{X}\beta$$

In $H_0 : \mathbf{L}\beta = \mathbf{0}$ hat \mathbf{L} in den genannten Tests der Haupteffekte die Form $[\mathbf{0}|\mathbf{I}|\mathbf{0}]$ und beim Test der Interaktion die Form $[\mathbf{0}|\mathbf{I}]$. Dabei ist \mathbf{I} der Reihe nach die $((p - 1) \times (p - 1))$ -, $((q - 1) \times (q - 1))$ - und $((p - 1) \cdot (q - 1) \times (p - 1) \cdot (q - 1))$ -Einheitsmatrix und $\mathbf{0}$ eine passende Matrix aus 0-Einträgen.

Zweifaktorielle Varianzanalyse: Quadratsummen vom Typ II

Auch bei Quadratsummen vom Typ II unterscheiden sich die eingeschränkten und umfassenderen Modelle um jeweils $p - 1$ (erster Haupteffekt), $q - 1$ (zweiter Haupteffekt) und $(p - 1) \cdot (q - 1)$

(Interaktion) freie Parameter. Das H₁-Modell beim Test beider Haupteffekte ist hier:

$$E(\mathbf{y}) = [\mathbf{1} | \mathbf{X}_1 | \mathbf{X}_2] \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{pmatrix}$$

Das jeweilige H₀-Modell für den Test des Haupteffekts eines Faktors wird bei Quadratsummen vom Typ II dadurch gebildet, dass ausgehend vom H₁-Modell die Parameter des zu testenden Effekts auf 0 gesetzt und die zugehörigen Spalten der Designmatrix entfernt werden.

$$\begin{aligned} E(\mathbf{y}) &= [\mathbf{1} | \mathbf{X}_2] \begin{pmatrix} \beta_0 \\ \beta_2 \end{pmatrix} && (\text{H}_0 - \text{Modell für Test Faktor 1}) \\ E(\mathbf{y}) &= [\mathbf{1} | \mathbf{X}_1] \begin{pmatrix} \beta_0 \\ \beta_1 \end{pmatrix} && (\text{H}_0 - \text{Modell für Test Faktor 2}) \end{aligned}$$

Die Reihenfolge der Faktoren ist anders als bei Quadratsummen vom Typ I beim Test der Haupteffekte unwesentlich, selbst wenn unbalancierte Zellbesetzungen vorliegen. Liegen proportional ungleiche Zellbesetzungen vor, stimmen die Ergebnisse für den Test der Haupteffekte mit jenen aus Quadratsummen vom Typ I überein. Bei Quadratsummen vom Typ II ist die Wahl für H₀- und H₁-Modell beim Test der Interaktion gleich jener bei Quadratsummen vom Typ I und III, die Ergebnisse sind daher identisch.

Zweifaktorielle Varianzanalyse: Quadratsummen vom Typ III

Auch bei Quadratsummen vom Typ III unterscheiden sich die eingeschränkten und umfassenderen Modelle um jeweils p-1 (erster Haupteffekt), q-1 (zweiter Haupteffekt) und (p-1)·(q-1) (Interaktion) freie Parameter. Hier ist beim Test aller Effekte das H₁-Modell das in Abschn. [12.9.3](#) vorgestellte vollständige Modell der zweifaktoriellen Varianzanalyse:

$$E(\mathbf{y}) = [\mathbf{1} | \mathbf{X}_1 | \mathbf{X}_2 | \mathbf{X}_{1 \times 2}] \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \beta_{1 \times 2} \end{pmatrix} = \mathbf{X}\boldsymbol{\beta}$$

Das jeweilige H₀-Modell für den Test aller Effekte wird bei Quadratsummen vom Typ III dadurch gebildet, dass ausgehend vom vollständigen Modell die Parameter des zu testenden

Effekts auf 0 gesetzt und die zugehörigen Spalten der Designmatrix gestrichen werden:

$$E(\mathbf{y}) = [\mathbf{1} | \mathbf{X}_2 | \mathbf{X}_{1 \times 2}] \begin{pmatrix} \beta_0 \\ \beta_2 \\ \beta_{1 \times 2} \end{pmatrix} \quad (\text{H}_0 - \text{Modell für Test Faktor 1})$$

$$E(\mathbf{y}) = [\mathbf{1} | \mathbf{X}_1 | \mathbf{X}_{1 \times 2}] \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_{1 \times 2} \end{pmatrix} \quad (\text{H}_0 - \text{Modell für Test Faktor 2})$$

$$E(\mathbf{y}) = [\mathbf{1} | \mathbf{X}_1 | \mathbf{X}_2] \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{pmatrix} \quad (\text{H}_0 - \text{Modell für Test Interaktion})$$

Die Reihenfolge der Faktoren ist anders als bei Quadratsummen vom Typ I beim Test der Haupteffekte unwesentlich, selbst wenn unbalancierte Zellbesetzungen vorliegen. Bei gleichen Zellbesetzungen liefern alle drei Typen von Quadratsummen dieselben Ergebnisse. Bei Quadratsummen vom Typ III stimmt die Wahl für H_0 - und H_1 -Modell beim Test der Interaktion mit jener bei Quadratsummen vom Typ I und II überein, die Ergebnisse sind daher identisch. Ein Test des absoluten Terms β_0 ist mit Quadratsummen vom Typ III ebenso wenig möglich wie ein Test von Haupteffekten, die nicht Teil einer Interaktion sind.

Die H_0 -Modelle für Haupteffekte machen bei Quadratsummen vom Typ III keine Einschränkung für den Parametervektor $\beta_{1 \times 2}$ der Interaktion, obwohl die Parameter des zu testenden, und an der Interaktion beteiligten Haupteffekts auf 0 gesetzt werden. Die Modellvergleiche verletzen damit anders als bei Quadratsummen vom Typ I und II das Prinzip, dass Vorhersageterme höherer Ordnung nur in das Modell aufgenommen werden, wenn alle zugehörigen Terme niederer Ordnung ebenfalls Berücksichtigung finden.

Modellvergleiche für Quadratsummen vom Typ III können deshalb bei ungleichen Zellbesetzungen in Abhängigkeit von der verwendeten Nebenbedingung für die Parameter samt Codierschema unterschiedlich ausfallen (Venables, 2018). Richtige Ergebnisse erhält man mit $\mathbf{v} = \mathbf{1}$, was alle Codierschemata gewährleisten, deren Koeffizienten sich über die Spalten der Codiermatrix \mathbf{C} zu 0 summieren (Abschn. 12.9.2). Dazu zählen Effekt- und Helmert-Codierung, nicht aber Treatment-Kontraste (Dummy-Codierung) – die Voreinstellung in R. Bei Quadratsummen vom Typ I und II spielt die Wahl von Nebenbedingung und Codierschema dagegen keine Rolle für den inferenzstatistischen Test.

Multiple Regression

Die Modellvergleiche beim Test von p Prädiktoren X_j in der multiplen Regression werden nach demselben Prinzip wie in der zweifaktoriellen Varianzanalyse gebildet: Das Gesamt-Nullmodell ist $E(\mathbf{Y}) = \mathbf{1}\beta_0^\top$, bei dem für alle Kriterien \mathbf{y}_l der zugehörige Parametervektor der p Prädiktoren $\beta_{p,l} = \mathbf{0}$ und damit $\mathbf{B}_{p,0} = \mathbf{0}$ sowie $\mathbf{X}_0 = \mathbf{1}$ ist. Das vollständige Modell ist jenes mit allen Prädiktoren $E(\mathbf{Y}) = \mathbf{X}\mathbf{B}$.

Die Prädiktoren werden über Modellvergleiche getestet, die wie in der zweifaktoriellen Varianzanalyse vom verwendeten Typ der Quadratsummen abhängen. Die Reihenfolge der Prädiktoren

ist im multivariaten Fall bei Quadratsummen vom Typ I bedeutsam, sofern nicht alle Prädiktoren paarweise unkorreliert sind: Für den Test des ersten Prädiktors ist das eingeschränkte H_0 -Modell das Gesamt-Nullmodell, das zugehörige umfassendere H_1 -Modell ist jenes mit nur dem ersten Prädiktor. Für den Test des zweiten Prädiktors ist das eingeschränkte H_0 -Modell das sequentielle H_1 -Modell des ersten Prädiktors, das H_1 -Modell ist jenes mit den ersten beiden Prädiktoren. Für die Tests aller folgenden Prädiktoren gilt dies analog.

Bei Quadratsummen vom Typ II ist das H_1 -Modell beim Test aller Prädiktoren gleich dem vollständigen Modell mit allen Prädiktoren, jedoch ohne deren Interaktionen. Das H_0 -Modell beim Test eines Prädiktors X_j entsteht aus dem H_1 -Modell, indem der zu X_j gehörende Parametervektor $\beta_j^\top = \mathbf{0}^\top$ gesetzt und die passende Spalte der Designmatrix gestrichen wird.

Bei Quadratsummen vom Typ III ist das H_1 -Modell beim Test aller Prädiktoren gleich dem vollständigen Modell mit allen Prädiktoren und deren Interaktionen, sofern letztere berücksichtigt werden sollen. Das H_0 -Modell beim Test eines Prädiktors X_j entsteht aus dem H_1 -Modell, indem der zu X_j gehörende Parametervektor $\beta_j^\top = \mathbf{0}^\top$ gesetzt und die passende Spalte der Designmatrix gestrichen wird. Quadratsummen vom Typ II und III unterscheiden sich also nur, wenn die Regression auch Interaktionsterme einbezieht.

Sollen allgemein die Parameter von m Prädiktoren gleichzeitig daraufhin getestet werden, ob sie 0 sind, ist das Vorgehen analog, wobei unter H_0 die zugehörigen, aus \mathbf{B} stammenden m Parametervektoren $\beta_j^\top = \mathbf{0}^\top$ gesetzt und entsprechend ausgehend vom H_1 -Modell die passenden Spalten der Designmatrix gestrichen werden.

12.9.7 Lineare Hypothesen testen

Liegt eine lineare Hypothese der Form $\psi = \mathbf{L}\beta$ vor (Fox, Friendly & Weisberg, 2013), soll \mathbf{A} die zur $((u - r) \times u)$ -Matrix \mathbf{L} gehörende $((u - r) \times n)$ -Matrix bezeichnen, die zeilenweise aus den n -Vektoren der Schätzerkoeffizienten $\mathbf{a}_j = \mathbf{X}_u(\mathbf{X}_u^\top \mathbf{X}_u)^{-1} \mathbf{c}_j$ gebildet wird (Abschn. 12.9.5). Es gilt also $\mathbf{A}^\top = \mathbf{X}_u(\mathbf{X}_u^\top \mathbf{X}_u)^{-1} \mathbf{L}^\top$ und $\text{Rang}(\mathbf{A}) = \text{Rang}(\mathbf{L}) = u - r$. Aus $\mathbf{c} = \mathbf{X}^\top \mathbf{a}$ (Abschn. 12.9.6, Fußnote 66) folgt damit $\mathbf{L}^\top = \mathbf{X}_u^\top \mathbf{A}^\top$, also $\mathbf{L} = \mathbf{A} \mathbf{X}_u$.⁶⁸

Der Vektor der Schätzungen $\hat{\psi} = \mathbf{L}\hat{\beta}$ kann mit \mathbf{A} analog zur Schätzung einer einfachen parametrischen Funktion $\hat{\psi}_j = \mathbf{c}_j^\top \hat{\beta} = \mathbf{a}_j^\top \mathbf{y}$ auch als Abbildung des Vektors der Beobachtungen \mathbf{y} formuliert werden:

$$\hat{\psi} = \mathbf{L}\hat{\beta} = \mathbf{L}(\mathbf{X}_u^\top \mathbf{X}_u)^{-1} \mathbf{X}_u^\top \mathbf{y} = \mathbf{A}\mathbf{y}$$

Mit der Vorhersage $\hat{\mathbf{y}}$ gilt zudem $\hat{\psi} = \mathbf{A}\hat{\mathbf{y}}$.⁶⁹ $\hat{\psi}$ ist ein erwartungstreuer Gauß-Markoff-Schätzer mit Verteilung $\hat{\psi} \sim \mathcal{N}(\psi, \sigma^2 \mathbf{A}\mathbf{A}^\top)$.⁷⁰ Der Rang der Kovarianzmatrix $\Sigma_{\hat{\psi}} = \sigma^2 \mathbf{A}\mathbf{A}^\top$ beträgt $u - r$. Er ist also gleich der Differenz der Anzahl zu schätzender Parameter beider Modelle sowie gleich der Differenz der Dimensionen von V_u und V_r einerseits und von V_r^\perp und V_u^\perp andererseits.

⁶⁸ $\mathbf{A}\mathbf{X}_u = (\mathbf{X}_u(\mathbf{X}_u^\top \mathbf{X}_u)^{-1} \mathbf{L}^\top)^\top \mathbf{X}_u = \mathbf{L}(\mathbf{X}_u^\top \mathbf{X}_u)^{-1} \mathbf{X}_u^\top \mathbf{X}_u = \mathbf{L}$.

⁶⁹ $\hat{\psi} = \mathbf{L}\hat{\beta} = \mathbf{L}(\mathbf{X}_u^\top \mathbf{X}_u)^{-1} \mathbf{X}_u^\top \mathbf{y} = \mathbf{A}\mathbf{X}_u(\mathbf{X}_u^\top \mathbf{X}_u)^{-1} \mathbf{X}_u^\top \mathbf{y} = \mathbf{A}\hat{\mathbf{y}}$.

⁷⁰ Zunächst gilt $E(\mathbf{L}\hat{\beta}) = \mathbf{L}E(\hat{\beta}) = \mathbf{L}\beta = \psi$. Weiter ist $V(\hat{\psi}) = V(\mathbf{L}\hat{\beta}) = \mathbf{L}V(\hat{\beta})\mathbf{L}^\top = \sigma^2 \mathbf{L}(\mathbf{X}_u^\top \mathbf{X}_u)^{-1} \mathbf{L}^\top = \sigma^2 \mathbf{L}(\mathbf{X}_u^\top \mathbf{X}_u)^{-1} \mathbf{X}_u^\top \mathbf{X}_u(\mathbf{X}_u^\top \mathbf{X}_u)^{-1} \mathbf{L}^\top = \sigma^2 \mathbf{A}\mathbf{A}^\top$. Insbesondere ist also $\mathbf{A}\mathbf{A}^\top = \mathbf{L}(\mathbf{X}_u^\top \mathbf{X}_u)^{-1} \mathbf{L}^\top$.

Univariate Teststatistik

Um die Teststatistik für den univariaten Fall zu motivieren, ist zunächst festzustellen, dass für die quadrierte Mahalanobisdistanz (Abschn. 12.1.4) $\|\hat{\psi} - \psi_0\|_{M, \Sigma_{\hat{\psi}}}^2$ von $\hat{\psi}$ zu $\psi_0 = \mathbf{0}$ bzgl. $\Sigma_{\hat{\psi}}$ gilt (Fußnoten 69 und 70):

$$\begin{aligned}\|\hat{\psi} - \psi_0\|_{M, \Sigma_{\hat{\psi}}}^2 &= (\hat{\psi} - \mathbf{0})^\top \Sigma_{\hat{\psi}}^{-1} (\hat{\psi} - \mathbf{0}) = \frac{\hat{\psi}^\top (\mathbf{L}(\mathbf{X}_u^\top \mathbf{X}_u)^{-1} \mathbf{L}^\top)^{-1} \hat{\psi}}{\sigma^2} \\ &= \frac{\hat{\psi}^\top (\mathbf{A}\mathbf{A}^\top)^{-1} \hat{\psi}}{\sigma^2} = \frac{\mathbf{y}^\top \mathbf{A}^\top (\mathbf{A}\mathbf{A}^\top)^{-1} \mathbf{A}\mathbf{y}}{\sigma^2}\end{aligned}$$

Hier ist $\mathbf{A}^\top (\mathbf{A}\mathbf{A}^\top)^{-1} \mathbf{A}$ die Projektion auf das orthogonale Komplement von V_r in V_u (also $V_r^\perp \cap V_u$), dessen spaltenweise Basis \mathbf{A}^\top ist.⁷¹

Für die Schätzung der Fehlervarianz σ^2 benötigt man das vollständige (*full*) Modell mit Designmatrix \mathbf{X}_f und Projektionsmatrix \mathbf{P}_f (bisher einfach als \mathbf{X} und \mathbf{P} bezeichnet), das alle Parameter beinhaltet. Der zugehörige Vektor der Residuen sei $e_f = \mathbf{y} - \hat{\mathbf{y}}_f$, die Residual-Quadratsumme also $SS_{ef} = \|e_f\|^2$ mit Freiheitsgraden $df_{ef} = n - \text{Rang}(\mathbf{X}_f)$, der Dimension des Fehlerraumes V_f^\perp (Abschn. 12.9.4). Als erwartungstreuer Schätzer $\hat{\sigma}^2$ dient in allen Tests $SS_{ef}/df_{ef} = \mathbf{y}^\top (\mathbf{I} - \mathbf{P}_f)\mathbf{y}/(n - \text{Rang}(\mathbf{X}_f))$ (Abschn. 12.9.5, Fußnote 64) – also auch dann, wenn das umfassendere Modell nicht mit dem vollständigen Modell übereinstimmt. Damit lassen sich lineare Nullhypotesen $\mathbf{L}\beta = \mathbf{0}$ im univariaten Fall mit der folgenden Teststatistik prüfen:

$$\begin{aligned}F &= \frac{\hat{\psi}^\top (\mathbf{L}(\mathbf{X}_u^\top \mathbf{X}_u)^{-1} \mathbf{L}^\top)^{-1} \hat{\psi} / \text{Rang}(\Sigma_{\hat{\psi}})}{\hat{\sigma}^2} = \frac{\hat{\psi}^\top (\mathbf{A}\mathbf{A}^\top)^{-1} \hat{\psi} / (u - r)}{\hat{\sigma}^2} \\ &= \frac{\mathbf{y}^\top \mathbf{A}^\top (\mathbf{A}\mathbf{A}^\top)^{-1} \mathbf{A}\mathbf{y} / (\text{Rang}(\mathbf{X}_u) - \text{Rang}(\mathbf{X}_r))}{\mathbf{y}^\top (\mathbf{I} - \mathbf{P}_f)\mathbf{y} / (n - \text{Rang}(\mathbf{X}_f))}\end{aligned}$$

Unter H_0 ist F zentral F -verteilt mit $\text{Rang}(\mathbf{X}_u) - \text{Rang}(\mathbf{X}_r)$ Zähler- und $n - \text{Rang}(\mathbf{X}_f)$ Nenner-Freiheitsgraden. Das Modell einer Regression mit p Prädiktoren und absolutem Term hat $n - (p + 1)$ Fehler-Freiheitsgrade, das Modell einer einfaktoriellen Varianzanalyse mit p Gruppen $n - p$.

Der Zähler der Teststatistik lässt sich äquivalent umformulieren, wobei explizit Bezug zum Modellvergleich genommen wird. Dafür sei e_r der Vektor der Residuen des eingeschränkten Modells mit zugehöriger Projektionsmatrix \mathbf{P}_r und analog e_u der Vektor der Residuen des umfassenderen Modells mit Projektionsmatrix \mathbf{P}_u . In der einfaktoriellen Varianzanalyse sowie im Gesamt-Test der Regressionsanalyse sind das umfassendere Modell und das vollständige Modell identisch ($\mathbf{P}_u = \mathbf{P}_f$), ebenso ist dort das eingeschränkte Modell gleich dem Gesamt-Nullmodell ($\mathbf{P}_r = \mathbf{P}_0$).

⁷¹Es sei $\mathbf{y}_r = \mathbf{X}_u \mathbf{r}$ aus V_r und $\mathbf{y}_a = \mathbf{A}^\top \mathbf{a}$ aus dem Erzeugnis der Spalten von \mathbf{A}^\top mit \mathbf{r} und \mathbf{a} als zugehörigen Koordinatenvektoren bzgl. \mathbf{X}_u und \mathbf{A}^\top . Mit $\mathbf{A}^\top = \mathbf{X}_u (\mathbf{X}_u^\top \mathbf{X}_u)^{-1} \mathbf{L}^\top$ liegt \mathbf{y}_a als Linearkombination der Spalten von \mathbf{X}_u in V_u . Da \mathbf{y}_r in V_r liegt, gilt $\mathbf{L}\mathbf{y}_r = \mathbf{0}$. Damit folgt $\mathbf{y}_a^\top \mathbf{y}_r = (\mathbf{A}^\top \mathbf{a})^\top \mathbf{X}_u \mathbf{r} = \mathbf{a}^\top \mathbf{A} \mathbf{X}_u \mathbf{r} = \mathbf{a}^\top \mathbf{L} (\mathbf{X}_u^\top \mathbf{X}_u)^{-1} \mathbf{X}_u^\top \mathbf{X}_u \mathbf{r} = \mathbf{a}^\top \mathbf{L} \mathbf{r} = \mathbf{a}^\top \mathbf{0} = 0$, d. h. $\mathbf{y}_a \perp \mathbf{y}_r$. Also liegt \mathbf{y}_a auch in V_r^\perp .

Dann ist die Residual-Quadratsumme des eingeschränkten Modells $SS_{er} = \mathbf{y}^\top (\mathbf{I} - \mathbf{P}_r) \mathbf{y}$ mit Freiheitsgraden $df_{er} = n - \text{Rang}(\mathbf{X}_r)$ und die des umfassenderen Modells $SS_{eu} = \mathbf{y}^\top (\mathbf{I} - \mathbf{P}_u) \mathbf{y}$ mit Freiheitsgraden $df_{eu} = n - \text{Rang}(\mathbf{X}_u)$. Für die Differenz dieser Quadratsummen gilt $SS_{er} - SS_{eu} = \mathbf{y}^\top (\mathbf{P}_u - \mathbf{P}_r) \mathbf{y}$, wobei ihre Dimension $df_{er} - df_{eu} = \text{Rang}(\mathbf{X}_u) - \text{Rang}(\mathbf{X}_r)$ und $\mathbf{P}_u - \mathbf{P}_r = \mathbf{A}^\top (\mathbf{A} \mathbf{A}^\top)^{-1} \mathbf{A}$ ist. Damit lautet die Teststatistik:

$$\begin{aligned} F &= \frac{\mathbf{y}^\top (\mathbf{P}_u - \mathbf{P}_r) \mathbf{y} / (\text{Rang}(\mathbf{X}_u) - \text{Rang}(\mathbf{X}_r))}{\mathbf{y}^\top (\mathbf{I} - \mathbf{P}_f) \mathbf{y} / (n - \text{Rang}(\mathbf{X}_f))} \\ &= \frac{(SS_{er} - SS_{eu}) / (df_{er} - df_{eu})}{SS_{ef} / df_{ef}} \end{aligned}$$

Multivariate Teststatistiken

Die Quadratsummen des univariaten Falls verallgemeinern sich im multivariaten Fall zu folgenden Matrizen, auf denen letztlich die Teststatistiken basieren (Fox et al., 2013):

- $\mathbf{T} = \mathbf{Y}^\top (\mathbf{I} - \mathbf{P}_0) \mathbf{Y}$: Dies ist gleich der SSP-Matrix der Residuen des Gesamt-Nullmodells und damit gleich der SSP-Matrix der Gesamt-Daten.⁷²
- $\mathbf{W} = \mathbf{Y}^\top (\mathbf{I} - \mathbf{P}_f) \mathbf{Y}$: Dies ist gleich der SSP-Matrix der Residuen des vollständigen Modells (Fußnote 72), verallgemeinert also SS_{ef} .
- $\mathbf{B} = \mathbf{Y}^\top (\mathbf{P}_u - \mathbf{P}_r) \mathbf{Y}$: Dies ist gleich der SSP-Matrix der Vorhersagedifferenzen beider Modelle,⁷³ verallgemeinert also $SS_{er} - SS_{eu}$.

In der einfaktoriellen Varianzanalyse handelt es sich bei den Matrizen um multivariate Verallgemeinerungen der Quadratsummen zwischen (*between*, \mathbf{B}) und innerhalb (*within*, \mathbf{W}) der Gruppen, sowie der totalen Quadratsumme $\mathbf{T} = \mathbf{B} + \mathbf{W}$. In der Diagonale dieser Gleichung findet sich für jede der r Zielgrößen die zugehörige Quadratsummenzerlegung aus der univariaten Varianzanalyse wieder. \mathbf{B} ist hier gleichzeitig die SSP-Matrix der durch die zugehörigen Gruppenzentroide ersetzen Daten.

In der zweifaktoriellen Varianzanalyse ist \mathbf{W} die multivariate Verallgemeinerung der Residual-Quadratsumme. Es gibt nun für den Test der ersten UV eine Matrix \mathbf{B}_1 , für den Test der zweiten UV eine Matrix \mathbf{B}_2 und für den Test der Interaktion eine Matrix $\mathbf{B}_{1\times 2}$ jeweils als multivariate Verallgemeinerung der zugehörigen Effekt-Quadratsumme. Mit diesen Matrizen gilt bei Quadratsummen vom Typ I dann analog zur univariaten Situation $\mathbf{T} = \mathbf{B}_1 + \mathbf{B}_2 + \mathbf{B}_{1\times 2} + \mathbf{W}$ (Abschn. 7.6.2). Bei Quadratsummen vom Typ II und III gilt dies nur in orthogonalen Designs.

In der multivariaten multiplen Regression ist analog für den Test jedes Prädiktors j eine Matrix \mathbf{B}_j zu bilden, die sich aus der Differenzprojektion des zugehörigen Paares von eingeschränktem und umfassenderem Modell ergibt.

⁷² $\mathbf{X}_0 = \mathbf{1}$, daher ist die Matrix der Residuen $\mathbf{E} = (\mathbf{I} - \mathbf{P}_0) \mathbf{Y} = \mathbf{Q} \mathbf{Y}$ zentriert (Abschn. 12.1.7, Fußnote 21) und $(\mathbf{Q} \mathbf{Y})^\top (\mathbf{Q} \mathbf{Y})$ deren SSP-Matrix. Als orthogonale Projektion ist \mathbf{Q} symmetrisch und idempotent, weshalb $(\mathbf{Q} \mathbf{Y})^\top (\mathbf{Q} \mathbf{Y}) = \mathbf{Y}^\top \mathbf{Q}^\top \mathbf{Q} \mathbf{Y} = \mathbf{Y}^\top \mathbf{Q} \mathbf{Y}$ gilt.

⁷³ Zunächst ist die Matrix der Vorhersagedifferenzen $\hat{\mathbf{Y}}_u - \hat{\mathbf{Y}}_r = \mathbf{P}_u \mathbf{Y} - \mathbf{P}_r \mathbf{Y} = \mathbf{Y} - \mathbf{P}_r \mathbf{Y} - \mathbf{Y} + \mathbf{P}_u \mathbf{Y} = (\mathbf{I} - \mathbf{P}_r) \mathbf{Y} - (\mathbf{I} - \mathbf{P}_u) \mathbf{Y} = \mathbf{E}_r - \mathbf{E}_u$ gleich der Matrix der Differenzen der Residuen. Als Differenz zweier zentrierter Matrizen ist sie damit ihrerseits zentriert. Für die weitere Argumentation s. Fußnote 72.

Auf Basis der Matrizen \mathbf{B} und \mathbf{W} berechnen sich die üblichen multivariaten Teststatistiken für lineare Hypothesen im ALM, die im univariaten Fall alle äquivalent zum oben aufgeführten F -Bruch sind.

- Wilks' Λ : $\frac{\det(\mathbf{W})}{\det(\mathbf{W}+\mathbf{B})}$
- Roys Maximalwurzel: entweder der größte Eigenwert θ_1 von $(\mathbf{B}+\mathbf{W})^{-1}\mathbf{B}$ oder der größte Eigenwert λ_1 von $\mathbf{W}^{-1}\mathbf{B}$ (so definiert in R). Für die Umrechnung von θ_1 und λ_1 gilt $\lambda_1 = \frac{\theta_1}{1-\theta_1}$ sowie $\theta_1 = \frac{\lambda_1}{1+\lambda_1}$.
- Pillai-Bartlett-Spur: $\text{tr}((\mathbf{B} + \mathbf{W})^{-1}\mathbf{B})$
- Hotelling-Lawley-Spur: $\text{tr}(\mathbf{W}^{-1}\mathbf{B})$

12.9.8 Beispiel: Multivariate multiple Regression

Das Ergebnis der multivariaten multiplen Regression in Abschn. 12.5 lässt sich nun mit dem in Abschn. 12.9.6 und 12.9.7 dargestellten Vorgehen manuell prüfen. Im Beispiel sollen anhand der Prädiktoren Alter, Körpergröße und wöchentliche Dauer sportlicher Aktivitäten die Kriterien Körpergewicht und Gesundheit (i. S. eines geeigneten quantitativen Maßes) vorhergesagt werden. Zunächst sind die Designmatrix \mathbf{X} und die Projektion \mathbf{P}_f für das vollständige Regressionsmodell zu erstellen.⁷⁴

```
> Y <- cbind(weight, health)                      # Matrix der Kriterien
> X <- cbind(1, height, age, sport)               # Designmatrix
> XR <- model.matrix(~ height + age + sport)    # Designmatrix aus R
> all.equal(X, XR, check.attributes=FALSE)         # Vergleich
[1] TRUE

> Xplus <- solve(t(X) %*% X) %*% t(X)          # X+
> B       <- Xplus %*% Y                          # Parameterschätzungen
> Pf      <- X %*% Xplus                         # Projektion (Hat-Matrix)
> Yhat   <- Pf %*% Y                            # Vorhersage

# Kontrolle: vergleiche manuelle Berechnungen mit R
> fit <- lm(Y ~ height + age + sport)           # mult. Regr.-Modell
> all.equal(B, coef(fit), check.attributes=FALSE)  # Parameter
[1] TRUE

> all.equal(Yhat, fitted(fit), check.attributes=FALSE) # Vorhersage
[1] TRUE
```

Im Gesamt-Nullmodell $E(\mathbf{Y}) = \mathbf{1}\beta_0^\top$ der multiplen Regression sind alle Parameter bis auf β_0 gleich 0, Designmatrix \mathbf{X}_0 ist der Vektor $\mathbf{1}$. Es folgt die Berechnung der zugehörigen Projektion

⁷⁴Der gewählte Weg zur Berechnung der Projektionsmatrizen soll die mathematischen Formeln direkt umsetzen, ist aber numerisch nicht stabil und weicht von in R-Funktionen implementierten Rechnungen ab (Bates, 2004).

Kapitel 12 Multivariate Verfahren

\mathbf{P}_0 . Die Residuen ergeben sich aus der Projektion $\mathbf{I} - \mathbf{P}_0$ auf das orthogonale Komplement des von \mathbf{X}_0 aufgespannten Raumes.

```
# Gesamt-Nullmodell
> X0 <- X[, 1, drop=FALSE] # Designmatrix X0=1-Vektor
> P0 <- X0 %*% solve(t(X0) %*% X0) %*% t(X0) # Projektion
> Id <- diag(N) # n x n Einheitsmatrix I
> WW <- t(Y) %*% (Id - Pf) %*% Y # W
```

Schließlich muss für den Test jedes der drei Prädiktoren das zugehörige sequentielle Paar aus eingeschränktem H_0 -Modell und umfassenderem H_1 -Modell mit zugehörigen Designmatrizen \mathbf{X}_r und \mathbf{X}_u sowie ihren orthogonalen Projektionen \mathbf{P}_r und \mathbf{P}_u berechnet werden. Aus $\mathbf{P}_u - \mathbf{P}_r$ ergibt sich für jeden Prädiktor die Matrix \mathbf{B}_j , die in die zugehörigen Teststatistiken eingeht.

```
# Test Prädiktor 1: eingeschränktes Modell (= Gesamt-Nullmodell)
> Xr1 <- X0 # Designmatrix
> Pr1 <- P0 # Projektion

# Test Prädiktor 1: umfassenderes Modell
> Xu1 <- X[, c(1, 2)] # Designmatrix
> Pu1 <- Xu1 %*% solve(t(Xu1) %*% Xu1) %*% t(Xu1) # Projektion
> B1 <- t(Y) %*% (Pu1 - Pr1) %*% Y # Matrix B1

# Test Prädiktor 2: eingeschränktes Modell (= umfass. Modell Präd. 1)
> Xr2 <- Xu1 # Designmatrix
> Pr2 <- Pu1 # Projektion

# Test Prädiktor 2: umfassenderes Modell
> Xu2 <- X[, c(1, 2, 3)] # Designmatrix
> Pu2 <- Xu2 %*% solve(t(Xu2) %*% Xu2) %*% t(Xu2) # Projektion
> B2 <- t(Y) %*% (Pu2 - Pr2) %*% Y # Matrix B2

# Test Prädiktor 3: eingeschränktes Modell (= umfass. Modell Präd. 2)
> Xr3 <- Xu2 # Designmatrix
> Pr3 <- Pu2 # Projektion

# Test Prädiktor 3: umfassenderes Modell (hier = vollst. Modell)
> Xu3 <- X # Designmatrix
> Pu3 <- Pf # Projektion
> B3 <- t(Y) %*% (Pu3 - Pr3) %*% Y # Matrix B3
```

Mit Hilfe der Matrizen \mathbf{B}_j und \mathbf{W} können die Teststatistiken für den Test jedes Prädiktors berechnet werden, was hier nur für den ersten Prädiktor gezeigt werden soll. Die Ergebnisse stimmen mit der Ausgabe von `summary(manova(...))` in Abschn. 12.5 überein.

```
> (WL1 <- det(WW) / det(B1 + WW)) # Wilks' Lambda
[1] 0.1104243

# Roys Maximalwurzel
```

```
> (RLR11 <- max(eigen(solve(WW) %*% B1)$values))          # lambda
[1] 8.055978

> (RLRt1 <- max(eigen(solve(B1 + WW) %*% B1)$values))    # theta
[1] 0.8895757

# Pillai-Bartlett-Spur: hier gleich Roys Maximalwurzel theta
> (PBT1 <- sum(diag(solve(B1 + WW) %*% B1)))
[1] 0.8895757

# Hotelling-Lawley-Spur: hier gleich Roys Maximalwurzel lambda
> (HLT1 <- sum(diag(solve(WW) %*% B1)))
[1] 8.055978
```

12.9.9 Beispiel: Einfaktorielle MANOVA

Das Ergebnis der einfaktoriellen MANOVA in Abschn. 12.7.1 lässt sich nun mit dem in Abschn. 12.9.6 und 12.9.7 dargestellten Vorgehen manuell prüfen. Im Beispiel sollen Daten von zwei Zielgrößen (Datenmatrix Y_{M1}) in drei Bedingungen (Faktor IV_{man}) vorliegen. Zunächst sind die Designmatrizen und Projektionen für das eingeschränkte H_0 -Modell sowie für das umfassendere H_1 -Modell zu erstellen.

```
# vollständiges Modell: ursprüngliche Inzidenzmatrix X*p
> XstarP <- cbind(as.numeric(IVman == 1),           # 1. Indikatorvariable
+                   as.numeric(IVman == 2),           # 2. Indikatorvariable
+                   as.numeric(IVman == 3))         # 3. Indikatorvariable

# vollständiges Modell für Treatment-Kontraste
> Ct   <- contr.treatment(ncol(XstarP))            # Codiermatrix C
> Xpm1 <- XstarP %*% Ct                            #  $X^{p-1} = X^p * C$ 
> X    <- cbind(1, Xpm1)                            # reduzierte Designmatrix [1| $X^{p-1}$ ]
> XR   <- model.matrix(~ IVman)                      # Designmatrix aus R
> all.equal(X, XR, check.attributes=FALSE)          # Vergleich
[1] TRUE

# orthogonale Projektion auf durch Designmatrix aufgespannten Raum
> Pf <- X %*% solve(t(X) %*% X) %*% t(X)
> Pu <- Pf                                         # hier:  $H_1$ -Modell = vollst. Modell,  $P_u = P_f$ 

# Gesamt-Nullmodell: Designmatrix = 1-Vektor
> X0 <- X[, 1, drop=FALSE]

# orthogonale Projektion auf durch Gesamt-Nullmodell aufgespannten Raum
> P0 <- X0 %*% solve(t(X0) %*% X0) %*% t(X0)
> Pr <- P0                                         # hier:  $H_0$ -Modell = Gesamt-0-Modell,  $P_r = P_0$ 
```

Die Kontrastschätzungen $\hat{\mathbf{B}} = \mathbf{X}^+ \mathbf{Y}$ bestehen bei den hier verwendeten Treatment-Kontrasten für jede Zielgröße l aus dem Mittelwert der ersten Gruppe ($\hat{\beta}_{0,l} = M_{1,l}$) sowie aus den Abweichungen der verbleibenden Gruppenmittel zu $M_{1,l}$ ($\hat{\beta}_{j,l} = M_{j,l} - M_{1,l}$ mit $j = 2, \dots, p$).

```
# Parameter- / Kontrastschätzungen für Treatment-Kontraste
> (Bt <- coef(lm(Ym1 ~ IVman, contrasts=list(IVman=contr.treatment))))
      [,1]   [,2]
(Intercept) -3.266667  5.60
IVman2       5.626667 -1.56
IVman3       2.916667 -6.00

# Kontrolle: teile Datenmatrix nach Gruppen auf und berechne Zentroide
> (Mj <- aggregate(cbind(X1, X2) ~ IVman,
+                     data=data.frame(Ym1), FUN=mean))
    IVman     X1     X2
1     1 -3.266667  5.60
2     2  2.360000  4.04
3     3 -0.350000 -0.40

> Mj[2, c("X1", "X2")] - Mj[1, c("X1", "X2")] # Abweichung M2-M1=beta2
      X1     X2
5.626667 -1.56

> Mj[3, c("X1", "X2")] - Mj[1, c("X1", "X2")] # Abweichung M3-M1=beta3
      X1     X2
2.916667 -6.00
```

Bei Treatment-Kontrasten stimmen die Schätzungen der ursprünglichen Parameter in $\hat{\mathbf{B}}_p^* = \mathbf{L}\hat{\mathbf{B}}_{p-1}$ für jede Zeile $j = 2, \dots, p$ mit jenen in $\hat{\mathbf{B}}$ überein, während durch die Nebenbedingung $\mathbf{v} = (1, 0, \dots, 0)^\top$ für die Parameter in der ersten Zeile $\hat{\beta}_{1,l}^* = 0$ gilt.

```
> (BstarPt <- Ct %*% Bt[-1, ])          # B*p
      [,1]   [,2]
1 0.000000  0.00
2 5.626667 -1.56
3 2.916667 -6.00
```

Bei der Effektcodierung bestehen die Kontrastschätzungen in $\hat{\mathbf{B}}$ für jede Zielgröße l aus dem ungewichteten Gesamtmittelwert ($\hat{\beta}_{0,l} = M_l = \frac{1}{p} \sum_j M_{j,l}$) sowie aus den Abweichungen der ersten $p - 1$ Gruppenmittel zu M_l ($\hat{\beta}_{j,l} = M_{j,l} - M_l$ mit $j = 1, \dots, p - 1$).

```
# Parameter- / Kontrastschätzungen für Effektcodierung
> (Be <- coef(lm(Ym1 ~ IVman, contrasts=list(IVman=contr.sum))))
    IVman     X1     X2
1     1 -3.266667  5.60
2     2  2.360000  4.04
3     3 -0.350000 -0.40
```

```
> Mj_num <- Mj[, c("X1", "X2")]
> colMeans(Mj_num) # ungewichtetes Gesamtmittel = beta0
      X1      X2
-0.4188889 3.0800000

# Mj.l - Ml = beta1, beta2
> scale(Mj_num, colMeans(Mj_num), scale=FALSE)
      X1      X2
1 -2.84777778 2.52
2  2.77888889 0.96
3  0.06888889 -3.48
```

Die Schätzungen der ursprünglichen Parameter in $\hat{\boldsymbol{B}}_p^*$ stimmen bei der Effektcodierung für jede Zeile $j = 1, \dots, p-1$ mit jenen in $\hat{\boldsymbol{B}}$ überein. Die Nebenbedingung $\boldsymbol{v} = \mathbf{1}$ legt für jede Zielgröße l die $\hat{\beta}_{p,l}^*$ dadurch fest, dass die Beziehung $\sum_{j=1}^p \hat{\beta}_{j,l}^* = 0$ gelten muss, was zu $\hat{\beta}_{p,l}^* = -\sum_{j=1}^{p-1} \hat{\beta}_{j,l}^*$ führt.

```
> Ce <- contr.sum(ncol(XstarP))           # Codiermatrix Effektcodierung
> (BstarPe <- Ce %*% betaE[-1, ])       # B*p
      [,1]      [,2]
1 -2.84777778 2.52
2  2.77888889 0.96
3  0.06888889 -3.48

> colSums(BstarPe)                      # Summe über beta*p = 0
[1] 0.000000e+00 1.110223e-16
```

Bei der Parametrisierung ohne β_0 in der Rolle des theoretischen Zentroids $\boldsymbol{\mu}$ (cell means Modell, Abschn. 12.9.2, Fußnote 50) erhalten die ursprünglichen Parameter in \boldsymbol{B}_p^* die Bedeutung der Gruppenzentroide $\boldsymbol{\mu}_j$ und werden entsprechend über die Gruppenmittelwerte geschätzt. Für jede Zielgröße l gilt also $\hat{\beta}_{j,l}^* = M_{j,l}$, zudem stimmt \boldsymbol{B}_p^* mit \boldsymbol{B}_p überein.

```
# Parameterschätzungen B = B* für cell means Modell
> (Bcm <- coef(lm(Ym1 ~ IVman - 1)))
      [,1]      [,2]
IVman1 -3.266667 5.60
IVman2  2.360000 4.04
IVman3 -0.350000 -0.40
```

Die Vorhersage $\hat{\boldsymbol{Y}}_f = \boldsymbol{P}_f \boldsymbol{Y}$ des vollständigen Modells liefert für eine Person in der Gruppe j für jede Zielgröße l den zugehörigen Gruppenmittelwert $M_{j,l}$.

```
> Yhat <- Pf %*% Ym1                  # Vorhersage vollständiges Modell
> unique(Yhat)                         # vorkommende Werte
      [,1]      [,2]
[1,] -3.266667 5.60
[2,]  2.360000 4.04
[3,] -0.350000 -0.40
```

Kapitel 12 Multivariate Verfahren

Die für den inferenzstatistischen Test notwendigen Matrizen \mathbf{T} , \mathbf{B} und \mathbf{W} ergeben sich aus den ermittelten Projektionen $\mathbf{P}_r = \mathbf{P}_0$ und $\mathbf{P}_u = \mathbf{P}_f$ jeweils auf den Unterraum, der durch die Designmatrix des eingeschränkten und des umfassenderen Modells aufgespannt wird.

```

> Id <- diag(sum(Nj))                      # n x n Einheitsmatrix I
> BB <- t(Ym1) %*% (Pu-Pr) %*% Ym1      # B
> WW <- t(Ym1) %*% (Id-Pf) %*% Ym1      # W
> TT <- t(Ym1) %*% (Id-P0) %*% Ym1      # T

# B: SSP-Matrix der durch Vorhersagedifferenzen ersetzten Daten
> all.equal(BB, (sum(Nj)-1) * cov((Pu-Pr) %*% Ym1))
[1] TRUE

# W: SSP-Matrix der durch Differenz zum jeweiligen
# Gruppenzentroid ersetzten Daten
> all.equal(WW, (sum(Nj)-1) * cov((Id-Pf) %*% Ym1))
[1] TRUE

# T: SSP-Matrix der Residuen des Nullmodells sowie der Daten
> all.equal(TT, (sum(Nj)-1) * cov((Id-P0) %*% Ym1))
[1] TRUE

> all.equal(TT, BB + WW)                      # SSP-Matrix Daten
[1] TRUE

> all.equal(TT, BB + WW)                      # Kontrolle: T = B + W
[1] TRUE

```

Mit Hilfe der Matrizen \mathbf{B} und \mathbf{W} können die Teststatistiken berechnet werden, die mit der Ausgabe von `summary(manova(...))` in Abschn. 12.7.1 übereinstimmen.

```

> (WL <- det(WW) / det(BB + WW))           # Wilks' Lambda
[1] 0.4201068

# Roys Maximalwurzel
> (RLR1 <- max(eigen(solve(WW) %*% BB)$values))          # lambda
[1] 0.6956387

> (RLRt <- max(eigen(solve(BB + WW) %*% BB)$values))       # theta
[1] 0.4102517

> (PBT <- sum(diag(solve(BB + WW) %*% BB)))    # Pillai-Bartlett-Spur
[1] 0.6979024

> (HLT <- sum(diag(solve(WW) %*% BB)))           # Hotelling-Lawley-Spur
[1] 1.099444

```

12.9.10 Beispiel: Zweifaktorielle MANOVA

Das Ergebnis der zweifaktoriellen MANOVA in Abschn. 12.7.2 lässt sich nun ebenfalls mit dem in Abschn. 12.9.6 und 12.9.7 dargestellten Vorgehen manuell prüfen. Es sollen Daten auf zwei Zielgrößen (Datenmatrix $\mathbf{Ym2}$) in 3×2 Bedingungen (Faktoren IV1 und IV2) vorliegen. Zunächst sind die ursprünglichen Inzidenzmatrizen $\mathbf{X}_1^*, \mathbf{X}_2^*, \mathbf{X}_{1 \times 2}^*$ zu erstellen. Ihr jeweiliges Produkt mit der zugehörigen Codiermatrix \mathbf{C} geht in die Designmatrizen der eingeschränkten und umfassenderen Modelle ein, die den Tests der drei Hypothesen (zwei Haupteffekte und Interaktionseffekt) zugrunde liegen.

```
# ursprüngliche Inzidenzmatrizen
> Xstar1 <- cbind(as.numeric(IV1 == 1),           # Faktor 1: X*1
+                   as.numeric(IV1 == 2),
+                   as.numeric(IV1 == 3))

> Xstar2 <- cbind(as.numeric(IV2 == 1),           # Faktor 2: X*2
+                   as.numeric(IV2 == 2))

# Interaktion: alle paarweisen Produkte Spalten von X*1 und X*2: X*1x2
> Xstar12 <- cbind(as.numeric(IV1 == 1) * as.numeric(IV2 == 1),
+                   as.numeric(IV1 == 2) * as.numeric(IV2 == 1),
+                   as.numeric(IV1 == 3) * as.numeric(IV2 == 1),
+                   as.numeric(IV1 == 1) * as.numeric(IV2 == 2),
+                   as.numeric(IV1 == 2) * as.numeric(IV2 == 2),
+                   as.numeric(IV1 == 3) * as.numeric(IV2 == 2))

# Codiermatrizen: Treatment-Kontraste
> C1  <- contr.treatment(ncol(Xstar1))          # Faktor 1
> C2  <- contr.treatment(ncol(Xstar2))          # Faktor 2
> C12 <- kronecker(C2, C1)                      # Interaktion

# reduzierte Inzidenzmatrizen
> X1  <- Xstar1 %*% C1                          # Faktor 1: X1 = X*1*C1
> X2  <- Xstar2 %*% C2                          # Faktor 2: X2 = X*2*C2
> X12 <- Xstar12 %*% C12                         # Interaktion: X*1x2*C1x2

# vollständiges Modell: Designmatrix für identifizierbare Parameter
> X  <- cbind(1, X1, X2, X12)                  # X = [1|X1|X2|X1x2]
> XR <- model.matrix(~ IV1*IV2)                # Designmatrix aus R
> all.equal(X, XR, check.attributes=FALSE)      # Vergleich
[1] TRUE
```

Im Gesamt-Nullmodell der zweifaktoriellen Varianzanalyse sind alle Parameter bis auf β_0 gleich 0, als Designmatrix \mathbf{X}_0 bleibt also der Vektor 1. Es folgt die Berechnung der zugehörigen Projektion \mathbf{P}_0 sowie der zum vollständigen Modell mit Designmatrix \mathbf{X}_f gehörenden Projektion \mathbf{P}_f . Aus der Projektion auf das orthogonale Komplement des jeweils von den Spalten von \mathbf{X}_0 und \mathbf{X}_f aufgespannten Unterraumes ergeben sich die Matrizen \mathbf{T} und \mathbf{W} .

Kapitel 12 Multivariate Verfahren

```
# Gesamt-Nullmodell
> X0 <- X[, 1, drop=FALSE] # Designmatrix X0 = 1-Vektor

# Nullmodell: orthogonale Projektion auf durch X0 aufgespannten Raum
> P0 <- X0 %*% solve(t(X0) %*% X0) %*% t(X0)

# vollst. Modell: orthog. Proj. auf von Designmatrix aufgespannten Raum
> Pf <- X %*% solve(t(X) %*% X) %*% t(X)

# Matrizen T und W für Teststatistiken
> Id <- diag(nrow(Ym2)) # n x n Einheitsmatrix I
> TT <- t(Ym2) %*% (Id - P0) %*% Ym2 # T
> WW <- t(Ym2) %*% (Id - Pf) %*% Ym2 # W
```

Die Vorhersage $\hat{\mathbf{Y}}_f = \mathbf{P}_f \mathbf{Y}$ des vollständigen Modells liefert für eine Person in der Bedingungskombination jk beider Faktoren für jede Zielgröße l den Gruppenmittelwert $M_{jk.l}$.

```
> Yhat <- Pf %*% Ym2 # Vorhersage vollständiges Modell
> unique(Yhat) # vorkommende Werte
      [,1]      [,2]
[1,] -3.26666667  5.600000
[2,]  2.36000000  4.040000
[3,] -0.35000000 -0.400000
[4,]  0.06666667  4.666667
[5,]  3.68000000  7.960000
[6,]  4.05000000 -0.400000

# Kontrolle: teile Datenmatrix nach Gruppen auf und berechne Zentroide
> aggregate(cbind(X1, X2) ~ IV1 + IV2, data=data.frame(Ym2), FUN=mean)
    IV1 IV2      X1      X2
1     1   1 -3.26666667  5.600000
2     2   1  2.36000000  4.040000
3     3   1 -0.35000000 -0.400000
4     1   2  0.06666667  4.666667
5     2   2  3.68000000  7.960000
6     3   2  4.05000000 -0.400000
```

Schließlich muss für jede der drei Hypothesen das zugehörige Paar aus eingeschränktem H_0 -Modell und umfassenderem H_1 -Modell mit zugehörigen Designmatrizen \mathbf{X}_r und \mathbf{X}_u sowie ihren orthogonalen Projektionen \mathbf{P}_r und \mathbf{P}_u berechnet werden. Dies soll hier für Quadratsummen vom Typ I geschehen. Aus der Differenz beider Projektionen ergibt sich für jeden Effekt die Matrix \mathbf{B}_j , die als Verallgemeinerung der univariaten Effekt-Quadratsumme in die zugehörigen Teststatistiken eingeht.

```
# Test UV 1: eingeschränktes Modell (= Gesamt-Nullmodell)
> Xr1 <- X0 # Designmatrix
> Pr1 <- P0 # Projektion
```

Kapitel 12 Multivariate Verfahren

```

# Test UV 1: umfassenderes Modell
> Xu1 <- X[, c(1, 2, 3)]                                # Designmatrix
> Pu1 <- Xu1 %*% solve(t(Xu1) %*% Xu1) %*% t(Xu1)      # Projektion
> B1   <- t(Ym2) %*% (Pu1-Pr1) %*% Ym2                # Matrix B1

# Test UV 2: eingeschränktes Modell (= umfassenderes Modell UV 1)
> Xr2 <- Xu1                                         # Designmatrix
> Pr2 <- Pu1                                         # Projektion

# Test UV 2: umfassenderes Modell
> Xu2 <- X[, c(1, 2, 3, 4)]                            # Designmatrix
> Pu2 <- Xu2 %*% solve(t(Xu2) %*% Xu2) %*% t(Xu2)      # Projektion
> B2   <- t(Ym2) %*% (Pu2-Pr2) %*% Ym2                # Matrix B2

# Test Interaktion: eingeschränktes Modell (= umfass. Modell UV 2)
> Xr12 <- Xu2                                         # Designmatrix
> Pr12 <- Pu2                                         # Projektion

# Test Interaktion: umfassenderes Modell (hier = vollst. Modell)
> Xu12 <- X                                         # Designmatrix
> Pu12 <- Pf                                         # Projektion
> B12  <- t(Ym2) %*% (Pu12 - Pr12) %*% Ym2          # Matrix B12

# Kontrolle: Verallgemeinerung der Quadratsummenzerlegung
# gilt allgemein nur für QS vom Typ I, sonst nur bei gleichen Njk
> all.equal(TT, B1 + B2 + B12 + WW)
[1] TRUE

```

Mit Hilfe der Matrizen B_1 , B_2 , $B_{1 \times 2}$ und W können die Teststatistiken für den Test jedes Effekts berechnet werden, was hier nur für den ersten Haupteffekt gezeigt werden soll. Die Ausgabe stimmt mit jener von `summary(manova(...))` in Abschn. 12.7.2 überein.

```

> (WL1 <- det(WW) / det(B1 + WW))                      # Wilks' Lambda
[1] 0.3846785

# Roys Maximalwurzel
> (RLR11 <- max(eigen(solve(WW) %*% B1)$values))       # lambda
[1] 1.042646

> (RLRt1 <- max(eigen(solve(B1 + WW) %*% B1)$values)) # theta
[1] 0.5104389

> (PBT1 <- sum(diag(solve(B1 + WW) %*% B1)))        # Pillai-Bartlett-Spur
[1] 0.7246769

> (HLT1 <- sum(diag(solve(WW) %*% B1)))              # Hotelling-Lawley-Spur
[1] 1.315296

```

Kapitel 13

Vorhersagegüte prädiktiver Modelle

Da empirische Daten fehlerbehaftet sind, bezieht die Anpassung eines statistischen Modells immer auch die Messfehler mit ein, die Parameterschätzungen orientieren sich daher zu stark an den zufälligen Besonderheiten der konkreten Stichprobe (*overfitting*). Das folgende Kapitel stellt Methoden vor, um die Vorhersagegüte von Modellen für neue, unabhängige Daten zu charakterisieren.

Die Güte der Passung eines Modells lässt sich als Funktion $f(\cdot)$ der Abweichungen $E = Y - \hat{Y}$ der Modellvorhersage \hat{Y} zu den tatsächlichen Werten der vorhergesagten Variable Y quantifizieren. Genauer soll $\hat{Y}'_{X,Y}(X')$ die folgende Vorhersage bezeichnen: Zunächst wird ein Modell an einer Stichprobe mit Werten für Prädiktoren X und Zielvariable Y (Kriterium) angepasst. In die Vorhersagegleichung mit den Parameterschätzungen dieses Modells werden dann (potentiell andere) Prädiktorwerte X' eingesetzt, um die Vorhersage \hat{Y}' zu berechnen, die mit den tatsächlichen Beobachtungen Y' zu vergleichen sind. $f(E)$ ist die *Verlustfunktion*, die alle individuellen absoluten Abweichungen e_i auf einen Gesamtwert für die Vorhersagegenauigkeit abbildet.

Angewendet auf die zur Modellanpassung verwendete *Trainingsstichprobe* selbst ($X' = X$) soll $f(E)$ hier als *Trainingsfehler* (auch *Resubstitutionsfehler*) bezeichnet werden, angewendet auf andere *Teststichproben* aus derselben Grundgesamtheit ($X' \neq X$) als *Vorhersagefehler*. Die Passung des Modells für die Trainingsstichprobe ist dabei im Mittel besser als für andere Teststichproben aus derselben Grundgesamtheit. Der Trainingsfehler ist dahingehend verzerrt, dass er den Vorhersagefehler systematisch unterschätzt, also zu optimistisch bzgl. der Generalisierbarkeit des angepassten Modells ist. Die Größe des Vorhersagefehlers liefert auch einen Anhaltspunkt für die Auswahl eines bestimmten Modells aus mehreren möglichen (Abschn. 6.3.3).

Die folgenden Abschnitte stellen Kreuzvalidierung und Bootstrapping für eine möglichst unverzerrte Schätzung des Vorhersagefehlers vor. Beide Methoden sind insofern Resampling-Verfahren (Kap. 11), als sie aus Daten einer gegebenen Basisstichprobe mehrfach neue Stichproben erstellen und mit ihnen den Vorhersagefehler schätzen.

[Hastie, Tibshirani und Friedman \(2009\)](#) und [James et al. \(2021\)](#) erläutern im Detail den theoretischen Hintergrund. Für viele der vorgestellten Verfahren und weiterführende Methoden bietet das Paket `cv` ([Fox & Monette, 2024](#)) spezialisierte Funktionen, die effizienter und damit schneller arbeiten als Funktionen des Basisumfangs von R.

13.1 Kreuzvalidierung linearer Regressionsmodelle

Für die Kreuzvalidierung ist eine vorliegende Gesamtstichprobe vom Umfang n in zwei komplementäre Teilmengen zu partitionieren: Die Trainingsstichprobe liefert die Datenbasis für die Parameterschätzung. Diese Modellanpassung wird dann auf die verbleibende Teststichprobe angewendet, indem die Werte der Kovariaten X' aus der Teststichprobe in die ermittelte Vorhersagegleichung eingesetzt werden. Als Verlustfunktion $f(E)$ dient etwa die mittlere Fehlerquadratsumme, im Falle einer linearen Regression der Standardschätzfehler.

13.1.1 k -fache Kreuzvalidierung

Für eine bessere Beurteilung der Vorhersagegenauigkeit sollte die dargestellte Prüfung mehrfach mit wechselnden Trainings- und Teststichproben durchgeführt werden. Häufig wird hierzu die Ausgangsstichprobe in k disjunkte Teilmengen partitioniert. Jede von ihnen dient daraufhin reihum als Teststichprobe, während die jeweils verbleibenden $k - 1$ Stichproben gemeinsam die Trainingsstichprobe bilden. Der Mittelwert der Verlustfunktion über die k Wiederholungen dient dazu, die Generalisierbarkeit der Parameterschätzung insgesamt zu beurteilen.¹ Häufig empfohlene Werte für k sind 5 oder 10. Für höhere Werte von k ist der steigende numerische Aufwand zur Kreuzvalidierung sehr umfassender Modelle zu berücksichtigen.²

Das Paket `boot` (Abschn. 11.1) stellt mit `cv.glm()` eine Funktion für die k -fache Kreuzvalidierung verallgemeinerter linearer Modelle (Kap. 8) bereit.

```
cv.glm(data=<Datensatz>, glmfit=<glm-Modell>, K=<Anzahl>,
       cost=<Verlustfunktion>)
```

Für `data` ist der Datensatz zu übergeben, aus dem die Variablen eines verallgemeinerten linearen Modells stammen, das unter `glmfit` zu nennen ist. Da die lineare Regression ein Spezialfall eines solchen Modells ist, kann `cv.glm()` auch für sie zum Einsatz kommen. Die Regression ist hierfür lediglich mit `glm()` zu formulieren (Abschn. 8.1.6). Mit `K` lässt sich die gewünschte Anzahl zufälliger Partitionen für die Aufteilung in Trainings- und Teststichproben wählen. In der Voreinstellung ist die Verlustfunktion die mittlere Fehlerquadratsumme, kann aber über `cost` auf eine selbst definierte Funktion geändert werden (Abschn. 13.2).

Das Ergebnis von `cv.glm()` ist eine Liste, die in der Komponente `delta` den Kreuzvalidierungsfehler (*cross validation error*, CVE) beinhaltet, in der Voreinstellung also den Mittelwert der k mittleren Fehlerquadratsummen in der Teststichprobe. Das zweite Element von `delta` berücksichtigt dabei die Korrektur einer aus der Wahl von k herrührenden Verzerrung.

¹Mit Stichproben der Größe n_k können die Einzelwerte der Verlustfunktion auch gewichtet mit dem Faktor $\frac{n_k}{n}$ in den Mittelwert eingehen. Die k -fache Kreuzvalidierung eines linearen Regressionsmodells ist für $k = n \left(1 - \frac{1}{\ln(n)-1}\right)$ asymptotisch äquivalent zum Informationskriterium BIC (Abschn. 6.3.3).

²Die *stratifizierte* k -fache Kreuzvalidierung sorgt bei der Einteilung in Teilmengen dafür, dass die Verteilung von Y in den Partitionen ähnlich ist. Für kontinuierliche Y führt dies zu einem annähernd konstanten Mittelwert von Y in den Partitionen. Für kategoriale Y bleibt so der Anteil der Kategorien weitgehend gleich. Für eine Umsetzung vgl. die Pakete `rsample` oder `ipred` (Peters & Hothorn, 2019).

```
# passe lineare Regression mit glm() an
> regrDf <- data.frame(weight, height, age, sport)      # Datensatz
> glmFit <- glm(weight ~ height + age + sport, data=regrDf,
+                  family=gaussian(link="identity"))

> library(boot)                                         # für cv.glm()
> k     <- 3                                           # Anzahl Partitionen
> kfCV <- cv.glm(regrDf, glmfit=glmFit, K=k) # k-fache Kreuzvalid.
> kfCV$delta           # Kreuzvalidierungsfehler und Korrektur
10.52608 10.38042
```

Für die manuelle Berechnung ist zunächst eine eigene Funktion zu erstellen, die auf Basis des numerischen Indexvektors einer Teststichprobe eine einzelne Kreuzvalidierung des gegebenen Regressionsmodells durchführt (Abschn. 17.3).

```
# einzelne Kreuzvalidierung gegeben numerische Indizes Teststichprobe
> doCV <- function(idxTst) {
+   # passe Regression in Trainingsstichprobe an
+   fitTrn <- lm(weight ~ height + age+sport, data=regrDf,
+                 subset=-idxTst)
+
+   # berechne für Teststichprobe Vorhersage aus angepasstem Modell
+   predTst <- predict(fitTrn, newdata=regrDf[idxTst, ])
+
+   # mittlere Fehlerquadratsumme für Teststichprobe
+   mean((predTst - regrDf$weight[idxTst])^2)
+ }
```

Daraufhin muss die gesamte Stichprobe vom Umfang n in k zufällige Gruppen partitioniert werden, die dann der Reihe nach als Teststichprobe dienen.

```
# Gruppen zufällig zuteilen
> tstGrp <- sample(seq_len(N), N, replace=FALSE) %% k
> foldsL <- split(seq_along(tstGrp), tstGrp) # Liste Teststichproben
> (cveL <- lapply(foldsL, doCV)) # Kreuzvalidierungsfehler Teststpr.
$`1`
[1] 11.96789

$`2`
[1] 10.45271

$`3`
[1] 11.85106

> mean(unlist(cveL))          # mittlerer Kreuzvalidierungsfehler
[1] 11.42389
```

Ein potentielles Problem der k -fachen Kreuzvalidierung ist die hohe Varianz ihrer Ergebnisse. Eine Strategie diese zu verringern, ist die *wiederholte* k -fache Kreuzvalidierung. Sie besteht darin, die Kreuzvalidierung mehrmals zu wiederholen und die jeweils ermittelten Schätzungen der Vorhersagegüte zu mitteln. Dies lässt sich manuell, etwa mit `replicate()` (Abschn. 17.2) erzielen oder über das Paket `rsample`.

```
> library(boot)                      # für cv.glm()
> n_repl    <- 5                     # Anzahl Wiederholungen
> rep_kfCV <- replicate(n_repl,
+                         cv.glm(data=regrDf, glmfit=glmFit, K=k),
+                         simplify=FALSE)

# extrahiere korrigierte Schätzungen der Vorhersagegüte
> (deltas <- vapply(rep_kfCV, function(x) { x$delta[2] }, numeric(1)))
[1] 11.055761 10.875137 10.935775 10.320764  9.891715

> mean(deltas)
[1] 10.61583
```

13.1.2 Leave-One-Out Kreuzvalidierung

Ein Spezialfall der k -fachen Kreuzvalidierung ist die *Leave-One-Out*-Kreuzvalidierung (LOOCV) für $k = n$.³ Hier dient nacheinander jede Einzelbeobachtung separat als Teststichprobe für ein Modell, das an der Trainingsstichprobe aus jeweils allen übrigen Beobachtungen angepasst wurde. Im Fall der linearen Regression lässt sich der Kreuzvalidierungsfehler hier numerisch effizient als Mittelwert der quadrierten PRESS-Residuen $\frac{1}{n} \sum_i \left(\frac{e_i}{1-h_i} \right)^2$ (*predicted residual error sum of squares*) direkt berechnen, die `rstandard(lm-Modell, type="predictive")` ausgibt (Abschn. 6.5.2). Dabei ist e_i das i -te Residuum $y_i - \hat{y}_i$ und h_i der Hebelwert der Beobachtung i (Abschn. 6.5.1).

```
# LOOCV-Kreuzvalidierungsfehler inkl. korrigierter Variante
> LOOCV <- cv.glm(data=regrDf, glmfit=glmFit, K=N)
> LOOCV$delta
      1
10.59404 10.58995

# Kreuzvalidierungsfehler hier = Mittelwert quadrierte PRESS-Residuen
> lmFit <- lm(weight ~ height + age + sport, data=regrDf)
> PRESS <- rstandard(lmFit, type="predictive")^2
> mean(PRESS)
[1] 10.59404

> mean((residuals(lmFit) / (1 - hatvalues(lmFit)))^2)   # manuell ...
```

³Sie ist asymptotisch äquivalent zum Informationskriterium AIC des Regressionsmodells (Abschn. 6.3.3).

Bei der Kontrolle wird die zuvor selbst erstellte Funktion `doCV()` mittels `sapply()` für jeden Index der Stichprobe aufgerufen.

```
> idx <- seq_len(N)                      # Indizes aller Beobachtungen
> res <- sapply(idx, function(x) { doCV(idx == x) } )    # LOOCV
> mean(res)                                # Kreuzvalidierungsfehler
[1] 10.59404
```

Für penalisierte Regressionsmodelle und verallgemeinerte additive Modelle (Abschn. 6.6) ist die Berechnung der Hat-Matrix \mathbf{H} – und damit der Diagonaleinträge h_i – aufwendig. In diesen Fällen lässt sich der Kreuzvalidierungsfehler über die verallgemeinerte Kreuzvalidierung (GCV) mit $\frac{1}{n} \sum_i \left(\frac{e_i}{1 - \text{Spur}(H)/n} \right)^2$ approximieren. Hier ist $\text{Spur}(H)$ die Anzahl zu schätzender Parameter.

13.2 Kreuzvalidierung verallgemeinerter linearer Modelle

Bei der Kreuzvalidierung verallgemeinerter linearer Modelle (Kap. 8) ist zu beachten, dass die Wahl der Verlustfunktion als Gütemaß der Vorhersagegenauigkeit für diskrete Variablen Besonderheiten mit sich bringt. Im Fall der logistischen Regression (Abschn. 8.1) wäre es etwa naheliegend, die Vorhersagegüte als Anteil der korrekten Vorhersagen umzusetzen, wenn die Schwelle bei $\hat{p} = 0.5$ gesetzt wird. Tatsächlich ist dieses Maß weniger gut geeignet, da mit ihm nicht das wahre statistische Modell den kleinsten Vorhersagefehler besitzt. Verlustfunktionen, die durch das wahre Modell minimiert werden, heißen *proper score*.

Das Argument `cost` von `cv.glm()` für selbst definierte Verlustfunktionen erwartet eine Funktion, die auf Basis je eines Vektors der beobachteten Werte Y und der vorhergesagten Werte \hat{Y} ein Abweichungsmaß zurückgibt. `cv.glm()` verwendet beim Aufruf von `cost()` für Y den Vektor `glm-Modell$y` und für \hat{Y} `predict(glm-Modell, type="response")`.

Für die logistische Regression ist der Brier-Score B eine geeignete Verlustfunktion. Für jede Beobachtung i berücksichtigt b_i die vorhergesagte Wahrscheinlichkeit $\hat{p}_{ij} = \hat{p}(y_i = j)$ jeder Kategorie j beim Vergleich mit der tatsächlich vorliegenden Kategorie y_i . Hier soll \hat{p}_i kurz für die vorhergesagte Trefferwahrscheinlichkeit $p(y_i = 1)$ stehen, und es gilt $1 - p_i = p(y_i = 0)$.

$$b_i = \begin{cases} (1 - \hat{p}_i)^2 + (0 - (1 - \hat{p}_i))^2 & \text{falls } y_i = 1 \\ (1 - (1 - \hat{p}_i))^2 + (0 - \hat{p}_i)^2 & \text{falls } y_i = 0 \end{cases}$$

Der Brier-Score für eine Stichprobe vom Umfang n ist dann $B = \frac{1}{n} \sum_i b_i$. Mit $k = 2$ Kategorien und dem Kronecker δ als Indikatorfunktion lässt sich b_i auch verkürzt schreiben, wobei $\delta_{ij} = 1$ ist, wenn $y_i = j$ gilt und $\delta_{ij} = 0$ sonst.

$$b_i = \sum_{j=1}^k (\delta_{ij} - \hat{p}_{ij})^2$$

Als Beispiel seien die Daten aus Abschnitt 8.1 zur logistischen Regression betrachtet.

```
> glmLR <- glm(postFac ~ DVpre, family=binomial(link="logit"),
+                 data=dfAncova)

# Verlustfunktion für Brier-Score
> brierA <- function(y, pHat) {
+   mean(((y == 1) * pHat)^2 + ((y == 0) * (1-pHat))^2)
+ }

> library(boot)                                # für cv.glm()
> B1 <- cv.glm(data=regDf, glmfit=glmFit, cost=brierA, K=10)
> B1$delta
[1] 0.5528585 0.5523104
```

Die genannte Definition des Brier-Score lässt sich unmittelbar auf Situationen mit $k > 2$ Kategorien verallgemeinern, etwa auf die multinomiale Regression oder die Diskriminanzanalyse, wo er ebenfalls ein proper score ist. Speziell für die logistische Regression führt auch die folgende vereinfachte Variante zu einem proper score, sie berücksichtigt nur die tatsächlich beobachtete Kategorie. Diese Verlustfunktion $b_i = (y_i - \hat{p}_i)^2$ entspricht der Voreinstellung von `cost` in `cv.glm()`.

```
# Verlustfunktion für Brier-Score - vereinfachte Variante
> brierB <- function(y, pHat) { mean((y-pHat)^2) }
> B2      <- cv.glm(data=regDf, glmfit=glmFit, cost=brierB, K=10)
> B2$delta
[1] 0.1787110 0.1773381
```

Eine Alternative zum Brier-Score basiert auf der logarithmierten geschätzten Wahrscheinlichkeit für die tatsächlich beobachtete Kategorie j von Y . Die Verlustfunktion $-2 \cdot \sum_i \ln \hat{p}_{ij}$ ist äquivalent zu Devianz-Residuen und führt ebenfalls zu einem proper score, der sich auch auf andere verallgemeinerte lineare Modelle anwenden lässt.

13.3 Bootstrap-Vorhersagefehler

Bootstrapping (Abschn. 11.1) liefert weitere Möglichkeiten zur unverzerrten Schätzung des Vorhersagefehlers prädiktiver Modelle, von denen hier die einfache Optimismus-Korrektur vorgestellt wird. Für weitere Ansätze wie den .632 bzw. .632+ bootstrap vgl. [Hastie et al. \(2009\)](#) und für eine Implementierung die Pakete `caret` oder `ipred`.

Für die Optimismus-Berechnung wird die (negative) systematische Verzerrung des Trainingsfehlers über viele Replikationen der Basisstichprobe hinweg geschätzt und danach als Bias-Korrektur vom ursprünglichen Trainingsfehler subtrahiert. Zur Erläuterung der notwendigen Schritte sei folgende Notation vereinbart:

- $\mathbb{E}[Y' - \hat{Y}'_{X,Y}(X')]$ ist der wahre Vorhersagefehler eines an den Beobachtungen X, Y der Basisstichprobe angepassten Modells für eine neue Stichprobe mit Beobachtungen X', Y' von Personen aus derselben Grundgesamtheit.

- $\bar{E}[Y - \hat{Y}_{X,Y}(X)]$ ist der mittlere Trainingsfehler eines an den Beobachtungen X, Y der Basisstichprobe angepassten Modells. $\bar{E}[Y - \hat{Y}_{X,Y}(X)]$ unterschätzt $\mathbb{E}[Y' - \hat{Y}'_{X,Y}(X')]$ systematisch, die Differenz wird als *Optimismus* des Trainingsfehlers bezeichnet.
- $\bar{E}[Y - \hat{Y}^*_{X^*,Y^*}(X)]$ ist der mittlere Vorhersagefehler der Bootstrap-Replikationen, deren jeweilige Vorhersagen für die Basisstichprobe berechnet werden. $\bar{E}[Y - \hat{Y}^*_{X^*,Y^*}(X)]$ ist plug-in-Schätzer für $\mathbb{E}[Y' - \hat{Y}_{X,Y}(X')]$.
- $\bar{E}[Y^* - \hat{Y}^*_{X^*,Y^*}(X^*)]$ ist der mittlere Trainingsfehler der Bootstrap-Replikationen und plug-in-Schätzer für $\mathbb{E}[Y - \hat{Y}_{X,Y}(X)]$.

Da bootstrapping die Beziehung zwischen Basisstichprobe und Population auf die Beziehung zwischen Replikation und Basisstichprobe überträgt, erfolgt die Bootstrap-Schätzung des wahren Trainingsfehler-Optimismus durch die Differenz der jeweiligen plug-in Schätzer. Die bias-korrigierte Schätzung des Vorhersagefehlers $\hat{\mathbb{E}}$ ist dann die Differenz vom Trainingsfehler der Basisstichprobe und dem (negativen) geschätzten Optimismus.

$$\hat{\mathbb{E}}[Y' - \hat{Y}_{X,Y}(X')] = \bar{E}[Y - \hat{Y}_{X,Y}(X)] - (\bar{E}[Y^* - \hat{Y}^*_{X^*,Y^*}(X^*)] - \bar{E}[Y - \hat{Y}^*_{X^*,Y^*}(X)])$$

Der Stichprobenumfang im hier verwendeten Beispiel einer logistischen Regression (Abschn. 8.1) ist gering. In den Replikationen können deshalb auch solche Zusammensetzungen der Beobachtungen auftreten, die zu einer vollständigen Separierbarkeit führen und ungültige Parameterschätzungen liefern (Abschn. 8.1.8). Mit den in Abschn. 17.3.4 und 17.4.2 vorgestellten Methoden kann diese Situation innerhalb der von `boot()` für jede Replikation aufgerufenen Funktion identifiziert werden, um dann einen fehlenden Wert anstatt der ungültigen Optimismus-Schätzung zurückzugeben. Als Verlustfunktion soll der vereinfachte Brier-Score für die logistische Regression (s.o.) dienen.

```
# Funktion, die Optimismus in einer Replikation schätzt
> getBSB <- function(dat, idx) {
+   op <- options(warn=2) # mache Warnungen zu Fehlern
+   on.exit(options(op)) # bei Verlassen: Option zurücksetzen
+
+   # logistische Regression, bei der Fehler abgefangen werden
+   bsFit <- try(glm(postFac ~ DVpre,
+                     family=binomial(link="logit"),
+                     subset=idx, data=dat))
+
+   fail <- inherits(bsFit, "try-error") # Fehler in glm()?
+   if(fail || !bsFit$converged) { # Fehler bzw. keine Konvergenz
+     return(NA)
+   } else { # alles ok
+     # Trainingsfehler Replikation
+     BbsTrn <- brierB(bsFit$y, predict(bsFit, type="response"))
+
+     # Vorhersagefehler Replikation bzgl. Basisstichprobe
+     BbsTst <- brierB(as.numeric(dat$postFac)-1,
+                       predict(bsFit, newdata=dat, type="response")))
+   }
}
```

Kapitel 13 Vorhersagegüte prädiktiver Modelle

```
+  
+      return(BbsTrn - BbsTst)           # Optimismus Replikation  
+ }  
+ }
```

Im Anschluss an das bootstrapping sollte geprüft werden, wie oft Konvergenz- oder Separierbarkeitsprobleme aufgetreten sind.

```
> library(boot)                      # für boot()  
> nR     <- 999                     # Anzahl Replikationen  
> bsRes <- boot(dfAncova, statistic=getBSB, R=nR)  
> sum(is.na(bsRes$t))              # Anzahl problematischer Anpassungen  
[1] 5  
  
# Traingsfehler Basisstichprobe  
> (Btrain <- brierB(glmLR$y, predict(glmLR, type="response")))  
[1] 0.1494605  
  
> (optimism <- mean(bsRes$t, na.rm=TRUE))      # mittlerer Optimismus  
[1] -0.02267715  
  
# Bootstrap-Schätzung Optimismus-korrigierter Vorhersagefehler  
> (predErr <- Btrain - optimism)  
[1] 0.1721376
```

Kapitel 14

Diagramme erstellen

Daten lassen sich in R mit Hilfe einer Vielzahl von Diagrammtypen grafisch darstellen, wobei hier nur auf eine Auswahl der verfügbaren Typen eingegangen werden kann. Für eine umfassende Dokumentation vgl. [Murrell \(2018\)](#) und [Unwin \(2015\)](#).¹ Während sich dieses Kapitel auf den Basisumfang von R konzentriert, geht Kap. 15 auf das beliebte Zusatzpaket `ggplot2` ein.

In R werden zwei Arten von Grafikfunktionen unterschieden: *High-Level*-Funktionen erstellen eigenständig ein komplettes Diagramm inkl. Achsen, während *Low-Level*-Funktionen lediglich ein bestimmtes Element einem bestehenden Diagramm hinzufügen. Einen kurzen Überblick über die Gestaltungsmöglichkeiten vermittelt `demo(graphics)`.

14.1 Grafik-Devices

14.1.1 Aufbau und Verwaltung von Grafik-Devices

Die Ausgabe von Befehlen zum Erstellen einer Grafik kann in verschiedenen Ausgabekanälen, den *devices* erfolgen. Ein device lässt sich wie eine leere Leinwand vorstellen, auf der mit Grafikfunktionen einzelne Inhalte wie mit einem Pinsel eingezeichnet werden. Die Fläche des device ist dabei in drei ineinander verschachtelte Regionen eingeteilt: die gesamte Device-Region mit den äußereren Rändern, die innerhalb dieser Ränder liegende *Figure*-Region und die in diese eingebettete *Plot*-Region, in die die Datenpunkte und andere Grafikelemente eingezeichnet werden (Abb. 14.1).² In der Voreinstellung ist ein device ein separates Grafikfenster, es können aber etwa auch Dateien in verschiedenen Grafikformaten als device dienen.

Sofern noch kein Grafikfenster existiert, öffnet es sich mit Eingabe des ersten High-Level-Grafikbefehls automatisch – in RStudio im *Plots* Tab. In dieses Fenster werden dann alle weiteren Ausgaben grafischer Funktionen hinein gezeichnet, wobei im Fall von High-Level-Funktionen ein ggf. bereits vorhandener Inhalt gelöscht wird. Soll für die Ausgabe einer Grafikfunktion zusätzlich zu bereits bestehenden ein neues, zunächst leeres Fenster geöffnet werden, geschieht dies unter Windows mit

```
windows(width=<Breite>, height=<Höhe>)
```

¹Viele Beispiele zeigt <https://r-graph-gallery.com/>

²Weitere Details erklärt <http://www.pmean.com/0000images/how-big-is-your-graph.pdf>

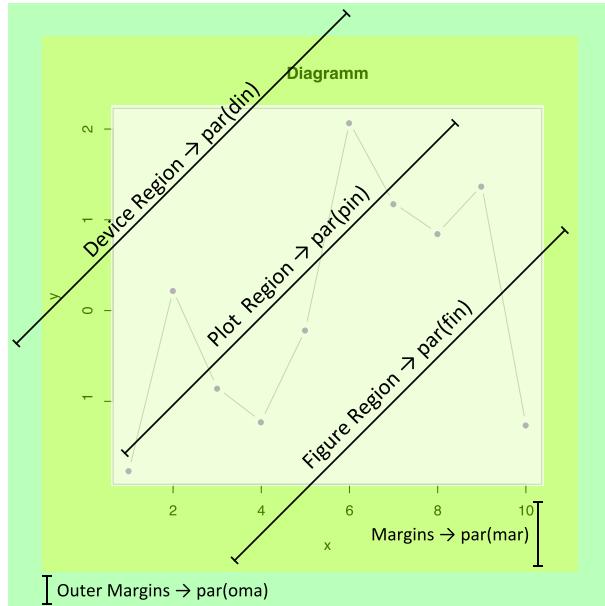


Abbildung 14.1: Regionen und Ränder eines device samt Möglichkeiten, ihre Größe mit `par()` ausgeben und ggf. verändern zu können (Abschn. 14.3.1)

Unter MacOS ist `quartz()` und unter Unix/Linux `x11()` der äquivalente Befehl. Unabhängig vom Betriebssystem erzielt `dev.new()` in der Voreinstellung denselben Effekt. Breite und Höhe des Fensters können über die Argumente `width` und `height` in der Einheit inch bestimmt werden. Bei mehreren geöffneten 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 Grafikfunktion gezeichnet wird, während die Inhalte der anderen devices unverändert bleiben. Um sich einen Überblick über alle aktuell geöffneten devices zu verschaffen, dient der Befehl `dev.list()`.

```
> dev.new(); dev.new(); dev.new()
> dev.list()
windows windows windows
      2      3      4
```

Die Ausgabe zeigt, welche Ausgabekanäle offen sind. Jedes device besitzt dafür eine laufende Nummer, wobei das erste device die 2 erhält. Um zu erfahren, welcher der Ausgabekanäle aktiv ist, dient `dev.cur()` (*device current*).

```
> dev.cur()
windows
      4
```

Die Ausgabe besteht in der fortlaufenden Nummer des aktiven device. `dev.prev()` (*device previous*) und `dev.next()` geben bei mehreren geöffneten devices die Nummer desjenigen device zurück, das sich unmittelbar vor bzw. unmittelbar hinter dem aktiven device befindet. Diese Information kann etwa zum Wechseln des aktiven device mit `dev.set(<Nummer>)` verwendet werden, damit die Ausgabe der folgenden Grafikfunktionen dort erfolgt.

```
> dev.set(3)                                # aktiviere device 3
windows
  3

> dev.set(dev.next())                      # aktiviere das folgende device
windows
  4
```

Das aktive device wird mit `dev.off()` geschlossen. Handelt es sich um ein Grafikfenster, hat dies denselben Effekt, wie das Fenster per Mausklick zu schließen. Die Ausgabe von `dev.off()` gibt an, welches fortan das aktive device ist. Ohne weitere geöffnete devices ist dies das *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
```

14.1.2 Grafiken speichern

Alles, was sich in einem Grafikfenster anzeigen lässt, kann auch als Datei gespeichert werden – in RStudio im *Plots*-Tab mit dem Eintrag *Export*. Ist ein Grafikfenster aktiviert, so ändert sich das Menü der R-Umgebung dahingehend, dass über *Datei: Speichern als:* die Grafik in vielen Formaten gespeichert werden kann. Als Alternative erlaubt ein sich durch Rechtsklick auf das Grafikfenster öffnendes Kontextmenü, die Grafik in wenigen Formaten zu speichern. Dasselbe Kontextmenü enthält auch Einträge, um die Grafik in die Zwischenablage zu kopieren und so direkt anderen Programmen verfügbar zu machen.

Grafiken lassen sich auch befehlsgesteuert ohne den Umweg eines Grafikfensters 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 device) festgelegt werden. Dazu dient etwa `pdf()`, wenn die Grafik im PDF-Format zu speichern ist. Es folgen Befehle zum Erstellen von Diagrammen oder Einfügen von Grafikelementen, deren Ausgabe dann nicht auf dem Bildschirm erscheint, sondern direkt in die Datei umgeleitet wird. Schließlich ist der Befehl `dev.off()` oder `graphics.off()` 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. Beispienhaft seien hier `pdf()` und `jpeg()` betrachtet.³

```
pdf(file="", width=<Breite>, height=<Höhe>)
jpeg(filename="", width=<Breite>, height=<Höhe>,
     units="px", quality=<Bildqualität>)
```

³Alternativ stellt das Paket `Cairo` ([Urbaneck & Horner, 2020](#)) die gleichnamige Funktion zur Verfügung, mit der Diagramme in vielen Dateiformaten auch in hoher Auflösung und mit automatischer Kantenglättung gespeichert werden können.

Unter `file` bzw. `filename` ist der Name der Ausgabedatei einzutragen – ggf. inkl. einer Pfadangabe (Abschn. 3.2). Sollen mehrere Grafiken 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 Grafik kontrolliert. Beide Angaben sind bei `pdf()` 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 – ein kleinerer Wert bedeutet hier eine geringere Bildqualität, die dann stärkere Kompression führt dafür aber auch zu einer geringeren Dateigröße.

Hier soll demonstriert werden, wie eine einfache Grafik im PDF-Format gespeichert wird.

```
> pdf("pdf_test.pdf")          # device öffnen (mit Dateinamen)
> plot(1:10, rnorm(10))       # Grafik einzeichnen
> dev.off()                  # device schließen
```

14.2 Streu- und Liniendiagramme

In zweidimensionalen Streudiagrammen (*scatterplots*) werden mit `plot()` Wertepaare in Form von Punkten in einem kartesischen Koordinatensystem dargestellt, wobei der eine Wert die Position des Punkts entlang der Abszisse (*x*-Achse) und der andere Wert die Position des Punkts entlang der Ordinate (*y*-Achse) bestimmt. Die Punkte können dabei für ein Liniendiagramm durch Linien verbunden oder für ein Streudiagramm als Punktfolke belassen werden.

14.2.1 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 Werte als *y*-Koordinaten interpretiert und die *x*-Koordinaten durch die Indizes des Vektors gebildet.⁴ Das Argument `type` hat mehrere mögliche Ausprägungen, die das Aussehen der Datenmarkierungen im Diagramm bestimmen (Tab. 14.1, Abb. 14.2). 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`.

Die Koordinaten der Punkte können auch über andere Wege angegeben werden: Als Modellformel lautet der Aufruf `plot(<y-Koord.> ~ <x-Koord.>, data=<Datensatz>)`.⁵ Wenn die in

⁴Dagegen erzeugt `plot(<Faktor>, ...)` ein Säulendiagramm der Häufigkeiten jeder Faktorstufe (Abschn. 14.4), da `plot()` eine generische Funktion ist (Abschn. 17.3.7).

⁵Hat die Modellformel die Form `<y-Koord.> ~ <Faktor>`, werden in einem Diagramm getrennt für die von `<Faktor>` definierten Gruppen Boxplots dargestellt (Abschn. 14.6.3).

der Modellformel verwendeten Variablen aus einem Datensatz stammen, ist dieser unter `data` zu nennen. Weiter ist es möglich, eine Liste mit zwei Komponenten `x` und `y` anzugeben, die die Koordinaten enthalten. Schließlich kann einfach eine Matrix mit zwei Spalten für die (x, y) -Koordinaten übergeben werden.

Tabelle 14.1: Mögliche Werte für das Argument `type` von `plot()`

| Wert für <code>type</code> | Bedeutung |
|----------------------------|---|
| "p" | Punkte |
| "l" | durchgehende Linien. Durch eng gesetzte Stützstellen 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 (<i>spike plot</i>) |
| "s" | Stufendiagramm |
| "S" | Stufendiagramm mit anderer Reihenfolge von vertikalen und horizontalen Schritten zur nächsten Stufe |
| "n" | fügt dem Diagramm keine Datenpunkte hinzu (<i>no plotting</i>) |

```
> vec <- rnorm(10)                                # Daten

# 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")
```

14.2.2 Datenpunkte eines Streudiagramms identifizieren

Werden viele Daten in einem Streudiagramm dargestellt, ist häufig nicht ersichtlich, zu welchem Wert ein bestimmter Datenpunkt gehört. Diese Information kann jedoch interessant sein, wenn etwa erst die grafische Betrachtung eines Datensatzes Besonderheiten der Verteilung verrät und die für bestimmte Datenpunkte verantwortlichen Untersuchungseinheiten identifiziert werden sollen. `identify()` erlaubt es, Werte in einem Streudiagramm interaktiv zu identifizieren.

```
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 Grafikfenster 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 über der Diagrammfläche zu einem Kreuz. Mit einem Klick der

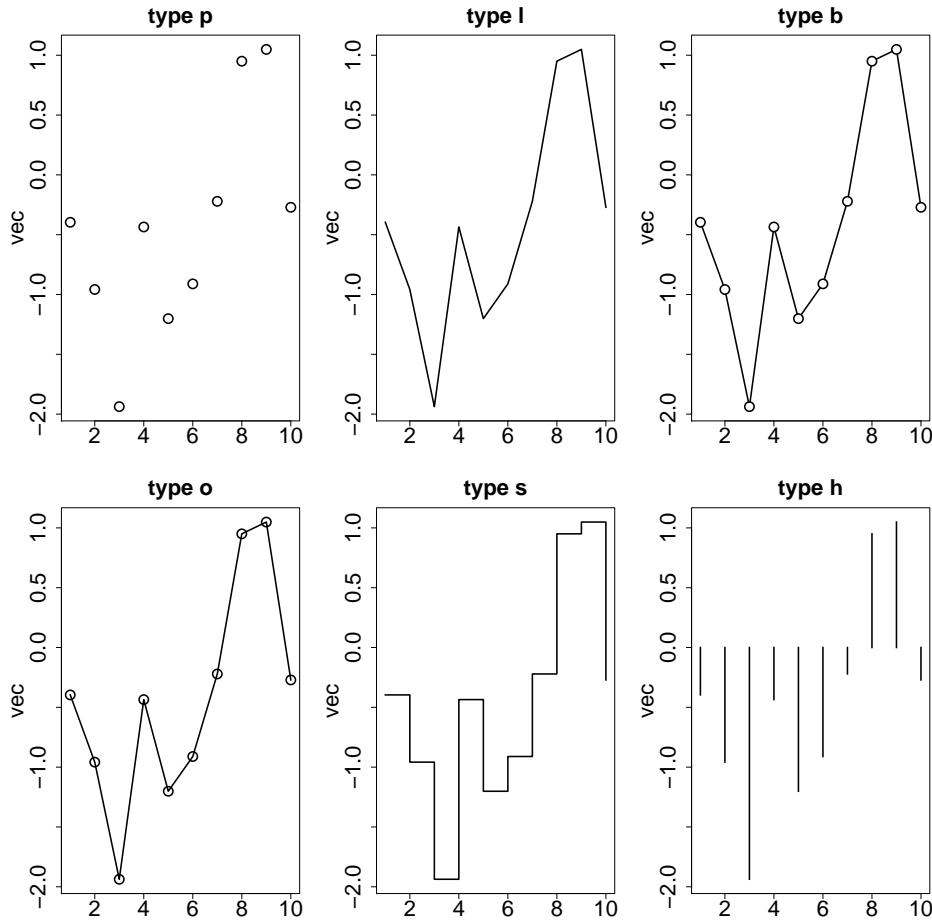


Abbildung 14.2: Mit `plot(type="<Option>")` erzeugbare Diagrammarten

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. In RStudio wird der Vorgang durch Drücken der ESC Taste beendet.

```
> vec <- rnorm(10)
> plot(vec)
> identify(vec)
# Mausklicks ...
[1] 1 4 5 8
```

14.2.3 Streudiagramme mit `matplot()`

So wie `plot()` ein Streudiagramm einer einzelnen Datenreihe erstellt, erzeugt `matplot()` ein Streudiagramm für mehrere Datenreihen gleichzeitig (Abb. 14.3).

```
matplot(x=<Matrix>, y=<Matrix>, type=<'Option>", pch=<'Symbol>")
```

Die Argumente sind dieselben wie für `plot()`, lediglich *x*- und *y*-Koordinaten können nun jeweils als Matrix an *x* bzw. *y* übergeben werden. Dabei wird jede ihrer Spalten als eine separate Datenreihe interpretiert. 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 die Ziffern 1–9, die mit der zur Datenreihe gehörenden Spaltennummer korrespondieren. Mit dem Argument *pch* können auch andere Symbole Verwendung finden (Abschn. 14.3.1, Abb. 14.5).

```
> 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")
```

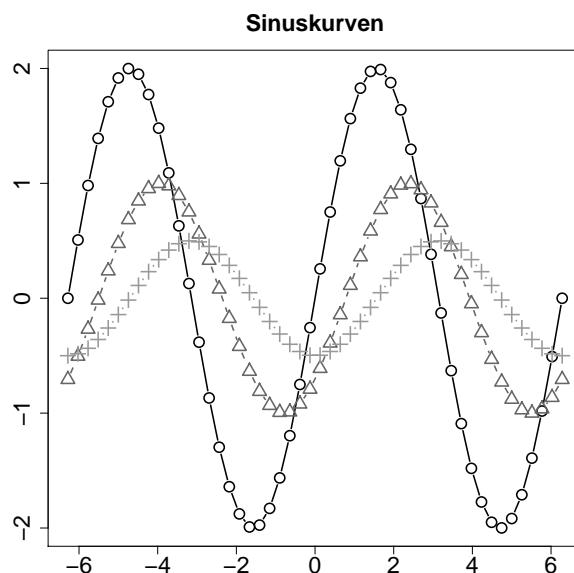


Abbildung 14.3: Mit `matplot()` erzeugtes Streudiagramm

14.3 Diagramme formatieren

`plot()` akzeptiert wie auch andere High-Level-Grafikfunktionen eine Vielzahl weiterer Argumente, mit denen ein Diagramm flexibel angepasst werden kann. Einige der wichtigsten Möglichkeiten zur individuellen Gestaltung werden im Folgenden vorgestellt.

14.3.1 Grafikelemente formatieren

Die Formatierung von Grafikelementen ist in vielen Aspekten variabel, etwa hinsichtlich der Art, Größe und Farbe der verwendeten Symbole oder der Orientierung der Achsenbeschrif-

tungen. Zu diesem Zweck akzeptieren die meisten High-Level-Funktionen eine gemeinsame Menge zusätzlicher Argumente, auch wenn diese nicht immer in der jeweiligen Hilfe-Seite mit aufgeführt sind. Die gebräuchlichsten von ihnen sind in Tab. 14.2 beschrieben.

Tabelle 14.2: Diagrammoptionen, die in `par()` und high-level Grafik-Funktionen gesetzt werden können

| Argument | Wert | Bedeutung |
|-----------------------|--|---|
| <code>bty</code> | "o", "n" | Bei "o" wird ein Rahmen um die die Datenpunkte enthaltende Plot-Region gezogen, bei "n" nicht |
| <code>cex</code> | $\langle \text{Zahl} \rangle$ | Vergrößerungsfaktor für die Datenpunkt-Symbole. Voreinstellung ist der Wert 1 |
| <code>cex.axis</code> | $\langle \text{Zahl} \rangle$ | Vergrößerungsfaktor für die Achsenbeschriftungen. Voreinstellung ist der Wert 1 |
| <code>cex.lab</code> | $\langle \text{Zahl} \rangle$ | Vergrößerungsfaktor für die Schrift der Achsenbezeichnungen. Voreinstellung ist der Wert 1 |
| <code>col</code> | " $\langle \text{Farbe} \rangle$ " | Farbe der Datenpunkt-Symbole sowie bei <code>par()</code> zusätzlich des Rahmens um die Plot-Region (s. Abschn. 14.3.2 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. " $\langle \text{Schlüsselwort} \rangle$ " | Linientyp: Schlüsselwörter sind "solid", "dashed", "dotted", "dotdash", "longdash", "twodash" (Abb. 14.5) |
| <code>lwd</code> | $\langle \text{Zahl} \rangle$ | Linienstärke, auch bei Datenpunktssymbolen. Voreinstellung ist der Wert 1. |
| <code>pch</code> | $\langle \text{Zahl 1-25} \rangle$ bzw. " $\langle \text{Buchstabe} \rangle$ " | Art der Datenpunkt-Symbole, vgl. <code>?points</code> und Abb. 14.5. Wird ein Buchstabe angegeben, dient dieser als Symbol der Datenpunkte |
| <code>xpd</code> | NA, TRUE, FALSE | Grafikelemente können nur in der Plot-Region eingefügt werden (Voreinstellung FALSE: clipping). TRUE: gesamte Figure-Region steht zur Verfügung, NA: gesamte Device-Region (Abschn. 14.1.1, 14.5.2) |

Anstatt die in Tab. 14.2 genannten Argumente direkt beim Aufruf von Grafikfunktionen mit anzugeben, können sie durch die separate Funktion `par(<Option>=<Wert>)` festgelegt werden. `par()` kann darüber hinaus noch weitere Einstellungen ändern, die in Tab. 14.3 aufgeführt sind. Die aktuell für das aktive device gültigen Einstellungen lassen sich durch `par("Opt1", ..., "<Opt2>, ...)` als Liste ausgeben, d. h. durch Nennung der relevanten Argumente ohne Zuweisung von Werten. Ohne weitere Argumente gibt `par()` die aktuellen Werte für alle veränderbaren Parameter aus.

Tabelle 14.3: Grafikoptionen, die nur über `par()` verändert werden können

| Argument | Wert | Bedeutung |
|------------------|-----------------------------|---|
| <code>mar</code> | <code><Vektor></code> | Ränder zwischen Plot- und Figure-Region eines Diagramms (Abb. 14.1). 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> | <code><Vektor></code> | Wie <code>mar</code> , jedoch in der Einheit inch |
| <code>oma</code> | <code><Vektor></code> | Ränder zwischen Figure- und Device-Region einer Grafik (Abb. 14.1). 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-Region vollständig aus |
| <code>omi</code> | <code><Vektor></code> | Wie <code>oma</code> , jedoch in der Einheit inch |

Die mit `par()` geänderten Parameter sind Einstellungen für das aktive device. Sie gelten für alle folgenden Ausgaben in dieses device bis zur nächsten expliziten Änderung, oder bis ein neues device aktiviert wird. Der auf der Konsole nicht sichtbare Rückgabewert von `par()` enthält die alten Einstellungen der geänderten Optionen 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. 14.4).

```
# Werte ändern und alte speichern
> op <- par(col="gray60", lwd=2, pch=16)
> plot(rnorm(10), main="Grau, fett, gefüllte Kreise")

# Parameter auf ursprüngliche Werte zurücksetzen
> par(op)
> plot(rnorm(10), main="Standardformatierung")
```

Abbildung 14.5 veranschaulicht die mit `lty` und `pch` einstellbaren Linientypen und Datenpunkt-Symbole (s. Abschn. 14.5 für das Einfügen von Elementen 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 (Abschn. 14.3.2).

```
# Koordinaten der Datenpunkte
> X <- row(matrix(numeric(6*11), nrow=6, ncol=11))
> Y <- col(matrix(numeric(6*11), nrow=6, ncol=11))

# leeres Diagramm mit der richtigen Skalierung
> plot(0:6, seq(1, 11, length.out=7), type="n", axes=FALSE,
+       main="pch Datenpunkt-Symbole und lty Linientypen")
```

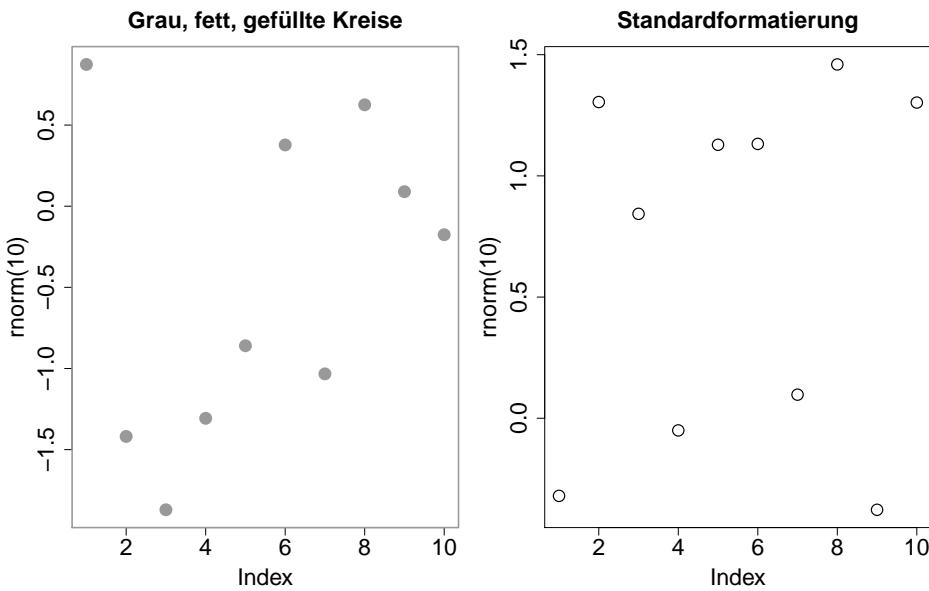


Abbildung 14.4: Verwendung von `par()` zur Diagrammformatierung

```
# Symbole für pch mit grauem Hintergrund für 21-25
> points(X[1:26], Y[1:26], pch=0:25, bg="gray", cex=2)

# Linien für lty
> matlines(X[ , 6:11], Y[ , 6:11], lty=6:1, lwd=2, col="black")

# erklärender Text - Nummer der pch Symbole und lty Linientypen
> text(X[1:26]-0.3, Y[1:26], labels=0:25)
> text(rep(0.7, 6), Y[1, 6:11], labels=6:1)
> text(0, 9, labels="Linientypen für lty", srt=90, cex=1.2)
> text(0, 3, labels="Symbole für pch", srt=90, cex=1.2)
```

14.3.2 Farben spezifizieren

Häufig ist es sinnvoll, Diagrammelemente farblich hervorzuheben, etwa um die Zusammengehörigkeit von Punkten innerhalb von Datenreihen zu kennzeichnen und verschiedene Datenreihen leichter voneinander unterscheidbar zu machen. 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.
- 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.

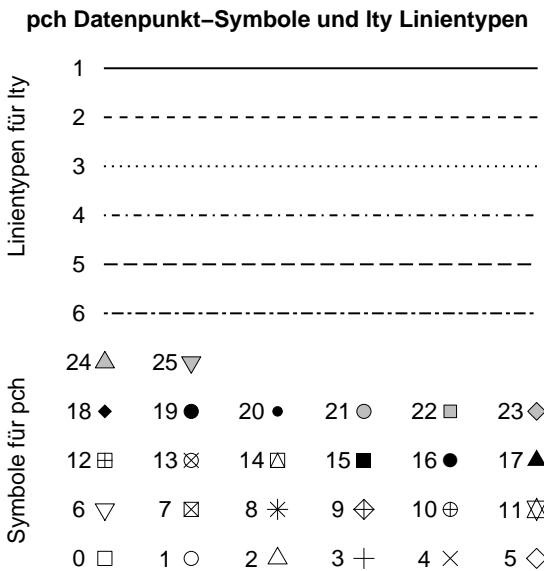


Abbildung 14.5: Datenpunkt-Symbole und Linientypen zur Verwendung für die Argumente `pch` und `lty` von Grafikfunktionen

- Als natürliche Zahl, dem Index für die derzeit aktive Farbpalette. Eine Farbpalette ist ein vordefinierter Vektor von Farben, der mit `palette()` ausgegeben werden kann. Die voreingestellte Palette beginnt mit den Farben "black", "#DF536B", "#61D04F" – der Index 2 entspricht einem entsättigten 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 temporären Wechsel der Palette in einem Vektor zwischengespeichert und später wieder an `palette()` übergeben werden. Alternativ stellt `palette("default")` die ursprünglich voreingestellte Palette wieder her. Für eine Beschreibung der verfügbaren Paletten vgl. `?hcl.colors`. Die dort genannten Funktionen können etwa dazu eingesetzt werden, die aktive Farbpalette zu ändern, indem ihre Ausgabe an `palette()` übergeben wird, z. B. mit `palette(hcl.colors(10))`.⁶

`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 intuitiv ist, stellt R verschiedene Funktionen bereit, mit denen Farben auf einfachere Art definiert werden können. Diese Funktionen geben dann die bezeichnete Farbe im Hexadezimalformat aus.

- Mit `rgb(red=<Rot>, green=<Grün>, blue=<Blau>)` können die Intensitäten der Monitor-Grundfarben Rot, Grün und Blau mit Zahlen im Wertebereich von 0–1 angegeben wer-

⁶Das Paket `colorspace` (Zeileis et al., 2019; Ihaka et al., 2020) definiert eine Reihe von Paletten, deren Farben für besonders gute Unterscheidbarkeit unter verschiedenen Randbedingungen optimiert wurden – etwa für Farbfehlsichtige oder Graustufen-Ausdrucke. Weiterführende Artikel zu diesem Thema nennt `vignette("colorspace")`.

den. Andere Höchstwerte lassen sich über das Argument `maxColorValue` festlegen.⁷ So gibt etwa `rgb(0, 1, 1)` die Farbe "#00FFFF" (Cyan) aus. Bei von `col2rgb()` erzeugten Vektoren ist `rgb()` mit dem Argument `maxColorValue=255` zu verwenden.

- 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.⁸ So entspricht etwa `hsv(0.1666, 1, 1)` der Farbe "#FFFF00" (Gelb). `rgb2hsv(r=<Rot>, g=<Grün>, b=<Blau>)` rechnet von RGB-Farben in HSV-Werte um.
- `hcl(h=<Farbton>, c=<Sättigung>, l=<Helligkeit>)` (*hue, chroma, luminance*) 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 `l` die Luminanz im Bereich von 0–100.
- `gray(<Grauwert>)` akzeptiert eine Zahl im Wertebereich von 0–1, die als Helligkeit einer achromatischen Farbe mit identischen RGB-Werten interpretiert wird. So erzeugt etwa `gray(0.5)` die graue Farbe "#808080". Das optionale Argument `alpha` kontrolliert den Grad simulierter Transparenz (Fußnote 7).
- `colorRamp(<Farben>)` und `colorRampPalette(<Farben>)` erstellen einen Farbverlauf, indem sie im Farbraum gleichmäßig zwischen den übergebenen Farben interpolieren. Über das Argument `alpha=TRUE` gilt dies auch für den Grad simulierter Transparenz. Das Ergebnis von `colorRamp()` ist dabei eine Funktion, die ihrerseits eine Zahl im Bereich [0, 1] akzeptiert und als relative Distanz zwischen der ersten und letzten zu interpolierenden Farbe interpretiert. Dagegen ist die Ausgabe von `colorRampPalette()` eine Funktion, die analog zu den vordefinierten Palettenfunktionen wie `rainbow()` oder `cm.colors()` arbeitet.
- Eine gegebene Farbe kann mit `adjustcolor()` hinsichtlich verschiedener Attribute kontrolliert verändert werden – etwa der Helligkeit, der Sättigung oder der simulierten Durchsichtigkeit.

14.3.3 Achsen formatieren

- Ob durch die Darstellung von Datenpunkten in einem Diagramm automatisch auch Achsen generiert werden, kontrollieren in High-Level-Funktionen 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` dafür den Wert `FALSE`.

⁷ Ein vierter Wert zwischen 0 und 1 kann den Grad des *alpha-blendings* für simulierte Transparenz definieren. Niedrige Werte stehen für sehr durchlässige, hohe Werte für opaque Farben (Abb. 12.2). Diese Art von Transparenz wird nur von manchen devices unterstützt, etwa von `pdf()` oder `png()`.

⁸ Für weitere Funktionen zur Verwendung verschiedener Farträume vgl. `?convertColor` sowie das Paket `colorspace`.

- Achsen können mit `axis()` auch separat einem Diagramm hinzugefügt werden, wobei sich Lage, Beschriftung und Formatierung der Achsenmarkierungen festlegen lassen (Abschn. 14.5.8).
- 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` gehen die Achsen auf beiden Seiten um 4% über den angegebenen Wertebereich hinaus. Um dies zu verhindern, sind in High-Level-Funktionen die Argumente `xaxs="i"` und `yaxs="i"` zu setzen.
- Die Achsenbezeichnungen können in High-Level-Funktionen über die Argumente `xlab="<Name>"` und `ylab="<Name>"` gewählt oder mit Setzen auf `NA` unterdrückt werden.
- Eine logarithmische Skalierung zur Basis 10 lässt sich in High-Level-Funktionen getrennt für jede Achse mit dem Argument `log="x"`, `log="y"` bzw. für beide Achsen gemeinsam mit `log="xy"` erzeugen.
- Die Orientierung der Achsenmarkierungen legt das Argument `las` von `par()` fest. Senkrecht ausgerichtete Beschriftungen beanspruchen dabei häufig mehr vertikalen Platz, als es die Randeinstellungen vorsehen. Mit dem Argument `mar` von `par()` lässt sich der Rand zwischen Plot- und Figure-Region entsprechend anpassen.

14.4 Säulen- und Punktdiagramme

Mit `barplot()` erstellte Säulendiagramme eignen sich vor allem zur Darstellung absoluter oder relativer Häufigkeiten von Gruppenzugehörigkeiten. Die Häufigkeit jeder Gruppe wird dabei durch die Höhe einer Säule repräsentiert.

14.4.1 Einfache Säulendiagramme

Sollen allgemein Kennwerte einer Variable getrennt für Gruppen dargestellt werden, die sich aus den Stufen eines einzelnen Faktors ergeben, lautet die Grundform von `barplot()`:

```
barplot(height=<Vektor>, <Argumente>)
```

- Unter `height` ist ein Vektor einzutragen, wobei jedes seiner Elemente den Kennwert für jeweils eine Bedingung repräsentiert und damit die Höhe einer Säule festlegt. Alternativ ist eine Modellformel der Form `<AV> ~ <UV>` möglich, wobei `<AV>` die Säulenhöhe festlegt und `<UV>` die zugehörigen Bedingungen bezeichnet. Stammen diese Variablen aus einem Datensatz, ist er unter `data` zu übergeben.
- `horiz` bestimmt, ob vertikale Säulen (Voreinstellung `FALSE`) oder horizontale Balken gezeichnet werden. 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, die Voreinstellung ist 0.2.

- `names.arg` nimmt einen Vektor von Zeichenketten für die Beschriftung der Säulen an.
- Ist das Minimum des mit `ylim` definierten Wertebereichs der y -Achse größer als 0, muss `xpd=FALSE` gesetzt werden, um Säulen nicht unterhalb der x -Achse zeichnen zu lassen. Andernfalls erscheinen in der Voreinstellung `xpd=TRUE` die Säulen unterhalb der x -Achse, auch wenn diese nicht bei 0 beginnen.

Als Beispiel diene das Ergebnis mehrerer simulierter Würfe eines sechsseitigen Würfels, wobei zunächst die absoluten Häufigkeiten auf Basis einer Häufigkeitstabelle dargestellt werden sollen (Abb. 14.6).

```
> dice <- sample(1:6, 100, replace=TRUE)      # Würfelwürfe
> (dTab <- xtabs(~ dice))                  # absolute Häufigkeiten
dice
 1   2   3   4   5   6
17  23  14  20  14  12

> barplot(dTab, ylim=c(0, 30), xlab="Augenzahl", ylab="N", col="black",
+           main="Absolute Häufigkeiten")
```

Für die Darstellung der relativen Häufigkeiten erfolgt der Aufruf mit einer Modellformel auf Basis von Variablen eines Datensatzes.

```
> pTab <- proportions(dTab)                 # relative Häufigkeiten
> (d_dice <- as.data.frame(pTab))          # als Datensatz
  dice Freq
1    1 0.17
2    2 0.23
3    3 0.14
4    4 0.20
5    5 0.14
6    6 0.12

> barplot(Freq ~ dice, data=d_dice, xlim=c(0, 0.25), horiz=TRUE,
+           xlab="relative Häufigkeit", ylab="Augenzahl",
+           col="gray50", main="Relative Häufigkeiten")
```

14.4.2 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 zwischen Säulengruppen kenntlich gemacht werden. Eine weitere Möglichkeit besteht darin, jede Einzelsäule nicht homogen, sondern als Stapel mehrerer Segmente darzustellen (Abb. 14.7).

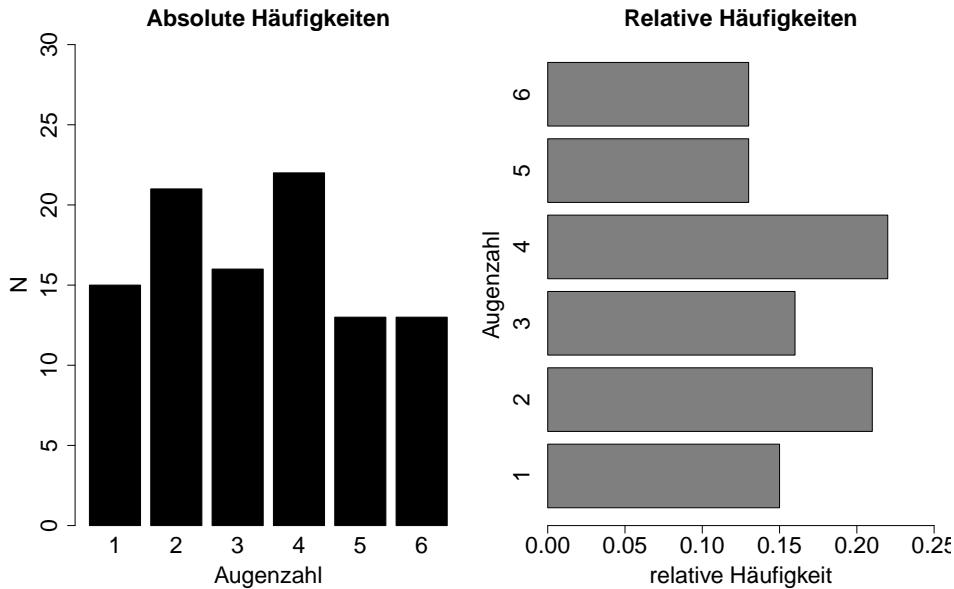


Abbildung 14.6: Säulendiagramme

- Für gruppierte oder gestapelte Säulendiagramme werden die Daten an `height` in Form einer Matrix übergeben, deren Werte die Säulenhöhen bzw. Balkenlängen festlegen. Alternativ eignet sich eine Modellformel der Form $\langle AV \rangle \sim \langle UV1 \rangle + \langle UV2 \rangle$ für Variablen aus einem Datensatz, der unter `data` zu nennen ist.
- Das Argument `beside` kontrolliert, welche Darstellungsart gewählt wird: `TRUE` steht für Säulengruppen, die Voreinstellung `FALSE` für gestapelte Säulen.
- Mit `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.
- In einer Modellformel $\langle AV \rangle \sim \langle UV1 \rangle + \langle UV2 \rangle$ legt $\langle UV1 \rangle$ die Segmente der für die Bedingungen von $\langle UV2 \rangle$ dargestellten Säulen fest (`beside=FALSE`). Mit `beside=TRUE` definiert $\langle UV2 \rangle$ die Säulengruppen und $\langle UV1 \rangle$ die Säulen innerhalb jeder Gruppe.
- 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.

```
> series <- rep(c("first", "second"), each=50)
> (rollAll <- xtabs(~ series + dice))
    dice
series   1 2 3 4 5 6
first    5 15 9 8 7 6
second   12 8 5 12 7 6

> barplot(rollAll, beside=FALSE, legend.text=TRUE, xlab="Augenzahl",
+           ylab="N", main="Abs. Häufigkeiten in 2 Substichproben")
```

Die in jeder der sechs Gruppen vorhandenen zwei Säulen sollten farblich getrennt werden, um die Zugehörigkeit zur ersten bzw. zweiten Substichprobe deutlich zu machen. Dazu wird an `col` ein Vektor mit zwei Farbnamen übergeben, den R intern so häufig recycled, wie es Säulengruppen gibt.

Das zugehörige gruppierte Säulendiagramm wird hier mit dem Aufruf mittels Modellformel erzeugt, die Variablen eines Datensatzes verwendet.

```
> d_rolls <- as.data.frame(rollAll)
> head(d_rolls, n=4)
  series dice Freq
1  first    1    5
2 second    1   12
3  first    2   15
4 second    2    8

> barplot(Freq ~ series + dice, data=d_rolls,
+           beside=TRUE, ylim=c(0, 15), col=c("red", "green"),
+           legend.text=TRUE, xlab="Augenzahl", ylab="N",
+           main="Abs. Häufigkeiten in 2 Substichproben")
```

Eine Verallgemeinerung gestapelter Säulendiagramme, mit denen die bedingten relativen Häufigkeiten einer kategorialen Variable in Abhängigkeit von einer anderen Variable dargestellt werden können, erzeugt `spineplot()` (Abb. 14.7).⁹

```
spineplot(x=<Vektor>, y=<Faktor>, breaks=<Grenzen>,
          xaxlabels=<Bezeichnungen>, yaxlabels=<Bezeichnungen>)
```

Die Daten in `x` und `y` gelten als Werte, die an denselben Beobachtungsobjekten erhoben wurden. Im Fall von `x` sind dies entweder Ausprägungen einer kategorialen (`x` ist ein Faktor) oder einer quantitativen Variable (`x` ist ein numerischer Vektor), 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. Der Flächeninhalt eines Segments spiegelt die

⁹Für die Verteilung eines dichotomen Merkmals in Abhängigkeit einer kontinuierlichen Variable vgl. `cdplot()` (Abb. 8.1). Vergleiche `mosaicplot()` (Abb. 8.5) für die gemeinsame Verteilung von mehr als zwei kategorialen Variablen.

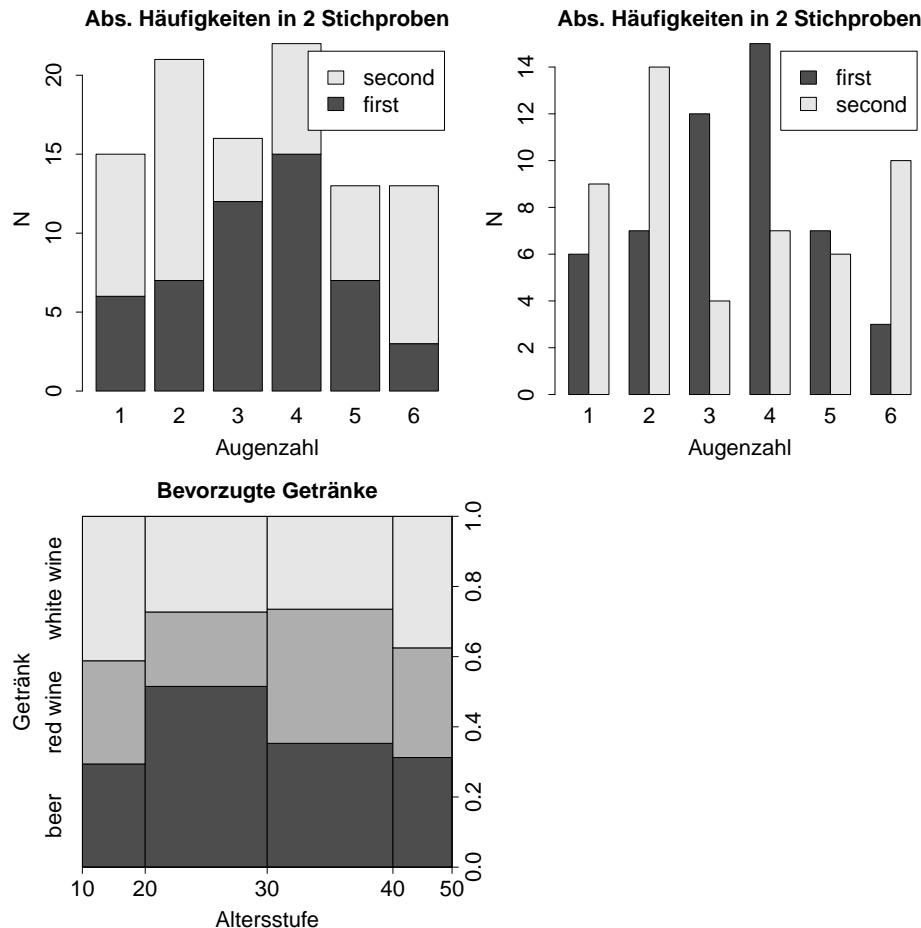


Abbildung 14.7: Gestapeltes und gruppiertes Säulendiagramm absoluter sowie spineplot bedingter relativer Häufigkeiten

relative Häufigkeit der zugehörigen Stufenkombination in der Gesamtstichprobe wider. Die bedingten relativen Häufigkeiten der Stufen von y in der Stufe von x werden dabei über die Höhe der Segmente visualisiert. Die Breite der Säulen repräsentiert 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` übergebenen Vektors mit den Intervallgrenzen. Alternativ kann anstelle von x und y eine Modellformel der Form $\langle y \rangle \sim \langle x \rangle$ mit Variablen eines für `data` zu übergebenden Datensatzes genannt werden. Auch lassen sich die Daten als zweidimensionale 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 simulieren das Ergebnis einer Umfrage, in der Personen unterschiedlichen Alters ihre Präferenz für ein alkoholisches Getränk nennen.

```

> N      <- 100                      # Anzahl Personen
> age    <- sample(18:45, N, replace=TRUE)   # Alter
> drinks <- c("beer", "red wine", "white wine") # mögliche Getränke
> pref   <- factor(sample(drinks, N, 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")

```

14.4.3 Dotchart

`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.

```
dotchart(x=<Daten>, labels=<'Namen'>, groups=<Faktor>, gdata=<Daten>)
```

Für *x* ist der Datenvektor anzugeben. Über das Argument `labels` lassen sich mittels eines Vektors aus Zeichenketten derselben Länge wie *x* die Bezeichnungen der Datenpunkte nennen. 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 Punktgruppen stärker räumlich getrennt sind.

Stellen die Daten Kennwerte verschiedener Gruppen dar, 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. Jeder Wert von `gdata` wird 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. 14.8).

```

> Nj  <- 5                         # Gruppengröße
> DV1 <- rnorm(Nj, 20, 2)           # Daten Gruppe 1
> DV2 <- rnorm(Nj, 25, 2)           # Daten Gruppe 2
> DV  <- c(DV1, DV2)               # Gesamt-Daten
> IV  <- gl(2, Nj)                 # Gruppenzugehörigkeit
> Mj  <- tapply(DV, IV, mean)      # Gruppenmittel

```

```
> dotchart(DV, gdata=Mj, pch=rep(c(17, 19), each=Nj),
+           color=rep(c("red", "blue", "green"), each=Nj),
+           gcolor="black", groups=IV, labels=rep(LETTERS[1:Nj], 2),
+           xlab="Messwerte", ylab="Gruppen",
+           main="Individuelle Ergebnisse und Mittel aus 2 Gruppen")
```

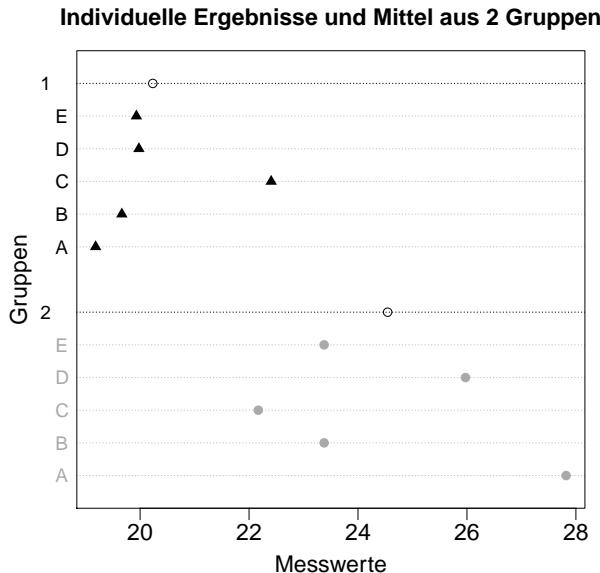


Abbildung 14.8: Dotchart von individuellen Messwerten in zwei Gruppen samt zugehöriger Mittelwerte

14.5 Elemente einem bestehenden Diagramm hinzufügen

Ein bereits erstelltes Diagramm lässt sich nachträglich durch zusätzliche Elemente erweitern, die durch Low-Level-Funktionen erzeugt werden (Tab. 14.4). Auch einige High-Level-Funktionen besitzen das Argument `add=TRUE`, durch das ihre Ausgabe dem derzeit aktiven device hinzugefügt wird, ohne dessen Inhalte zuvor zu löschen. In diesem Fall kann meist durch das Argument `at=<Position>` bestimmt werden, an welcher Stelle der Plot-Region 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 device erscheint. Später eingefügte Elemente werden immer über bereits bestehende gezeichnet – neue Inhalte übermalen also ältere Inhalte, die sich an derselben Stelle befinden.

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 device mit `dev.new()` sowie anschließend `plot.new()` geöffnet und zum Einfügen von Elementen vorbereitet werden.

Tabelle 14.4: Mögliche Diagrammelemente und die sie hinzufügenden Funktionen

| Diagrammelement | hinzufügende Low-Level-Grafikfunktion |
|----------------------|---|
| 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>
Abschn. 16.1 |
| Funktionsgraphen | <code>curve()</code> |
| Text | <code>title()</code> , <code>legend()</code> , <code>text()</code> , <code>mtext()</code> |
| Achsen | <code>axis()</code> |

14.5.1 Koordinaten in einem Diagramm identifizieren

Die meisten nachträglich einzufügenden Diagrammelemente erfordern für ihre Positionierung die Angabe von (x, y) -Koordinaten in einem Koordinatensystem, das durch die in der Plot-Region (Abschn. 14.1.1, Abb. 14.1) bereits dargestellten Daten festgelegt ist. Diese *User-Koordinaten* neu einzufügender Elemente ergeben sich oft direkt 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 `locator(n=<Anzahl>)`, nachdem ein Diagrammfenster erstellt wurde und noch geöffnet ist.

Das Argument `n` legt fest, wie viele Koordinaten zu bestimmen sind. Durch den Aufruf von `locator()` ändert sich der Mauszeiger zu einem Kreuz, sobald er sich über der Diagrammfläche befindet. 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 (in RStudio durch Drücken der `ESC` Taste). Daraufhin gibt `locator()` eine Liste mit den Komponenten `x` und `y` zurück, die in Form jeweils eines Vektors die (x, y) -Koordinaten der angeklickten Punkte enthalten.

```
> plot(rnorm(10))                      # Streudiagramm
> (xy <- locator(n=3))                 # 3 Punkte durch Klicken auswählen
$x
[1] 2.402735 7.008204 3.141016

$y
[1] 0.656082382 -0.006072532 -0.484488124
```

14.5.2 In beliebige Diagrammbereiche zeichnen

Die Voreinstellungen für ein device legen fest, dass zusätzliche Elemente nur in der Plot-Region gezeichnet werden können (Abschn. 14.1.1, Abb. 14.1). Diese *clipping* genannte Beschränkung kann über das Argument `xpd=NA` von `par()` aufgehoben werden (Abb. 14.9). Für zusätzliche Grafikelemente müssen auch außerhalb der Plot-Region letztlich User-Koordinaten verwendet werden, die durch die bereits dargestellten Daten definiert sind. Die einzelnen Device-Regionen besitzen jedoch auch eigene Koordinatensysteme, die es erlauben, allgemeingültige Koordinaten zur Plazierung von Elementen zu verwenden, die unabhängig von den konkreten Daten sind: Die Ränder jeder Device-Region definieren dafür jeweils ein Rechteck mit (x, y) -Koordinaten im Bereich 0–1, wobei (0, 0) die linke untere und (1, 1) die rechte obere Ecke bezeichnet. Diese Koordinaten müssen vor der Darstellung eines Elements in User-Koordinaten umgewandelt werden, wozu `cnvrt.coords()` aus dem Paket `Hmisc` dient.

```
cnvrt.coords(x=<x-Koordinaten>, y=<y-Koordinaten>,
              input=c("usr", "plt", "fig", "dev", "tdev"))
```

Unter `x` und `y` sind die x bzw. y -Koordinaten der einzusteckenden Elemente jeweils als Vektor einzutragen. Auf welches Koordinatensystem sie sich beziehen, legt das Argument `input` fest. Dabei steht "`usr`" für User-Koordinaten, "`plt`" für Koordinaten der Plot-Region, entsprechend "`fig`" für die Figure-Region, "`dev`" für die Device-Region ohne die äußeren Ränder und "`tdev`" für die Device-Region inkl. dieser Ränder.

`cnvrt.coords()` gibt eine Liste zurück, die für jedes Koordinatensystem eine Komponente mit den konvertierten Koordinaten besitzt. Die Komponenten tragen dabei dieselben Namen wie die möglichen Werte für das Argument `input`. Jede Komponente ist wiederum eine Liste mit den Komponenten `x` und `y` für die (x, y) -Koordinaten.

```
# clipping ausschalten und äußere Ränder setzen
> par(xpd=NA, oma=c(2, 2, 2, 2))
> plot(rnorm(10), xlab=NA, ylab=NA, pch=20)          # Streudiagramm
> library(Hmisc)                                     # für cnvrt.coords()

# Figure-Region links-unten
> pt1 <- cnvrt.coords(0, 0, input="fig")            # Figure-Koordinaten
> pt1$usr                                         # User-Koordinaten
$x
[1] -0.9669358

$y
[1] -3.389251

# Kreuz und zugehörigen Text einfügen
> points(pt1$usr, pch=4, lwd=5, cex=5, col="darkgray")
> text(pt1$usr$x + 0.5, pt1$usr$y, adj=c(0, 0), cex=1.5,
+       labels="Kreuz links-unten Figure-Region")

# gesamte Device-Region links-oben und rechts-unten
```

```
> pt2 <- cnvrt.coords(c(0.05, 0.95), c(0.95, 0.05), input="tdev")
> pt2$usr
# User-Koordinaten
$x
[1] -1.064920 11.281049

$y
[1] 2.163520 -3.433674

# Pfeil und Text einfügen
> arrows(x0=pt2$usr$x[1], y0=pt2$usr$y[1], x1=pt2$usr$x[2],
+         y1=pt2$usr$y[2], lwd=4, code=3, angle=90, lend=2,
+         col="darkgray")

> text(pt2$usr$x[1] + 0.5, pt2$usr$y[1], adj=c(0, 0), cex=1.5,
+       labels="Pfeil über gesamte Device-Region")
```

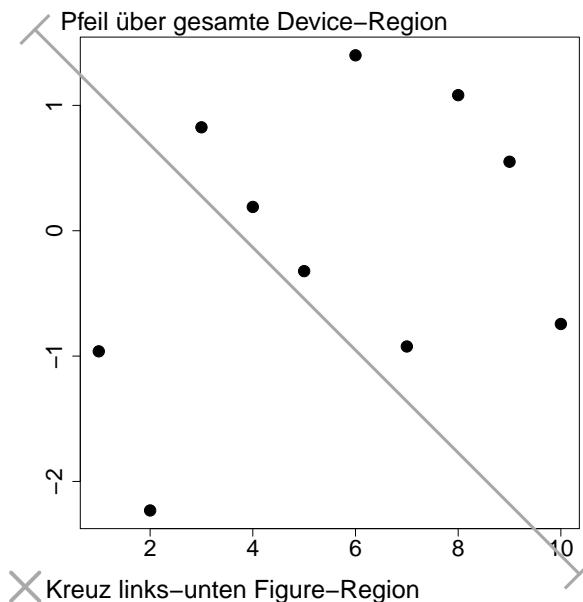


Abbildung 14.9: Grafikelemente an beliebiger Stelle eines Diagramms einfügen

14.5.3 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, die Argumente `x`, `y` und `type` stimmen in ihrer Bedeutung mit jenen von `plot()` überein. 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)
```

14.5.4 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 Punkte an, das Argument `type` bestimmt wie in `plot()` die genaue Art der Linien.¹⁰ `matlines()` arbeitet wie `matpoints()`, die (x, y) -Koordinaten können also für mehrere Datenreihen gleichzeitig in Form jeweils einer Matrix an die Argumente `x` und `y` übergeben werden (Abb. 14.10).

```
> 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 $Y = bX + a$ mit den Parametern `a` für den Schnittpunkt mit der y -Achse und `b` für die Steigung beschrieben werden. Diese beiden Parameter erwartet alternativ auch das Argument `coef` in Form eines Vektors mit zwei Elementen. Statt dieser Parameter akzeptiert `abline()` auch ein mit `lm()` erstelltes Objekt, um die angepasste Vorhersagegerade einer einfachen linearen Regression darzustellen (Abschn. 6.2). Ein über die gesamte Breite der Plot-Region gehender horizontaler Geradenabschnitt kann 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, etwa um ein Gitter zu zeichnen. Sowohl für `h` als auch für `v` kann dafür gleichzeitig jeweils ein Vektor von Koordinaten angegeben werden (Abb. 14.10).

```
> abline(h=0, v=0, col="green", lwd=1.6)

grid(nx=NULL, ny=nx)
```

Die Argumente `nx` und `ny` von `grid()` bestimmen, wie viele Elemente in das einzufügende 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

¹⁰Das Aussehen von Linienenden sowie das Verhalten von sich treffenden Endpunkten bestimmen die Argumente `lend` und `ljoin` von `par()`.

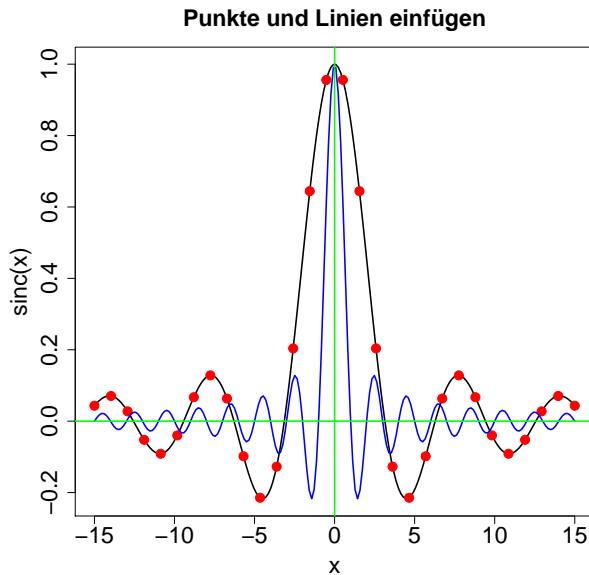


Abbildung 14.10: Einfügen von Diagrammelementen: Punkte und Linien

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 grafisch veranschaulicht, insbesondere die Residuen als Differenz der tatsächlichen zu den vorhergesagten Werten (Abb. 14.11).

```
> height <- rnorm(20, 175, 7)                      # Prädiktor
> weight <- 0.5*height + 10 + rnorm(20, 0, 4)       # Kriterium
> fit      <- lm(weight ~ height)                  # Regression
> pred     <- fitted(fit)                         # Vorhersage
> plot(weight ~ height, asp=1, col="blue", pch=16,
+       main="Gitter, Segmente und Pfeile einfügen")

> abline(fit, lwd=2)                                # Regressionsgerade
> grid(lwd=2, col="darkgray")
```

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.

```
# Differenz Vorhersage-Kriterium (Residuen)
> segments(x0=height, y0=pred, x1=height, y1=weight, col="gray")
```

Ähnlich wie Liniensegmente können auch Pfeile mit `arrows()` 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 eingezeichnet, mit `code=1` am Startpunkt, mit `code=3` an beiden Enden (Abb. 14.11).

```
> 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)
```

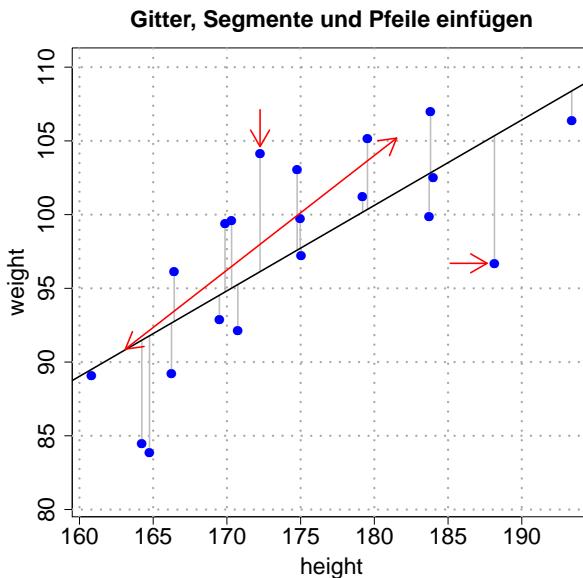


Abbildung 14.11: Einfügen von Diagrammelementen: Gitter, Liniensegmente und Pfeile

14.5.5 Polygone

Einem Diagramm kann mit `box(which="<Region>")` an verschiedenen Stellen ein rechteckiger Rahmen hinzugefügt werden. Das Argument `which` bestimmt, ob der Rahmen um die Plot-Region ("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.

`rect()` erzeugt beliebig dimensionierte Rechtecke, die sich in der Plot-Region 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 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 (Abb. 14.12; 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), times=n)  # Grün-Anteil
> cols  <- rgb(colsR, colsG, 0)                # Farben in RGB

# Koordinaten der Quadrate festlegen
> xLeft  <- rep(seq(0, 1-len, by=len), times=n)      # x-Koord. links
> yBot   <- rep(seq(0, 1-len, by=len), each=n)        # y-Koord. unten
> xRight <- rep(seq(len, 1, by=len), times=n)        # x-Koord. rechts
> yTop   <- rep(seq(len, 1, by=len), each=n)          # y-Koord. oben

# zunächst ein leeres Diagramm erzeugen, dann Rechtecke einzeichnen
> 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")
> 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, labels=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, labels=cols[idx])
```

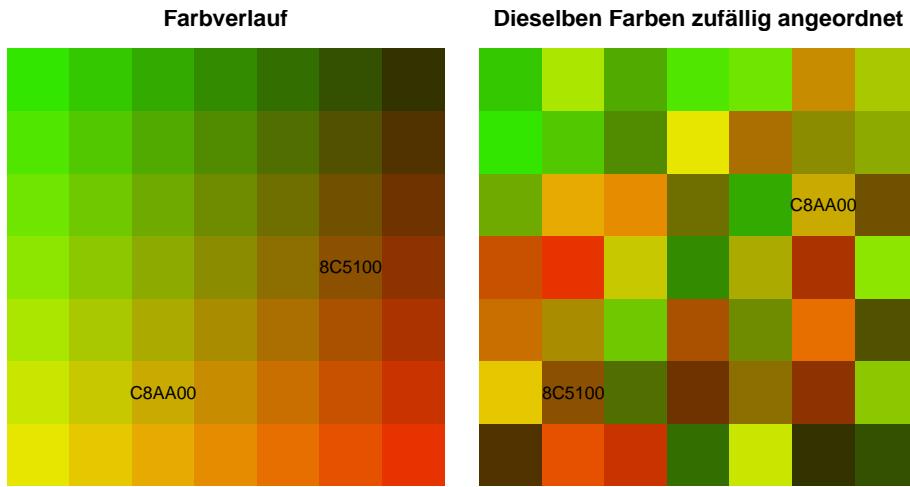


Abbildung 14.12: Mit `rect()` erzeugte Rechtecke zur Veranschaulichung eines Farbphänomens:
Beide Grafiken sind aus denselben Quadranten zusammengesetzt, die jedoch – auch abgesehen vom Chevreul-Effekt links – in den beiden Konfigurationen unterschiedlich aussehen. Zwei jeweils links und rechts identische Farben sind mit ihrem Hexadezimalwert bezeichnet

`polygon()` 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.¹¹ Soll kein Rahmen gezogen werden, ist `border=NA` zu setzen. Mittels eng gesetzter Eckpunkte lassen sich auch runde Formen durch Polygone approximieren.

```
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. 14.13).

```
> mu      <- 0                      # Erwartungswert
> sigma   <- 3                      # Streuung
> xLims  <- c(mu-4*sigma, mu+4*sigma) # Grenzen x-Achse
```

¹¹Mit `polypath()` lassen sich auch Polygone mit sich selbst überschneidenden Kanten zeichnen.

```

> X      <- seq(xLims[1], xLims[2], length.out=100)      # x-Werte
> Y      <- dnorm(X, 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(X, mu, sigma)      # Werte der Verteilungsfunktion

# Grafik leer mit breitem rechten Rand öffnen
> par(mar=c(5, 4, 4, 5))
> plot(X, Y, type="n", xlim=xLims-c(-2, 2), xlab=NA, ylab=NA,
+       main="Dichtefunktion und Verteilungsfunktion N(0, 3)")

# Rahmen, 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")

> lines(X, Y, lwd=2)                  # Dichtefunktion

# Verteilungsfunktion einzeichnen
> par(new=TRUE)                      # folgendes Diagramm in bestehendes zeichnen
> plot(X, 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)
```

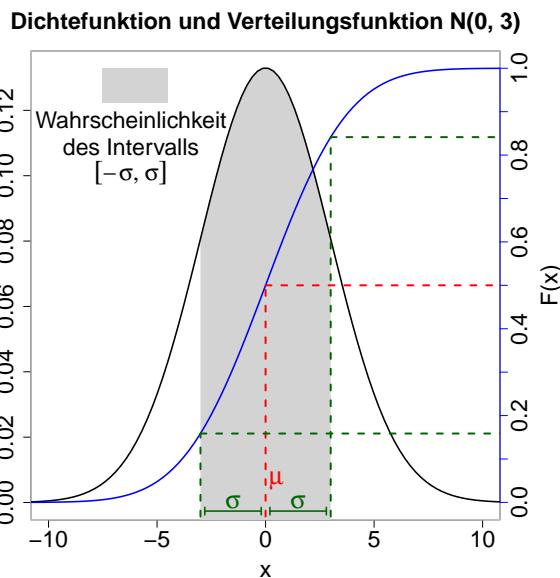


Abbildung 14.13: Diagrammelemente einfügen: Rechtecke, Polygone, Achsen, Text und Symbole

14.5.6 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 als Funktionsgleichung mit `x` als Variable spezifiziert werden (z. B. `dnorm(x, mean=1, sd=1)` oder `x^2 + 10`). Hier wird also dieselbe Notation benutzt, wie um aus einem bestehenden Vektor `x` einen neuen Vektor mit den Funktionswerten zu generieren. Als Kurzform lässt sich auch nur der Name einer Funktion angeben (z. B. `dnorm`), die dann mit den Voreinstellungen für ihre Argumente aufgerufen wird. In welchem Wertebereich die Funktion ausgewertet werden soll, bestimmen die Argumente `from` und `to`. Dabei legt das Argument `n` die Anzahl der Stützstellen fest, an wie vielen gleichabständigen Stellen in diesem Bereich also Funktionswerte bestimmt und eingezeichnet werden. Um den Funktionsgraphen dem aktuell aktiven device hinzuzufügen, ohne

dessen Inhalte zu löschen, ist `add=TRUE` zu setzen (Abb. 14.14). Als auf der Konsole unsichtbaren Rückgabewert liefert `curve()` eine Liste mit den (x, y) -Koordinaten der gezeichneten Punkte.

```
> mu      <- 0                      # Normalverteilung Erwartungswert
> sigma   <- 2                      # theoretische Streuung
> curve(dnorm(x, mean=1, sd=1), 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)
```

14.5.7 Text und mathematische Formeln

```
title(main="<Titel>", sub="<Untertitel>")
```

Der Diagrammtitel lässt sich in High-Level-Grafikfunktionen 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 Escape-Sequenz `\n` dient innerhalb einer Zeichenkette als Symbol für den Zeilenwechsel (Abb. 14.14).

```
> title(main="zwei Normalverteilungskurven", sub="Untertitel")
```

Eine Legende zur Erläuterung der verwendeten Symbole erstellt `legend()`.

```
legend(x=x-Koordinate, y=y-Koordinate, legend="<Text>",
       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, s. Abschn. 14.3.1). Für diese Argumente ist jeweils ein Vektor derselben Länge wie `legend` einzugeben, wobei `NA` Einträge bedeuten, 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(1, 1)", "N(0, 2)"),
+         col=c("blue", "black"), lty=c(1, 2))
```

Mit `text()` lässt sich allgemein Text an beliebiger Stelle in die Plot-Region eines Diagramms einfügen.

```
text(x=<x-Koordinaten>, y=<y-Koordinaten>, adj=<Positionskorrektur>,
      labels=<Name>, srt=<Rotationswinkel>)
```

Zunächst sind mit den Argumenten `x` und `y` die Koordinaten des Texts festzulegen. Sollen mehrere Textelemente gleichzeitig eingefügt werden, sind hier Vektoren zu übergeben. In der Voreinstellung beziehen sich die Koordinaten auf den Mittelpunkt des Texts, 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 `labels` in Form eines Vektors von Zeichenketten übergeben werden. Das Argument `srt` erwartet eine Winkelangabe in Grad und erlaubt es, den Text um den mit `adj` definierten Drehpunkt zu rotieren.

```
> text(x=3.6, y=0.35, labels="Normalverteilung\nN(1, 1)")
> text(x=-3.5, y=0.1, labels="N(0, 2)")
```

Sollen Textelemente nicht innerhalb der Plot-Region, sondern an deren äußere Ränder geschrieben werden, ist `mtext()` hilfreich.

```
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 Plot-Region entspricht.

```
> mtext(text="Wahrscheinlichkeitsdichte", side=3)
```

In allen Funktionen zum Einfügen von Text können mit Hilfe einer an das Textsatzsystem L^AT_EX angelehnten Syntax auch jedwede Art von Symbolen (griechische Buchstaben, mathematische Sonderzeichen, etc.) und mathematische Formeln definiert werden (Ligges, 2002). Zu diesem Zweck wird eine Formel in Textform an `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 Hilfe-Seite `?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, ")")))
```

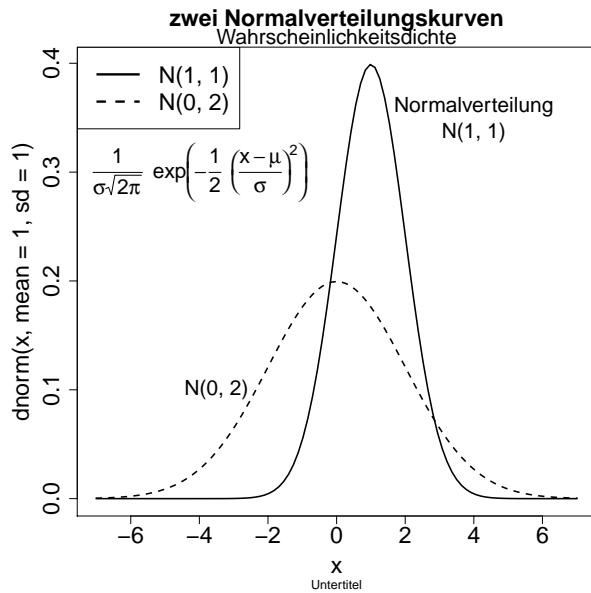


Abbildung 14.14: Diagrammelemente einfügen: Funktionsgraphen, Text, Legende und mathematische Formeln

14.5.8 Achsen

Ob durch die Darstellung von Datenpunkten in einem Diagramm automatisch auch Achsen generiert werden, kontrollieren in High-Level-Grafikfunktionen 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 `axes` dafür den Wert `FALSE`.

Für eine feinere Kontrolle über den Wertebereich sowie über Aussehen und Lage der Wertemarkierungen der Achsen empfiehlt es sich, ihre automatische Generierung zunächst mit `axes=FALSE` zu unterdrücken. Nachdem das Diagramm erstellt wurde, können dann mit dem Befehl `axis()` samt seiner Argumente zur Formatierung und Positionierung Achsen hinzugefügt werden (Abb. 14.13, 14.15).

```
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), 3 (oben, alternative x -Achse) und 4 (rechts, alternative y -Achse). Die Wertemarkierungen der Achse lassen sich über `at` in Form eines Vektors festlegen. 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 Rändern der Plot-Region 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, lty=c(1, 2, 4), col=c(12, 14, 17), type="l",
+           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, lwd=2)

> abline(h=0, v=0, lwd=2)
> legend(x="bottomleft", legend=c("sin(x)", "cos(x)", "tan(x)"),
+          lty=c(1, 2, 4), col=c(12, 14, 17))
```

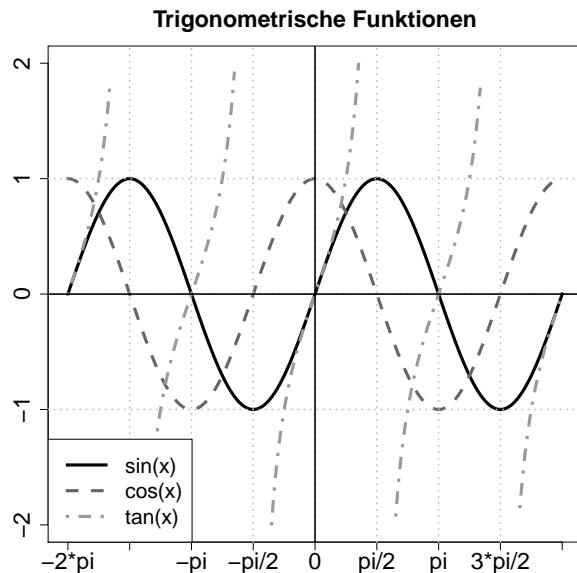


Abbildung 14.15: Diagrammachsen mit `axis()` anpassen

14.5.9 Fehlerbalken

Fehlerbalken werden zusätzlich zu Kennwerten von Variablen vor allem in Punktdiagrammen eingezeichnet, um die Variabilität der Daten auszudrücken. Als Maß der Variabilität kann dabei u. a. die Breite eines statistischen Konfidenzintervalls für einen Parameter (z. B. den Erwartungswert) oder ein deskriptives Maß wie die Streuung verwendet werden.¹²

¹²Für Konfidenzellipsen als Maß für die Variabilität zweidimensionaler Daten vgl. Abschn. 14.6.8.

Das Paket `DescTools` enthält die Funktion `ErrBars()`, die Fehlerbalken einem bestehenden Diagramm hinzufügt. Das vertikale Zentrum der Fehlerbalken wird als Punkt gezeichnet, so dass es nicht unbedingt notwendig ist, zusätzlich Punkte zur Veranschaulichung des Parameters zu zeichnen, dessen Variabilität über einen Fehlerbalken dargestellt wird (Abb. 14.16).

```
ErrBars(from=<y-Koordinate unten>, to=<y-Koordinate oben>,
        pos=<x-Koordinate>, horiz=FALSE)
```

Unter `from` und `to` werden die y -Koordinaten der unteren bzw. oberen Grenzen der Fehlerbalken definiert. Für `pos` sind die x -Koordinaten der Fehlerbalken anzugeben. Mit `horiz` wird definiert, ob die Balken vertikal (Voreinstellung `FALSE`) oder horizontal (`TRUE`) zu zeichnen sind. Die Funktion verfügt über weitere Optionen zur Formatierung der Fehlerbalken bzgl. ihrer Farbe, Linienstärke, etc.

```
# Simulation von Messwerten in 4 Gruppen ohne Gruppeneffekt
> Nj <- c(15, 20, 18, 22)                      # Gruppengrößen
> P <- length(Nj)                               # Anzahl Gruppen
> DV <- rnorm(sum(Nj), rep(c(30, 20, 25, 15), Nj), 6)    # Daten
> IV <- factor(rep(1:P, Nj))                  # Gruppenzugehörigkeit
> Mj <- tapply(DV, IV, mean)                  # Gruppenmittel
> Sj <- tapply(DV, IV, sd)                    # Gruppenstreuungen

# halbe Breite 95% t-Konfidenzintervalle für Erwartungswerte
> ciWidths <- qt(0.975, Nj-1) * Sj/sqrt(Nj)

# Punktdiagramm getrennt nach Gruppen
> stripchart(DV ~ IV, method="jitter", xaxt="n", xlab="Gruppe",
+             ylim=c(0, 40), main="Rohdaten & Konfidenzintervalle",
+             col="darkgray", pch=16, vert=TRUE)

> library(DescTools)                           # für ErrBars()
> ErrBars(from=Mj-ciWidths, to=Mj+ciWidths, pos=1:P, length=0.1,
+           col="blue", col.pch="blue", lwd=2, pch=19)

> axis(side=1, at=1:P, labels=LETTERS[1:P])    # 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.

```
> Mj1 <- c(2, 3, 6, 3, 5)                      # Mittelwerte Gruppe 1
> Sj1 <- c(1.7, 1.8, 1.7, 1.9, 1.8)          # Streuungen Gruppe 1
> Mj2 <- c(4, 3, 2, 1, 3)                      # Mittelwerte Gruppe 2
> Sj2 <- c(1.4, 1.7, 1.7, 1.3, 1.5)          # Streuungen Gruppe 2
> Q <- length(Mj1)                             # Anzahl Mittelwerte
> xOff <- 0.1        # horizontaler offset zwischen Werten einer Gruppe

# Mittelwerte
> plot(c((1:Q)-xOff, (1:Q)+xOff), c(Mj1, Mj2), pch=19,
```

```

+     main="Mittelwerte & SDs im 5x2 Design",
+     xlab="Faktor A", ylab="Mittelwert",
+     col=rep(c("blue", "red"), each=5), ylim=c(0, 8))

# Fehlerbalken
> ErrBars(from=c(Mj1, Mj2) - c(Sj1, Sj2),
+           to=c(Mj1, Mj2) + c(Sj1, Sj2),
+           pos=c((1:Q)-xOff, (1:Q)+xOff), lty=rep(1:2, each=Q),
+           col=rep(c("blue", "red"), each=Q),
+           col.pch=rep(c("blue", "red"), each=Q), pch=19)

> legend(x="topleft", legend=c("B-1", "B-2"), pch=c(19, 19),
+         col=c("blue", "red"))

```

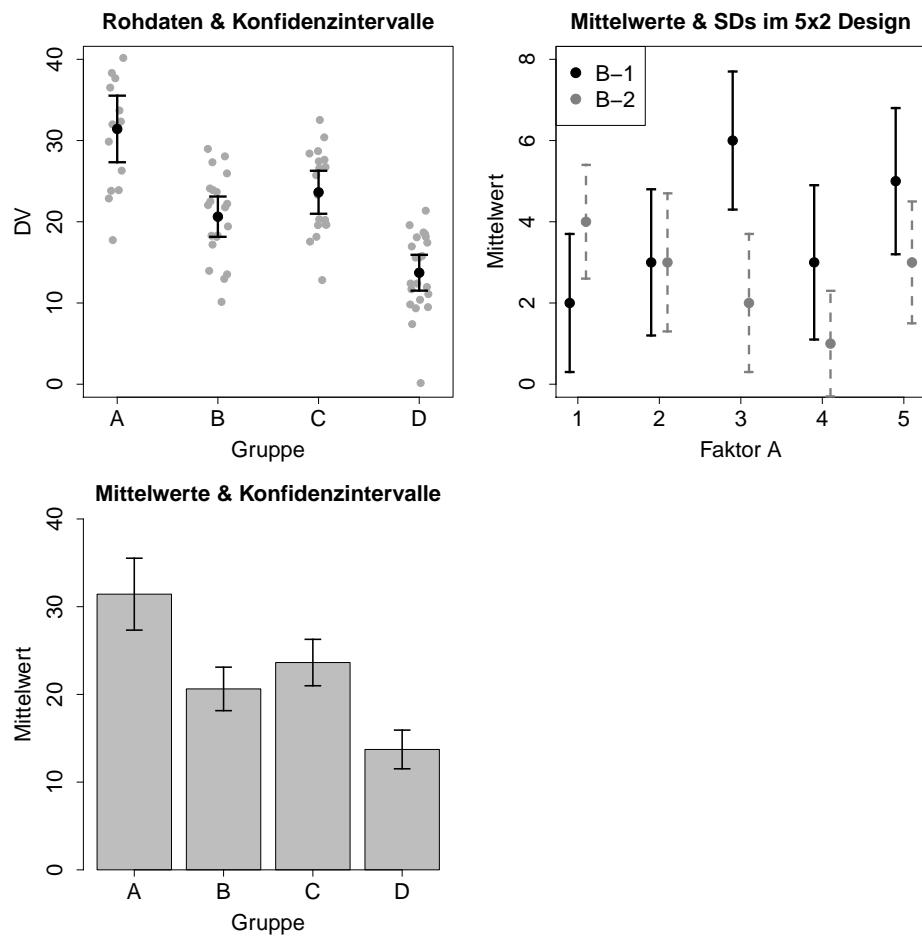


Abbildung 14.16: Fehlerbalken mit `ErrBars()` oder als Pfeile einfügen

Um selbst konstruierte Fehlerbalken in ein Diagramm einzufügen, können mit `arrows()` erstellte Pfeile „zweckentfremdet“ werden (Abschn. 14.5.4). 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`, Abb.

[14.16](#)). Die benötigten x -Koordinaten der Fehlerbalken sind im von `barplot()` zurückgegebenen Vektor enthalten. Bei einem gruppierten Säulendiagramm handelt es sich stattdessen um eine Matrix. Die Höhe der Säulen liefert den vertikalen Mittelpunkt der Fehlerbalken und ergibt sich direkt aus den Daten, ebenso die y -Koordinaten der Endpunkte der Fehlerbalken als Grenzen des Konfidenzintervalls für den geschätzten statistischen Kennwert.

```
# Säulendiagramm
> barsX <- barplot(height=Mj, xaxt="n", xlab="Gruppe", ylim=c(0, 40),
+                      ylab="Mittelwert", main="Mittelwerte & Konfidenzintervalle")

> axis(side=1, at=barsX, labels=LETTERS[1:P])      # Gruppenbezeichnungen
> ciLo <- Mj - ciWidths                            # Fehlerbalken untere Grenze
> ciHi <- Mj + ciWidths                            # Fehlerbalken obere Grenze

# Fehlerbalken mit arrows()
> arrows(x0=barsX, y0=ciLo, x1=barsX, y1=ciHi,
+         code=3, angle=90, length=0.1, col="blue")
```

14.5.10 Rastergrafiken

Rastergrafiken definieren ein Bild als Ansammlung diskreter Bildpunkte (*pixel – picture elements*), während Vektorgrafiken dies durch eine strukturelle Beschreibung der im Bild dargestellten Objekte tun. Für jeden Bildpunkt speichern Rastergrafiken den Farbwert, den das Bild an diesem Punkt besitzen soll. Rastergrafiken werden auch als Bitmap- bzw. Pixel-Grafiken bezeichnet. Sie lassen sich in zwei Schritten erzeugen und darstellen: Zunächst ist eine Matrix mit Farbwerten zu füllen, wobei jedes ihrer Elemente einen Bildpunkt festlegt. `rasterImage()` fügt das so definierte Bild dann einem bestehenden Diagramm hinzu (Abb. [14.17](#)).¹³

```
rasterImage(image=<Farb-Matrix>,
            xleft=<x-Koordinate>, ybottom=<y-Koordinate>,
            xright=<x-Koordinate>, ytop=<y-Koordinate>,
            angle=<Drehwinkel>, interpolate=TRUE)
```

An `image` ist eine Matrix mit Farbwerten zu übergeben (Abschn. [14.3.2](#)). Anstatt die Farbwerte direkt in eine Matrix zu schreiben, lässt sich auch eine Matrix bzw. ein array mit Zahlen im Bereich von 0–1 mit Hilfe von `as.raster()` in eine solche Farbwert-Matrix umwandeln. Wird dabei eine Matrix übergeben, symbolisiert jede Zahl den Grauwert eines Bildpunkts. Für farbige Bilder ist ein array mit drei Ebenen zu verwenden: Die Matrix der ersten Ebene definiert den Rot-Anteil, die der zweiten Ebene den Grün- und die der dritten Ebene den Blau-Anteil der Farbe jedes Bildpunkts.

Mit den Argumenten `xleft` und `ybottom` werden die (x, y) -Koordinaten der Position im Diagramm definiert, an der die linke untere Bildecke liegen soll, mit `xright` und `ytop` entsprechend die Koordinaten für die rechte obere Bildecke. `angle` erlaubt es mit einer Winkelangabe in Grad, die Rastergrafik gegen den Uhrzeigersinn um die linke untere Bildecke zu drehen. Der

¹³Als Grafikdatei vorhandene Bitmap-Bilder können mit Funktionen aus den Paketen `imager` ([Barthelme, 2020](#)) oder `magick` ([Ooms, 2024](#)) eingelesen werden.

über die Koordinaten der Bildecken ausgewählte Bereich des Diagramms kann in der Darstellung letztlich mehr oder weniger Bildpunkte abdecken, als in `image` definiert sind. Das Bild muss deshalb auf den Abbildungsbereich gestreckt oder gestaucht werden. In der Voreinstellung `interpolate=TRUE` wird beim Strecken linear zwischen der Farbe vormals angrenzender Bildpunkte interpoliert, was in etwas weniger abrupten Farbabstufungen resultiert.

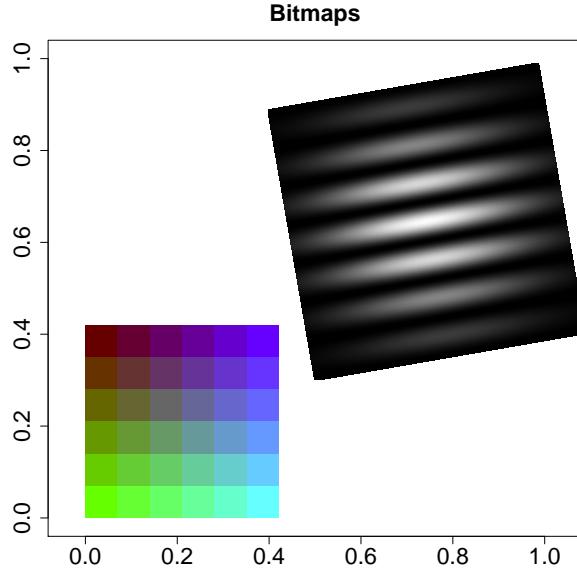


Abbildung 14.17: Rastergrafiken in einem Diagramm anzeigen

Im Beispiel soll ein Bild aus farbigen Rechtecken und zusätzlich eine Gabor-Funktion dargestellt werden, also eine orientierte zweidimensionale Cosinus-Funktion, deren Amplitude einer zweidimensionalen Normalverteilung folgt.

```
> pxSq <- 6                                # Anzahl Bildpunkte pro Achse
> colsR <- rep(0.4, pxSq^2)                 # Rot-Anteil: hier konstant
> colsG <- rep(seq(0, 1, length.out=pxSq), times=pxSq)      # Grün
> colsB <- rep(seq(0, 1, length.out=pxSq), each=pxSq)       # Blau

# array mit drei Ebenen: Rot, Grün, Blau
> arrSq <- array(c(colsR, colsG, colsB), c(pxSq, pxSq, 3))
> sqIm <- as.raster(arrSq)                  # Zahlen -> Farbwerte

# Gabor-Funktion
> pxGab <- 500                               # Anzahl Bildpunkte pro Achse
> alpha <- 0.5                                # Faktor für Winkel 2D-Cosinus
> beta <- min(c(1-alpha, 1+alpha))            # für komplementären Winkel
> freq <- 3.5                                 # Frequenz 2D-Cosinus

# x- und y-Wertebereich für 2D-Cosinus und 2D-Normalverteilung
> vals <- rep(seq(-2*pi, 2*pi, length.out=pxGab), pxGab)

# Matrizen mit den x- bzw. y-Koordinaten aller Bildpunkte
```

```

> x    <- matrix(vals, nrow=pxGab, byrow=TRUE)
> y    <- matrix(vals, nrow=pxGab, byrow=FALSE)
> phi <- alpha*x + beta*y           # konvexe Mischung der Koordinaten

# 2D-Cosinus skaliert auf Werte im Bereich [0, 1]
> cosMat <- 0.5*cos(freq*phi) + 0.5

# Werte unkorrelierter 2D-Normalverteilung
> library(mvtnorm)                 # für dmvnorm()
> mu       <- c(0, 0)               # Zentroid
> sigma    <- diag(2)*10            # Kovarianzmatrix
> gaussVal <- dmvnorm(cbind(c(x), c(y)), mu, sigma)

# als Matrix skaliert auf Werte im Bereich [0, 1]
> gaussMat <- matrix(gaussVal, nrow=pxGab) / max(gaussVal)

# Gabor = Produkt von 2D-Cos und 2D-NV, Zahlen -> Farbwerte
> gabIm <- as.raster(cosMat*gaussMat)

# öffne leeres Diagramm
> plot(c(0, 1), c(0, 1), type="n",
+       main="Bitmaps", xlab="", ylab="", asp=1)

# füge Matrizen mit Farbwerten als Rastergrafiken ein
> rasterImage(sqIm,     0, 0,   0.3, 0.3, angle=0, interpolate=FALSE)
> rasterImage(gabIm, 0.5, 0.3, 1.1, 0.9, angle=10, interpolate=TRUE)

```

R verfügt auch über die grundlegenden Funktionen zur digitalen Signalverarbeitung, die sich insbesondere zur Bildanalyse und -manipulation eignen: Mit `fft()` ist die schnelle Fourier-Transformation und ihre Rücktransformation möglich, der Faltungsoperator ist in `convolve()` implementiert. Weit darüber hinausgehende Funktionen zur Bildbearbeitung bieten die Pakete `imager` und `magick`.

14.6 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 können (Abschn. 4.3). Dies kann entweder anhand summarischer Kennwerte oder aber durch Darstellung von Einzelwerten, ggf. in vergrößerter Form, geschehen.

14.6.1 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 ist (Abb. 14.18).¹⁴

```
hist(x=<Vektor>, breaks=<Grenzen>, freq=NULL)
```

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 und mit einem Vektor deren genaue Lage vorgegeben werden kann. In der Voreinstellung wird beides nach einem in der Hilfe beschriebenen Algorithmus entsprechend den Daten in `x` gewählt.¹⁵ Bei gleichabständigen Klassengrenzen werden in der Voreinstellung `freq=NULL` absolute Häufigkeiten angezeigt. Mit `freq=FALSE` sind es stattdessen die Dichten, also die relativen Häufigkeiten geteilt durch die Klassenbreite. `TRUE` erzwingt absolute Häufigkeiten auch bei ungleichen Klassenbreiten. 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, mean=175, sd=7)      # Körpergröße 100 Personen
> hist(height, xlab="height [cm]", ylab="N")
```

Für die individuelle Wahl der Klassengrenzen empfiehlt es sich, zunächst die Spannweite 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 <- round(range(height), -1) + c(-10, 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 & Normalverteilung")
```

Als weitere Information lässt sich im Anschluss an den Aufruf von `hist()` mit `rug()` auch die Lage der Einzelwerte mit darstellen. Dies geschieht in Form senkrechter Striche entlang der Abszisse, wobei jeder Strich für einen Wert steht. Um Bindungen in Form separater Striche in ihre Einzelwerte aufzulösen, sollte die Funktion mit `jitter()` gekoppelt werden. Diese Funktion verändert die Werte um einen kleinen zufälligen Betrag, wodurch sich die Lage der Striche horizontal leicht verschiebt. Dies führt zu dickeren Strichen als Repräsentation der Bindungen.

```
> rug(jitter(height))
```

Soll über das Histogramm zum Vergleich eine theoretisch vermutete Dichtefunktion gelegt werden, kann diese etwa mit `curve(..., add=TRUE)` hinzugefügt werden. Für eine passende Skalierung muss das Histogramm mit `freq=FALSE` die Dichte darstellen.

¹⁴PlotFdist() aus dem Paket DescTools stellt das Histogramm gemeinsam mit der kumulierten empirischen Häufigkeitsverteilung sowie einem Boxplot in einem Diagramm dar. Für den Vergleich der Verteilungen einer Variable in zwei Bedingungen zeigt histbackback() aus dem Hmisc Paket die zugehörigen Histogramme Rücken-an-Rücken angeordnet simultan in einem Diagramm.

¹⁵Wird die Anzahl der Klassengrenzen genannt, behandelt R diesen Wert nur als Vorschlag, nicht als zwingend. Der in der Voreinstellung verwendete Algorithmus erzeugt häufig nur wenige Intervalle. Durch `breaks="FD"` wird ein anderer Algorithmus verwendet, der meist zu einer etwas größeren Anzahl führt.

```
# füge Dichtefunktion einer Normalverteilung hinzu
> curve(dnorm(x, mean(height), sd(height)), lwd=2, col="blue", add=TRUE)
```

Die Wahl der Intervallgrenzen hat einen starken Einfluss auf die Form von Histogrammen. Aufgrund dieser Abhängigkeit von einem willkürlich festzulegenden Parameter eignen sich Histogramme nicht immer gut, um sich einen Eindruck von der empirischen Verteilung einer Variable zu verschaffen.

Als Alternative lässt sich mit `density(<Vektor>)` auf Basis einer im Vektor gespeicherten Stichprobe von Werten die Dichtefunktion der zugehörigen Variable schätzen und grafisch darstellen (Abschn. 16.1.4).¹⁶ Über `plot(density(<Vektor>))` wird die Schätzung der Dichtefunktion grafisch in einem separaten Diagramm veranschaulicht, während `lines(density(<Vektor>))` die geschätzte Dichtefunktion einem bereits geöffneten Diagramm hinzufügt. Bei einem Histogramm ist dabei darauf zu achten, für die Darstellung der Dichte das Argument `freq=FALSE` zu setzen.

```
> hist(height, freq=FALSE, xlim=fromTo, xlab="height [cm]",
+       main="Histogramm & Dichte-Schätzung")
> lines(density(height, bw="SJ"), lwd=2, col="blue")
> rug(jitter(height))
```

14.6.2 Stamm-Blatt-Diagramm

Das Stamm-Blatt-Diagramm mischt die Darstellung von Häufigkeiten einzelner Werte mit einem Histogramm. Seine Ausgabe erfolgt nicht in einem device, sondern in Textform auf der Konsole. Der Wertebereich wird dafür zunächst wie bei einem Histogramm in disjunkte Intervalle eingeteilt, die dieselbe Breite besitzen. Diese Intervalle bilden den *Stamm* und werden durch die Ziffernfolge repräsentiert, mit der alle Werte im Intervall beginnen. Geht das erste Intervall etwa von 120 bis (ausschließlich) 130, ist der Wert des Stamms 12. Die *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 > 10 fest, wie viele Blätter maximal gezeigt werden, wobei diese Anzahl `width-10` beträgt. Bei Werten für `width` ≤ 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.

¹⁶Zweidimensionale Dichten werden von `smoothScatter()` geschätzt und dargestellt.

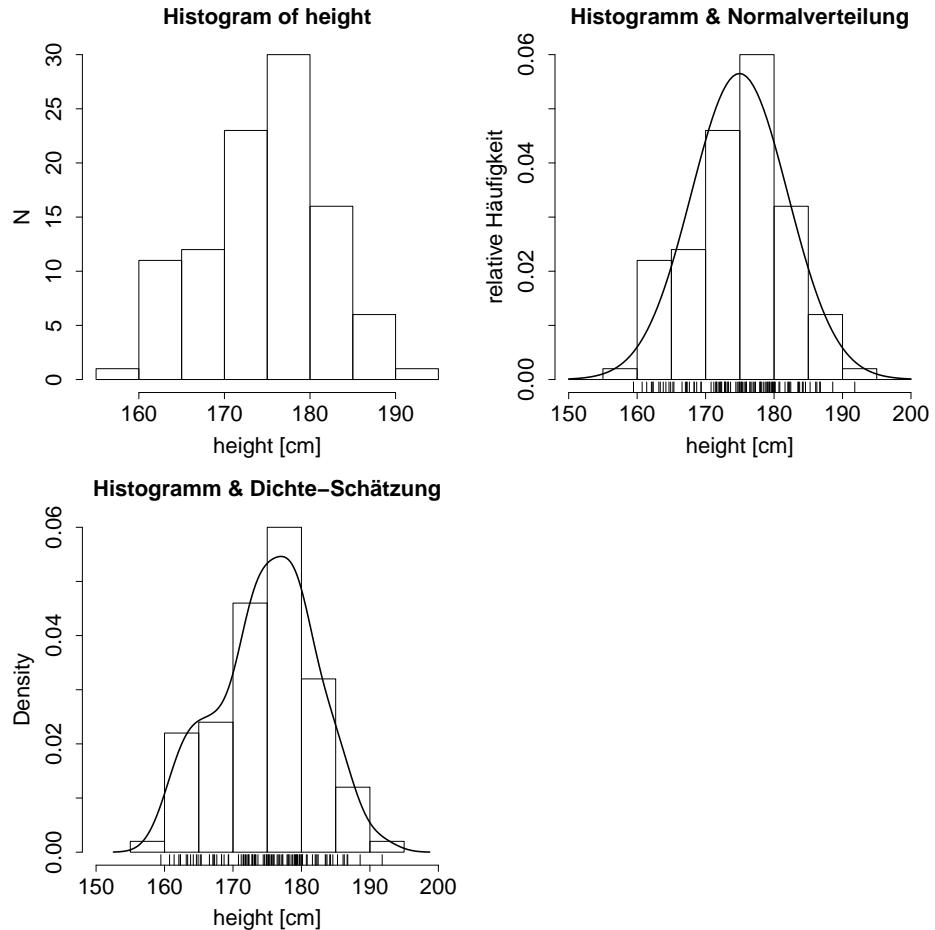


Abbildung 14.18: Histogramm, zusätzlich mit Einzelwerten und Normalverteilung bzw. mit Schätzung der Dichtefunktion

```
> stem(rnorm(100, mean=175, sd=7))
The decimal point is 1 digit(s) to the right of the |
```

```
15 | 8
16 | 12344
16 | 5556678888999999
17 | 001111222222223334444
17 | 555555556666677777788889999999
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.

14.6.3 Boxplot

Ein Boxplot (*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.¹⁷ Innerhalb des so gebildeten Rechtecks liegen damit die mittleren 50% der Werte, seine Höhe ist gleich dem Interquartilabstand. 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 u. a. 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. 14.19).

```
boxplot(x=<Vektor>, notch=FALSE, horizontal=FALSE)
```

Bei einem einzelnen Boxplot wird unter `x` der Datenvektor eingegeben. Stattdessen kann für `x` auch eine Modellformel der Form `<Messwerte> ~ <Faktor>` übergeben werden, wobei `<Faktor>` dieselbe Länge wie der Vektor `<Messwerte>` besitzt und für jeden von dessen Werten codiert, zu welcher Bedingung er gehört. Stammen die in der Modellformel verwendeten Variablen aus einem Datensatz, ist dieser unter `data` zu nennen. Hiermit werden getrennt für die von `<Faktor>` definierten n Gruppen Boxplots nebeneinander in einem Diagramm dargestellt, wobei als x -Koordinaten die Zahlen $1, \dots, n$ verwendet werden. Dasselbe Ergebnis lässt sich erzielen, indem an `x` eine Matrix übergeben wird, deren Spalten die Gruppen definieren, für die jeweils ein Boxplot darzustellen ist. Das Argument `notch` bestimmt, ob ein gekerbter Boxplot 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.

```
> Nj <- 40                                     # Gruppengröße
> P   <- 3                                      # Anzahl Gruppen
> DV  <- rnorm(P*Nj, mean=100, sd=15)          # Messwerte

# Faktor Gruppenzugehörigkeit
> IV <- gl(P, Nj, labels=c("Control", "Group A", "Group B"))
> Mj <- tapply(DV, IV, FUN=mean)                # Gruppenmittel
> boxplot(DV ~ IV, ylab="Score", col=c("red", "blue", "green"),
+           main="Boxplots und Scores in 3 Gruppen")

> points(1:P, Mj, pch=16, cex=2)                 # zeige Gruppenmittel
```

¹⁷Liegen geradzahlig viele Werte vor, wird das Rechteck nach oben und unten nicht exakt durch die Quartile begrenzt, vgl. `?boxplot.stats` sowie Abschn. 2.7.5, Fußnote 23.

Da Boxplots wichtige Eigenschaften der Verteilung wie etwa Multimodalität nicht repräsentieren können, ist es oft sinnvoll, zusätzlich die Rohdaten darzustellen. `beeswarm()` aus dem Paket `beeswarm` (Eklund, 2016) zeigt alle einzelnen Datenpunkte, wobei die horizontale Position so versetzt gewählt wird, dass die Verteilungsform gut ersichtlich ist (Abb. 14.19). Der Aufruf erfolgt analog zu `boxplot()`, wobei die Rohdaten mit der Option `add=TRUE` einem schon bestehenden Diagramm hinzugefügt werden.

Damit Ausreißer nicht doppelt dargestellt werden, sollte man sie im Aufruf von `boxplot()` mit der Option `outline=FALSE` ausblenden. Der Darstellungsbereich der y -Achse muss dann gleichzeitig so gewählt werden, dass `beeswarm()` anschließend die Ausreißer noch sichtbar darstellen kann.

```
> library(beeswarm) # für beeswarm()
> DVrange <- round(range(DV), digits=-1) # Darstellungsbereich y-Achse
> boxplot(DV ~ IV, ylab="Score", col=c("red", "blue", "green"),
+           main="Boxplots und Scores in 3 Gruppen",
+           outline=FALSE, ylim=DVrange)

> beeswarm(DV ~ IV, add=TRUE, pch=16, col="#00000077") # mit Transparenz
```

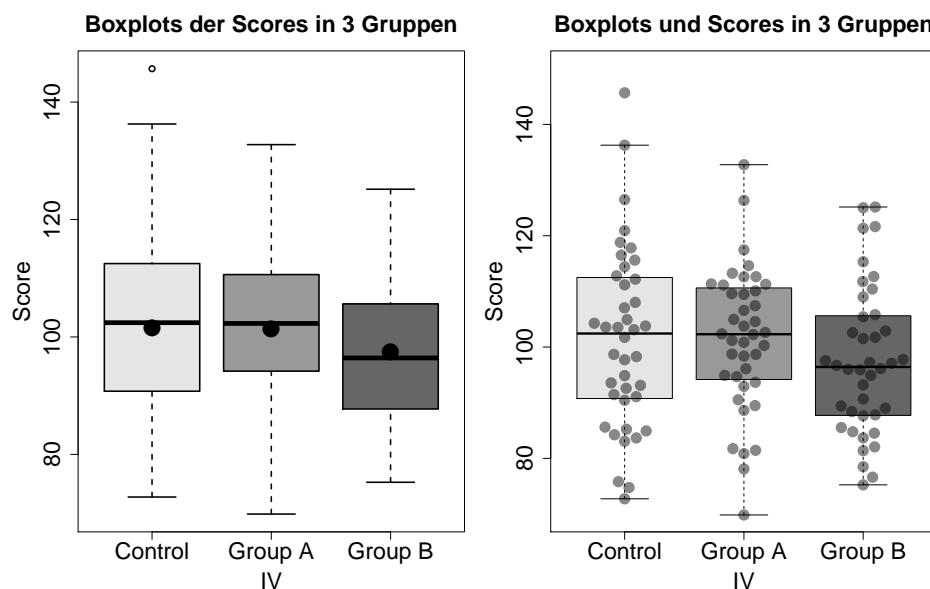


Abbildung 14.19: Boxplots getrennt nach Gruppen mit Mittelwerten sowie zusätzlich den Einzelwerten

14.6.4 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. 14.20).

```
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 Bindungen durch dasselbe Symbol repräsentiert werden. Die so erstellte Grafik 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.

Ein `stripchart` kann auch die Verteilung einer quantitativen Variable für mehrere Gruppen gleichzeitig veranschaulichen. Hierfür gibt es die Möglichkeit, mit `add=TRUE` das Ergebnis eines `stripchart()` Aufrufs einem schon bestehenden Diagramm hinzuzufügen – etwa einem Boxplot zur gleichzeitigen Veranschaulichung der Rohdaten und wichtiger Kennwerte (Abb. 1.2). In diesem Fall kontrolliert das Argument `at` die vertikale Position der Achse, auf der die Symbole einzulegen sind. Beim ersten Aufruf von `stripchart()` wird die Achse auf einer Höhe von 1 eingezeichnet.

Alternativ können die Daten auch als Modellformel $\langle \text{Messwerte} \rangle \sim \langle \text{Faktor} \rangle$ übergeben werden, wobei $\langle \text{Faktor} \rangle$ dieselbe Länge wie der Vektor $\langle \text{Messwerte} \rangle$ besitzt und für jeden von dessen Werten codiert, zu welcher Bedingung er gehört. Stammen die in der Modellformel verwendeten Variablen aus einem Datensatz, ist dieser unter `data` zu nennen. In diesem Fall wird für jede Stufe von $\langle \text{Faktor} \rangle$ jeweils ein `stripchart` der zugehörigen Daten im Diagramm eingezeichnet, wobei unterschiedliche Faktorstufen durch vertikal getrennte Achsen kenntlich gemacht werden (Abb. 14.20).

```
> Nj    <- 25                                # Würfe pro Gruppe
> P     <- 4                                  # Anzahl Gruppen
> dice <- sample(1:6, P*Nj, replace=TRUE)      # Würfelwürfe
> IV   <- gl(P, Nj)                          # Gruppenzugehörigkeit
> stripchart(dice ~ IV, main="Würfelwürfe - 4 Gruppen",
+             xlab="Augenzahl - jitter-Methode", ylab="Gruppe",
+             pch=1, col="blue", method="jitter")

> stripchart(dice ~ IV, main="Würfelwürfe - 4 Gruppen",
+             xlab="Augenzahl - stack-Methode", ylab="Gruppe",
+             pch=16, col="red", method="stack")
```

14.6.5 Quantil-Quantil-Diagramm

Viele statistische Auswertungen setzen voraus, dass die zu analysierenden Variablen auf Ebene der Population eine bestimmte Verteilung aufweisen, etwa normalverteilt sind oder der

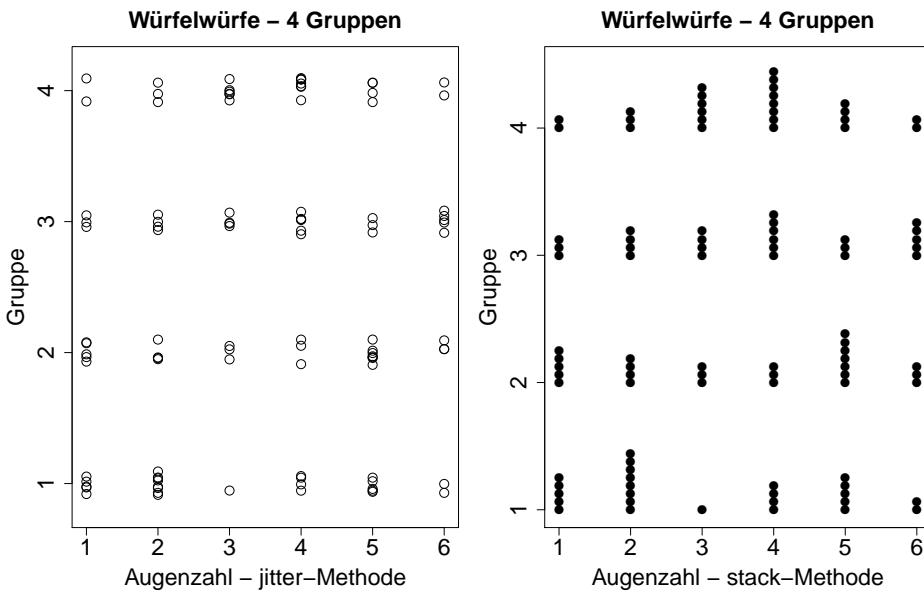


Abbildung 14.20: Stripcharts mit verschiedenen Methoden zur Darstellung einzelner Werte

Verteilung in einer anderen Population gleichen (Abschn. 10.1). Inwieweit die empirischen Werte mit dieser Annahme verträglich sind, kann mit einem Q-Q-Diagramm (Quantil-Quantil-Darstellung) heuristisch abgeschätzt werden.

Vergleich zweier Stichproben

Wenn die Verteilung eines Merkmals in zwei Gruppen identisch ist, stimmen auch die Quantile zu denselben Wahrscheinlichkeiten überein (Abschn. 5.3.3). Auf empirischer Ebene lassen sich dafür die beobachteten Quantile von zwei Variablen zu denselben Wahrscheinlichkeiten in einem Diagramm gegeneinander auftragen. Sind ihre Quantile identisch, fallen die Punkte bei gleicher Achsenkalierung auf die Winkelhalbierende. Unterscheiden sich die Verteilungen lediglich durch eine lineare Transformation, fallen die Punkte auf eine Gerade. Voneinander abweichende Verteilungen werden entsprechend durch Abweichungen von einer Referenzgeraden deutlich. Ein solches Diagramm erstellt `qqplot(x=<Vektor>, y=<Vektor>)` (Abb. 14.21). Für x und y sind dafür die zu vergleichenden Variablen einzutragen, die auch unterschiedliche Länge haben können.

```
> DV1 <- rnorm(200)          # simuliere normalverteilte Messwerte
> DV2 <- rf(100, df1=3, df2=15) # simuliere F-Verteilung, anderes N
> qqplot(DV1, DV2, xlab="Quantile N(0, 1)", ylab="Quantile F(3, 15)",
+         main="Quantile N(0, 1) vs. F(3, 15)-Verteilung", pch=19)
```

Vergleich mit theoretischer Verteilung

Die empirischen Quantile von Daten einer Stichprobe können auch mit Quantilen verglichen werden, die sich aus der Annahme einer bestimmten Verteilung ergeben. Hierfür werden die

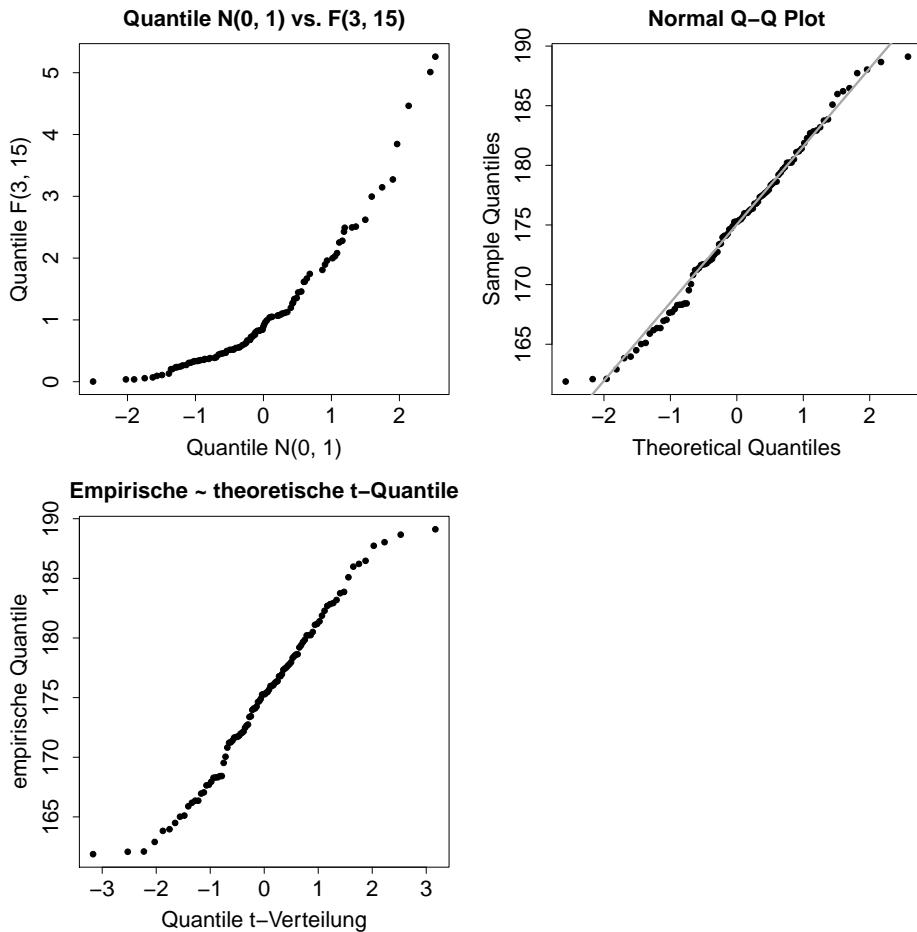


Abbildung 14.21: Quantil-Quantil Darstellung zum Vergleich von zwei Stichproben, zur Überprüfung von Normalverteiltheit und zum Vergleich mit t -Verteilung

erwarteten Quantile als x - und die tatsächlichen Quantile als y -Koordinaten von Punkten in einem Streudiagramm verwendet. Die empirischen Quantile sind einfach die sortierten Werte eines Vektors. Die Funktion `ppoints(<Vektor>)` bestimmt, welches der zu einem Quantil gehörende Wert der empirischen kumulierten Häufigkeitsverteilung als Schätzung der Verteilungsfunktion ist.¹⁸ Ihre Ausgabe kann als Argument für die Quantilfunktion einer infrage kommenden Verteilung verwendet werden, um die theoretischen Quantile zu erhalten (Abschn. 5.3.3). Quantile zu denselben Wahrscheinlichkeiten lassen sich dann mit `plot()` als Streudiagramm gegeneinander auftragen.

```
> cProb <- ppoints(height)      # kumulierte W'keit für t-Quantile
> qTheo <- qt(cProb, df=10)    # t-Quantile
> qEmp  <- sort(height)        # empirische Quantile

# Q-Q-Plot Vergleich empirische Quantile mit t-Quantilen
> plot(qTheo, qEmp, main="Empirische ~ theoretische t-Quantile",
```

¹⁸Dies sind die Mittelpunkte der Intervalle, die durch die empirischen kumulierten relativen Häufigkeiten gebildet werden (Abschn. 2.12.6).

```
+     xlab="Quantile t-Verteilung", ylab="empirische Quantile", pch=19)
```

Für den häufigen Spezialfall eines Vergleichs mit der Standardnormalverteilung existiert `qqnorm()`. Eine Referenzgerade wird durch `qqline()` in das Diagramm eingetragen (Abb. 14.21).

```
qqnorm(y=<Vektor>, datax=FALSE)
qqline(y=<Vektor>, datax=FALSE)
```

Mit `datax=FALSE` wird in der Voreinstellung festgelegt, dass die Quantile der empirischen Werte auf der y -Achse abgetragen werden. `datax` ist beim Aufruf von `qqnorm()` und `qqline()` auf denselben Wert zu setzen. Im Vergleich zu `qqplot()` entfällt hier die Angabe des ersten Datenvektors, da die x -Koordinaten der Punkte von den theoretisch erwarteten Quantilen gebildet werden.

```
> height <- rnorm(100, mean=175, sd=7)
> qqnorm(height, pch=19)
> qqline(height, col="red", lwd=2)
```

14.6.6 Empirische kumulierte Häufigkeitsverteilung

Die kumulierte Häufigkeitsverteilung empirischer Daten lässt sich durch die `ecdf()` Funktion ermitteln, die ihrerseits eine Funktion erzeugt (Abschn. 2.12.6). Zur grafischen Darstellung wird diese neue Funktion an `plot()` übergeben (Abb. 14.22).

```
> vec <- round(rnorm(10), 1)
> Fn  <- ecdf(vec)
> plot(Fn, main="Empirische kumulierte Häufigkeitsverteilung")
> curve(pnorm, add=TRUE, col="gray", lwd=2) # Vergleich mit Standard-NV
```

14.6.7 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 den Anteil des zugehörigen Wertes an der Summe aller Werte. 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. 14.23).

```
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, kann ein Vektor aus entsprechenden Zeichenketten für das Argument `labels` übergeben werden. Das Argument `col` kontrolliert die Farbe der Sektoren.

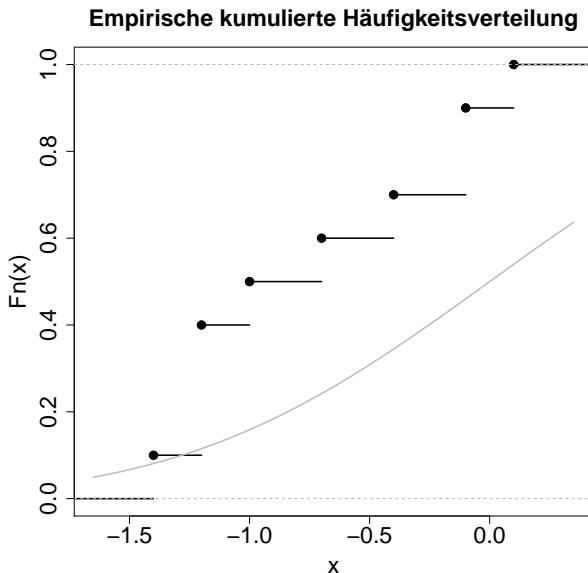


Abbildung 14.22: Vergleich von empirischen kumulierten Häufigkeiten mit der Verteilungsfunktion der Standardnormalverteilung

```
> 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 hier `text()` verwendet. Die zur Plazierung notwendigen (x, y) -Koordinaten ergeben sich aus den relativen Häufigkeiten der Würfelergebnisse. Dabei ist dafür zu sorgen, dass die Beschriftung 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 <- proportions(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, labels=dTabFreq) # Sektorbeschriftungen
```

14.6.8 Gemeinsame Verteilung zweier Variablen

Die in Abschn. 14.2 vorgestellte `plot()` Funktion eignet sich dafür, die gemeinsame Verteilung von zwei Variablen in Form eines Streudiagramms zu untersuchen. Stammen die Daten aus verschiedenen Gruppen, kann die Gruppenzugehörigkeit über die Farbe oder den Typ

Relative Häufigkeiten beim Würfeln

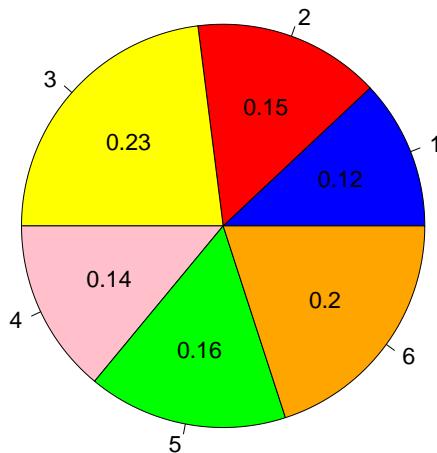


Abbildung 14.23: Kreisdiagramm als Mittel zur Darstellung von Kategorienhäufigkeiten

der Datenpunktsymbole gekennzeichnet werden. Hierfür lässt sich die Eigenschaft von Faktoren ausnutzen, dass ihre Stufen intern über natürliche Zahlen repräsentiert sind, die sich mit `unclass()` ausgeben lassen und damit als Indizes dienen können (Abb. 14.24).

```
> N <- 200 # Anzahl Personen
> P <- 2 # Anzahl Gruppen
> x <- rnorm(N, 100, 15) # Daten Variable 1
> y <- 0.5*x + rnorm(N, 0, 10) # Daten Variable 2
> IV <- gl(P, N/P, labels=LETTERS[1:P]) # Gruppierungsfaktor

# gemeinsame Verteilung mit 2 Datenpunktsymbolen und Legende
> plot(x, y, pch=c(4, 16)[unclass(IV)], lwd=2,
+       col=c("black", "darkgray")[unclass(IV)],
+       main="Gemeinsame Verteilung je Gruppe")

> legend(x="topleft", legend=c("Gruppe A", "Gruppe B"),
+          pch=c(4, 16), col=c("black", "darkgray"))
```

Weiterhin kann es hilfreich sein, mit `ellipse()` aus dem Paket `car` auch die Streuungsellipse der gemeinsamen Verteilung einzuzeichnen, wie sie durch die Eigenwerte und Eigenvektoren der Kovarianzmatrix der Variablen definiert wird (Abschn. 12.1.5, 12.2).¹⁹

```
ellipse(<Zentroid>, shape=<Kovarianzmatrix>, radius=<Radius>)
```

Die Ellipse hat ihren Mittelpunkt im als ersten Argument zu übergebendem Zentroid der Daten. Die Hauptachsen sind durch die Eigenvektoren der für `shape` zu nennenden Matrix gegeben, wobei die Länge jeder Halbachse gleich der Wurzel aus dem zugehörigen Eigenwert ist (Abb. 14.24). Mit `radius` kann diese Länge um einen beliebigen Faktor skaliert werden.

¹⁹Für Konfidenzellipsen im inferenzstatistischen Sinn s. `confidenceEllipse()` aus demselben Paket.

```
> library(car) # für ellipse()
> mat <- cbind(x, y) # Datenmatrix
> ctr <- colMeans(mat) # Zentroid
> plot(mat, xlab="x", ylab="y", asp=1, # gemeinsame Verteilung
+       main="Gemeinsame Verteilung 2 Variablen")

> ellipse(ctr, shape=cov(mat), radius=1, col="blue") # Streuungsellipse
```

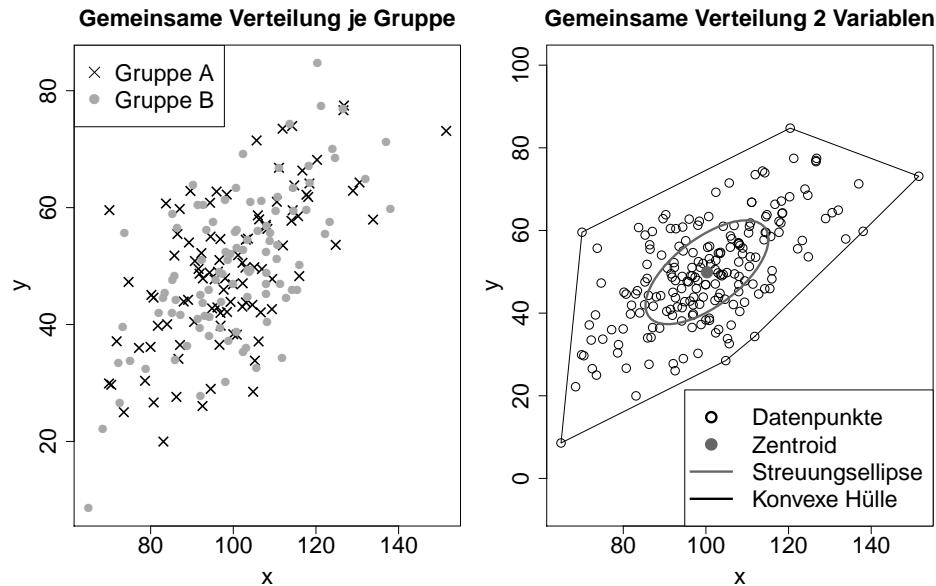


Abbildung 14.24: Gemeinsame Verteilung zweier Variablen getrennt nach Gruppen. Gesamtdaten mit Streuungsellipse, deren Hauptachsen und der konvexen Hülle

Die konvexe Hülle der gemeinsamen Verteilung zweier Variablen ist definiert als kleinstes konkaves 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. 14.24).

`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 im Uhrzeigersinn die Indizes der Ecken des Polygons.

```
> hullIdx <- chull(x, y) # Indizes Ecken konvexe Hülle
> polygon(x[hullIdx], y[hullIdx]) # konvexe Hülle einzeichnen
> legend(x="bottomright", legend=c("Datenpunkte", "Zentroid",
+ "Streuungsellipse", "konvexe Hülle"),
+ pch=c(1, 19, NA, NA), lty=c(NA, NA, 1, 1),
+ col=c("black", "blue", "blue", "black"))
```

Sind nur wenige Wertepaare möglich und deswegen Bindungen in den Daten vorhanden, 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. 14.25).

```
> vec1 <- sample(1:10, 100, replace=TRUE)
> vec2 <- sample(1:10, 100, replace=TRUE)
> plot(vec2 ~ vec1, main="Punktfolge")      # Datenpunkte ohne jitter()

# Datenpunkte: zufälliger Versatz entlang y-Achse durch jitter()
> plot(jitter(vec2) ~ vec1, main="Punktfolge mit jitter")
```

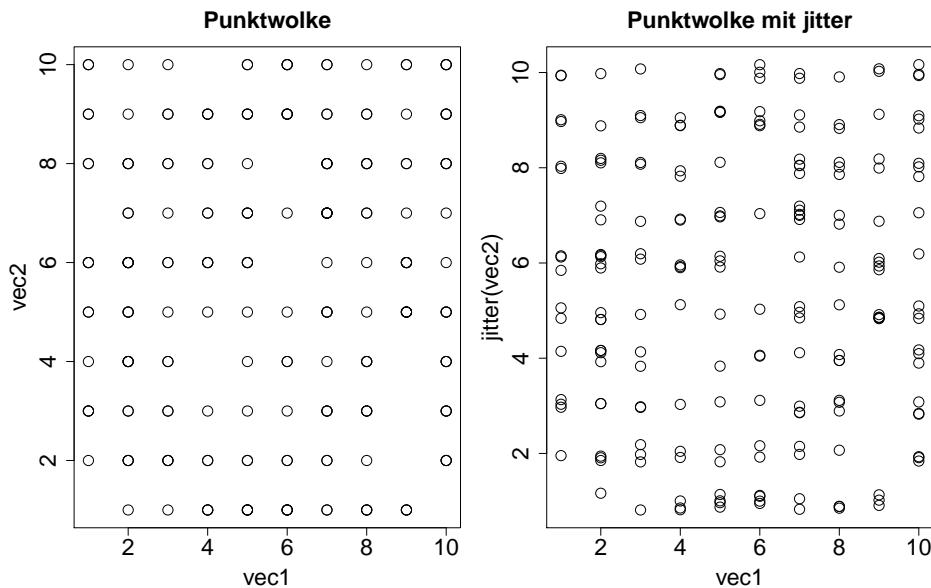


Abbildung 14.25: Streudiagramm von Daten mit Bindungen ohne und mit Anwendung von `jitter()`

Bei sehr vielen darzustellenden Punkten taucht mitunter das Problem auf, dass Datenpunkt-symbole einander überdecken und einzelne Werte 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 (Abschn. 14.3.2, Fußnote 7).

Alternativ lässt sich mit `hexbin()` aus dem gleichnamigen Paket (Carr, Lewin-Koh & Maechler, 2020) ein Diagramm erstellen, das die Diagrammfläche in hexagonale Regionen einteilt und die Dichte der Datenpunkte in jeder Region ähnlich einem Höhenlinien-Diagramm (Abschn. 14.7.1) farblich repräsentiert. Auch `smoothScatter()` codiert die Dichte von Datenpunkten über farblich abgesetzte Diagrammregionen, verwendet dabei jedoch einen 2D-Kerndichteschätzer zur Glättung der Regionengrenzen (Abschn. 16.1.4).

14.7 Multivariate Daten visualisieren

In unterschiedlichen Situationen kann es erstrebenswert sein, multivariate Daten grafisch darzustellen: So können Messwerte in ihrer Ausprägung von mehr als einer numerischen Variable abhängen, etwa im Kontext einer multiplen linearen Regression. Oder aber mehrere Variablen werden gleichzeitig erhoben, um ihre gemeinsame Verteilung zu analysieren. Schließlich hängt etwa im Rahmen mehrfaktorieller Varianzanalysen eine Variable von der Kombination verschiedener qualitativer Faktoren ab. Die Visualisierung solcher Daten muss dann alle beteiligten Komponenten berücksichtigen.

Für dreidimensionale Daten existieren 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 grafische 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 Grafik verzichtet werden, in der eine Serie uni- oder bivariater Diagramme nebeneinander für alle Stufen bzw. Stufenkombinationen weiterer Variablen angeordnet ist.

14.7.1 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, y) -Koordinaten enthält und deswegen so viele Zeilen wie x - und so viele Spalten wie y -Koordinaten besitzt. Die Werte dieser Matrix definieren die z -Koordinate als dritte Komponente für jedes (x, y) -Koordinatenpaar.

Höhenlinien symbolisieren die z -Koordinate wie topografische 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 als Matrix. 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 gemeinsam normalverteilten Variablen mit positiver Korrelation dargestellt werden (Abb. 14.26). Um die Dichte zu berechnen, wird zunächst eine eigene Funktion definiert (Abschn. 17.3), die `dmvnorm()` aus dem `mvtnorm` Paket für ein (x, y) -Koordinatenpaar aufruft. Mit `outer()` lässt sich die Funktion dann auf alle Paare von (x, y) -Koordinaten anwenden.

```
> mu      <- c(1, 3)           # Erwartungswerte Normalverteilung
> sigma   <- matrix(c(1, 0.6, 0.6, 1), nrow=2)       # Kovarianzmatrix
> rng     <- 2.5             # Wertebereich in Std.-Abw.
> N       <- 50              # Anzahl der x- und der y-Koordinaten

# x- und y-Koordinaten
> X <- seq(from=mu[1] - rng*sigma[1, 1],
+           to=mu[1] + rng*sigma[1, 1], length.out=N)

> Y <- seq(from=mu[2] - rng*sigma[2, 2],
+           to=mu[2] + rng*sigma[2, 2], length.out=N)

# Funktion, um Dichte z für gegebenes (x, y)-Paar zu erzeugen
> library(mvtnorm)          # für dmvnorm()
> genZ <- function(x, y) { dmvnorm(cbind(x, y), mu, sigma) }

# Dichte z für alle (x, y)-Koordinatenpaare berechnen
> matZ <- outer(X, Y, "genZ")
> contour(X, Y, matZ, main="Höhenlinien für 2D-NV Dichte")
```

Die durch die Höhenlinien definierten Regionen können mit `filled.contour()` auch eingefärbt werden. Dabei stammen die Farben aus einem Farbverlauf, dessen Zuordnung zu Kategorien auf der rechten Seite des Diagramms in einer Legende erläutert wird (Abb. 14.26). Der Aufruf gleicht dem für `contour()` stark, jedoch kann über das zusätzliche Argument `color.palette` eine andere Farbpalette spezifiziert werden (Abschn. 14.3.2).

```
> filled.contour(X, Y, matZ, main="Farbige Höhenlinien")
```

Mit `symbols()` lassen sich zwei – typischerweise quantitative – Variablen durch die räumliche Lage von Datenpunktssymbolen gemeinsam mit weiteren Variablen durch die Gestaltung dieser Symbole grafisch repräsentieren (Abb. 14.26).²⁰

```
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 Modellformel der Form $\langle y \rangle \sim \langle x \rangle$ möglich. Stammen diese Variablen aus einem Datensatz, ist er unter `data` zu nennen.

Welche Datenpunktssymbole an den (x, y) -Koordinaten erscheinen, richtet sich danach, für welches weitere Argument Daten übergeben werden: `circles` erwartet einen Vektor derselben

²⁰Ähnliche Abbildungen erzeugen `stars()` und `sunflowerplot()`.

Länge wie `x` und `y` mit positiven Werten für die Größe der als Datenpunktsymbol 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` jeweils 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` fest, die der von ihnen umschlossenen Flächen `bg`.

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 gemessene Variable simulieren.

```
> N      <- 10
> age    <- rnorm(N, 30, 8)
> sport   <- abs(-0.25*age + rnorm(N, 60, 30))
> weight <- -0.3*age -0.4*sport + 100 + rnorm(N, 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
> symbols(age, sport, circles=wScale, inch=0.6, fg=NULL, bg=rainbow(N),
+           main="Gewicht vs. Alter und Sport")
```

14.7.2 Dreidimensionale Gitter und Streudiagramme

`persp()` erzeugt Diagramme mit Tiefeneindruck, in der die Datenpunkte zu einer Gitterfläche verbunden sind. Der Augpunkt, d. h. die Perspektive, aus der die simulierte dreidimensionale Szene gezeigt wird, ist frei wählbar (Abb. 14.27).

```
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 (Längengrad) und `phi` die Höhe über dem Äquator (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(X, Y, matZ, xlab="x", ylab="y", zlab="Dichte", theta=5,
+        phi=35, main="Dichte einer 2D Normalverteilung")
```

Mit Funktionen aus dem Paket `rgl` (Adler & Murdoch, 2020) erstellte Diagramme – etwa `plot3d()`, `hist3d()` oder `persp3d()` als Pendants zu den konventionellen Funktionen `plot()`,

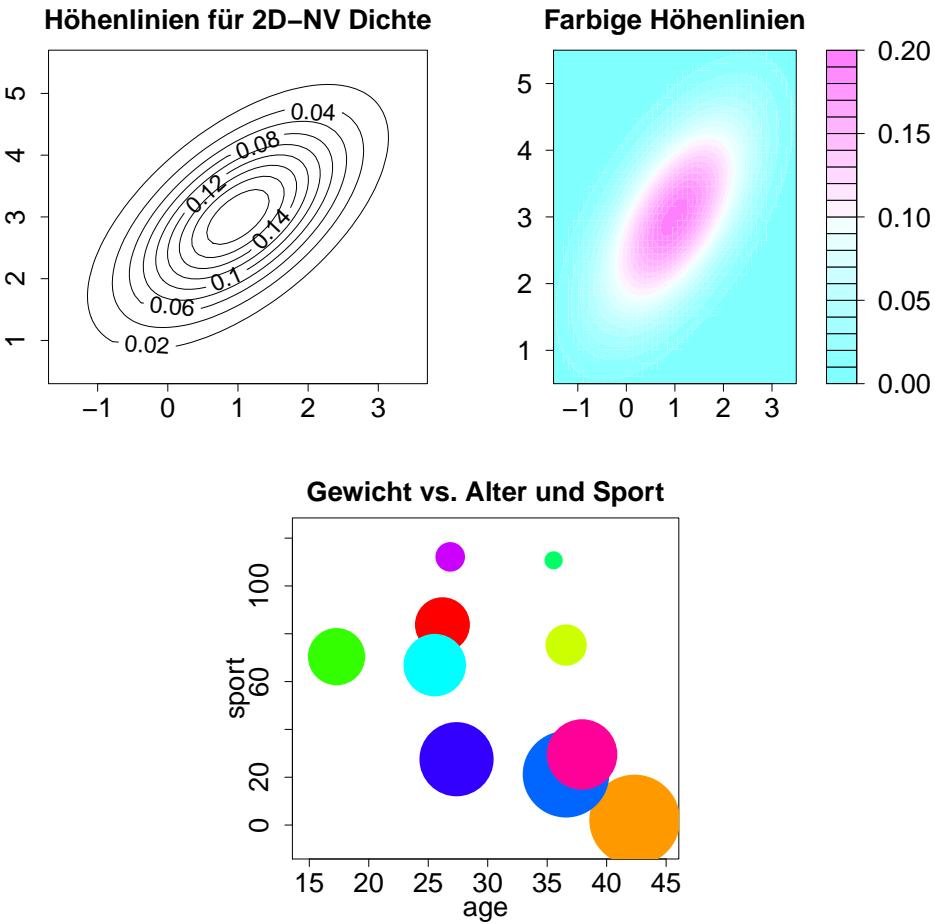


Abbildung 14.26: Visualisierungsmöglichkeiten für dreidimensionale Daten: Höhenlinien und variable Datenpunktsymbole

`hist()` und `persp()`, erlauben durch den Einsatz der Grafikbibliothek OpenGL eine interaktive Manipulation:²¹ Durch Anklicken der Diagrammfläche lässt sich die Perspektive auf das Diagramm beliebig ändern, indem die linke Maustaste gedrückt gehalten und die Maus bewegt wird (Abb. 14.27).

```
plot3d(x=<x-Koordinaten>, y=<x-Koordinaten>, z=<x-Koordinaten>)
```

Im Unterschied zu `contour()` und `persp()` müssen die (x, y, z) -Koordinaten der Punkte hier in Form dreier Vektoren gleicher Länge an die Argumente `x`, `y` und `z` übergeben werden. Das `rgl` Paket bringt eigene Funktionen zum Einfügen aller in Abschn. 14.5 für zweidimensionale Diagramme beschriebenen Grafikelemente mit, vgl. `help(package="rgl")`.

```
> vecX <- rep(seq(-10, 10, length.out=10), times=10)
> vecY <- rep(seq(-10, 10, length.out=10), each=10)
```

²¹Hinweise auf weitere relevante Zusatzpakete liefert der Abschnitt *Dynamic Visualizations and Interactive Graphics* der CRAN Task Views (Zhang & Cook, 2024). Sehr dynamisch ist derzeit die Entwicklung von Paketen, die interaktive Diagramme für Webseiten mit Hilfe von JavaScript-Bibliotheken implementieren, etwa `plotly` (Sievert et al., 2020; Sievert, 2020).

```
> vecZ <- vecX*vecY
> library(rgl)                                     # für plot3d()
> plot3d(vecX, vecY, vecZ, main="3D Scatterplot",
+         col="blue", type="h", aspect=TRUE)

> spheres3d(vecX, vecY, vecZ, col="red", radius=2)
> grid3d(c("x", "y+", "z"))
```

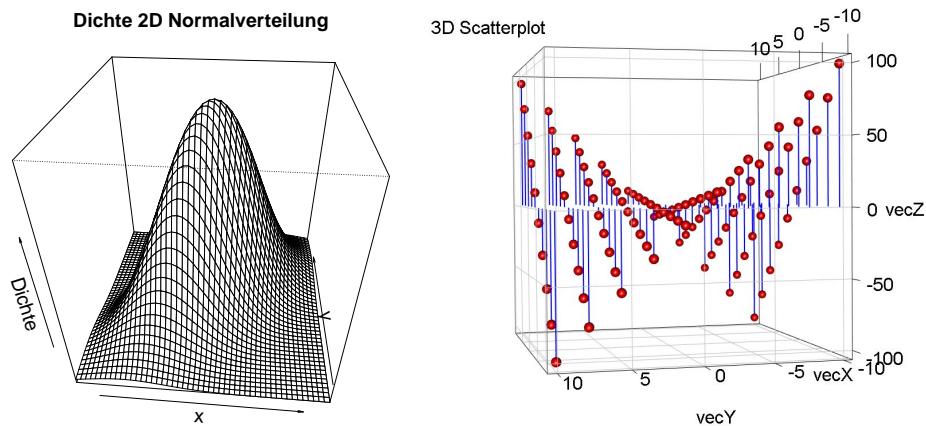


Abbildung 14.27: Visualisierungsmöglichkeiten für dreidimensionale Daten: Gitter und interaktives Streudiagramm

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. Nach Laden des Pakets mit `library(rgl)` erhält man durch `demo(rgl)` sowie insbesondere `example(persp3d)` einen Überblick über die zahlreichen Gestaltungsmöglichkeiten.

Photorealistisch gerenderte Darstellungen dreidimensionaler Karten oder Daten erzeugt das Paket `rayshader` (Morgan-Wall, 2020). Das Paket `rayrender` (Morgan-Wall, 2024) erweitert diese Möglichkeiten auf ganze Szenen.²²

14.7.3 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 (Panels (i, i)) werden in der Voreinstellung die Variablennamen ausgegeben. An der Hauptdiagonale gespiegelte Panels $((i, j)$ und (j, i)) stellen dieselbe gemeinsame Verteilung mit vertauschten Achsen dar (Abb. 14.28).

²²Beispiele zeigt <https://github.com/tylermorganwall/rayshader/> sowie <https://github.com/tylermorganwall/rayrender/>.

```
pairs(x=<Formel>, data=<Datensatz>,
      panel=<Zeichenfunktion>,
      lower.panel=<Zeichenfunktion>,
      upper.panel=<Zeichenfunktion>,
      diag.panel=<Zeichenfunktion>)
```

Für `x` ist die rechte Seite einer Modellformel $\sim \langle \text{Var1} \rangle + \langle \text{Var2} \rangle + \dots$ mit Variablen anzugeben, die aus einem Datensatz `data` stammen, und deren paarweise gemeinsame Verteilungen dargestellt werden sollen. `panel` erwartet den Namen einer Zeichenfunktion, deren Ergebnis in den Panels außerhalb der Diagonale der erzeugten Grafik-Matrix abgebildet wird. Die Funktion muss als Argumente zwei Vektoren (Variablen aus `data`) verarbeiten können und darf kein neues Diagramm öffnen, muss also eine Low-Level-Funktion sein. Voreinstellung ist `points` für ein Streudiagramm. Sollen die Panels oberhalb und unterhalb der Hauptdiagonale unterschiedliche Diagrammtypen zeigen, können für `lower.panel` und `upper.panel` separate Zeichenfunktionen angegeben werden. Durch Angabe einer Zeichenfunktion für `diag.panel`, die als Argument die Daten der Variable i erhält, lassen sich auch auf der Diagonale Diagramme einfügen (Voreinstellung ist `NULL` für die Ausgabe der Variablennamen).

Im Beispiel seien Personen in zwei Gruppen aufgeteilt und an ihnen vier Variablen erhoben worden. Die Streudiagramme sollen die Gruppenzugehörigkeit über Art und Farbe der Datenpunktssymbole kenntlich machen (Abschn. 14.6.8).

```
> N <- 20 # Gesamt-N
> P <- 2 # Anzahl Gruppen
> IV <- factor(rep(c("CG", "T"), each=N/P)) # Gruppenzugehörigkeit

# abhängige Variablen
> age <- sample(18:35, N, replace=TRUE)
> IQ <- round(rnorm(N, rep(c(100, 115), each=N/P), 15))
> rating <- round(0.4*IQ - 30 + rnorm(N, 0, 10), 1)
> score <- round(-0.3*IQ + 0.7*age + rnorm(N, 0, 8), 1)
> mvDf <- data.frame(IV, age, IQ, rating, score) # Datensatz

# Matrix aus Streudiagrammen darstellen
> pairs(~ age + IQ + rating + score, main="Streudiagramm-Matrix",
+       data=mvDf,
+       pch=c(4, 16)[unclass(IV)], col=c("red", "blue")[unclass(IV)])
```

Da sich nur Low-Level Zeichenfunktionen für die `panel` Argumente eignen, ist häufig der Umweg über selbst erstellte Funktionen notwendig, um bestimmte Grafiken in der Matrix darzustellen (Abschn. 17.3): Für High-Level-Funktionen muss dafür zunächst `par(new=TRUE)` aufgerufen werden, damit sie selbst kein neues device öffnen (Abschn. 14.8.2).

Hier soll dies zunächst für in der Hauptdiagonale darzustellende Histogramme geschehen. Zusätzlich sollen die für beide Gruppen getrennten Streuungsellipsen der gemeinsamen Verteilung in den Panels oberhalb der Hauptdiagonale gezeigt werden (Abb. 14.24, Abschn. 14.6.8).

```
# Funktion, um Histogramm darzustellen, ohne neue Grafik zu öffnen
> myHist <- function(x, ...) { par(new=TRUE); hist(x, ..., main="") }
```

```
# Funktion, um Streuungsellipsen getrennt für beide Gruppen zu zeichnen
> library(car) # für ellipse()
> myEll <- function(x, y, ...) { # ... für weitere Argumente aus pairs()
+   # trenne Beobachtungen in Variablen x und y nach Gruppen und
+   # verbinde Variablen getrennt nach Gruppen zu einer Datenmatrix
+   splLL <- split(data.frame(x, y), mvDf$IV)
+   matCG <- data.matrix(splLL$CG) # Daten Gruppe 1
+   matT <- data.matrix(splLL$T) # Daten Gruppe 2
+   ctrCG <- colMeans(matCG) # Zentroid Gruppe 1
+   ctrT <- colMeans(matT) # Zentroid Gruppe 2
+
+   # Ellipsen einzeichnen
+   ellipse(ctrCG, shape=cov(matCG), col="red", radius=1)
+   ellipse(ctrT, shape=cov(matT), col="blue", radius=1)
}

> pairs(~ age + IQ + rating + score, diag.panel=myHist,
+        upper.panel=myEll, main="Streudiagramm-Matrix", pch=16,
+        col=c("red", "blue") [unclass(IV)])
```

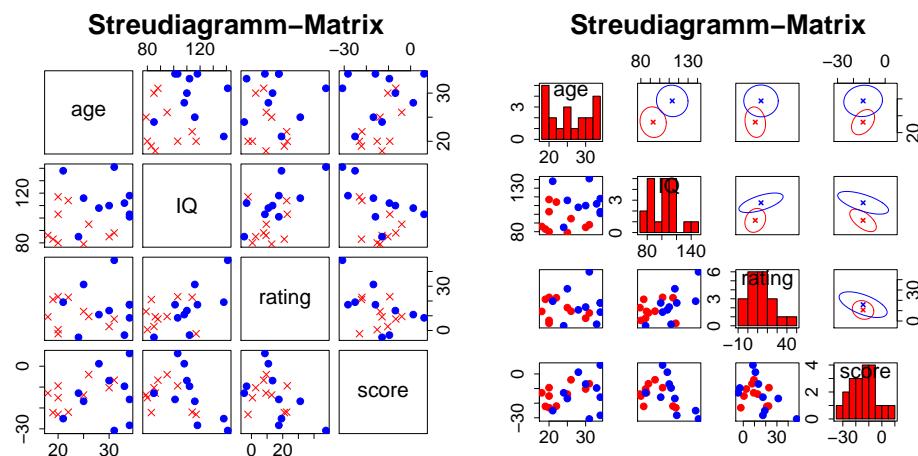


Abbildung 14.28: Matrizen von paarweisen Streudiagrammen mehrerer Variablen mit `pairs()`

14.7.4 Heatmap

Eine *heatmap* ist eine Falschfarben-Darstellung der Werte einer Matrix, wobei für jede Zelle eine Farbe entsprechend der Größe des Zellenwertes gewählt wird. Mit `heatmap()` können auf diese Weise etwa die Einträge der Korrelationsmatrix vieler Variablen so visualisiert werden, dass unmittelbar ersichtlich ist, welche Variablen einen starken linearen Zusammenhang aufweisen.²³

²³Ästhetisch ansprechendere heatmaps mit sehr viel mehr Gestaltungsmöglichkeiten bietet das Paket `pheatmap` ([Kolde, 2019](#)).

```
heatmap(<Matrix>, symm, revC, Rowv, Colv, recV, col,
       RowSideColors, ColSideColors)
```

Das erste Argument erwartet eine numerische Matrix. Handelt es sich um eine symmetrische Matrix (z. B. eine Korrelationsmatrix), kann dies mit `symm=TRUE` angezeigt werden. In diesem Fall sorgt `revC=TRUE` dafür, dass die Reihenfolge von Zeilen und Spalten übereinstimmt. `Rowv` und `Colv` kontrollieren, ob an den Grafikrändern ein *Dendrogramm* angezeigt werden soll, das die mit einer Clusteranalyse ermittelten hierarchischen Zusammenhänge von Variablen darstellt. Setzt man diese Argumente auf `NA`, wird kein Dendrogramm angezeigt. Die zu verwendenden Farben können an `col` übergeben werden, wobei soviele Farben wie Zellen vorhanden sein müssen. Um die Reihenfolge der Farben in einer Legende an den Seiten des Diagramms darzustellen, können sie an `RowSideColors` (soviele Farben wie Zeilen) oder `ColSideColors` (soviele Farben wie Spalten) übergeben werden.

Die Ausgabe entspricht der Struktur der übergebenen Matrix, wobei jede Zelle durch eine Farbe dargestellt wird (Abb. 14.29). Farben werden Werten so zugeordnet, dass die erste Farbe in `col` zum kleinsten Wert gehört und die letzte Farbe zum größten Wert in der Matrix.

```
> corMat <- cor(data.matrix(mvDf))           # Korrelationsmatrix
> round(corMat, digits=3)
      IV    age     IQ rating   score
IV    1.000 0.578  0.570  0.209  0.003
age   0.578 1.000  0.210  0.041  0.254
IQ    0.570 0.210  1.000  0.546 -0.435
rating 0.209 0.041  0.546  1.000 -0.458
score  0.003 0.254 -0.435 -0.458  1.000

> cmCol  <- rev(heat.colors(nrow(corMat))) # Basisfarben
> cmRamp <- colorRampPalette(cmCol)        # Funktion für Mischfarben
> cols    <- cmRamp(length(corMat))         # Mischfarben
> heatmap(corMat, symm=TRUE, revC=TRUE, Rowv=NA, Colv=NA, col=cols,
+          RowSideColors=cmCol, main="Heatmap einer Korrelationsmatrix")
```

`heatmap()` verfügt über viele weitere Darstellungsmöglichkeiten, die insbesondere im Rahmen einer Clusteranalyse sinnvoll sind und in `?heatmap` erläutert werden.

14.8 Mehrere Diagramme in einem Grafik-Device darstellen

Die in Abschn. 14.7.3 vorgestellten Funktionen verdeutlichen, dass auch mehr als ein Diagramm in ein device gezeichnet werden kann. Sollen nicht nur Diagramme gleichen Typs, sondern auch unterschiedliche Grafiken in einem device dargestellt werden, lässt sich dessen Fläche manuell in mehrere rechteckige Bereiche aufteilen. Diese können dann mit jeweils einem Diagramm gefüllt werden. Auf diese Weise simultan dargestellte Diagramme erleichtern den Vergleich von Ergebnissen in verschiedenen Teilstichproben, lassen sich aber auch nutzen, um dieselben Daten parallel auf verschiedene Arten zu visualisieren.

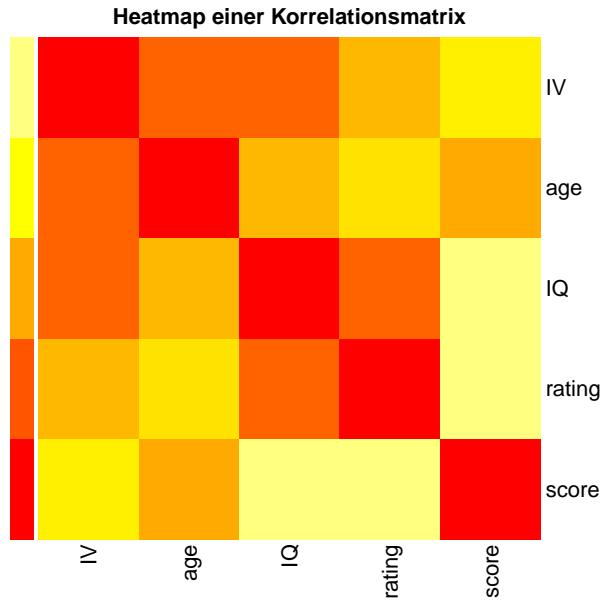


Abbildung 14.29: Heatmap einer Korrelationsmatrix

14.8.1 `layout()`

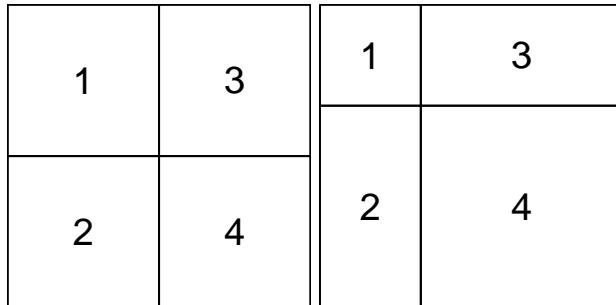


Abbildung 14.30: Mit `layout()` eingeteilte Regionen eines device

Um n Diagramme in einem device darzustellen, kann dessen Fläche mit `layout()` unter Zuhilfenahme einer Matrix `mat` aufgeteilt werden, 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 device zu überschreiben (Abb. 14.30, 14.31).

```
layout(mat=<Matrix>, widths=<Vektor>, heights=<Vektor>)
```

`mat` bestimmt die Einteilung der Device-Region in Planquadrate, die durch die Zellen von `mat` symbolisiert werden. 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 Planquadrate. 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. 14.32). 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 `1cm(<Zahl>)` in der Einheit cm eingegeben werden. Um die n Planquadrate durch Umrandung sichtbar zu machen, ist `layout.show(n)` zu verwenden.

```
> (mat1 <- matrix(1:4, 2, 2))    # 4 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

# vier Diagramme einfügen
> barplot(table(round(rnorm(100))), horiz=TRUE, main="Barplot")
> boxplot(rt(100, 5), main="Boxplot")
> stripchart(sample(1:20, 40, replace=TRUE), method="stack",
+             main="Stripchart")

> pie(table(sample(1:6, 20, replace=TRUE)), main="Kreisdiagramm")
```

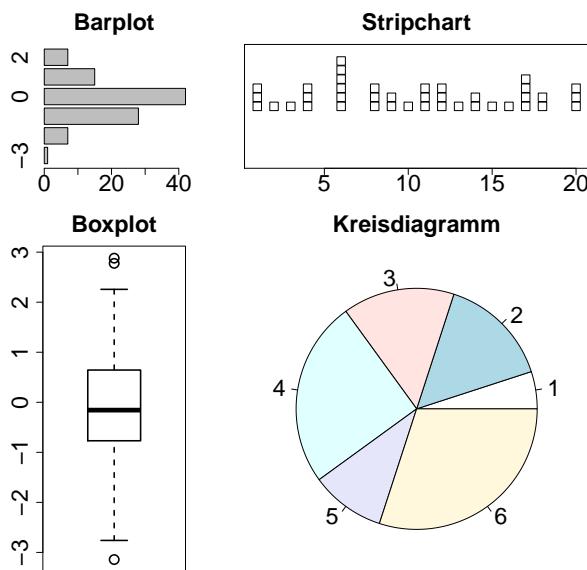


Abbildung 14.31: Verschiedene Diagramme in einem Diagrammfenster mittels `layout()`

Benachbarte Planquadrate der Device-Fläche lassen sich zusammenfassen, indem den zugehörigen Matrixelementen dieselbe Zahl zugeordnet wird (Abb. 14.32). Auch hier ist die Anzahl n der dargestellten Diagramme kleiner als die Zahl der Zellen von `mat`.

```
# Zellen der ersten Zeile zusammenfassen, 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")
```

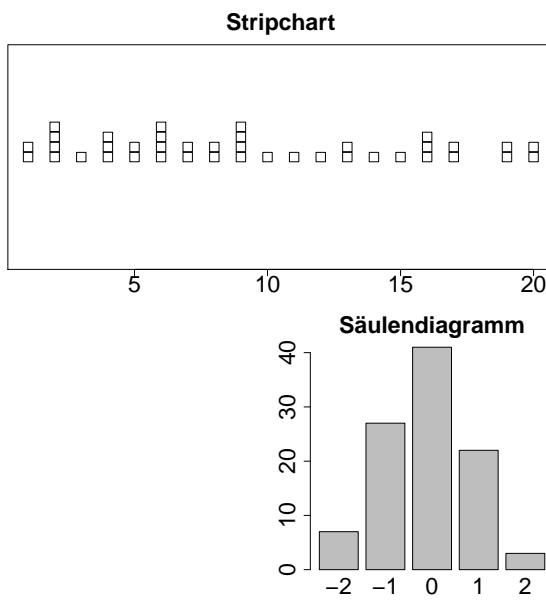


Abbildung 14.32: Mehrere, teils zusammengefasste, teils leere Regionen in einem Diagrammfenster mittels `layout()`

14.8.2 `par(mfrow, mfcoll, fig)`

Ein weiterer Weg Device-Flächen zu unterteilen, besteht in der Verwendung der Argumente `mfrow` oder `mfcoll` von `par()`.

```
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 jeweils 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 Device-Fläche in einem rechteckigen Layout letztlich `<Anzahl>` viele Diagramme unterzubringen, kann alternativ auch die Funktion `n2mfrow(<Anzahl>)` zur Erstellung eines geeigneten Vektors verwendet werden, der dann an `mfrow` zu übergeben ist.

```
> dev.new(width=7, height=4)                      # Diagrammgröße ändern
> par(mfrow=c(1, 2))                            # Diagrammfenster aufteilen
> boxplot(rt(100, 5), xlab=NA, notch=TRUE, main="Boxplot")
> plot(rnorm(10), pch=16, xlab=NA, ylab=NA, main="Streudiagramm")
```

Mit `par(fig=<Vektor>, new=TRUE)` kann alternativ vor einem High-Level-Befehl zum Erstellen eines Diagramms die rechteckige Figure-Region innerhalb der Device-Fläche definiert werden, in die das folgende Diagramm gezeichnet wird.

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 Grafikbefehl dem aktiven device hinzugefügt werden soll. Indem `par(fig)` mehrfach alternierend mit High-Level-Grafikfunktionen aufgerufen wird, lassen sich mehrere Diagramme in einem device plazieren (Abb. 14.33).

```
# Simulation von 1000 mal 10 Bernoulli-Experimenten
> resBinom <- rbinom(1000, size=10, prob=0.3)

# in Faktor umwandeln für mögliche Kategorien, die nicht auftauchen
> facBinom <- factor(resBinom, levels=0:10)

# Häufigkeitsverteilung bestimmen und als senkrechte Linien darstellen
> tabBinom <- table(facBinom)
> par(fig=c(0, 1, 0.10, 1))
> plot(tabBinom, type="h", bty="n", xlim=c(0, 10), xlab=NA,
+       xaxt="n", ylab="Häufigkeit", main="Ergebnisse
+       1000*10 Bernoulli Experimente (p=0.3)")

# Werte zusätzlich als Punkte einzeichnen
> points(names(tabBinom), tabBinom, pch=16, cex=2, col="gray")

# im unteren Diagrammbereich gekerbten 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")
```

14.8.3 `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

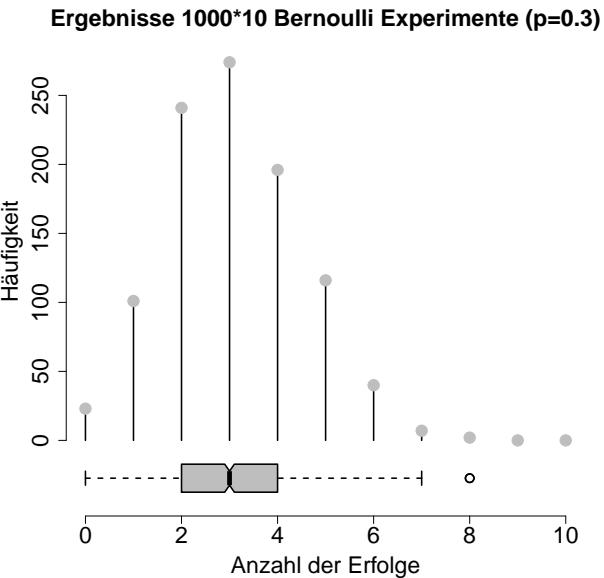


Abbildung 14.33: Anpassen der Größe verschiedener Regionen mit `par(fig)`

`screen()` und `close.screen()` zu verwenden ist.

```
split.screen(figs=<2-Vektor oder Nx4-Matrix>, screen=<Nummer>)
```

Das Argument `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. 14.34).

Die mit `split.screen()` gebildeten Regionen sind intern bei 1 beginnend numeriert 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 `screen(<Nummer>)` 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, aller-

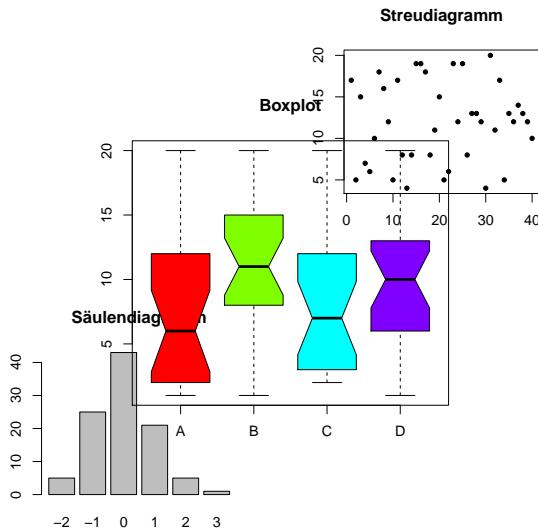


Abbildung 14.34: Sich überlappende Regionen mit `screen()` erzeugen

dings mit dem Risiko unerwünschter Effekte. Der Inhalt einer mit `split.screen()` erzeugten Teilfläche kann mit `erase.screen(<Nummer>)` wieder gelöscht werden.

Um eine mit `split.screen()` erzeugte Teilfläche abzuschließen und damit ein weiteres Hinzufügen von Elementen zu verhindern, dient `close.screen(<Nummer>, all.screens=FALSE)`. Mit ihr können über das Argument `all.screens=TRUE` auch alle Teilflächen simultan gesperrt werden. Dies sollte etwa geschehen, wenn alle Regionen eines device mit Diagrammen gefüllt und keine weiteren Änderungen gewünscht sind.

```
# 3 Regionen als Zeilen einer Matrix 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
> x <- gl(4, 25, labels=LETTERS[1:4])        # Bedingungen
> y <- sample(1:20, 100, replace=TRUE)         # Zielgröße
> boxplot(y ~ x, col=rainbow(4), notch=TRUE, main="Boxplot")
> 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 15

Diagramme mit `ggplot2`

Mit dem Zusatzpaket `ggplot2` lassen sich die in Kap. 14 vorgestellten Diagrammtypen ebenfalls erstellen. Dabei ist die Herangehensweise eine grundsätzlich andere: Während der Basisumfang von R für verschiedene Diagrammarten einzelne Funktionen bereitstellt, werden mit `ggplot2` alle Diagrammtypen mit einem einheitlichen System erzeugt.¹ Sind Diagramme des Basisumfangs analog zu einer Leinwand, auf der jede Funktion später nicht mehr änderbare Elemente aufmalt, repräsentiert `ggplot2` dagegen alle Diagrammelemente explizit in einem Objekt. Erstellte Diagramme lassen sich über dieses Objekt weiter verändern, an Funktionen übergeben und speichern.

Zu den Vorteilen von `ggplot2`-Diagrammen gegenüber jenen des Basisumfangs zählt, dass sie oft ästhetisch ansprechender sind und automatisch Legenden für alle verwendeten visuellen Attribute besitzen. Besser sind auch der einheitliche Zugang zu verschiedenen Diagrammtypen und die besonders einfache Möglichkeit, für alle Teilgruppen von Daten bedingte Diagramme in vielen Panels gleichzeitig darzustellen. Ein Nachteil ist die ungewohnte Syntax. Auch machen große Datenmengen die Darstellung langsam.

Eine detaillierte Einführung in `ggplot2` findet sich bei [Wickham und Sievert \(2024\)](#) oder [Chang \(2018\)](#). [Wilke \(2019\)](#) verwendet `ggplot2` als Basis, um allgemein Grundlagen der Diagrammgestaltung zu erklären. Teils sind diese Bücher vollständig online erhältlich, teils zeigen sie auf einer Webseite zahlreiche Beispiele.²

15.1 Grundprinzip

In `ggplot2` wird ein Diagramm als Liste repräsentiert, die alle Informationen über die Darstellung beinhaltet und sich als Objekt speichern oder verändern lässt. `ggplot`-Objekte erscheinen erst dann als Diagramm in einem device, wenn – explizit oder implizit (Abschn. 1.4.3) – `print(<ggplot-Objekt>)` aufgerufen wird.

¹Dies orientiert sich an Ideen der *grammar of graphics* ([L. Wilkinson, 2005](#)) und basiert intern auf dem Paket `grid` aus dem Basisumfang von R.

²Insbesondere <https://r-graphics.org/>. Siehe auch das `ggplot2` *cheat sheet* unter <https://posit.co/resources/cheatsheets/>. Eine interaktive Web-Oberfläche zum Aufbau eines vollständigen Diagramms bietet die Shiny-App des Pakets `esquisse` ([F. Meyer & Perrier, 2024](#)). Das Paket `gganimate` ([Pedersen & Robinson, 2020](#)) ermöglicht es Animationen zu erstellen.

15.1.1 Grundsicht

Für alle Diagrammtypen ist im ersten Schritt mit `ggplot()` eine Grundsicht zu erzeugen, die die darzustellenden Daten festlegt und definiert, wie Eigenschaften der Daten in verschiedene visuelle Attribute umgesetzt werden. Einer Grundsicht lassen sich dann weitere Schichten hinzufügen, um ein konkretes Diagramm zu erhalten und Diagrammoptionen zu ändern.

```
ggplot(<Datensatz>, aes(x=<x-Koordinaten>,
                           colour=<Variable>,
                           fill=<Variable>,
                           shape=<Faktor>,
                           linetype=<Faktor>,
                           group=<Faktor>))
```

Als erstes Argument wird der Datensatz erwartet, aus dem die im Diagramm dargestellten Variablen stammen müssen. Bei Daten aus Messwiederholungen sollte der Datensatz im Long-Format vorliegen (Abschn. 3.3.11). Das zweite Argument bestimmt mit der Funktion `aes()`, welche Werte für die *x*- und *y*-Koordinaten herangezogen werden. Zusätzliche Argumente für `aes()` kontrollieren die Zuordnung von weiteren Variablen zu visuellen Attributen der Diagrammelemente, die dann automatisch in der Legende erklärt werden.³ Zu beachten ist hierbei, dass alle Variablennamen ohne Anführungszeichen verwendet werden.⁴

- **colour:** Eine übergebene Variable sorgt dafür, dass die Datenpunkte getrennt nach ihrer Ausprägung eingefärbt sind. Welche Farbe welcher Gruppe zugeordnet ist, wählt `ggplot()` in der Voreinstellung selbst (Abschn. 15.5.4). Möglich sind stetige Variablen wie auch Faktoren.
- **fill:** Bei flächigen Diagrammelementen – etwa Säulen – kontrolliert die übergebene Variable analog zu `colour` deren Farbe.
- **shape:** Der übergebene Faktor bestimmt bei Punktdiagrammen die Art der Datenpunkt-symbole (Abschn. 15.5.4).
- **linetype:** Der übergebene Faktor bestimmt den Linientyp bei Liniendiagrammen (Abschn. 15.5.4).
- **group:** Ein genannter Faktor sorgt dafür, dass in hinzugefügten Schichten jeweils alle Datenpunkte einer Gruppe (Faktorstufe) als zusammengehörig betrachtet werden. Unterschiedliche Gruppen werden dagegen getrennt behandelt, was etwa für Liniendiagramme oder Boxplots relevant ist.⁵

Die hier vorgestellten Beispiele verwenden die folgenden simulierten Daten.

³`vignette("ggplot2-specs")` erläutert die visuellen Attribute.

⁴Wo notwendig, lässt sich alternativ `aes(x=.data[["<Name>"]], y=.data[["<Name>"]])` mit Variablennamen als Zeichenkette verwenden. Hier repräsentiert `.data` den Datensatz der Grundsicht unabhängig vom übergebenen ursprünglichen Namen des Datensatzes.

⁵Übergibt man statt eines Faktors die 1, gehören alle Datenpunkte zusammen, z. B. für ein Liniendiagramm einer einzelnen Datenreihe.

```

> Njk <- 50                                # Gruppengröße
> P    <- 3                                 # Anzahl Stufen Gruppe
> Q    <- 2                                 # Anzahl Stufen Geschlecht
> sex <- factor(rep(c("f", "m"), times=P*Njk))      # Geschlecht

# Gruppenzugehörigkeit
> group  <- factor(rep(c("control", "placebo", "treatment"), each=Q*Njk))
> sgComb <- interaction(sex, group)  # kombinierte Gruppenzugehörigkeit
> mood   <- round(rnorm(P*Q*Njk,       # Stimmung
+                           mean=c(85,80,110,90,130,100)[sgComb], sd=25))

> height <- rnorm(P*Q*Njk, mean=c(170, 180)[sex], sd=7) # Körpergröße
> rating <- factor(rbinom(Njk*P*Q, size=3,           # Zustimmung
+                           prob=c(0.1,0.25,0.4,0.6,0.75,0.9))[sgComb]), levels=0:3,
+                           labels=c("Not at all", "Somewhat", "Mostly", "Completely"))

# Gesamt-Datensatz
> myDf <- data.frame(sex, group, sgComb, mood, height, rating)

```

Als Grundsicht soll hier die Stimmung gegen die Körpergröße aufgetragen werden, wobei das Geschlecht die Farbe der Datenpunktsymbole und die Gruppenzugehörigkeit die Art der Datenpunktsymbole bestimmt. Die Grundsicht allein erzeugt ein leeres Diagramm.

```

> library(ggplot2)                         # für ggplot()
> ggplot(myDf, aes(x=height, y=mood, colour=sex, shape=group))

```

15.1.2 Diagramme speichern

Mit den in Abschn. 14.1.2 vorgestellten Funktionen wie `pdf()` oder `jpeg()` lassen sich ebenfalls `ggplot`-Objekte speichern. Eine bequeme Alternative dazu ist `ggsave()`.

```
ggsave("<Dateiname>", plot, device, width=<Breite>, height=<Höhe>,
       units=<Maßeinheit>, dpi=<Auflösung>)
```

Nach dem Dateinamen im ersten Argument kann optional ein `ggplot`-Objekt als zweites Argument `plot` übergeben werden. Fehlt dies, speichert die Funktion automatisch das letzte dargestellte `ggplot`-Diagramm. Anhand der gewählten Endung des Dateinamens (etwa "`<Name>.pdf`") wählt `ggsave()` selbst automatisch das passende Grafikformat. Dies kann aber auch für `device` explizit angegeben werden. Breite und Höhe der Grafikdatei lassen sich über `width` und `height` kontrollieren, wobei die gemeinte Maßeinheit für `units` zu nennen ist – Voreinstellung ist "`in`" für inch, andere Möglichkeiten sind "`cm`" und "`mm`". Die Auflösung beträgt in der Voreinstellung 300 dpi (*dots per inch*) und kann über `dpi` erhöht oder gesenkt werden.

15.2 Diagrammtypen

Jede der folgenden Funktionen fügt einer mit `ggplot()` erstellten Grundschicht einen konkreten Diagrammtyp hinzu (für weitere vgl. Wickham & Sievert, 2024), wenn sie mit dieser Schicht über `+` verknüpft wird.⁶ Dafür besitzen die genannten Funktionen eigene Argumente, um Eigenschaften zu kontrollieren, die spezifisch für den jeweiligen Diagrammtyp sind (s. u. für Beispiele und die jeweils zugehörige Hilfe-Seite für Details).

- `geom_point()` für ein Punkt- oder Streudiagramm
- `geom_bar()` sowie `geom_col()` für ein Säulendiagramm
- `geom_histogram()` für ein Histogramm
- `geom_density()` für einen Kerndichteschätzer
- `geom_line()` für ein Liniendiagramm
- `geom_boxplot()` für einen Boxplot

Die Diagrammtypen lassen sich auch kombinieren: So kann etwa einer Grundschicht sowohl ein Boxplot als auch ein Punktdiagramm für die Rohdaten hinzugefügt werden, die das Diagramm dann beide darstellt. Später angefügte Schichten verdecken dabei ggf. Inhalte zuvor definierter Schichten.

Alle `geom_<Name>()` Funktionen verwenden in der Voreinstellung den in der Grundschicht definierten Datensatz. Sie können aber auch auf einen eigenen Datensatz Bezug nehmen, der für das Argument `data` zu nennen ist. Analog lässt sich über `aes()` eine Zuordnung von Variablen des Datensatzes zu visuellen Attributen des Diagramms festlegen, die für die mit `geom_<Name>()` erstellte Schicht spezifisch ist.

15.2.1 Punkt-, Streu- und Liniendiagramme

Als Beispiel soll zunächst mit `geom_point()` ein Streudiagramm erzeugt werden (Abb. 15.1).

```
# füge Grundschicht Streudiagramm mit größeren Datenpunktsymbolen hinzu
> ggplot(myDf, aes(x=height, y=mood, colour=sex, shape=group)) +
+     geom_point(size=3)
```

Das mit `geom_line()` erzeugte Liniendiagramm zeigt die Gruppenmittelwerte der Stimmung getrennt für Männer und Frauen (Abb. 15.1). Dafür soll zunächst mit `aggregate()` (Abschn. 3.3.12) ein geeigneter Datensatz erzeugt werden.

⁶Die ungewöhnliche Syntax ist möglich, weil der Operator `+` eine Funktion ist, `a + b` also in `"+"(a, b)` übersetzt wird (Abschn. 1.2.5). Weiter ist `"+"` generisch, kann sich also für `ggplot`-Objekte anders als für Zahlen verhalten (Abschn. 17.3.7).

```
> (groupM <- aggregate(mood ~ sex + group, data=myDf, FUN=mean))
   sex      group     mood
1   f    control  76.06667
2   m    control  82.60000
3   f    placebo 118.76667
4   m    placebo  95.60000
5   f treatment 132.70000
6   m treatment 103.93333

> ggplot(groupM,
+         aes(x=group, y=mood, color=sex, shape=sex, group=sex)) +
+         geom_point(size=2) +
+         geom_line(linewidth=0.8)
```

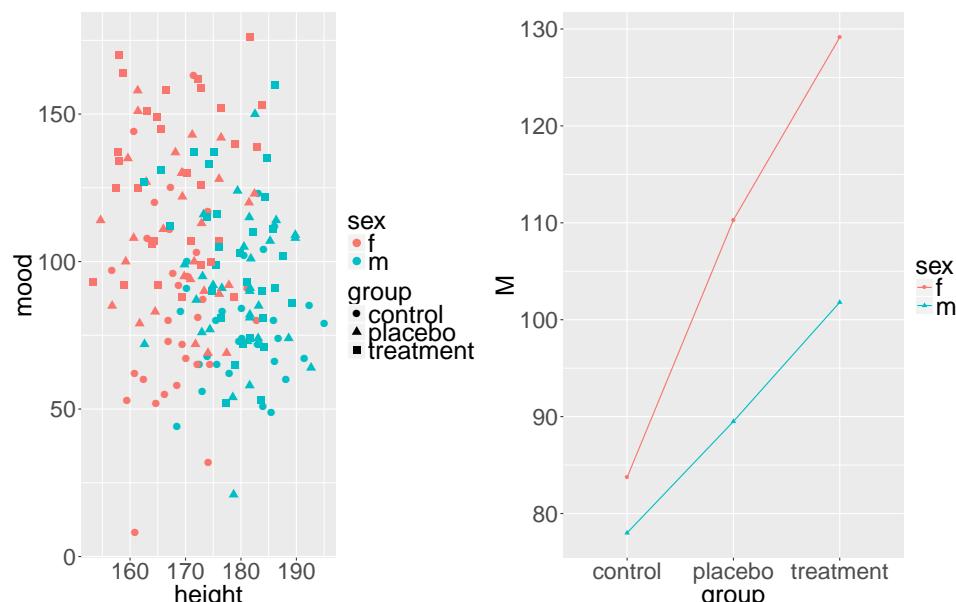


Abbildung 15.1: Streudiagramm mit `geom_point()` und Liniendiagramm mit `geom_line()`

15.2.2 Säulendiagramm

Für ein Säulendiagramm mit `geom_bar()` ist im vorherigen Aufruf von `ggplot(..., aes(x=<Variable>))` für `x` eine Variable anzugeben, deren Wertehäufigkeiten die Höhe der Säulen bestimmen.

Bei `geom_bar()` ist `x` eine kategoriale Variable, deren absolute Wertehäufigkeiten automatisch ausgezählt und als Säulenhöhe visualisiert werden. Dieses Verhalten wird kontrolliert durch die Voreinstellung `stat="count"`, die definiert, wie die Daten vor der Darstellung transformiert werden. Relative Häufigkeiten erhält man mit `geom_bar(aes(y=after_stat(count / sum(count))))`.⁷

⁷count ist der Name einer mit `stat="count"` intern erzeugten Variable mit den berechneten Gruppenhäufigkeiten, auf die man innerhalb des Aufrufs von `aes()` mit `after_stat()` Zugriff hat, um sie in eigenen

```
# absolute Häufigkeiten
> ggplot(myDf, aes(x=rating)) +
+   geom_bar()

# relative Häufigkeiten
> ggplot(myDf, aes(x=rating)) +
+   geom_bar(aes(y=after_stat(count / sum(count))))
```

Weist man in der Grundschicht mit `aes(y=<Variable>)` die auszuzählende Variable der *y*-Koordinate zu, erzeugt `geom_bar()` horizontale Balken, deren Länge die Wertehäufigkeiten darstellen.

Anders als die Voreinstellung stellt `geom_bar(stat="identity")` die nicht weiter aggregierten Werte von `x` selbst als Säulenhöhe dar – etwa wenn dies bereits vorher berechnete Häufigkeiten sind (Abb. 15.2). Eine Kurzform dafür ist die Funktion `geom_col()`, für die im vorherigen Aufruf von `aes()` sowohl *x*- als auch *y*-Koordinate definiert werden müssen.

```
# Datensatz der Häufigkeiten selbst erstellen
> (myDf_freq <- as.data.frame(xtabs(~ rating, data=myDf)))
  rating Freq
1 Not at all  49
2 Somewhat   51
3 Mostly     36
4 Completely  44

# horizontal orientierte Balken
> ggplot(myDf_freq, aes(x=Freq, y=rating)) +
+   geom_col()
```

Bei Klassifikation der Daten durch zwei Faktoren sorgt das Argument `position` mit der Voreinstellung `position_stack()` für ein gestapeltes Säulendiagramm (Abb. 15.3). Sollen im gestapelten Säulendiagramm alle Säulen auf die Länge 1 normiert sein, damit die Segmente die bedingten relativen Anteile der ausgezählten kategorialen Variable darstellen, ist `position_fill()` zu verwenden (Abschn. 15.5.1). Abschnitt 15.5.2 zeigt, wie Achsenbeschriftungen angepasst werden können.

```
# gestapeltes Säulendiagramm
> ggplot(myDf, aes(x=rating, group=sex, fill=sex)) +
+   geom_bar(position=position_stack())
```

Demgegenüber führt `geom_bar(position=position_dodge())` zu einem gruppierten Säulendiagramm, das hier die bedingten relativen Häufigkeiten der ratings je Geschlecht darstellen soll.

```
# bedingte relative Häufigkeiten ratings je Geschlecht
> tab_freq <- xtabs(~ sex + rating, data=myDf)
> (tab_crel <- proportions(tab_freq, margin=1))
rating
```

Rechnungen zu verwenden.

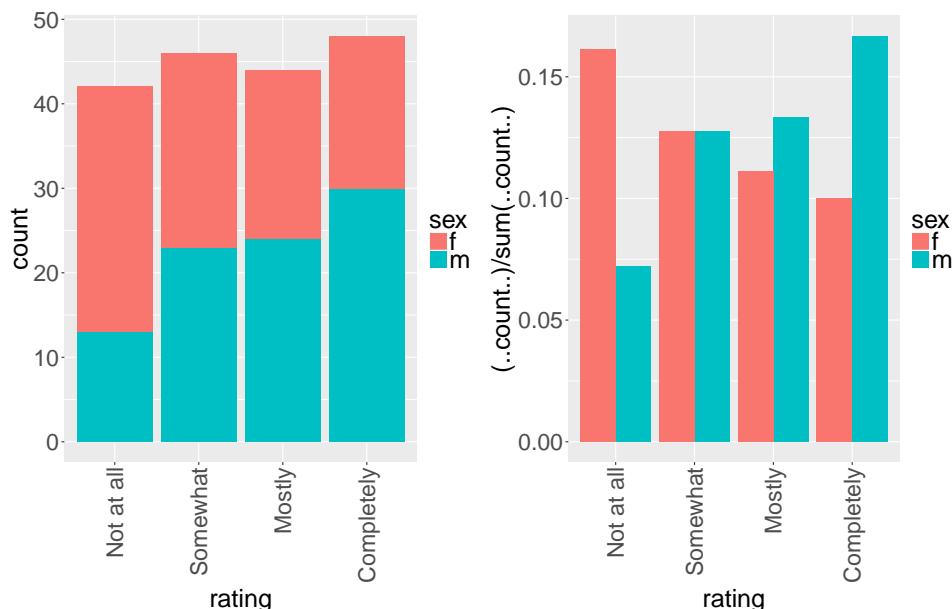


Abbildung 15.2: Einfaches Säulendiagramm der absoluten Häufigkeiten vertikal und horizontal orientiert mit `geom_bar()` und `geom_col()`.

```

sex Not at all Somewhat Mostly Completely
  f  0.3666667 0.2666667 0.2000000 0.1666667
  m  0.1777778 0.3000000 0.2000000 0.3222222

# als Datensatz inkl. gerundeter Angabe für labels
> myDf_crel <- as.data.frame(tab_crel) |>
+   transform(Freq_round=round(Freq, 2))

# gruppiertes Säulendiagramm - schmalere Säulen, default 0.9
> ggplot(myDf_crel, aes(x=sex, y=Freq, label=Freq_round,
+                         group=rating, fill=rating)) +
+   geom_col(position=position_dodge(width=0.8),
+             width=0.8)

```

15.2.3 Histogramm

Bei einem Histogramm mit `geom_histogram()` ist in der Grundschicht bei `aes(x=<Variable>)` für `x` die Variable anzugeben, deren Häufigkeit von Werte-Intervallen vom Histogramm als Höhe senkrechter Säulen dargestellt werden. Für horizontale Balken ist `aes(y=<Variable>)` zu verwenden.

Um die geschätzte Dichte (die relative Häufigkeit geteilt durch die Intervallbreite) statt der absoluten Häufigkeiten der Kategorien darzustellen, kann man `geom_histogram(aes(y=after_stat(density)))`

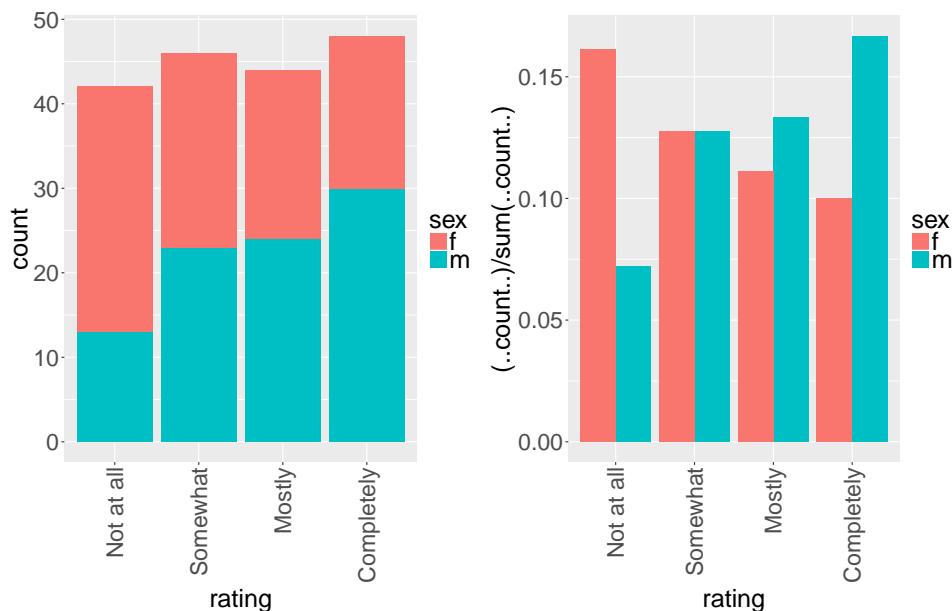


Abbildung 15.3: Gestapeltes Säulendiagramm der absoluten Häufigkeiten sowie gruppiertes Säulendiagramm der bedingten relativen Häufigkeiten mit `geom_bar()`. Für senkrecht orientierte Beschriftungen der x -Achse s. Abschn. 15.5.2.

verwenden.⁸ In `geom_histogram()` lässt sich über `binwidth=⟨Breite⟩` die Breite der Intervalle kontrollieren, alternativ über `bins=⟨Anzahl⟩` ihre Anzahl (Abb. 15.4).

```
> ggplot(myDf, aes(x=mood)) +
+   geom_histogram(bins=30)
```

Weil die Form eines Histogramms von der willkürlichen Wahl für Anzahl und Grenzen der Intervalle abhängt, ist ein zusätzlich dargestellter Kerndichteschätzer oft sinnvoll. Dieser lässt sich mit `geom_density()` hinzufügen (Abb. 15.4). Das Argument `adjust` ist dabei ein Faktor mit der Voreinstellung 1 für die Bandbreite des Faltungskerns. Faktoren < 1 erhöhen die Variabilität des Ergebnisses, solche > 1 senken sie. Die Farbwahl erläutert Abschn. 15.5.4.

```
> ggplot(myDf, aes(x=mood)) +
+   geom_histogram(aes(y=after_stat(density))) +
+   geom_density(color="darkgrey", fill="grey", alpha=0.6))
```

15.2.4 Boxplot

`geom_boxplot()` erzeugt einen Boxplot. Dabei ist im vorherigen Aufruf von `ggplot()` innerhalb von `aes(x=⟨Faktor⟩, y=⟨Variable⟩, ...)` bei Bedarf für `x` ein Faktor zu übergeben, für dessen Stufen dann separate Boxplots angezeigt werden, die die Kennwerte der für `y` genannten Variable darstellen (Abb. 15.5).⁹

⁸Analog zu Fußnote 7 ist `density` der Name einer intern berechneten Variable, auf die man innerhalb von `aes()` mit `after_stat()` Zugriff hat.

⁹Die Methode zur Berechnung der Quantile weicht leicht von der in `boxplot()` (Abschn. 14.6.3) ab, dass in seltenen Fällen andere Darstellungen resultieren können.

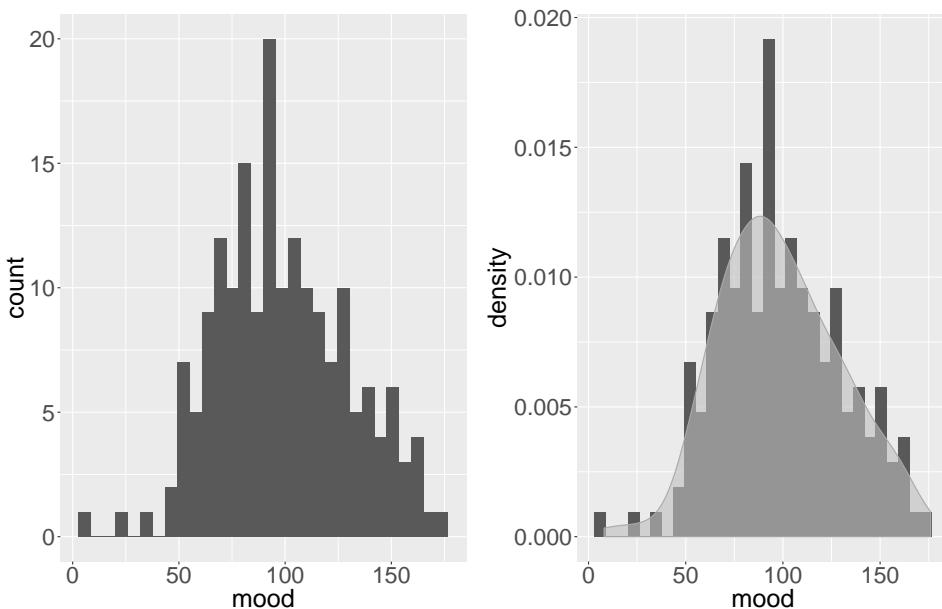


Abbildung 15.4: Histogramm der absoluten Häufigkeiten mit `geom_histogram()` sowie Dichte-Histogramm und Kerndichteschätzer mit `geom_density()`

Mit `notch=TRUE` werden gekerbte Boxplots zur Hervorhebung des Medians dargestellt. Abschnitt 15.5.3 zeigt, wie die hier unnötige Legende entfernt werden kann. Um die Orientierung zu ändern und horizontale Boxplots zu erhalten, kann man im Aufruf von `aes()` in der Grundschicht die Zuweisung an die Koordinaten tauschen und die quantitative Variable für `x` nennen.

```
> ggplot(myDf, aes(x=sex, y=height, fill=sex)) +
+   geom_boxplot(notch=TRUE)
```

Da Boxplots wichtige Eigenschaften der Verteilung wie etwa Multimodalität nicht repräsentieren können, ist es oft sinnvoll, zusätzlich die Rohdaten darzustellen. `geom_beeswarm()` aus dem Paket `ggbeeswarm` (Clarke & Sherrill-Mix, 2017) zeigt alle einzelnen Datenpunkte, wobei die horizontale Position so versetzt gewählt wird, dass die Verteilungsform gut ersichtlich ist (Abb. 15.5). Damit Ausreißer nicht doppelt dargestellt werden, sollte man sie in `geom_boxplot()` mit der Option `outlier.shape=NA` ausblenden.

```
> library(ggbeeswarm)                      # für geom_beeswarm()
> ggplot(myDf, aes(x=sex, y=height, fill=sex)) +
+   geom_boxplot(outlier.shape=NA) +    # Boxplot: outlier ausblenden
+   geom_beeswarm(alpha=0.5)           # mit simulierter Transparenz
```

Insbesondere bei vielen Datenpunkten sind Violin-Plots eine Alternative, die `geom_violin()` erstellt. Sie sind ähnlich wie Beeswarm-Plots aufgebaut, zeigen aber statt der Originaldaten selbst das Profil des Kerndichteschätzers, gespiegelt entlang der vertikalen Achse. Das Argument `adjust` ist dabei ein Faktor mit der Voreinstellung 1 für die Bandbreite des Faltungskerns. Faktoren < 1 erhöhen die Variabilität des Ergebnisses, solche > 1 senken sie.

```
> ggplot(myDf, aes(x=sex, y=height, fill=sex)) +
```

```
+     geom_boxplot() +
+     geom_violin(alpha=0.5)
```

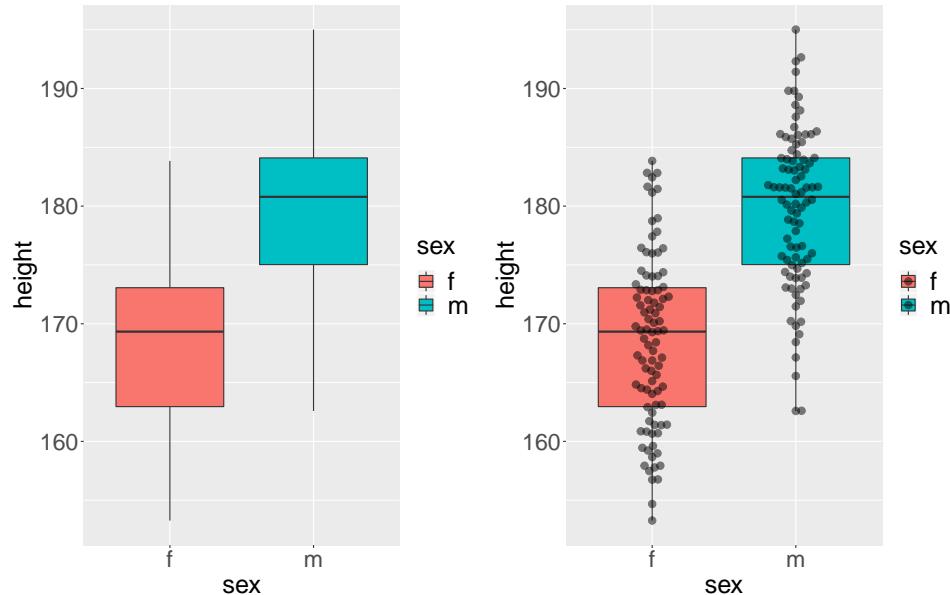


Abbildung 15.5: Nach Geschlecht getrennte Boxplots mit `geom_boxplot()` sowie Rohdaten mit `geom_beeswarm()`

15.2.5 Quantil-Quantil-Diagramm

Quantil-Quantil-Diagramme vergleichen eine empirische mit einer theoretisch vermuteten Verteilung mit Hilfe der jeweiligen Quantile zu denselben Wahrscheinlichkeiten. Sie können auch analog zwei empirische Verteilungen vergleichen (Abschn. 14.6.5). Die Diagramme werden durch `geom_qq()` erzeugt, woraufhin `geom_qq_line()` eine entsprechende Referenzlinie hinzufügt (Abb. 15.6). Im vorausgehenden Aufruf von `ggplot()` muss dafür innerhalb von `aes()` die Option `sample` die Variable definieren, deren empirische Quantile gegen die theoretischen Quantile – in der Voreinstellung die der Standardnormalverteilung – in einem Streudiagramm aufgetragen werden.

```
> ggplot(myDf, aes(sample=height)) +
+     geom_qq() +
+     geom_qq_line(color="blue")
```

Für den Vergleich mit einer anderen theoretischen Verteilung kann deren Quantilfunktion in `geom_qq()` und `geom_qq_line()` an das Argument `distribution` übergeben werden. Zusätzliche Parameter für diese Funktion sind in Form einer Liste mit entsprechend benannten Komponenten für das Argument `dparams` zu nennen.¹⁰

```
> parL <- list(df=10, ncp=0)           # Parameter für t-Verteilung
> ggplot(myDf, aes(sample=height)) +
```

¹⁰Für die Schätzung der Parameter geläufiger Verteilungen s. Abschn. 16.4.1.

Kapitel 15 Diagramme mit ggplot2

```
+     geom_qq(distribution=qt, dparams=parL) +
+     geom_qq_line(distribution=qt, dparams=parL, color="blue")
```

Für den Vergleich der Quantile zweier empirischer Verteilungen sind zunächst mit `ppoints()` die zu einem Quantil gehörende Werte der empirischen kumulierten Häufigkeitsverteilung als Schätzung der Verteilungsfunktion zu bilden. Als Argument kann die gewünschte Anzahl von Werten oder ein Vektor übergeben werden, dessen Länge dann die Anzahl bestimmt. Nach Berechnung der zugehörigen Quantile in beiden relevanten Variablen für die von `ppoints()` ausgegebenen Werte kann das Streudiagramm der Quantile manuell erstellt werden.

```
> probs <- ppoints(50)      # Wahrscheinlichkeiten 0.01 bis 0.99
> qDat  <- data.frame(qmood=quantile(myDf$mood, probs),
+                       qheight=quantile(myDf$height, probs))

> head(qDat, n=4)          # Quantile beider Variablen
   qmood  qheight
1%  28.79 156.9509
3%  44.00 161.5984
5%  52.95 162.8354
7%  59.00 163.3587

> ggplot(qDat, aes(x=qmood, y=qheight)) +
+     geom_point()
```

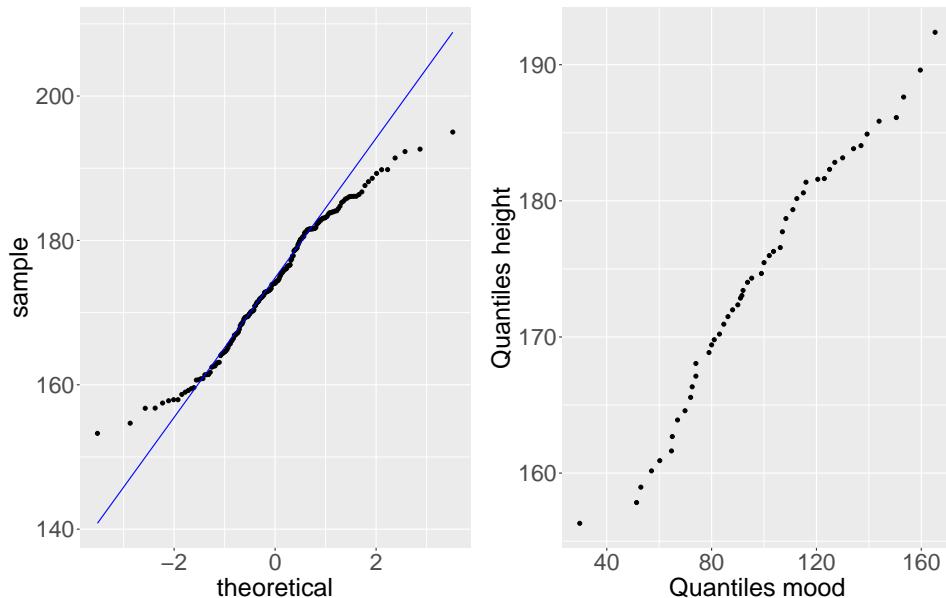


Abbildung 15.6: Quantil-Quantil-Diagramme mit `geom_qq()` und `geom_qq_line()`

15.3 Bedingte Diagramme in Panels darstellen

Multivariate Daten, deren Variablen teilweise kategorial sind, lassen sich durch bedingte Diagramme visualisieren: Dafür wird die gesamte Diagrammfläche in *Panels (Facetten)* unterteilt und in den Panels getrennt für jede Stufe einer kategorialen Variable derselbe Diagrammtyp dargestellt. Bei mehreren Faktoren ist dies für jede Kombination von Stufen verschiedener Faktoren möglich.

- Um die Diagrammregion in einzelne Panels aufzuteilen, die jeweils einer Gruppe zugeordnet sind, kann eine mit `facet_grid(<Zeilen> ~ <Spalten>)` erstellte Schicht hinzugefügt werden. Dabei definiert die übergebene Modellformel die Aufteilung der Diagrammfläche jeweils in Zeilen vs. Spalten durch links vs. rechts der `~` genannte Faktoren. Sollen Zeilen bzw. Spalten jeweils aus der Kombination zweier Faktoren gebildet werden, kann der zugehörige Teil der Modellformel `<Faktor1> + <Faktor2>` lauten – analog für weitere Faktoren.
- `facet_wrap(~ <Faktor>, nrow=<Anzahl>, ncol=<Anzahl>)` erstellt ebenfalls ein Panel für jede Gruppe des rechts der `~` genannten Faktors, ordnet die Panels aber selbstständig in einer rechteckig gekachelten Fläche an. Sollen die Gruppen aus der Kombination zweier Faktoren gebildet werden, kann die Modellformel `~ <Faktor1> + <Faktor2>` lauten. Die Anzahl der Zeilen und Spalten des Diagramms lässt sich durch `nrow` und `ncol` vorgeben.
- In der Voreinstellung erhalten alle Panels dieselben Achsenbereiche. Es ist jedoch möglich, mit dem Argument `scales="free_y"` von `facet_grid()` pro Zeile eine eigene *y*-Achse und mit `scales="free_x"` pro Spalte eine eigene *x*-Achse zu definieren. `scales="free"` kombiniert beide Optionen. Auch `facet_wrap()` akzeptiert diese Optionen, gibt dann jedoch jedem Panel eine eigene *x*- bzw. *y*-Achse, also mehrere *y*-Achsen pro Zeile bzw. mehrere *x*-Achsen pro Spalte.

Zunächst sollen nach Geschlecht getrennte Boxplots in separaten Panels für die Gruppen dargestellt werden (Abb. 15.7). Abschnitt 15.5.3 zeigt, wie die hier unnötige Legende entfernt werden kann.

```
> ggplot(myDf, aes(x=sex, y=height, fill=sex)) +
+   geom_boxplot() +
+   facet_grid(~ group)
```

Nun soll für jede Kombination von Geschlecht und Gruppe jeweils ein Panel das bedingte Streudiagramm von Körpergröße und Stimmung zeigen (Abb. 15.7).

```
> ggplot(myDf, aes(x=height, y=mood)) +
+   geom_point() +
+   facet_wrap(~ sex + group, ncol=2, scales="free_y")
```

Analog zu den in Abschn. 14.8 vorgestellten Lösungen für Basis-Diagramme kann das Paket `patchwork` (Pedersen, 2020) verschiedene `ggplot`-Objekte zu einem gemeinsamen Diagramm kombinieren. Dabei lässt sich die Aufteilung der Diagrammfläche sehr flexibel steuern. Auch ist es möglich, Teildiagramme mit Beschriftungen wie a) oder b) zu versehen und weiteren beliebigen Text auf dem Diagramm einzutragen. In Kombination mit der Funktion `as_gtable()` aus dem Paket `gt` können auch aufwendig formatierte Tabellen Teil eines Diagramms sein.

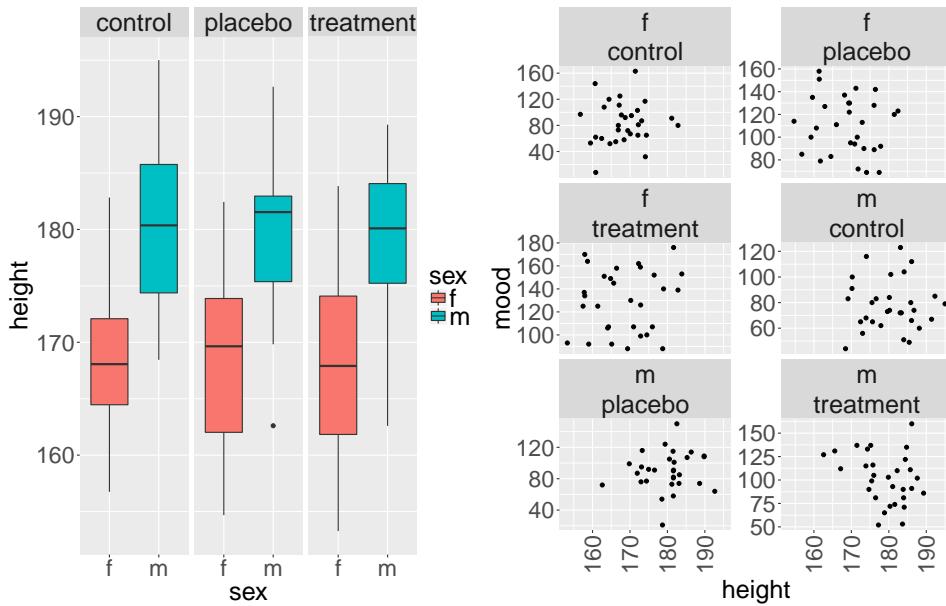


Abbildung 15.7: Bedingte Diagramme in mehreren Panels mit `facet_grid()` und `facet_wrap()`.

15.4 Diagrammelemente hinzufügen

Einer Grundschicht mit konkretem Diagramm lassen sich mit `+` weitere Schichten mit Diagrammelementen hinzufügen, etwa Linien, Text oder Regressionsgeraden. Diese Elemente sind frei miteinander kombinierbar und können sich über die Argumente `data` sowie `aes()` der hinzufügenden Funktion ggf. auch auf andere Daten als die der Grundschicht beziehen.

Geometrische Grundformen:

- `geom_segment(x=<Vektor>, y=<Vektor>, xend=<Vektor>, yend=<Vektor>)` für Liniensegmente, die bei Koordinaten `x`, `y` beginnen und `xend`, `yend` enden. Über das Argument `arrow` lassen sie sich auch als Pfeile zeichnen.
- `geom_polygon()` für Polygone, wodurch man z. B. Umrisse in Karten zeichnen kann.
- `geom_rect(xmin=<Vektor>, xmax=<Vektor>, ymin=<Vektor>, ymax=<Vektor>)` für Rechtecke, deren linke untere Ecke über Koordinaten `xmin`, `ymin` und deren rechte obere Ecke über `xmax`, `ymax` definiert ist.

Besondere Diagrammelemente:

- `labs(title=<Text>, subtitle=<Text>)` für einen Diagrammtitel und -untertitel.
- `geom_vline(aes(xintercept=<x-Koord.>))` für vertikale Linien, die die `x`-Achse in `xintercept` schneiden.
- `geom_hline(aes(yintercept=<y-Koord.>))` für horizontale Linien, die die `y`-Achse in `yintercept` schneiden.

- `geom_ribbon()` für eine Fläche zwischen den in `aes()` definierten y -Koordinaten `ymin=⟨untere Grenze⟩` und `ymax=⟨obere Grenze⟩`. Eine mögliche Anwendung ist die Darstellung eines durchgehenden Konfidenzbereichs um eine Vorhersage.
- `geom_linerange()` für vertikale Liniensegmente, die in `aes()` über die Argumente `ymin=⟨Vektor⟩` und `ymax=⟨Vektor⟩` definiert werden. Mögliche Anwendungen sind die Darstellung von punktweisen Konfidenzintervallen oder Spike-Plots.
- `geom_errorbar()` für klassische Fehlerbalken, die in `aes()` über die Argumente `ymin=⟨Vektor⟩` und `ymax=⟨Vektor⟩` definiert werden. Im Gegensatz zu `geom_linerange()` besitzen sie Querlinien am oberen und unteren Ende des dargestellten Intervalls.
- `geom_rug()` für eine Darstellung der Einzelwerte einer Verteilung als vertikale Striche entlang der x - oder y -Achse – z. B. kombiniert mit einem Kerndichteschätzer aus `geom_density()`.
- `geom_smooth()` für eine automatisch berechnete Regressionslinie. Dies ist etwa die Vorhersage einer linearen Regression mit `method="lm"` (Abschn. 6.2) oder ein lokaler LOESS-Glätter mit `method="loess"` (Abschn. 16.1.3, Voreinstellung für kleinere Datensätze).
 - Die angegebene Modellierungsfunktion kann weitere Argumente benötigen, etwa mit `method="glm"`. Solche Argumente lassen sich als Liste für `method.args` angeben, etwa `list(family=binomial())` (Abschn. 8.1.1).
 - Das Argument `formula` kann mit einer Modellformel das Regressionsmodell spezifizieren, etwa um polynomiale Terme für nichtlineare Verläufe hinzuzufügen (Abschn. 5.2). Die Voreinstellung ist `⟨y⟩ ~ ⟨x⟩`, mit den unter `aes()` angegebenen Variablen für die x - und y -Achse.
 - Mit `se=TRUE` erhält man zusätzlich den Konfidenzbereich für die Regressionslinie zum Niveau `level`.
 - `fullrange=TRUE` sorgt dafür, dass sich die Regressionslinie bis an den Rand eines Diagramms erstreckt, also über den beobachteten Wertebereich der x -Variable hinaus extrapoliert.
- `geom_function(fun=⟨Funktion⟩, args=⟨Liste⟩)` für Funktionsgraphen. Dabei ist `fun` die Funktion, z. B. `dnorm`, und `args` die Liste der notwendigen Funktionsargumente, z. B. `list(mean=0, sd=1)`.
- `geom_text(aes(x=⟨x-Koord.⟩, y=⟨y-Koord.⟩, label=⟨Variable⟩))` für Text, der in jedem Panel des Diagramms an den Koordinaten `x` und `y` erscheinen soll. Für `label` ist die Variable des Datensatzes im Aufruf von `ggplot()` zu nennen, die diesen Text für jede Beobachtung enthält. Das Paket `ggrepel` (Slowikowski, 2024) bietet eine Lösung für das Problem, dass Datenpunkte mit Text beschriftet werden sollen, ohne diese Punkte oder andere Beschriftungen zu überdecken.
- `annotate("text", x=⟨x-Koord.⟩, y=⟨y-Koord.⟩, label="⟨Anmerkung⟩")` für Textanmerkungen, die in jedem Panel des Diagramms identisch sind und an den Koordinaten `x` und `y` erscheinen sollen, wobei dieser Text an das Argument `label` zu übergeben ist.

Der in Abschn. 15.2.1 erstellte Datensatz der Gruppenmittelwerte der Stimmung soll hier um die gruppenweise Streuung der Stimmung sowie um die jeweilige Zellbesetzung ergänzt werden. Die einzelnen Datensätze lassen sich mit `merge()` (Abschn. 3.3.9) zu einem gemeinsamen Datensatz zusammenführen, dem der gruppenweise Standardfehler hinzugefügt wird.

```
# einzelne Datensätze für Zellbesetzung und gruppenweise Streuung
> groupN <- as.data.frame(xtabs(~ sex + group, data=myDf))
> groupSD <- aggregate(mood ~ sex + group, data=myDf, FUN=sd)

# führe einzelne Datensätze zusammen
> groupMSD <- merge(groupM, groupSD, by=c("sex", "group"),
+                      suffixes=c(".M", ".SD"))

> groupMSDN <- merge(groupMSD, groupN, by=c("sex", "group"))

# berechne Standardfehler
> (groupMSDN <- transform(groupMSDN,
+                           SEMlo=mood.M - mood.SD/sqrt(Freq),
+                           SEMup=mood.M + mood.SD/sqrt(Freq))) # ...
```

Das in Abschn. 15.2.1 gezeigte Liniendiagramm der Gruppenmittelwerte soll nun mit Fehlerbalken erweitert werden, die ± 1 Standardfehler darstellen (Abb. 15.8).

```
> ggplot(groupMSDN,
+         aes(x=group, y=mood.M, ymin=SEMlo, ymax=SEMup,
+              color=sex, shape=sex, group=sex)) +
+     geom_point(size=2) +
+     geom_linerange() +
+     geom_line()
```

Abbildung 15.9 soll nach Gruppen getrennte Streudiagramme inkl. Regressionsgerade samt Konfidenzbereich, eine vertikale Referenz-Linie sowie Text-Labels in einem multi-Panel Layout kombinieren. Abschn. 15.5.3 zeigt, wie die Legende verbessert werden kann, indem man das Symbol für `geom_text()` unterdrückt.

```
> ggplot(myDf, aes(x=height, y=mood, colour=sex:group, shape=sex)) +
+     geom_vline(aes(xintercept=180), linetype=2) +
+     geom_point(size=3) +
+     geom_smooth(method="lm", se=TRUE, linewidth=1, fullrange=TRUE) +
+     facet_grid(sex ~ group) +
+     labs(title="mood ~ height getrennt nach Geschlecht + Gruppe") +
+     geom_text(aes(x=190, y=70, label=sgComb)) +
+     annotate("text", x=165, y=5, label="Annotation")
```

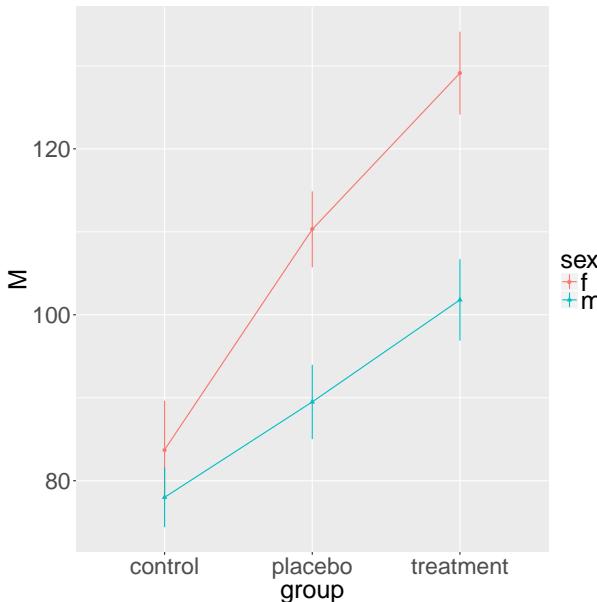


Abbildung 15.8: Liniendiagramm mit `geom_line()` und Fehlerbalken mit `geom_linerange()`

15.5 Diagramme formatieren

Das Aussehen von `ggplot2`-Diagrammen lässt sich über eine Vielzahl weiterer Argumente im Detail anpassen. Die wichtigsten Möglichkeiten zur individuellen Gestaltung werden im Folgenden vorgestellt.

15.5.1 Elementposition kontrollieren

Bei gruppiert dargestellten Daten mit kategorialer x -Achse zeigt das Diagramm für jede Kategorie der x -Achse mehrere Elemente – z. B. Boxplots oder Säulen. Diese Elemente lassen sich jeweils relativ zueinander auf verschiedene Arten anordnen. Dazu können die folgenden Funktionsaufrufe an das Argument `position` von `geom_<Name>()` Funktionen übergeben werden:

- `position_stack()` sorgt bei Säulendiagrammen dafür, dass die zu den Gruppen gehörenden Segmente aufeinander gestapelt werden (gestapeltes Säulendiagramm).
- `position_fill()` stapelt ebenfalls die Segmente aufeinander, skaliert jedoch alle Gesamtsäulen auf dieselbe Höhe. Diese Anordnung ist etwa für die Präsentation bedingter relativer Häufigkeiten geeignet, die sich pro Kategorie der x -Achse zu 1 addieren.
- `position_dodge(width=<Breite>)` ordnet Säulen oder Boxplots nebeneinander an, kann also etwa ein gruppiertes Säulendiagramm erzeugen. Über `width` lässt sich eine Lücke zwischen Diagrammelementen herstellen.
- Überlappen sich bei eigentlich stetigen x - oder y -Koordinaten viele mit `geom_point()` dargestellte Datenpunkte, kann man von der präzisen Positionierung abweichen, um alle Datenpunkte sichtbar zu machen: Übergibt man an das `position` Argument `position_jitter(width=<Streubreite>, height=<Streubreite>)`

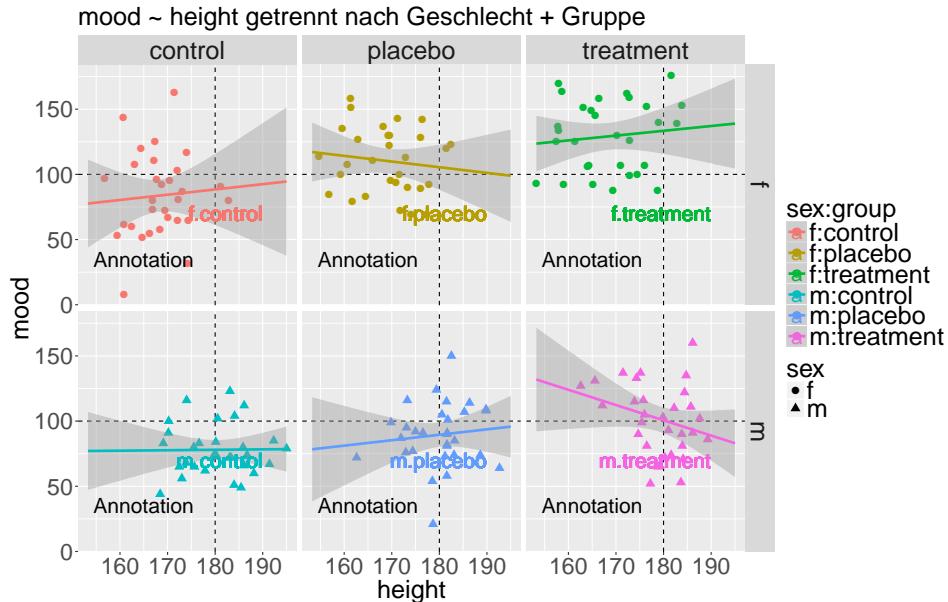


Abbildung 15.9: Nach Gruppen getrennte Streudiagramme inkl. Regressionsgerade samt Konfidenzbereich mit `geom_smooth()` sowie Text-Labels mit `geom_text()` und `annotate()`

sorgt dies dafür, dass x - oder y -Koordinaten mit einem kleinen zufälligen Versatz dargestellt werden. Die erzeugte halbe Streubreite legen `width` bzw. `height` fest – Voreinstellung ist 0.4. Bei sehr vielen Datenpunkten ist u. U. die Kombination mit simulierter Transparenz sinnvoll, die das Argument `alpha` von `geom_<Name>()` Funktionen kontrolliert (Abschn. 15.5.4). Alternativ kann auf einen 2D-Dichteschätzer mit `geom_hex()` analog zu `hexbin()` (Abschn. 14.6.8) ausgewichen werden.

Nach Gruppen getrennte Histogramme sollen zunächst versetzt nebeneinander dargestellt werden (Abb. 15.10).

```
> ggplot(myDf, aes(x=mood, fill=sex)) +
+   geom_histogram(aes(y=after_stat(density)),
+                  position=position_dodge())
```

Das folgende Diagramm stellt die bedingten relativen Häufigkeiten einer kategorialen Variable für verschiedene Gruppen in einem normierten Säulendiagramm dar (Abb. 15.10). Abschnitt 15.5.2 zeigt, wie die hier nicht sinnvollen Achsenbeschriftungen angepasst werden können.

```
> ggplot(myDf, aes(x=rating, group=sex, fill=sex)) +
+   geom_bar(stat="count",
+            aes(y=after_stat(count / sum(count))),
+            position=position_fill())
```

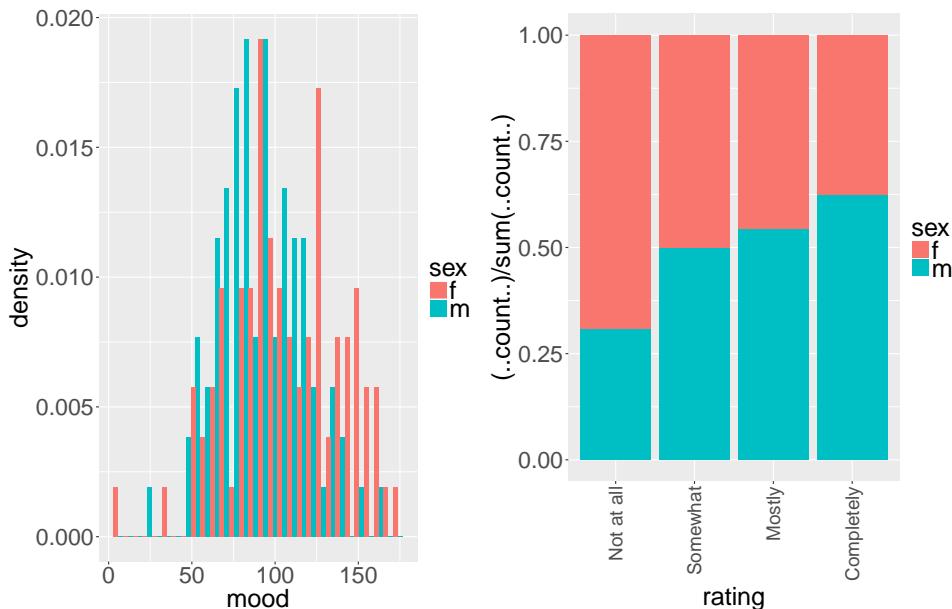


Abbildung 15.10: Versetzte gruppenweise Histogramme mit `position_dodge()` und gestapeltes normiertes Säulendiagramm mit `position_fill()`.

15.5.2 Achsen anpassen

Die folgenden Funktionen legen Skalierung, Wertebereich und genaue Position der Wertemarkierungen der Diagrammachsen fest, wenn sie mit + einer Grundschicht hinzugefügt werden.

- `labs(x="", y="")` für den Titel von x - und y -Achse, separat auch über `xlab()` bzw. `ylab()`
- `guides(x=guide_axis(angle=90))` sorgt für vertikal orientierte Wertebeschriftungen der x -Achse. Auch beliebige andere Winkel können für `angle` angegeben werden. Analog lässt sich die Ausrichtung der Wertebeschriftungen der y -Achse mit dem Argument `y` ändern.
- Als letzte Schicht entfernt `theme(axis.ticks=element_blank())` die Wertmarkierungen der Achsen, `theme(axis.text.x=element_blank())` löscht die Beschriftung der Wertmarkierungen der x -Achse (Abschn. 15.5.5).
- `scale_x_date(date_labels="")` bestimmt das Aussehen der x -Achse, wenn dort Werte einer Datumsvariable dargestellt sind. Insbesondere lässt sich die Formatierung des Datums mit einem Format-String für `date_labels` festlegen, der wie in `strptime()` aufgebaut ist (Abschn. 2.15.1).
- `scale_x_continuous()` definiert das Aussehen der x -Achse, wenn dort Werte einer stetigen Variable dargestellt sind.
 - Ein für `limits` angegebener Vektor beschränkt den dargestellten Wertebereich. Die nicht in diesem Bereich liegenden Daten werden in allen weiteren Schichten nicht verwendet, etwa zur Definition von Boxplots oder Regressionsgeraden (*clipping*). Tatsächlich geht der dargestellte Wertebereich zu beiden Seiten etwas über die in

`limits` genannten Werte hinaus. Um die Achsen exakt bei diesen Werten enden zu lassen, kann `expand=c(0, 0)` gesetzt werden. Die Anzahl der automatisch gesetzten Wertemarkierungen orientiert sich an der für `n.breaks` genannten Anzahl. Ein für `breaks` übergebener Vektor definiert dagegen exakt, an welchen Stellen sich Wertemarkierungen befinden.

- Mit `position="top"` wird die *x*-Achse am oberen Diagrammrand gezeichnet.
- Bei einer kategorialen *x*-Achse ist die Reihenfolge der Einträge gleich der Reihenfolge der Stufen der dargestellten Variable, die dafür automatisch in einen Faktor umgewandelt wird. Die Reihenfolge lässt sich deswegen durch Anpassen dieses Faktors (Abschn. 2.6.4) vor der Diagrammerstellung verändern.
- Die Skalierung der *x*-Achse kann auf zwei Wegen transformiert werden, etwa um sie entsprechend des dekadischen Logarithmus zu stauchen. Über das Argument `transform="log10"` von `scale_x_continuous` bleiben die Wertemarkierungen der Achse und vertikalen Gitternetzlinien in ihrer Darstellung gleichabständig, repräsentieren dabei aber Werte, die auf der transformierten Skala gleichabständig sind. Dagegen führt `coord_trans(x="log10")` bei gleicher Lage der Datenpunkte zu gestauchten Wertemarkierungen bzw. vertikalen Gitternetzlinien, die Werte repräsentieren, die auf der nicht transformierten Skala gleichabständig sind. Über weitere vordefinierte Transformationen wie "log2" oder "sqrt" informiert `?scale_x_continuous`.
- `coord_cartesian(xlim=<Vektor>, ylim=<Vektor>)` definiert den sichtbaren Wertebereich über die Argumente `xlim` und `ylim`. Im Unterschied zu `scale_x_continuous()` werden die Datenpunkte dabei nicht von späteren Schichten ausgeschlossen, es wird also ohne *clipping* gezoomt.
- `coord_fixed(ratio=<Zahl>)` definiert das Längenverhältnis einer Einheit der *y*-Achse zu einer Einheit der *x*-Achse. Gleich skalierte Achsen erhält man mit `ratio=1`. Demgegenüber sorgt `theme(aspect.ratio=<Zahl>)` (Abschn. 15.5.5) in der letzten Schicht dafür, dass die Länge der *y*-Achse in einem bestimmten Verhältnis zur Länge der *x*-Achse steht, unabhängig vom dargestellten Wertebereich.
- `coord_flip(xlim=<Vektor>, ylim=<Vektor>)` vertauscht die Rolle von *x*- und *y*-Achse. Vertikale Linien parallel zur *y*-Achse werden so zu horizontalen Linien parallel zur *x*-Achse, etc. Sollen die dargestellten Wertebereiche dabei geändert werden, muss dies innerhalb von `coord_flip()` über die Argumente `xlim` und `ylim` geschehen. Einfacher ist es, die Zuweisung von *x*- und *y*-Achse gleich in der Grundschicht mit `ggplot(..., aes())` passend zu wählen.

Alle `scale_x_<Name>()` Funktionen haben ein Pendant für die *y*-Achse, das entsprechend `scale_y_<Name>()` heißt.

Achsen können mit folgendem Befehl in der letzten Schicht als Pfeile gezeichnet werden:

```
> theme(axis.line=element_line(arrows=arrow(length=unit(0.4, "cm"),
+                                         type="closed")))
```

Das folgende Diagramm zeigt geänderte Achsentitel, eine modifizierte Reihenfolge der *x*-Achsen Kategorien sowie senkrecht orientierte Wertebeschriftungen der *x*-Achse (Abb. 15.11).

```
> levels(myDf$rating)      # aktuelle Reihenfolge der Stufen
[1] "Completely" "Mostly" "Somewhat" "Not at all"

# umgekehrte Reihenfolge der Stufen
> myDf$rating <- ordered(myDf$rating,
+                         levels=rev(levels(myDf$rating)))

> ggplot(myDf, aes(x=rating, group=sex, fill=sex)) +
+     geom_bar(stat="count",
+               aes(y=after_stat(count / sum(count))),
+               position=position_fill()) +
+     labs(x="Rating category", y="Cumulative relative frequency") +
+     guides(x=guide_axis(angle=90))
```

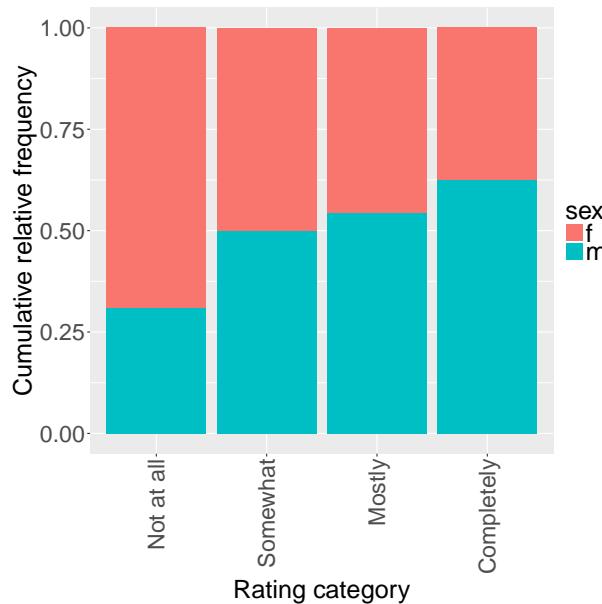


Abbildung 15.11: Angepasste Achsen-eigenschaften mit `scale_x_discrete()`, `labs()` und `guides()`

Das folgende Streudiagramm passt die Gitterabstände für x - und y -Achse an und verwendet senkrecht orientierte Wertebeschriftungen der x -Achse (Abb. 15.12).

```
> ggplot(myDf, aes(x=height, y=mood, colour=sex, shape=group)) +
+     geom_point(size=3) +
+     scale_x_continuous(limits=c(150, 200),
+                        expand=c(0, 0),
+                        breaks=seq(150, 200, by=5)) +
+     scale_y_continuous(n.breaks=8) +
+     guides(x=guide_axis(angle=90))
```

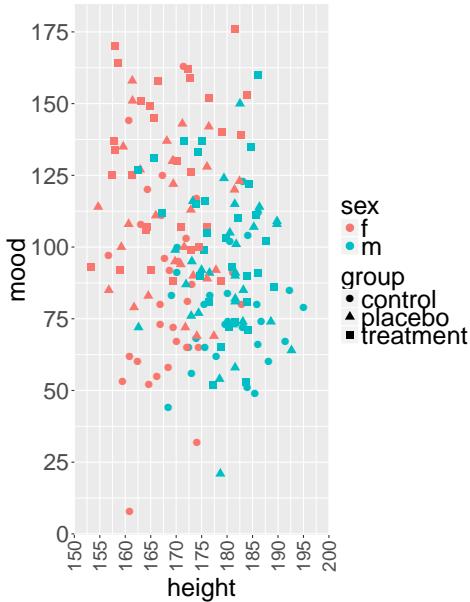


Abbildung 15.12: Angepasste Achseneigenschaften mit `scale_x_continuous()` und `guides()`

15.5.3 Legende ändern

Die automatisch dargestellte Legende lässt sich in vielen Details an eigene Vorgaben anpassen.

- `theme(legend.position="none")` sorgt dafür, dass das Diagramm keine Legende zeigt. Um dagegen die Legende an verschiedenen Seiten zu positionieren, kann `legend.position` auf "left", "right", "bottom", oder "top" gesetzt werden. Eine Legende im Zeichnungsbereich erlaubt `legend.position="inside"` in Kombination mit `legend.position.inside=<Vektor>`, wobei der Vektor die relativen (x, y) -Koordinaten im Bereich 0–1 für die Position der Legende nennt.
- Damit keine Legendeneinträge erstellt werden, die aus einer mit `geom_<Name>()` erzeugten Schicht abgeleitet sind, kann der Aufruf von `geom_<Name>()` um `show.legend=FALSE` ergänzt werden.
- `guides(<Eigenschaft>="none")` entfernt aus der Legende das zu <Eigenschaft> gehörende Element. So sorgt etwa `shape="none"` dafür, dass die Legende keine Einträge zur Art der verwendeten Datenpunktsymbole enthält.
- `guides(<Eigenschaft>=guide_legend(title=NULL))` bewirkt, dass die eine bestimmte Eigenschaft erklärende Legende keinen Titel trägt – in der Voreinstellung der Name der Variable, mit der die Eigenschaft innerhalb von `aes()` verknüpft wurde. Alternativ kann an `title` eine Zeichenkette übergeben werden, um den Titel festzulegen. Kürzer ist dies auch mit `labs(<Eigenschaft>="<Titel>")` möglich. Analog entfernt `labs(<Eigenschaft>=NULL)` den Titel der Legende.

- Die Reihenfolge der Legendeneinträge ist dieselbe wie die Reihenfolge der Stufen des dargestellten Faktors, der innerhalb von `aes()` mit dem zugehörigen visuellen Attribut verknüpft wurde. Die Reihenfolge lässt sich durch Anpassen dieses Faktors (Abschn. 2.6.4) vor Diagrammerstellung verändern. Im Diagramm ist es mit `guides(<Eigenschaft>=guide_legend(rev=TRUE))` möglich, die Reihenfolge umzukehren.
- Formatungsdetails der Legende steuert `theme()`, etwa Schriftart und -größe oder die Farbe der Rahmen, s. in `?theme` die mit `legend.` beginnenden Einträge.

Nun wird die Legende am unteren Diagrammrand positioniert, der Titel der Farblegende geändert und der überflüssige Legendeneintrag entfernt, der die Zuordnung von Geschlecht zu Datenpunktssymbolen erläutert (Abb. 15.13).

```
> ggplot(myDf, aes(x=height, y=mood, colour=sex:group, shape=sex)) +
+   geom_vline(aes(xintercept=180), linetype=2) +
+   geom_point(size=3) +
+   geom_smooth(method="lm", se=TRUE, linewidth=1, fullrange=TRUE) +
+   facet_grid(sex ~ group) +
+   labs(title="mood ~ height getrennt nach Geschlecht + Gruppe") +
+   geom_text(aes(x=190, y=70, label=sgComb), show.legend=FALSE) +
+   annotate("text", x=165, y=5, label="Annotation") +
+   guides(shape="none",
+         colour=guide_legend(title="Sex/Group")) +
+   theme(legend.position="bottom")
```

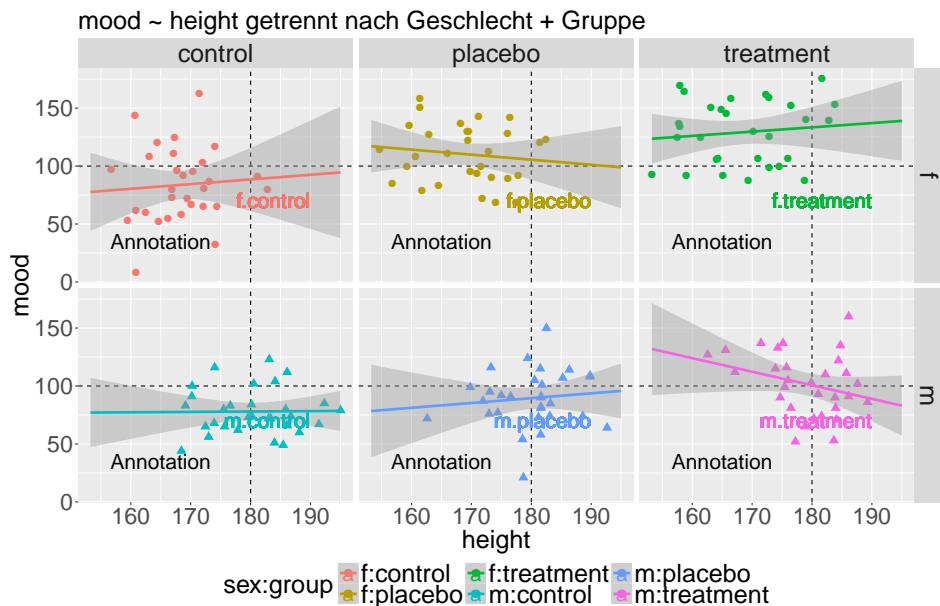


Abbildung 15.13: Angepasste Position und Einträge der Legende mit `guides()` sowie `theme()`

15.5.4 Farben, Datenpunktsymbole und Linientypen

Farben, Datenpunktsymbole und Linientypen lassen sich auf zwei Wegen wählen: Entweder werden diese visuellen Attribute im Aufruf von `ggplot()` bzw. von `geom_<Name>()` innerhalb von `aes()` mit Variablen eines Datensatzes verknüpft, oder aber außerhalb von `aes()` auf einen festen Wert gesetzt.

- Für Farben existiert das Argument `colour` (für mögliche Werte s. Abschn. 14.3.2). Zusätzlich zu `colour` kann `alpha` mit einer Zahl im Bereich von 0–1 den Grad der simulierten Transparenz festlegen. Dies kann etwa sinnvoll sein, wenn Diagrammelemente übereinander plaziert werden sollen, ohne dass später gezeichnete Elemente früher gezeichnete vollständig verdecken.
- Die Datenpunktsymbole eines Punktdiagramms bestimmt `shape` (für mögliche Werte s. Abschn. 14.3.1).
- Für Linientypen eines Liniendiagramms ist `linetype` vorgesehen (mögliche Werte sind "solid", "dashed", "dotted" oder die in Abschn. 14.3.1 genannten Zahlen).
- Die Größe der Datenpunkte gibt `size` vor, die Dicke der Umrisse nicht ausgefüllter Formen `stroke` und die Linienbreite `linewidth`.

Bei der Zuordnung von Farben zu Variablen existiert eine große Vielfalt von Gestaltungsmöglichkeiten, die sich aus dem möglichen Austausch der verwendeten Farbpaletten ergibt. Dazu fügt man der Grundschicht eine der folgenden Schichten mit + hinzu:

- `scale_colour_hue(h=<Farbton>, c=<Sättigung>, l=<Helligkeit>)` ist die Voreinstellung und sorgt für annähernd gleich helle Farben unterschiedlichen Farbtönen. Die Farben lassen sich bzgl. des abgedeckten Farbbereichs (Vektor `h` mit zwei Winkeln im Farbkreis 0–360), der vom Farbton abhängigen Sättigung `c` und der Helligkeit `l` (0–100) modifizieren. Andere Farbgradienten erzeugt `scale_colour_gradient()`.
- `scale_colour_grey(start=0.2, end=0.8)` stellt Linien in Graustufen im Bereich von `start` bis `end` dar, wobei der mögliche Wertebereich von 0–1 reicht.
- Das im Basisumfang von R enthaltene Paket `colorspace` verfügt über eine Reihe von Farbpaletten, die für unterschiedliche Zwecke optimiert sind – etwa zur Visualisierung stetiger Übergänge, zur besseren Unterscheidbarkeit unterschiedlicher Gruppen oder für Ausdrucke in Graustufen. Zu diesen Paletten wechseln Funktionen, deren Namen analog zu `scale_color_continuous_sequential()` oder `scale_fill_discrete_qualitative()` aufgebaut sind. Das Argument `palette` erlaubt es dabei, zwischen mehreren Varianten zu wählen.¹¹

Alle Funktionen nach dem Muster `scale_color_<...>()` legen Farben für Linien und Umrisse fest. Die Farbe gefüllter Flächen wird analog über die zugehörigen Funktionen `scale_fill_<...>()` definiert. Nun soll die Größe der Datenpunktsymbole sowie der Linientyp angepasst werden (Abb. 15.14).

¹¹https://colorspace.r-forge.r-project.org/articles/ggplot2_color_scales.html zeigt Details und Beispiele.

```
> ggplot(groupM, aes(x=group, y=mood, color=sex, shape=sex, group=sex)) +
+   geom_point(size=8, stroke=2) +
+   geom_line(linewidth=2, linetype="dashed") +
+   scale_shape_discrete(solid=FALSE)
```

Das folgende nach Gruppen getrennte Histogramm mit Kerndichteschätzer verwendet eine der `colorspace` Farbpaletten und setzt simulierte Transparenz für eine bessere Sichtbarkeit der einzelnen Flächen in sich überlappenden Regionen ein (Abb. 15.14).

```
> library(colorspace)           # für scale_fill_discrete_qualitative()
> ggplot(myDf, aes(x=mood, fill=group)) +
+   geom_histogram(aes(y=after_stat(density)), alpha=0.5) +
+   geom_density(alpha=0.7) +
+   scale_fill_discrete_qualitative()
```

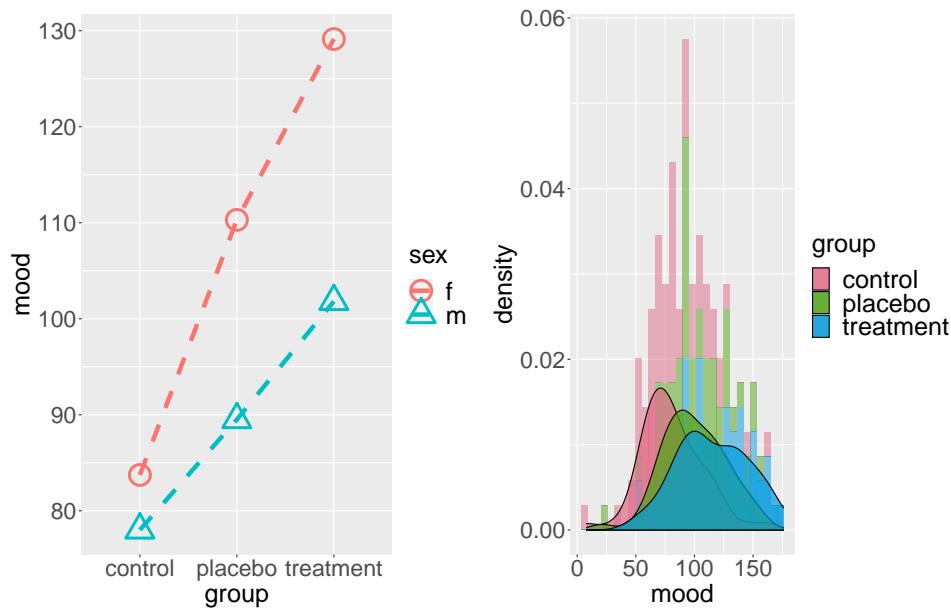


Abbildung 15.14: Angepasste Datenpunktssymbole, Linientypen und Farben mit `scale_shape_discrete()` und `scale_fill_discrete_qualitative()`

15.5.5 Aussehen im Detail verändern

Um Details im Aussehen von ggplot2-Diagrammen zu ändern, kann man auf zwei Wegen vorgehen: Eine mit `+` zu einer Grundschicht hinzugefügte `theme_<Name>()` Schicht verändert gleichzeitig viele Einstellungen, die den grundsätzlichen Diagrammstil bestimmen.¹² Alle folgenden Funktionen verfügen insbesondere über das Argument `base_size=<Schriftgröße>`, um die Basis-Schriftgröße festzulegen. Voreinstellung sind 11 Punkte.

¹²Weitere stellt das Paket `ggthemes` (Arnold, 2019) zur Verfügung.

- `theme_grey()` ist die Voreinstellung und sorgt für einen leicht grauen Hintergrund der ohne Rahmen dargestellten Zeichnungsfläche, auf der dünne weiße Gitterlinien zu sehen sind.
- `theme_bw()` setzt den Hintergrund der Zeichnungsfläche auf weiß und rahmt sie mit einer dünnen schwarzen Linie ein. Die Gitterlinien sind grau.
- `theme_minimal()` entfernt gegenüber `theme_bw()` den Rahmen um die Zeichnungsfläche und um Elemente der Legende. Linien für die Achsen fehlen ebenfalls.
- `theme_classic()` entfernt gegenüber `theme_minimal()` noch alle Gitterlinien.

Eine Reihe von optischen Details wie Schriftgröße, -art und -farbe des Titels, der Achsenbeschriftungen und der Legende lassen sich zudem auch einzeln mit `theme()` verändern. Diese Funktion kontrolliert u. a. auch die Randgrößen sowie die Farbe der Panel-Beschriftungen und Gitterlinien. Alle Einstellmöglichkeiten erläutert `?theme`.

Das in Abb. 15.15 gezeigte Diagramm reduziert die visuellen Elemente von Abb. 15.14 aus Abschn. 15.5.4.

```
> library(colorspace)          # für scale_fill_discrete_qualitative()
> ggplot(myDf, aes(x=mood, fill=group)) +
+   geom_histogram(aes(y=after_stat(density)), alpha=0.5) +
+   geom_density(alpha=0.7) +
+   scale_fill_discrete_qualitative() +
+   theme_bw()
```

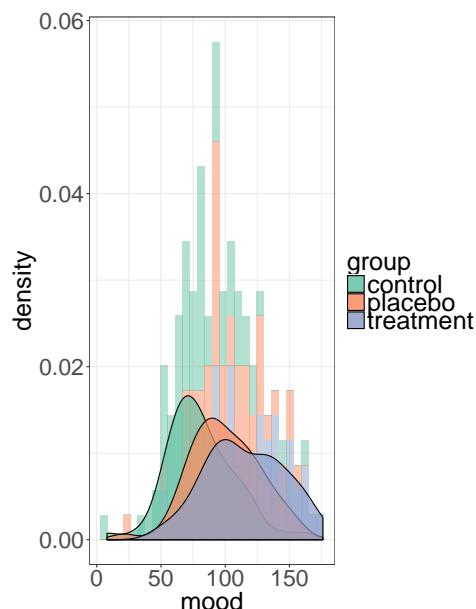


Abbildung 15.15: Reduzierte Diagrammelemente mit `theme_bw()`

Kapitel 16

Numerische Methoden

Numerische Methoden spielen in der Datenanalyse u. a. deswegen eine wichtige Rolle, weil nur in Spezialfällen geschlossene Formeln für die Parameterschätzung existieren, die zur bestmöglichen Passung eines statistischen Modells für beobachtete Daten führt. Der Einsatz numerischer Methoden bleibt dem Anwender aber verborgen, weil die eingesetzten Funktionen zur Modellanpassung zwar intern typischerweise solche Methoden verwenden, sie die gewählten Algorithmen dem Anwender aber nicht unmittelbar offen legen.

Gemein ist allen numerischen Methoden, dass die von ihnen bestimmten Ergebnisse anders als analytische Lösungen von begrenzter numerischer Genauigkeit und auch nicht immer optimal sind. Auch ist es möglich, dass gar keine Lösungen gefunden werden, obwohl sie eigentlich existieren. Um numerische Methoden in eigenen maßgeschneiderten Auswertungen einsetzen zu können, sollte man sich daher auch mit den technischen Hintergründen vertraut machen. Details beschreiben [Bloomfield \(2014\)](#) und [Nash \(2014a\)](#), Hinweise auf relevante Zusatzpakete liefert der Abschnitt *Numerical Mathematics* der CRAN Task Views ([Borchers, 2019a](#)).¹

In den folgenden Abschnitten müssen an vielen Stellen selbst Funktionen definiert werden. Die dafür notwendigen Grundlagen vermittelt Abschn. 17.3.

16.1 Daten interpolieren und glätten

R verfügt über verschiedene vorbereitete Möglichkeiten, zwischen gegebenen Datenpunkten zu interpolieren und Daten zu fitten. Hierbei werden Datenpunkte generiert, die horizontal und vertikal zwischen den bestehenden plaziert sind und diese i. S. einer zugrundeliegenden Funktion ergänzen. Für Möglichkeiten zum Anpassen verschiedener Funktionsfamilien s. Abschn. 16.4.1.

16.1.1 Lineare Interpolation

`approx()` interpoliert konstant oder linear zwischen Datenpunkten (Abb. 16.1).

```
approx(x=<x-Koordinaten>, y=<y-Koordinaten>,
       method="<Interpolationsmethode>", n=<Anzahl>)
```

¹Wichtige numerische Methoden für die lineare Algebra erläutern Abschn. 12.1.2 und 12.1.6. Für Pakete zur Bearbeitung der hier nicht besprochenen gewöhnlichen oder partiellen Differentialgleichungen vgl. den entsprechenden Abschnitt *Differential Equations* ([Soetaert & Petzoldt, 2020](#)) der CRAN Task Views.

Die (x, y) -Koordinaten der Datenpunkte sind für x und y in Form von Vektoren derselben Länge zu übergeben. Mit dem Argument `method` wird kontrolliert, wie die Interpolation erfolgt – "linear" bewirkt eine lineare Interpolation. Mit "constant" erhalten alle interpolierten Punkte die y -Koordinate des in horizontaler Richtung vorangehenden Datenpunkts. Wie viele Punkte der interpolierten Funktion erzeugt werden sollen, bestimmt das Argument `n`. Die Ausgabe von `approx()` besteht aus einer Liste mit den Komponenten `x` und `y`, den Vektoren der (x, y) -Koordinaten der interpolierten Punkte.

```
> xOne      <- 1:9                                # x-Koordinaten
> yOne      <- rnorm(9)                            # y-Koordinaten
> ptsLin    <- approx(xOne, yOne, method="linear") # linear
> ptsConst <- approx(xOne, yOne, method="constant")# konstant

> plot(xOne, yOne, pch=19, cex=3, main="Datenpunkte interpolieren")
> points(ptsLin, pch=16, col="red")
> points(ptsConst, pch=22, col="blue", lwd=2)
> legend(x="topright", c("Daten", "linear", "konstant"),
+         pch=c(19, 16, 22), col=c("black", "red", "blue"))
```

`approxfun()` gibt stattdessen eine Funktion zurück, die x -Koordinaten akzeptiert und y -Koordinaten ausgibt, so dass die ursprünglichen Daten interpoliert werden. `approxfun()` besitzt dieselben Argumente wie `approx()` bis auf die nicht benötigte Angabe `n`

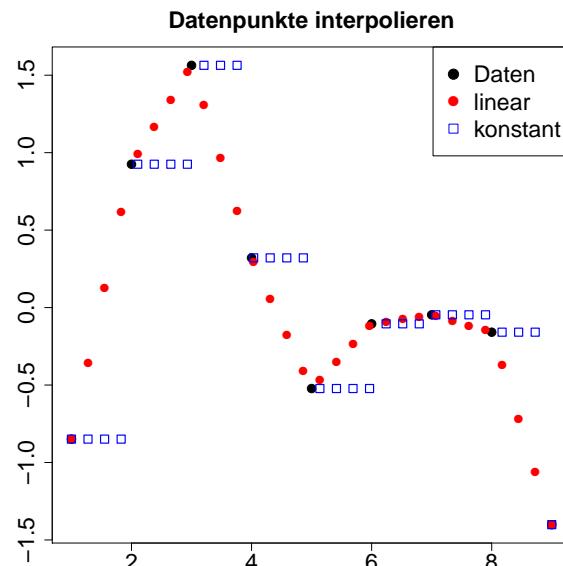


Abbildung 16.1: Lineare und konstante Interpolation von Daten

16.1.2 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 bekannt ist. Kubische Splines² können mit `spline()` erzeugt werden: Die ermittelten Punkte liegen auf einer Kurve durch alle vorhandenen Datenpunkte. `spline()` gibt eine Liste mit den (x, y) -Koordinaten in den Komponenten `x` und `y` aus. `splinefun()` gibt stattdessen eine Funktion zurück, die x -Koordinaten akzeptiert und y -Koordinaten ausgibt, so dass die ursprünglichen Daten interpoliert werden. `splinefun()` besitzt dieselben Argumente wie `spline()` bis auf die nicht benötigte Angabe `n`.

Die von `smooth.spline()` berechnete Kurve führt nicht direkt durch die übergebenen Datenpunkte. Über das Argument `spar` (*smoothing parameter*) kann die Glattheit der Kurve, d. h. das Ausmaß ihrer Variation kontrolliert werden. Dieser Parameter bestimmt auch, wie nah die Kurve an den Datenpunkten liegt. Das Ergebnis von `smooth.spline()` ist ein Objekt, mit dem über `predict()` die interpolierten y -Koordinaten für neue x -Koordinaten erzeugt werden können (Abb. 16.2).

```
> 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=0.35) # 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="topright", c("data", "Spline", "spar=0.25", "spar=0.35",
+                         "spar=0.45"), pch=c(19, NA, NA, NA, NA), lty=c(NA, 1, 1, 1, 1),
+           col=c("black", "darkgray", "blue", "green", "orange"))
```

`xspline()` stellt weitere Splines bereit, mit denen nicht nur Funktionswerte interpoliert, sondern allgemein über *Kontrollpunkte* beliebige, auch geschlossene und sich überschneidende Formen angenähert werden können (Abb. 16.2). Sofern gewünscht zeichnet `xspline()` das Ergebnis direkt in ein Diagramm ein.

```
> xTwo <- rnorm(100) # x-Koordinaten
> yTwo <- 0.4 * xTwo + rnorm(100, 0, 1) # y-Koordinaten
```

²Für weitere Spline-Typen, etwa monotone Splines, vgl. `help(package="splines")`. Auch das Paket `interp` (Gebhardt, Bivand & Sinclair, 2024) implementiert verschiedene Splines zur glatten Interpolation zweidimensionaler Daten. Das Paket `MBA` (Finley, Banerjee & Hjelle, 2024) liefert Methoden zur Spline-Interpolation dreidimensionaler Daten.

```
> ord <- order(xTwo) # für geordnete x-Koordinaten
> idx <- seq(8, 88, by=20) # wähle 4 Kontrollpunkte
> xspline(xTwo[ord][idx], yTwo[ord][idx], c(1, -1, -1, 1, 1),
+           border="darkgreen", lwd=2, open=FALSE)
```

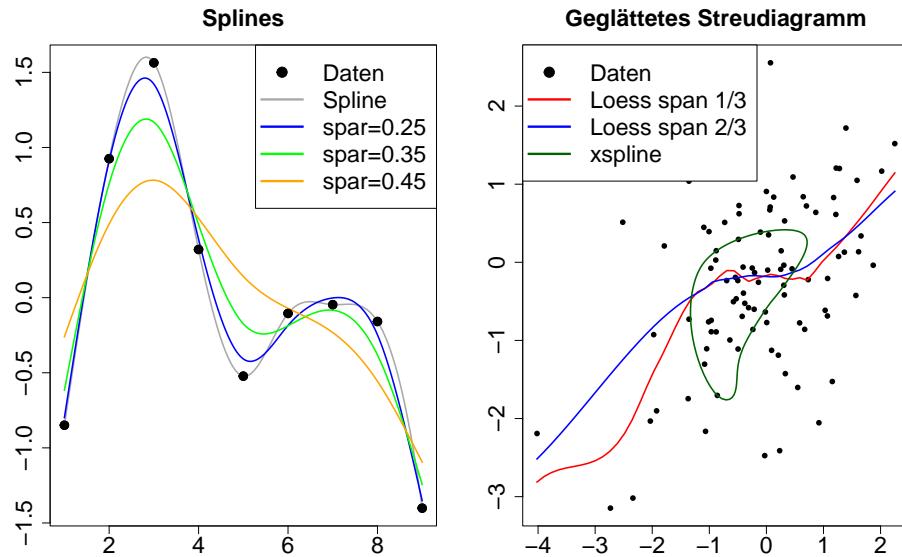


Abbildung 16.2: Unterschiedlich glatte Splines und polynomiale LOESS Glätter

16.1.3 LOESS-Glättter

Um (x, y) -Streudiagramme nonparametrisch durch glatte Kurven zu approximieren, existieren verschiedene Glätter, die auf lokal gewichteten Polynomen basieren – darunter die in `loess()` und `loess.smooth()` implementierten (Abb. 16.2).³

```
loess(<Modellformel>, data=<Datensatz>, span=<Bandbreite>)
loess.smooth(x=<x-Koordinaten>, y=<y-Koordinaten>, span=<Bandbreite>)
```

Die (x, y) -Koordinaten der Datenpunkte sind in `loess()` als Modellformel $y \sim x$ zu spezifizieren, wobei ggf. der die Variablen enthaltende Datensatz für `data` genannt werden kann. Das Ergebnis von `loess()` ist ein Objekt, mit dem über `predict()` die interpolierten y -Koordinaten für neue x -Koordinaten erzeugt werden können.

In `loess.smooth()` lassen sich die Koordinaten für `x` und `y` in Form von Vektoren derselben Länge übergeben. Das Argument `span` gibt die Bandbreite des Glätters als Anteil der Punkte an, deren Position in jedem geglätteten Wert berücksichtigt wird. Größere Anteile bewirken dabei glattere Interpolationen (Voreinstellung ist $\frac{2}{3}$). Als Ergebnis gibt `loess.smooth()` eine Liste mit den Komponenten `x` und `y` zurück, den Vektoren der (x, y) -Koordinaten der interpolierten Punkte.

³Das im Basisumfang enthaltene Paket `KernSmooth` (Wand, 2019) stellt mit `locpoly()` eine weitere Alternative bereit, deren optimale Bandbreite über `dpline()` aus demselben Paket berechnet werden kann.

```
> 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")

> lines(ptsL1, lwd=2, col="red")                  # LOESS-Glätter einzeichnen
> lines(ptsL2, lwd=2, col="blue")

# Legende einzeichnen
> 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"))
```

Das folgende Beispiel demonstriert, wie in `loess()` über die Modellformel die Glättung dreidimensionaler Daten als zweidimensionale Oberfläche spezifiziert wird (Abbildung 16.3).

```
> zTwo      <- -0.5 * xTwo + rnorm(100, 0, 0.5)  # z-Koordinaten
> d_xyz     <- data.frame(x=xTwo, y=yTwo, z=zTwo) # Datensatz Koord
> loess_xyz <- loess(z ~ x + y, data=d_xyz)      # 3D Glättung
```

Zur Darstellung mit dem Paket `rgl` (Abschn. 14.7.2) wird ein Datensatz mit (x, y) -Gitterkoordinaten definiert, für die `predict()` dann als Vorhersage eine Matrix der z -Koordinaten auf Basis der glatten Oberfläche erzeugt.

```
> library(rgl)                                     # für plot3d(), surface3d()

# definiere Gitter von (x,y)-Koordinaten als Datensatz
> x_grid <- seq(from=min(xTwo), to=max(xTwo), length.out=50)
> y_grid <- seq(from=min(yTwo), to=max(yTwo), length.out=50)
> d_grid <- expand.grid(x=x_grid, y=y_grid)

# erstelle Vorhersage der glatten Oberfläche für Gitter
> d_loess_pred <- predict(loess_xyz, newdata=d_grid, se=TRUE)

# stelle Box Punkte und glatte Oberfläche interaktiv dar
> plot3d(xTwo, yTwo, zTwo, type="s", size=0.75, lit=FALSE, col="red")
> points3d(xTwo, yTwo, zTwo, size=5, col="red")
> surface3d(x_grid, y_grid, d_loess_pred[[1]],
+             alpha=0.4, front="lines", back="lines")
```

16.1.4 Nonparametrische Kerndichteschätzer

Histogramme zur Veranschaulichung einer empirischen Verteilung von Werten bergen den Nachteil, dass ihre Form stark von der willkürlichen Wahl der Anzahl der Klassen abhängen kann (Abschn. 14.6.1). Abhilfe schaffen hier Kerndichteschätzer, die eine stetige Verteilungsschätzung ermöglichen. Für den eindimensionalen Fall eignet sich die Funktion `density()`, deren Schätzung darauf basiert, die Daten mit einem Glättungskern zu falten:

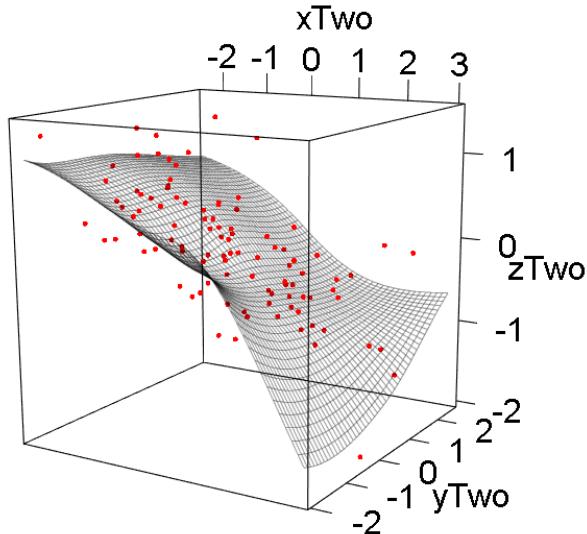


Abbildung 16.3: LOESS Glättung dreidimensionaler Daten als zweidimensionale Oberfläche

```
density(x=<Vektor>, bw=<'Bandbreite'>, kernel=<'Kern'>, n=512, ...)
```

Neben den Daten in Form eines Vektors `x` ist der Faltungskern `kernel` zu wählen. Die bekanntesten Glättungskerne sind die Dichtefunktion der Normalverteilung ("`gaussian`") und der Epanechikov-Kern ("`epanechikov`"). Die Option `bw` zur Wahl der Bandbreite sollte abweichend von der historisch bedingten Voreinstellung auf "`SJ`" gesetzt werden. Die Dichteschätzung wird an `n` vielen gleichabständigen Stützstellen ausgewertet. Ein Beispiel zeigt Abschn. 14.6.1.

Aus dem Paket `KernSmooth` stammt `bkde2d()` zur zweidimensionalen Kerndichteschätzung, die sich mit `smoothScatter()` als Diagramm darstellen lässt.

16.2 Nullstellen finden

Für eine gegebene streng monotone Funktion f findet `uniroot()` die Nullstelle (`root`) in einem vorher festgelegten Intervall. Details erläutert Nash (2014a).

```
uniroot(f=<Funktion>, interval=<Intervall>,
        extendInt=c("no", "yes", "downX", "upX"), ...)
```

Zunächst sind die Funktion `f` und ein Vektor mit zwei Elementen `interval` anzugeben, der das abzusuchende Intervall definiert. `uniroot()` ruft `f` wiederholt mit einzelnen Werten aus `interval` für das erste Argument von `f` auf, `f` muss also nicht vektorisiert sein. Für den Fall, dass die Nullstelle nicht in diesem Intervall gefunden wird, kann `uniroot()` über das Argument `extendInt` das Intervall eigenständig schrittweise erweitern – nach unten ("`downX`"), nach oben ("`upX`"), oder in beide Richtungen ("`yes`"). Weitere Argumente für `f` lassen sich anstelle der `...` nennen. Die zurückgegebene Liste enthält die Nullstelle in der Komponente `root`.

Nützlich ist die Nullstellensuche u. a. dafür, die Umkehrfunktion F^{-1} einer gegebenen Funktion F numerisch zu bestimmen. In statistischen Anwendungen ist F dabei häufig eine Verteilungsfunktion, deren Umkehrfunktion (die Quantilfunktion) unbekannt oder nicht in geschlossener Form darstellbar ist. Für eine gegebene Wahrscheinlichkeit p ist also das Quantil $q = F^{-1}(p)$ gesucht, für das $F(q) = p$ gilt. Anders formuliert gilt $F(q) - p = 0$, q ist also die gesuchte Nullstelle der Funktion $f(x) = F(x) - p$.

Als Beispiel sei die Verteilungsfunktion der Hoyt-Verteilung betrachtet. Für einen zufällig gezogenen Vektor \mathbf{x} einer zweidimensionalen Normalverteilung $\mathcal{N}_2(\mathbf{0}, \Sigma)$ gibt sie die Wahrscheinlichkeit p dafür an, dass seine euklidische Länge einen gegebenen Wert q nicht überschreitet, also $P(\|\mathbf{x}\| \leq q)$. Sind λ_1, λ_2 die absteigend geordneten Eigenwerte der Kovarianzmatrix Σ , wird die Hoyt-Verteilung typischerweise spezifiziert über die Parameter $q = 1/\sqrt{((\lambda_1 + \lambda_2)/\lambda_2) - 1}$ (Elliptizität, $q \in (0, 1)$) und $\omega = \lambda_1 + \lambda_2$ (Gesamtvarianz i. S. der Spur von Σ , $\omega > 0$).

```
# Verteilungsfunktion der Hoyt-Verteilung
> pHoyt <- function(q, qpar, omega) {
+   alphaQ <- (sqrt((1-qpar^4))/(2*qpar)) * sqrt((1+qpar)/(1-qpar))
+   betaQ <- (sqrt((1-qpar^4))/(2*qpar)) * sqrt((1-qpar)/(1+qpar))
+   y <- q / sqrt(omega)
+   pchisq((alphaQ*y)^2, df=2, ncp=(betaQ*y)^2) -
+   pchisq((betaQ*y)^2, df=2, ncp=(alphaQ*y)^2)
+ }

# streng monotone Funktion, deren Nullstelle zu finden ist
> f <- function(x, p, qpar, omega) {
+   pHoyt(x, qpar=qpar, omega=omega) - p
+ }

# Quantilfunktion der Hoyt-Verteilung
> qHoyt <- function(p, qpar, omega) {
+   uniroot(f, interval=c(0, omega), extendInt="upX",
+           p=p, qpar=qpar, omega=omega)$root
+ }

> qHoyt(p=0.7, qpar=0.5, omega=10)
[1] 3.351123
```

Eine weitere Anwendung besteht darin, über die Inversionsmethode Zufallszahlen aus einer Verteilung mit gegebener Verteilungsfunktion F zu ziehen. Im ersten Schritt sind dafür mit `runif()` die Wahrscheinlichkeiten p_i als gleichverteilte Zufallszahlen im Intervall $[0, 1]$ zu ziehen. Die Zufallszahlen mit der gewünschten Verteilung ergeben sich dann als $F^{-1}(p_i)$, also als Nullstellen von $F(x) - p_i$. Da die oben definierte Quantilfunktion `qHoyt()` nicht vektorisiert ist, wird hier `sapply()` verwendet, um zu den zunächst gezogenen zufälligen Wahrscheinlichkeiten die zugehörigen Quantile zu finden.

```
> U <- runif(1000)      # gleichverteilte Wahrscheinlichkeiten
# Hoyt-verteilte Zufallszahlen
```

```
> rh <- sapply(U, function(x) { qHoyt(p=x, qpar=0.5, omega=10) })

# Kontrolle: kumulierte relative Häufigkeiten der Zufallszahlen
# vs. Verteilungsfunktion der Hoyt-Verteilung (hier nicht gezeigt)
> plot(ecdf(rh), col="blue")
> curve(pHoyt(x, qpar=0.5, omega=10), from=0, to=10, add=TRUE)
```

Die Nullstellen von Polynomen bestimmt `polyroot()`. Um die Nullstellen auch nicht-monotoner Funktionen zu bestimmen, ist das Paket `rootSolve` ([Soetaert, 2019](#)) geeignet.

16.3 Integrieren und differenzieren

16.3.1 Numerisch integrieren

Zur numerischen Integration wird eine Funktion wiederholt an Stützstellen ausgewertet, um z. B. mit einer Variante der Trapezregel die Fläche unter der Funktionskurve zwischen gegebenen Grenzen zu bestimmen. Im Basisumfang von R ist dafür `integrate()` vorhanden.

```
integrate(f=<Funktion>, lower=<Grenze unten>, upper=<Grenze oben>, ...)
```

Als erstes Argument ist eine vektorisierte Funktion `f` zu übergeben, die aus einem Vektor von Eingangswerten einen Vektor von Funktionswerten berechnen können muss. Da `f` nur an endlich vielen Stützstellen ausgewertet wird, ist es für korrekte Ergebnisse notwendig, dass `f` weitgehend glatt ist. Die Integrationsgrenzen sind für `lower` und `upper` zu nennen, wobei diese ggf. auf `-Inf` bzw. `Inf` gesetzt werden können. Benötigt `f` weitere Argumente, lassen sich diese anstelle der `...` nennen. Die zurückgegebene Liste enthält den berechneten Wert für das Integral in der Komponente `value`.

Als Beispiel sei zunächst das Integral der Normalverteilung $\mathcal{N}(\mu = 1, \sigma = 2)$ im Intervall $(-\infty, 1)$ betrachtet.

```
> integrate(dnorm, lower=-Inf, upper=1, mean=1, sd=2)$value
[1] 0.5

> pnorm(1, mean=1, sd=2)                                # Kontrolle
[1] 0.5
```

Die in Abschn. [16.2](#) angewendete Inversionsmethode zum Ziehen von Zufallszahlen aus einer Verteilung ohne bekannte Quantilfunktion soll nun auf Fälle verallgemeinert werden, in denen auch keine Formel für die Verteilungsfunktion F bekannt ist und nur die Dichtefunktion in geschlossener Form vorliegt. Beispiel sei die Normalverteilung.⁴

⁴Die hier selbst definierten Funktionen dienen nur als Illustration des allgemeinen Prinzips. Zu ihren Nachteilen gegenüber den R-eigenen Funktionen `{d,p,q}norm()` zählt, dass sie die übergebenen Argumente nicht auf zulässige Werte prüfen und zudem weit weniger genau sind. So funktioniert `pGauss()` nur für betragsmäßig kleine μ und kleine obere Integrationsgrenzen.

```
# Dichtefunktion der Normalverteilung - naive Implementierung
> dGauss <- function(x, mu=0, sigma=1) {
+   (1/(sigma*sqrt(2*pi))) * exp(-0.5 * ((x-mu)/sigma)^2)
+ }

# Verteilungsfunktion Normalverteilung als Integral der Dichtefunktion
> pGauss <- function(x, mu=0, sigma=1) {
+   integrate(dGauss, lower=-Inf, upper=x, mu=mu, sigma=sigma)$value
+ }
```

Wie in Abschn. 16.2 ist nun die streng monotone Funktion $f(x) = F(x) - p$ zu definieren, deren Nullstelle in der Quantilfunktion über `uniroot()` gefunden werden soll.

```
> f <- function(x, p, mu=0, sigma=1) {
+   pGauss(x, mu=mu, sigma=sigma) - p
+ }

# Quantilfunktion der Normalverteilung
> qGauss <- function(u, mu=0, sigma=1) {
+   interval <- c(mu-10*sigma, mu+10*sigma)
+   uniroot(f, interval=interval, extendInt="yes",
+           p=u, mu=mu, sigma=sigma)$root
+ }

> U <- runif(5)                      # gleichverteilte zufällige W'keit
> sapply(U, qGauss, mu=0, sigma=1)    # zufällige Quantile
[1] -1.4179136 1.5202963 1.2529418 -0.1385798 -0.1990648

> qnorm(U, mean=0, sd=1)              # Kontrolle -> leichte Abweichungen
[1] -1.4179197 1.5203254 1.2529324 -0.1386019 -0.1990879
```

Unterschiedliche Varianten der Gauß-Quadratur stellt das Paket `statmod` (Smyth, 2020) in der Funktion `gauss.quad()` bereit. Numerische Integration über quadratischen Flächen und ihre höherdimensionalen Verallgemeinerungen setzt das Paket `cubature` (Balasubramanian, Johnson, Hahn, Bouvier & Kieu, 2020) um. Mit `polyCub` (S. Meyer & Held, 2014) lässt sich über beliebige zweidimensionale Polygone integrieren.

16.3.2 Numerisch differenzieren

Die Ableitung einer Funktion kann für viele statistische Anwendungen hilfreich sein. Ohne symbolische Lösung dafür können Funktionen numerisch differenziert werden, wofür sich das Paket `numDeriv` (Gilbert & Varadhan, 2019) mit der Funktion `grad()` eignet. Detaillierte Hintergründe liefert Nash (2014a).⁵

```
grad(func=<Funktion>, x=<Stützstelle(n)>, ...)
```

⁵Mit `D()` lässt sich die Ableitung bestimmter Funktionen auch symbolisch ermitteln.

Als erstes Argument ist die zu differenzierende Funktion `func` zu nennen. Diese muss als erstes Argument einen Vektor von Werten akzeptieren, den `x` definiert. Ist `func` eine Abbildung $f : \mathbb{R} \rightarrow \mathbb{R}$, wird der Wert ihrer ersten Ableitung an jeder der Stützstellen `x` berechnet. Bildet `func` dagegen als Abbildung $f : \mathbb{R}^n \rightarrow \mathbb{R}$ einen Vektor von Werten auf eine einzelne Zahl ab, ist das Ergebnis der *Gradient* ∇f , also der Vektor der partiellen ersten Ableitungen an der Stelle `x`. Benötigt `func` weitere Argumente, können diese anstelle der `...` übergeben werden – sie dürfen aber nicht ihrerseits `x` heißen.

Als Beispiel sei die erste Ableitung der Verteilungsfunktion der Normalverteilung betrachtet, also die zugehörige Dichtefunktion.

```
> library(numDeriv)                      # für grad()
> x <- seq(-2, 2, length.out=5)          # Stützstellen
> grad(pnorm, x, mean=0, sd=1)           # Ableitung Verteilungsfunktion
[1] 0.05399097 0.24197072 0.39894228 0.24197072 0.05399097

> dnorm(x, mean=0, sd=1)                 # Kontrolle: Dichtefunktion
[1] 0.05399097 0.24197072 0.39894228 0.24197072 0.05399097
```

Ebenso soll kontrolliert werden, dass die Ableitung der Exponentialfunktion gleich der Exponentialfunktion ist.

```
> grad(exp, x)
[1] 0.1353353 0.3678794 1.0000000 2.7182818 7.3890561

> exp(x)                                # Kontrolle
[1] 0.1353353 0.3678794 1.0000000 2.7182818 7.3890561
```

Als Anwendung soll mit Hilfe der Delta-Methode der approximative Standardfehler einer nicht-linearen Transformation f von geschätzten Parametern $\hat{\beta} = (\hat{\beta}_0, \hat{\beta}_1)^\top$ einer Poisson-Regression (Abschn. 8.4.1) bestimmt und daraus ein Konfidenzintervall berechnet werden.

```
> N <- 100                                 # Anzahl Beobachtungen
> X <- rnorm(N, 0, 2)                      # Prädiktor
> mu <- exp(1 + 0.5*X)                     # Erwartungswert Poisson-Rate
> Y <- rpois(N, mu)                         # zufällige Zähldaten

# Anpassung Poisson-Regression
> glmFit <- glm(Y ~ X, family=poisson(link="log"))

# Spaltenvektor der Maximum-Likelihood-Schätzer der Parameter
> (bML <- cbind(coef(glmFit)))
      [,1]
(Intercept) 0.9700897
X           0.5077277
```

Ziel ist es, ein Konfidenzintervall für den Schätzer $f(\hat{\beta}) = \hat{\beta}_0/\hat{\beta}_1$ zu finden. Mit dem Gradienten ∇f an der Stelle $\hat{\beta}$ und der geschätzten Kovarianzmatrix \mathbf{S} von $\hat{\beta}$ gilt für den ap-

proximativen Standardfehler $\text{SE}(f(\hat{\beta})) = \sqrt{\nabla f(\beta)^\top \cdot S \cdot \nabla f(\hat{\beta})}$. Dabei erhält man S mit `vcov(glm-Fit)`.

```
# nichtlineare Transformation, die Wertevektor auf 1 Zahl abbildet
> f <- function(b) { b[1] / b[2] }
> f(bML)                                # Wert für beobachtete ML-Schätzung
[1] 1.91065

> (bGrad <- grad(f, bML))            # Gradient
[1] 1.969560 -3.763139

# approximativer Standardfehler aus Delta-Methode
> (SEdelta <- sqrt(t(bGrad) %*% vcov(glmFit) %*% bGrad))
[1,] 0.2193467

# zugehöriges Konfidenzintervall
> (CIdelta <- c(lo=f(bML) - c(qnorm(1-(0.05/2))*SEdelta),
+               up=f(bML) + c(qnorm(1-(0.05/2))*SEdelta)))
      lo        up
1.480738 2.340561
```

In vielen statistischen Modellen lässt sich die Kovarianzmatrix der Parameterschätzer durch die Inverse der Hesse-Matrix H schätzen. Dies ist die Matrix der zweiten partiellen Ableitungen der negativen log-likelihood der beobachteten Daten für die Maximum-Likelihood-Schätzung der Parameter. H lässt sich mit `hessian()` aus dem Paket `numDeriv` berechnen.

```
hessian(func=<Funktion>, x=<Spaltenvektor>, ...)
```

Die Funktion `func` muss als erstes Argument einen Vektor von Parametern akzeptieren, der in Form eines Spaltenvektors für `x` zu nennen ist. Weitere Argumente von `func` können anstelle der `...` übergeben werden. Dabei ist darauf zu achten, dass keines dieser Argumente den Namen `x` trägt. Wenn `func` die negative log-likelihood von Daten für den Parametervektor β eines verallgemeinerten linearen Modells (Kap. 8) berechnet und `x` die Maximum-Likelihood-Schätzung $\hat{\beta}_{\text{ML}}$ dieser Parameter ist, erhält man die geschätzte Kovarianzmatrix von $\hat{\beta}$ als Inverse der Hesse-Matrix.

Als Beispiel sei wieder die schon durchgeführte Poisson-Regression betrachtet. Zunächst ist die Funktion zu definieren, die für n beobachtete Zähldaten im Vektor y , gegeben die Designmatrix X (Vektor der Kovariaten für Person i ist x_i) und den Parametervektor β die negative Poisson log-likelihood $-\ell$ bestimmt.

$$\begin{aligned} -\ell &= - \sum_{i=1}^n \ln \left(\frac{\mu_i^{y_i} e^{-\mu_i}}{y_i!} \right) = - \sum_{i=1}^n y_i \cdot \ln \mu_i - \mu_i - \ln y_i! \\ &= - \sum_{i=1}^n y_i \cdot x_i^\top \beta - e^{x_i^\top \beta} - \ln y_i! \end{aligned}$$

Für den Term $\ln y_i!$ sei an die Beziehung $n! = \Gamma(n+1)$ für eine natürliche Zahl n erinnert.

```
> nllPois <- function(b, X, Y) {
+   mu <- exp(b[1] + b[2]*X)    # Erwartungswert Poisson-Verteilung
+   # negative log-likelihood der Daten
+   -sum(Y*log(mu) - mu - lgamma(Y+1))
+ }

> h <- hessian(nllPois, bML, X=X, Y=Y)      # Hesse-Matrix
> solve(h)                                     # Inverse Hesse-Matrix -> Kovarianzmatrix
      [,1]      [,2]
[1,]  0.004437647 -0.0010525902
[2,] -0.001052590  0.0005259292

> vcov(glmFit)                                # Kontrolle
            (Intercept)          X
(Intercept)  0.004437647 -0.0010525901
X           -0.001052590  0.0005259292
```

16.4 Numerisch optimieren

Durch numerische Optimierung lassen sich die Parameter einer *Zielfunktion* (*objective function*) so bestimmen, dass sie ihr Minimum annimmt. Optimierungsverfahren sind u. a. die Grundlage der Anpassung vieler statistischer Modelle. [Nash \(2014a\)](#) liefert eine umfassende Einführung in ihre Anwendung. Für Pakete, die Implementierungen unterschiedlicher Optimierungsverfahren bereitstellen, vgl. den Abschnitt *Optimization and Mathematical Programming* der CRAN Task Views ([Schwendinger & Borchers, 2019](#)).

16.4.1 Maximum-Likelihood-Parameterschätzung

Ein wichtiger Anwendungsfall von Optimierung ist die Maximum-Likelihood-Methode zur Parameterschätzung statistischer Modelle. Hier sind die Modellparameter gesucht, für die die likelihood der erhobenen Daten ihr Maximum annimmt und damit die – numerisch besser zu behandelnde – negative log-likelihood ihr Minimum. Für die einfachere Aufgabe, nur die Maximum-Likelihood-Schätzungen für die Parameter der geläufigsten Verteilungsfamilien für einen Datenvektor zu finden, eignet sich `fitdistr()` aus dem Paket MASS.⁶

```
fitdistr(x=<Vektor>, densfun=<'Dichtefunktion>', start=<Startwert>, ...)
```

Welche Verteilungsfamilie für die Daten im Vektor `x` angepasst werden soll, ist über `densfun` festzulegen. Mögliche Werte sind etwa "Poisson", "gamma" oder "lognormal" – für andere vgl. `?fitdistr`, wenn das Paket MASS geladen ist. Für einige Verteilungsfamilien müssen die Parameterschätzungen über eine numerische Suche gefunden werden, weil keine explizite Formel für die Lösung existiert. In diesem Fall sind Startwerte für die Suche als Liste mit

⁶Einen erweiterten Funktionsumfang bietet das Paket `fitdistrplus` ([Delignette-Muller & Dutang, 2015](#)).

benannten Komponenten an `start` zu übergeben. Die Ausgabe zeigt die Punktschätzer sowie deren Standardfehler.

Als Beispiel sollen die Parameter Weibull-verteilter Daten geschätzt werden.

```
> library(MASS)                      # für fitdistr()
> X <- rweibull(100, shape=1.5, scale=100) # Weibull-Zufallszahlen
> fitdistr(X, densfun="weibull", start=list(shape=1, scale=50))
      shape          scale
 1.4453031    101.1767531
(0.1093255)   ( 7.3871757)
```

Die allgemeine Maximum-Likelihood-Schätzung von Parametern eines statistischen Modells ist mit der Funktion `mle()` aus dem im Basisumfang enthaltenen Paket `stats4` möglich. Sie schätzt auch die Kovarianzmatrix der Parameterschätzer. `mle()` wesentlich auf der im folgenden Abschnitt vorgestellten Funktion `optim()`.

16.4.2 Allgemeine Optimierung

Die hier nicht beschriebene Funktion `optimize()` eignet sich für den Spezialfall einer Zielfunktion mit nur einem Parameter, während `optim()` allgemein Zielfunktionen mit mehreren Parametern minimieren kann.⁷

```
optim(par=<Startwerte>, fn=<Funktion>, gr=<Gradient>, ...,
      method=<Optimierungsverfahren>, hessian=FALSE)
```

`par` erwartet einen Vektor mit den Startwerten der Parameter, an denen die numerische Suche nach einem Minimum beginnt. Die Zielfunktion ist an `fn` zu übergeben. Als erstes Argument muss sie einen Vektor mit den Parametern akzeptieren, aus denen sie den Funktionswert bestimmt. Im Zuge der numerischen Suche ruft `optim()` die Zielfunktion immer wieder mit wechselnden Werten für die Parameter auf und prüft so für verschiedene Bereiche im Parameterraum, wo der Funktionswert minimal wird. Lässt sich die lokale Veränderungsrate (Ableitung) der Zielfunktion in geschlossener Form oder über numerische Methoden (Abschn. 16.3.2) berechnen, kann eine entsprechende Gradientenfunktion an `gr` übergeben werden. Andernfalls ist `gr=NULL` zu setzen. Benötigt die Zielfunktion weitere Argumente, können sie anstelle der `...` genannt werden.

Für den Suchalgorithmus im Parameterraum stehen verschiedene Verfahren zur Auswahl, die `method` kontrolliert. Voreinstellung ist "Nelder-Mead", andere Methoden sind "CG" (*conjugate gradient*), "SANN" (*simulated annealing*) oder "BFGS" (ein quasi-Newton-Verfahren). Mit der Methode "L-BFGS-B" können auch einfache Nebenbedingungen für die Parameter formuliert werden, die den zulässigen Wertebereich auf ein Intervall beschränken (*box constraints*). Welches Optimierungsverfahren hinsichtlich der Genauigkeit und Geschwindigkeit am besten geeignet ist, hängt sehr von der Zielfunktion ab (Nash, 2014a, 2014b). Damit eine numerische Approximation der Hesse-Matrix (Abschn. 16.3.2) als Komponente der zurückgegebenen Liste enthalten ist, muss man `hessian=TRUE` wählen.

⁷Das Paket `optimx` (Nash & Ravi, 2011) verfolgt das Ziel, eine genauso wie `optim()` bedienbare Funktion bereitzustellen, die jedoch i. d. R. bessere Ergebnisse liefern soll.

Wur die numerische Suche erfolgreich, enthält die zurückgegebene Liste in der Komponente `par` den Wert des Parametervektors, für den `fn` ihr Minimum annimmt. Die Komponente `convergence` enthält Informationen darüber, ob die Suche auf ein Minimum konvergiert ist, ehe die maximale Anzahl von Suchschritten erreicht wurde.

Als Beispiel sei der χ^2 -Test auf Normalverteilung betrachtet (Abschn. 10.1.5). Für ihn werden die Werte einer Stichprobe in mehrere disjunkte Intervalle gruppiert und deren beobachtete Häufigkeiten mit den erwarteten Häufigkeiten unter der Annahme verglichen, dass Normalverteilung vorliegt. Für die Berechnung der erwarteten Häufigkeiten ist die Normalverteilung zunächst durch ihre Parameter μ und σ zu spezifizieren. Damit die aus beobachteten und erwarteten Klassenhäufigkeiten gebildete Teststatistik auch tatsächlich χ^2 -verteilt ist, müssen μ und σ aus den Daten unter Berücksichtigung der Klassengrenzen geschätzt werden und nicht einfach über Mittelwert und Standardabweichung der Stichprobe. Geeignet ist etwa eine Minimum- χ^2 -Schätzung – dies sind jene Werte $\hat{\mu}$ und $\hat{\sigma}$, bei denen die χ^2 -Teststatistik für die gegebenen Daten und für die festgelegten Klassengrenzen ihr Minimum annimmt.

Zum Vergleich seien zunächst der gewöhnliche Mittelwert und die gewöhnliche Streuung einer Stichprobe berechnet.

```
> DV <- rnorm(50, mean=0, sd=1)          # Stichprobe
> mean(DV)                                # Mittelwert
[1] 0.02526254

> sd(DV)                                   # Streuung
[1] 1.037666
```

Weiter seien hier vier, bei Normalverteilung mit $\mu = 0$ und $\sigma = 1$ gleichwahrscheinliche Intervalle gebildet und deren beobachtete Häufigkeiten für die entsprechend gruppierten Daten berechnet.

```
> nCls    <- 4                           # Anzahl Intervalle
> limits <- qnorm(seq(from=1/nCls, to=(nCls-1)/nCls,
+                   length.out=nCls-1), mean=0, sd=1)

> breaks   <- c(-Inf, limits, Inf)       # alle Intervallgrenzen
> DVcut    <- cut(DV, breaks=breaks)     # gruppierte Daten
> observed <- table(DVcut)                # beobachtete Häufigkeiten
```

Die zu minimierende Zielfunktion erhält als erstes Argument den Vektor der geschätzten Parameter $\hat{\mu}$ und $\hat{\sigma}$, als zweites Argument den Vektor der Intervallgrenzen und als drittes Argument die Tabelle der in der gezogenen Stichprobe beobachteten Klassenhäufigkeiten. Die Funktion gibt den Wert der zugehörigen χ^2 -Teststatistik zurück, im Fall einer zu kleinen Varianz einen fehlenden Wert. Als Startwert für die Minimierung werden hier der gewöhnliche Mittelwert und die gewöhnliche Streuung der Stichprobe verwendet.

```
> minFunMin <- function(param, brks, obs) {
+   if(param[2] < 1e-4) { return(NA_real_) }    # zu kleine Streuung
+   # erwartete Klassenwahrscheinlichkeiten bei Normalverteilung
+   probs    <- diff(pnorm(brks, mean=param[1], sd=param[2]))
+   expected <- sum(obs) * probs               # erwartete Häufigkeiten
```

```

+     sum((obs-expected)^2 / expected)      # chi^2 Teststatistik
+ }

> resMinChisq <- optim(c(mean(DV), sd(DV)), minFunMin,
+                         brks=breaks, obs=observed, gr=NULL, method="BFGS")

> resMinChisq$par                      # min-chi^2-Schätzer
[1] -0.006693669  1.094088449

```

Alternativ führt auch eine gruppierte Maximum-Likelihood-Schätzung von μ und σ dazu, dass die Teststatistik des χ^2 -Tests auf Normalverteilung tatsächlich χ^2 -verteilt ist. Die likelihood lässt sich maximieren, indem man die negative log-likelihood minimiert. Die likelihood ist hier gleich der aus der Multinomialverteilung berechneten Punktwahrscheinlichkeit der beobachteten Klassenhäufigkeiten, gegeben die aus der Normalverteilung abgeleiteten Klassenwahrscheinlichkeiten.

```

> minFunGML <- function(param, brks, obs) {
+   if(param[2] < 1e-4) { return(NA_real_) }      # zu kleine Streuung
+   # erwartete Klassenwahrscheinlichkeiten bei Normalverteilung
+   probs <- diff(pnorm(brks, mean=param[1], sd=param[2]))
+   # negative log-likelihood Multinomialverteilung
+   -dmultinom(obs, size=sum(obs), prob=probs, log=TRUE)
+ }

> resGrML <- optim(c(mean(DV), sd(DV)), minFunGML,
+                     brks=breaks, obs=observed, gr=NULL, method="BFGS")

> resGrML$par                      # gruppierter ML-Schätzer
[1] -0.01175457  1.15656851

```

Kapitel 17

R als Programmiersprache

R bietet nicht nur Mittel zur numerischen und grafischen Datenanalyse, sondern ist gleichzeitig eine Programmiersprache, die dieselbe Syntax wie die bisher behandelten Auswertungen verwendet. Das sehr umfangreiche Thema der Programmierung mit R soll in den folgenden Abschnitten nur soweit angedeutet werden, dass nützliche Sprachkonstrukte wie z. B. Kontrollstrukturen verwendet sowie einfache Funktionen selbst erstellt und analysiert werden können.

Eine ausführliche Behandlung sei der hierauf spezialisierten Literatur überlassen ([Chambers, 2016](#); [Gillespie & Lovelace, 2017](#); [Wickham, 2019a](#)). Die Entwicklung eigener R-Pakete behandeln [R Core Team \(2020d\)](#) und [Wickham \(2023\)](#). R kann auch direkt Bibliotheken oder Funktionen nutzen, die in anderen Programmiersprachen erstellt wurden, etwa Python ([J. J. Allaire, Ushey & Tang, 2020](#)), Java ([Urbanek, 2019](#)), C/C++ ([Eddelbuettel, 2013](#)) oder Julia ([Li, 2019](#)).

17.1 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 *Schleifen* dieselben Befehle, solange bestimmte Nebenbedingungen gelten. Hilfe zu diesem Thema zeigt `?Control`.

17.1.1 Fallunterscheidungen

Mit `if()` können Befehle im Rahmen einer Fallunterscheidung in Abhängigkeit davon ausgeführt werden, ob eine Bedingung gilt.

```
if(<logischer Ausdruck>) {  
  <Befehlsblock>          # ausgeführt, wenn <Ausdruck> TRUE ist  
}
```

Als Argument erwartet `if()` zur Formulierung der Bedingung einen Ausdruck, der sich zu einem einzelnen Wahrheitswert `TRUE` oder `FALSE` auswerten lässt. Hierbei ist auszuschließen, dass der Ausdruck einen Vektor mit Länge $\neq 1$ oder aber `NA`, `NaN` bzw. `NULL` ergibt. Sicherstellen kann man dies etwa, indem der Ausdruck in `isTRUE()` eingeschlossen wird (Abschn. [1.4.6](#)).

Mögliche Ausdrücke sind u. a. logische Vergleiche. Im Folgenden, durch {} eingeschlossenen Block, sind jene Befehle aufzuführen, die nur dann ausgewertet werden sollen, wenn der Ausdruck gleich TRUE ist.¹ Finden die Befehle in einer Zeile hinter if() Platz, sind die geschweiften Klammern optional.² Das Ergebnis von if() ist der Wert des letzten ausgewerteten Ausdrucks und lässt sich direkt einem Objekt zuweisen.

```
> (x <- round(rnorm(1, mean=100, sd=15)))
[1] 115

> y <- NA          # default Wert
> y <- if(x > 100) { TRUE }
> y
[1] TRUE
```

Für den Fall, dass der Ausdruck gleich FALSE ist, kann eine Verzweigung des Befehlsflusses auch einen alternativen Block von Befehlen vorsehen, der sich durch das Schlüsselwort else eingeleitet an den auf if() folgenden Block anschließt.

```
if(<Ausdruck>) {
  <Befehlsblock>      # ausgeführt, wenn <Ausdruck> TRUE ist
} else {
  <Befehlsblock>      # ausgeführt, wenn <Ausdruck> FALSE ist
}
```

Hier ist zu beachten, dass sich das Schlüsselwort else in derselben Zeile wie die schließende Klammer } des if() Blocks befinden muss und nicht separat in der auf die Klammer folgenden Zeile stehen kann.

```
> x <- round(rnorm(1, mean=100, sd=15))
> text_out <- if(x > 100) {
+   cat("x is", x, "(greater than 100)\n")
+ } else {
+   cat("x is", x, "(100 or less)\n")
+ }

> text_out
[1] x is 83 (100 or less)
```

Innerhalb der auf if() oder else folgenden Befehlssequenz können wiederum if() Abfragen stehen. So verschachtelt lassen sich auch Bedingungen prüfen, die mehr als zwei Ausprägungen annehmen können.

¹ 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.

² Allgemein gilt, dass in einer Zeile stehende Befehle einen Block bilden und zusammen ausgeführt werden. Geschweifte Klammern sorgen dafür, dass die zwischen ihnen stehenden Befehle auch dann als einzelner Block gewertet werden, wenn sie sich über mehrere Zeile erstrecken. Die Auswertung erfolgt erst, wenn die Klammer geschlossen wird, auch wenn Abschnitte davor bereits für sich genommen syntaktisch vollständig sind.

```
> (day <- sample(c("Mon", "Tue", "Wed"), size=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 neither Monday nor Tuesday")
+   }
+ }
[1] "The day is neither Monday nor Tuesday"
```

Kürzer und mit weniger Verschachtelungsebenen lässt sich die mehrfache Verzweigung formulieren, indem das `else` und folgende `if` einander direkt angeschlossen werden.

```
> text_out <- if(day == "Mon") {
+   print("The day is Monday")
+ } else if(day == "Tue") {
+   print("The day is Tuesday")
+ } else {
+   print("The day is neither Monday nor Tuesday")
+ }

> text_out
[1] "The day is neither Monday nor Tuesday"
```

Die `switch()` Anweisung ist für eben solche Situationen gedacht, in denen eine Nebenbedingung mehr als zwei Ausprägungen besitzen kann und für jede dieser Ausprägungen anders verfahren werden soll. `switch()` 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 folgende Form:

`switch(EXPR=<Ausdruck>, <Befehl 1>, <Befehl 2>, ...)`

Als erstes Argument `EXPR` 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. Das Ergebnis des Ausdrucks `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, size=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 die folgende Gestalt.

```
switch(EXPR=<Ausdruck>,
      <Wert 1>=<Befehl 1>, <Wert 2>=<Befehl 2>, ...
      <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**.

Abschnitt 17.3 erläutert, wie Funktionen wie hier im Beispiel selbst erstellt werden können.

```
> 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])           # default / catch all
+ }

> (vals <- round(rnorm(2, 1, 10)))
[1] -5 -7

> myCalc("minus", vals)
[1] 2

> myCalc("XX", vals)                  # -> catch all in switch()
[1] 0.7142857
```

17.1.2 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.³

```
for(<Variable> in <Vektor>) {
  <Befehlsblock>
}
```

Die Funktion **for()** 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 *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

³ Anders als in kompilierten Programmiersprachen wie etwa C oder Fortran sind Schleifen in R als interpretierter Sprache oft ineffizient. Abschnitt 17.5.1 gibt Hinweise für die Steigerung der Effizienz.

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 100.000 Zufallszahlen einer normalverteilten Variable simuliert. Daraufhin werden die Zahlen quadriert und logarithmiert, also einer nichtlinearen Transformation unterzogen. In einer Schleife werden aus dieser Grundgesamtheit 1.000 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.

```
> src      <- log(rnorm(100000, 0, 1)^2)      # Grundgesamtheit
> alpha    <- 0.05                            # nominelles alpha
> nTests   <- 1000                            # Anzahl simulierter Tests
> Nj       <- 20                               # Gruppengröße
> sigVecT <- numeric(nTests)                  # für Ergebnisse t.test()
> sigVecV <- numeric(nTests)                  # für Ergebnisse var.test()

> for(i in seq(length.out=nTests)) {
+   group1     <- sample(src, size=Nj, replace=FALSE) # Stichprobe 1
+   group2     <- sample(src, size=Nj, 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) # F 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 F-Tests Varianzhomogenität:", sum(sigVecV), "\n")
Signifikante F-Tests Varianzhomogenität: 197
```

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 lässt.

Schleifen können zur Effizienzsteigerung 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. Eine Alternative zum obigen Vorgehen zeigt Abschn. 17.2.

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>) { <Befehlsblock> }
```

Als Argument erwartet `while()` einen Ausdruck, der sich zu einem einzelnen Wahrheitswert `TRUE` oder `FALSE` auswerten lässt. Hierbei ist auszuschließen, dass der Ausdruck einen Vektor mit Länge $\neq 1$ oder aber `NA`, `NaN` bzw. `NULL` ergibt. Sicherstellen kann man dies etwa, indem der Ausdruck in `isTRUE()` eingeschlossen wird (Abschn. 1.4.6).

Der in `{}` eingefasste Schleifenrumpf wird immer wieder durchlaufen, solange der Ausdruck gleich `TRUE` ist. Typischerweise ändern Berechnungen im Schleifenrumpf den 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 *Endlosschleife*, die den weiteren Programmablauf blockiert und auf der Konsole mit der `ESC` Taste abgebrochen werden müsste (unter Linux mit `Strg+c`).

```
> x <- 37
> y <- 10
> while(x >= y) { x <- x-y }    # Modulo-Berechnung (für positive Werte)
> x
[1] 7
```

Die Schlüsselwörter `break` und `next` erlauben es, die Ausführung von Schleifen innerhalb des Schleifenrumpfes zu steuern. Sie stehen i. d. R. innerhalb eines durch `if()` eingeleiteten Blocks. 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 { <Befehlsblock> }

> i <- 0
> repeat {
+   i <- i+1
+   if((i %% 4) == 0) { break }
+   print(i)
+ }
[1] 1
[1] 2
[1] 3
```

17.2 Funktionsaufrufe ohne Schleifen wiederholen

Als vereinfachte Form von `sapply()` (Abschn. 3.3.13) sorgt `replicate(n=<Anzahl>, expr=<Befehl>)` dafür, 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(6, round(rnorm(4), digits=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
```

Im vorangehenden Beispiel einer Simulation für die Power-Schätzung (Abschn. 17.1.2) ließe sich die `for()` Schleife auch vermeiden, indem `replicate()` sowie Möglichkeiten zur Vektorisierung genutzt werden. Häufig ist insbesondere letzteres aus Effizienzgründen erstrebenswert.

```
# wiederhole nTests mal für beide Stichproben Ziehen von je Nj Personen
> group1 <- replicate(nTests, sample(src, size=Nj, replace=FALSE))
> group2 <- replicate(nTests, sample(src, size=Nj, 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/Nj) * sqrt(var(x[1:Nj]) + var(x[(Nj+1):(2*Nj)])) } )

# berechne alle Mittelwertsdifferenzen
> meanDiffs <- apply(groups, 2, function(x) {
+   mean(x[1:Nj]) - mean(x[(Nj+1):(2*Nj)]) } )

# vektorisierte Befehle, um t-Werte und p-Werte zu erhalten
```

```
> tVals <- meanDiff / estSigDiff      # t-Werte
> pVals <- pt(tVals, (2*Nj)-2, lower.tail=FALSE)      # p-Werte
> sum(pVals < alpha)                  # Anzahl signifikanter Tests
[1] 55
```

17.3 Eigene Funktionen erstellen

Funktionen sind eine Zusammenfassung von Befehlen auf Basis von beim Aufruf mitgelieferten Eingangsinformationen, den *Funktionsargumenten*. Ebenso wie etwa Matrizen als Objekte der Klasse `matrix` können Funktionen als Objekte der Klasse `function` erstellt werden. Dafür wird über `function()` eine Funktion definiert und das Ergebnis einem Objekt zugewiesen. Selbst erstellte Funktionen haben denselben Status und dieselben Möglichkeiten wie mit R mitgelieferte Funktionen.

```
<Name> <- function(<Argument1>=<Voreinstellung>,
                     <Argument2>=<Voreinstellung>, ...)
  <Befehlsblock>
}
```

Als Beispiel soll eine Funktion mit dem Namen `.First` erstellt werden, über die in der Datei `Rprofile.site` im `etc/` Ordner des R-Programmordners individuell festgelegt werden kann, welche Befehle zu Beginn jeder R-Sitzung automatisch auszuführen sind (Abschn. 1.2.2).

```
> .First <- function() {
+   library(car)                      # lade automatisch Paket car
+   print("Have a nice day!")          # gib Begrüßung aus
+ }
```

Eine Kurzschreibweise von `function()`, die sich insbesondere bei der Definition anonymer Funktionen innerhalb von Aufrufen von `lapply()`, `sapply()`, etc. eignet, ist `\()` (Abschn. 17.3.6).

17.3.1 Funktionskopf

Zunächst ist bei einer Funktion der *Funktionskopf* innerhalb der runden Klammern () zu definieren, in dem durch Komma getrennt jene *formalen* Argumente benannt werden, die eine Funktion als Eingangsinformation akzeptiert. Die hier benannten Argumente stehen innerhalb des Funktionsrumpfes (s. u.) als Objekte zur Verfügung.⁴ Auch ein leerer Funktionskopf ist möglich, wenn die folgenden Befehle nicht von äußeren Informationen abhängen.

Jedem formalen Argument kann mit = 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, muss beim Aufruf der Funktion zwingend ein Wert für das

⁴Die aufgerufene Funktion kann nur Kopien der ursprünglichen Objekte, nicht aber die Objekte selbst ändern, da Argumente als *Wertparameter* (*call by value*) übergeben werden.

Argument übergeben werden. Argumente mit Voreinstellung sind beim Aufruf dagegen optional (Abschn. 1.2.5).

Argumente können von jeder Klasse, also auch ihrerseits Funktionen sein – eine Möglichkeit, die etwa bei `lapply()` oder `curve()` Verwendung findet. Eine solche Funktion höherer Ordnung, die eine Funktion als Argument akzeptiert und auf einen Wert abbildet, ist ein *Funktional*.

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. Im Funktionsrumpf sollte deshalb eine Prüfung erfolgen, ob beim Funktionsaufruf tatsächlich ein für die weiteren Berechnungen geeignetes Objekt für ein Argument übergeben wurde (Abschn. 17.3.3).

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 innerhalb der Funktion gemeinsam mittels `Objekt <- list(...)` in einem Objekt speichern und weitergeben lassen. Das Argument `...` sollte das letzte Argument im Funktionskopf sein. Es 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.

17.3.2 Funktionsrumpf

Auf den Funktionskopf folgt eingeschlossen in geschweifte Klammern `{}` der sich ggf. über mehrere Zeilen erstreckende *Funktionsrumpf* als Block von Befehlen und durch `#` eingeleiteten Kommentaren. Findet die vollständige Funktionsdefinition in einer Zeile Platz, sind die geschweiften Klammern optional.

Die Befehle im Funktionsrumpf haben Zugriff auf alle übergebenen Argumente und auf die in einer R-Sitzung zuvor erstellten *globalen* Objekte. Objekte, die innerhalb eines Funktionsrumpfes definiert werden, stehen dagegen nur *lokal*, d. h. innerhalb der Funktion zur Verfügung und verfallen nach dem Aufruf der Funktion.⁵

Werden innerhalb des Funktionsrumpfes Argumente aus dem Funktionskopf verwendet, ist das *Lazy-Evaluation*-Prinzip zu beachten, nach dem der Inhalt von Funktionsargumenten erst mit deren tatsächlichen Verwendung ausgewertet wird.⁶ 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 Funktionsrümpfe `{ var2 }` und `{ var1 <- var1^2; var2 }` hätten

⁵Lokale Objekte existieren in einer beim Funktionsaufruf eigens erstellten Umgebung (Abschn. 1.4.1). Für das in diesem Kontext relevante, aber komplexe Thema der Regeln für die Gültigkeit von Objekten (*scoping*) vgl. Chambers (2016).

⁶Bis zur Auswertung ist das Argument ein *promise*. Für gewöhnliche Zuweisungen an Objekte gilt dagegen das *Eager-Evaluation*-Prinzip, nach dem der Inhalt von neuen Objekten schon bei der Zuweisung ausgewertet wird. Um auch für diese Objekte das Lazy-Evaluation-Prinzip zu nutzen, muss `delayedAssign()` verwendet werden.

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 später im Funktionsrumpf erstellt werden. Beispiel in Abschn. 17.3.3 sei eine Plot-Funktion mit einem optionalen Argument für den Wertebereich der Abszisse, wobei die Funktion bei Vektoren ungleicher Länge automatisch den längeren passend kürzt.

Die Befehle im Funktionsrumpf führen zu keiner Ausgabe 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 (Abschn. 2.14.2).

17.3.3 Funktionsargumente prüfen

Häufig setzen Auswertungsschritte im Rumpf einer Funktion voraus, dass die als Argumente übergebenen Objekte von einer vorher festgelegten Struktur sind, es sich etwa um Matrizen einer bestimmten Dimensionierung, Vektoren von Zeichenketten, o. ä. handelt. Um sicherzustellen, dass erwartete Argumente auch tatsächlich beim Funktionsaufruf übergeben wurden und die richtige Form besitzen, ist es sinnvoll, den Funktionsrumpf mit entsprechenden Prüfungen zu beginnen.⁷

Zunächst prüft `missing("<Argumentname>")`, ob ein konkreter Wert für ein Argument fehlt. Ein weiteres Hilfsmittel ist die Funktion `match.arg()`. Sie ermittelt, ob der für ein Funktionsargument übergebene Wert tatsächlich aus einer vorgegebenen Menge möglicher Werte stammt.

```
match.arg(arg, choices=<Werte>, several.ok=FALSE)
```

Ohne weitere Angaben für `choices` und `several.ok` wird geprüft, ob der für das formale Funktionsargument `arg` übergebene Wert aus dem Vektor stammt, der im Funktionskopf als Voreinstellung benannt wurde. Ist dies nicht der Fall, bricht die Funktion mit einer Fehlermeldung ab. Wichtig ist hierbei, dass *partial matching* erlaubt ist. Es ist also ausreichend, wenn der Beginn des Werts mit einem der vordefinierten Werte zumindest soweit übereinstimmt, dass eine eindeutige Zuordnung möglich ist. Sollen nicht die im Funktionskopf als Voreinstellung genannten Werte für die Prüfung herangezogen werden, lässt sich alternativ für das Argument `choices` ein Vektor zulässiger Werte nennen. `several.ok` definiert, ob ein einzelner Wert notwendig ist, oder auch ein Vektor von Werten übergeben werden kann.

```
# Funktion zur Transformation von Werten
> modify <- function(x, how=c("standardize", "normalize")) {
+   how <- match.arg(how)
+   if(how == "standardize") {
+     scale(x, center=TRUE, scale=TRUE)
```

⁷Für einfache, nur zum eigenen Gebrauch bestimmte Funktionen mag dies überflüssig erscheinen, da man dann selbst darauf achten kann, sie korrekt zu verwenden. Allerdings ist dies gefährlich, da sich Funktionen durch andere als die vorgesehenen Eingangsinformationen stillschweigend anders als beabsichtigt verhalten können und so u. U. schwer entdeckbare Fehler verursachen (Abschn. 17.3.7).

```

+     } else if(how == "normalize") {
+         (x - min(x)) / diff(range(x))
+     }
+ }

> x      <- rnorm(100, mean=5, sd=3)
> x_mod1 <- modify(x, how="standardize")
> mean(x_mod1)
[1] 6.531613e-17
> sd(x_mod1)
[1] 1

# Aufruf mit abgekürzter Wahl für das Argument how
> x_mod2 <- modify(x, how="norm")
> range(x_mod2)
[1] 0 1

# Aufruf mit nicht vorgesehenem Wert für how
> x_mod3 <- modify(x, how="boxcox")
Fehler in match.arg(how) :
'arg' should be one of "standardize", "normalize"

```

Funktionen wie `inherits(<Objekt>, "<Klasse>")` (Abschn. 1.4) eignen sich dazu sicherzustellen, dass ein übergebenes Objekt eine bestimmte Klasse und damit eine vorgegebene Struktur besitzt.

Selbst erstellte Prüfungen können mit `stop("<Fehlermeldung>")` kombiniert werden, um die Funktion abzubrechen, wenn Voraussetzung nicht erfüllt sind. Soll die Funktion dagegen weiter ausgeführt werden und nur eine Warnmeldung ausgeben, ist `warning("<Warnhinweis>")` zu verwenden. `stopifnot()` liefert eine andere Möglichkeit zum Abbruch einer Funktion, nachdem sie einen logischen Ausdruck geprüft hat.

```
stopifnot(<logischer Ausdruck 1>, <logischer Ausdruck 2>, ...)
```

`stopifnot()` beendet die Funktion, sofern bereits einer der übergebenen Ausdrücke nicht TRUE, sondern FALSE, NA, NaN, NULL oder ein leerer Vektor ist und kann damit eine Konstruktion wie die folgende ersetzen:

```

> if(!isTRUE(<logischer Ausdruck>)) { stop("<Fehlermeldung>") }

# Voreinstellung für xLims ist ein erst im Rumpf erstelltes Objekt
> myPlot <- function(x, y, xLims=xRange) {
+   # stelle sicher, dass x und y numerisch und nicht leer sind
+   stopifnot(is.numeric(x), is.numeric(y))
+   stopifnot(length(x) > 0, length(y) > 0)
+
+   # prüfe, ob ein Vektor automatisch gekürzt werden muss
+   if(length(x) != length(y)) {
+     warning("x und y haben ungleiche Länge -> kürze")

```

```

+
+     lenMin <- min(c(length(x), length(y)))
+
+     x <- x[seq_len(lenMin)]
+     y <- y[seq_len(lenMin)]
+
+ }
+
+
+ # fehlt xLims im Aufruf, verwende diesen Wertebereich für x
+ xRange <- round(range(x), -1) + c(-10, 10)
+
+
+ # Auswertung von Argument xLims erfolgt erst im nächsten Befehl
+ plot(x, y, xlim=xLims)
}

> myPlot(1:5, 1:10)
Warnmeldung:
In myPlot(1:5, 1:10) : x und y haben ungleiche Länge -> kürze

```

17.3.4 Fehler behandeln

Führt ein im Funktionsrumpf verwendeter Ausdruck zur Laufzeit zu einem Fehler, bricht die gesamte Funktion an dieser Stelle ab. Eine Funktion kann aber auch fehlertolerant werden, indem potentiell zu Fehlern führende Ausrücke in `try(<Ausdruck>)` eingeschlossen werden. Mit dieser Konstruktion ist die Funktion immun gegenüber vom Ausdruck produzierten Fehlern und setzt sich im Anschluss wie gewohnt fort. Schlägt der Ausdruck mit Fehlern fehl, gibt `try()` ein Objekt zurück, das aus der Klasse "try-error" abgeleitet ist. Mit `inherits(<try-Objekt>, what="try-error")` lässt sich dies prüfen, um ggf. dann notwendige Befehle auszuführen (s. Abschn. 13.3 für ein Beispiel).

Um die Fehlerbehandlung mit `try()` auch bei Warnungen nutzen zu können, lässt sich mit `options(warn=2)` das Verhalten von R so ändern, dass Warnungen wie Fehler behandelt werden – Voreinstellung ist `options(warn=0)`.

Präziser lassen sich durch Ausrücke im Funktionsrumpf ausgelöste Fehler und Warnungen mit `tryCatch()` behandeln.

```

tryCatch({ <Ausdruck> },
         error=function(e) { <Ausdruck> },
         warning=function(w) { <Ausdruck> },
         finally=           { <Ausdruck> })

```

Als erstes Argument ist der potentiell zu Fehlern oder Warnungen führende Ausdruck anzugeben – erstreckt er sich über mehrere Zeilen, ist er in `{ }` einzuschließen. Das Argument `error` erwartet eine Funktion, die ausgeführt wird, wenn es zu einem Fehler kommt. Sie erhält ihrerseits die Fehlermeldung als Argument. Analog ist für `warning` eine Funktion zu nennen, die bei Warnmeldungen aufgerufen werden soll. Ein für `finally` übergebener Ausdruck wird in jedem Fall im Anschluss an den eigentlichen Ausdruck sowie nach den für `error` und `warning` genannten Funktionen ausgewertet – auch wenn keine Fehler oder Warnungen auftreten.

17.3.5 Rückgabewert und Funktionsende

Das von einer Funktion zurückgegebene Ergebnis ist die Ausgabe des letzten Befehls im Funktionsrumpf. Empfehlenswert ist es allerdings, die Rückgabe eines Objekts explizit über `return(<Objekt>)` erfolgen zu lassen. Dies 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 (Abschn. 17.1.1).

Mehrere Werte lassen sich gemeinsam zurückgeben, indem sie zuvor im Funktionsrumpf in einem geeigneten Objekt zusammengefasst werden, etwa einem Vektor, einer Matrix, einer Liste oder einem Datensatz.

Soll die Funktion keinen Wert zurückgeben, etwa weil sie nur Grafiken zu erstellen hat, ist als letzte Zeile der Befehl `return(invisible(NULL))` zu verwenden. Mit `return(invisible(<Objekt>))` gibt die Funktion ein Objekt zurück, das jedoch nach ihrem Aufruf nicht auf der Konsole sichtbar ist – wie etwa bei `barplot()`.

Mit `on.exit(<Ausdruck>)` kann zu Beginn im Funktionsrumpf ein Ausdruck festgelegt werden, der beim Beenden der Funktion ausgeführt wird – unabhängig davon, an welcher Position die Funktion tatsächlich verlassen wird. Dies ist etwa sinnvoll, wenn eine Funktion temporär globale Optionen mit `options()` ändern und am Schluss wieder auf den ursprünglichen Wert zurücksetzen soll (s. Abschn. 13.3 für ein Beispiel).

Da Funktionen selbst reguläre (*first class*) Objekte sind, lassen sie sich ebenfalls als Rückgabewert verwenden. Dabei ist die Besonderheit zu beachten, dass eine zurückgegebene Funktion eine Kopie der Daten der Umgebung mit einschließt, in der sie definiert wurde (*closure*). In der closure bleiben diese Daten konstant, auch wenn sie sich außerhalb der closure später ändern. Gemeinsam mit anderen Eigenschaften sind closures ein wesentliches Merkmal funktionaler Programmiersprachen Wickham (2019a). Weitere solche Merkmale sind die Möglichkeit, Funktionen höherer Ordnung (Funktionale, Abschn. 17.3.1) zu definieren, Funktionen in Listen zu speichern, anonyme Funktionen zu verwenden (Abschn. 17.3.6) und die Strategie, Funktionen so zu gestalten, dass sie keine *Seiteneffekte*, also Auswirkungen auf Objekte außerhalb der Funktion haben.

17.3.6 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 u. a. dann nützlich, wenn Funktionen an Befehle wie `apply()` oder `Map()` übergeben werden können, s. etwa Abschn. 7.10.3, 10.1.2, 11.1, 14.7.1, oder 14.7.3 für Beispiele.

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 *anonyme*, weil namenlose Funktionen. Diese lassen sich direkt innerhalb eines Funktionsaufrufs definieren, wo eine Funktion als Argument zu übergeben ist, s. etwa Abschn. 10.3.3 und 17.1.2 für Beispiele.

```
> (function(arg1, arg2) { arg1^2 + arg2^2 })(-3, 4)
[1] 25

# euklidische Länge jeder Spalte einer Matrix
> mat <- matrix(rnorm(16, mean=100, sd=15), nrow=4)
> apply(mat, 2, \(\{x\} \{sqrt(sum(x^2))\}) # Kurzform \()
[1] 218.9201 203.0049 205.1682 209.4517

> sqrt(colSums(mat^2)) # Kontrolle: effizienter
[1] 218.9201 203.0049 205.1682 209.4517
```

17.3.7 Generische Funktionen

Generische Funktionen stellen eine Möglichkeit dar, mit der R *objektorientiertes* Programmieren unterstützt. Funktionen werden als generisch bezeichnet, wenn sie sich abhängig von der Klasse der übergebenen Argumente automatisch unterschiedlich verhalten. Es existieren dann mehrere Varianten (*Methoden*) einer Funktion, die alle unter demselben allgemeinen Namen angesprochen werden können. Die Methoden 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 Funktionsarguments 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.

Es handelt sich bei der hier vorgestellten Technik um das S3-Paradigma – in Abgrenzung zum flexibleren, aber auch komplizierteren S4-Paradigma, das etwa beim beschriebenen *method dispatch* nicht auf das erste Argument beschränkt ist.⁸

Vorhandene S3-Methoden einer Funktion nennt `methods("<Funktionsname>")`, auf die auch ihre Hilfe-Seite unter `Usage` verweist. Analog erhält man mit `methods(class="<Klasse>")` Auskunft darüber, welche Funktionen spezielle Methoden für Objekte einer bestimmten Klasse besitzen.⁹ 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.

⁸Eine als Paket implementierte Alternative, die mehr aus anderen Programmiersprachen bekannte Funktionalität der Objektorientierung anbietet, sind *R6* Klassen (<https://r6.r-lib.org/>). Dagegen sind *S7* Klassen (<https://rconsortium.github.io/S7/>) bei Drucklegung des Buchs noch experimentell, könnten mittelfristig aber Teil des Basisumfangs von R werden.

⁹Für S4-Methoden analog `showMethods("<Funktionsname>")` sowie `showMethods(classes="<Klasse>")`.

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 übergebane 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.

```
> info <- function(x) { UseMethod("info") }      # generische Funktion

# Methoden für Objekte verschiedener Klassen
> info.numeric   <- function(x) { range(x) } # für numerischen Vektor
> info.matrix    <- function(x) { dim(x) }   # für Matrix
> info.data.frame <- function(x) { names(x) } # für Datensatz
> info.default   <- function(x) { length(x) } # für andere Objekte

# Objekte verschiedener Klassen
> vec  <- round(runif(12, 20, 100), 2) # numerischer Vektor
> char <- LETTERS[sample(1:26, 5)]       # Vektor von Zeichenketten
> mat  <- matrix(vec, nrow=3)            # Matrix
> myDf <- setNames(data.frame(mat),      # Datensatz mit Variablennamen
+                   LETTERS[seq_len(ncol(myDf))])

# Anwendung von info() auf Objekte verschiedener Klassen
> info(vec)                           # ruft info.numeric() auf
[1] 26.05 91.62

> info(mat)                           # ruft info.matrix() auf
[1] 3 4

> info(myDf)                          # ruft info.data.frame() auf
[1] "A" "B" "C" "D"

> info(char)                           # ruft info.default() auf
[1] 5
```

Die verschiedenen Methoden einer generischen Funktion können im Prinzip unterschiedliche Argumente erwarten. In der Praxis empfiehlt es sich jedoch, diese Freiheit nur dahingehend zu nutzen, dass jede Methode zunächst alle Argumente der generischen Funktion in derselben Reihenfolge mit denselben Voreinstellungen verwendet. Zusätzlich können Methoden danach weitere Argumente besitzen, die sich je nach Methode unterscheiden. In diesem Fall muss die generische Funktion als letztes Argument ... besitzen.

17.4 Funktionen analysieren

Ein wichtiger Bestandteil der vertieften Arbeit mit fremden und selbst erstellten Funktionen ist die Analyse der ausgeführten Befehlsabfolge, insbesondere hinsichtlich der Frage, ob die Funktion fehlerfrei und effizient arbeitet (*debugging*). Hierfür eignet sich zum einen die Begutachtung ihres Quelltextes, zum anderen die schrittweise Untersuchung ihres Verhaltens zur Laufzeit.¹⁰

17.4.1 Quelltext fremder Funktionen begutachten

Aus welchen Befehlen sie besteht, ist bei selbst erstellten Funktionen bekannt. Bei fremden Funktionen hilft der Umstand, dass R den Quelltext auf der Konsole ausgibt, wenn der Funktionsname ohne runde Klammern als Befehl eingegeben wird.¹¹ Mit `capture.output(<Funktionsname>)` lässt sich dieser Quelltext in einem Vektor aus Zeichenketten zur späteren Analyse speichern. Bei `sd()` etwa ist erkennbar, dass letztlich `var()` aufgerufen und die Streuung als Wurzel der Varianz berechnet wird.

```
> sdSource <- capture.output(sd)
> sdSource
[1] "function (x, na.rm = FALSE) "
[2] "sqrt(var(if (is.vector(x) || is.factor(x)) x else as.double(x), "
[3] "      na.rm = na.rm))"
[4] "<bytecode: 0x000001cfccf4a7c20>"
[5] "<environment: namespace:stats>"
```

Um explizit auf Objekte einer bestimmten Umgebung (etwa der eines Pakets) zuzugreifen, ist dem Objektnamen der Paketname mit zwei Doppelpunkten voranzustellen. Alternativ eignet sich auch `get()`. Zusatzpakete können über *Namensräume* (*namespaces*) dafür sorgen, dass manche ihrer Objekte nur für Funktionen aus dem Paket selbst verwendbar sind, außerhalb davon jedoch nicht. Indem man den mit dem Paketnamen übereinstimmenden Namensraum mit `:::` dem Objektnamen voranstellt, lassen sich auch solche Objekte untersuchen.

```
> base::mean      # mean() aus Paket base ...
> boot:::basic.ci # basic.ci() aus Paket boot, sonst nicht sichtbar...
```

Ist eine Funktion im S3-Paradigma generisch (Abschn. 17.3.7), enthält sie nur den `UseMethod()` Befehl. Über `methods("<Funktionsname>")` erfährt man jedoch, welche Methoden für eine solche Funktion existieren und sich über ihren vollständigen Namen ausgeben lassen.¹² Manche Methoden sind dabei zunächst unsichtbar, was durch ein `*` Symbol deutlich gemacht wird, mit dem der Name der Methode in der Ausgabe von `methods()` versehen wird. `getS3method("<Funktionsname>",
→<Klasse>")` hilft, auch den Inhalt solcher Methoden anzuzeigen.¹³ Demselben Zweck dient auch `getAnywhere("<Funktionsname>")`, wobei hier der vollständige Funktionsname i. S. von "`<Basisname>.<Klasse>`" anzugeben ist.

¹⁰ Entwicklungsumgebungen wie RStudio (Abschn. 1.1.4) bieten grafische Oberflächen, die das debugging sehr erleichtern.

¹¹ Operatoren sind in rückwärts gerichtete Hochkommata zu setzen, etwa ``+`` für die Addition.

¹² Für S4-Methoden analog `showMethods("<Funktionsname>")`.

¹³ Für S4-Methoden analog `showMethods("<Funktionsname>", classes="<Klasse>", includeDefs=TRUE)`.

```
> get("fligner.test")                      # fligner.test ist generisch
function (x, ...)
UseMethod("fligner.test")
<environment: namespace:stats>

> methods("fligner.test")                  # Methoden für fligner.test
[1] fligner.test.default* fligner.test.formula*
Non-visible functions are asterisked

> get("fligner.test.default")             # Fehler: unsichtbare Methode
Fehler: Objekt 'fligner.test.default' nicht gefunden

> getS3method("fligner.test", "default")  # findet Methode ...
> getAnywhere("fligner.test.default")     # findet Methode ...
```

In jedem Fall ist es über den auf CRAN erhältlichen Quelltext von R¹⁴ und den der Zusatzpakete¹⁵ möglich, die Befehlsabfolge einer Funktion zu analysieren.

17.4.2 Funktionen zur Laufzeit untersuchen

Mit `cat()` und `print()` lassen sich innerhalb einer selbst geschriebenen Funktion Zwischenergebnisse auf der Konsole ausgeben, um während ihrer Entwicklung Anhaltspunkte darüber zu gewinnen, wie sich die Funktion zur Laufzeit verhält. Treten Fehler im Endergebnis auf, kann so womöglich die Ursache auf einen bestimmten Teilschritt eingegrenzt werden. Ebenso sollten Zwischenergebnisse, die von einem bestimmten Typ sein müssen, analog zu übergebenen Funktionsargumenten daraufhin geprüft werden, ob die Vorgaben auch eingehalten sind. Andernfalls ist die Funktion ggf. mit `stop("<Fehlermeldung>")` oder `stopifnot()` abzubrechen (Abschn. 17.3.2).

Treten Fehler beim Ausführen einer Funktion auf, zeigt `traceback()` den *call stack* an, welche Funktionen also zuletzt ineinander verschachtelt aufgerufen wurden. Um diese Möglichkeit auch bei Warnungen nutzen zu können, lässt sich mit `options(warn=2)` das Verhalten von R so ändern, dass Warnungen wie Fehler behandelt werden. Damit bricht eine Funktion ab, sobald es zu einer Warnung kommt. Im Beispiel wird deutlich, dass zuletzt `stop()` ausgeführt und vorher aus der Funktion `stopifnot()` aufgerufen wurde, die wiederum innerhalb von `myPlot()` gestartet wurde (Abschn. 17.3.2).

```
> myPlot(1:2, "ABC")
Fehler: is.numeric(y) is not TRUE

> traceback()
3: stop(paste(ch, " is not ", if (length(r) > 1L) "all ", "TRUE",
           sep = ""), call. = FALSE)
```

¹⁴Unter <https://github.com/wch/r-source/> kann eine Kopie des Quelltext online gelesen und durchsucht werden.

¹⁵<https://r-pkg.org/> verlinkt für jedes Paket auf CRAN eine URL, unter der der Quelltext eines Pakets online gelesen und durchsucht werden kann.

```
2: stopifnot(is.numeric(x), is.numeric(y)) at selection.r#2
1: myPlot(1:2, "ABC")
```

Wenn durch den Aufruf von `traceback()` klar ist, in welcher Funktion ein Problem auftritt, kann man sich mit `debug(<Funktionsname>)` in diese direkt hinein begeben, während sie ausgeführt wird. Die zu analysierende Funktion ist als Argument zu übergeben. Dadurch wird sie zum debugging markiert und fortan bei ihrem Aufruf immer im *browser* ausgeführt, der zunächst alle Befehle der Funktion anzeigt und dann die schrittweise Abarbeitung jedes einzelnen Befehls ermöglicht. Jeder Ausdruck innerhalb der Funktion (eine Zeile oder ein durch `{ ... }` definierter Block mehrerer Zeilen) wird dabei zunächst separat angezeigt. Daraufhin ist `n (next)` bzw. die `Return` Taste zu drücken, um ihn auszuführen und zum nächsten Ausdruck zu gelangen.

Darüber hinaus kann im *browser* die Konsole von R wie gewohnt verwendet werden, um sich etwa mit `ls()`, `print()`, `head()` oder `str()` einen Überblick über den Inhalt der vorhandenen Objekte zu verschaffen und die verarbeiteten Daten zu begutachten. `c (continue)` setzt die Abarbeitung der Funktion ohne weitere Unterbrechung fort, `s (step into)` führt das debugging innerhalb der in der aktuellen Zeile auszuführenden Funktion weiter, `f (finish)` beendet den aktuellen Funktionsaufruf oder Schleifendurchlauf. Die Eingabe von `where` dient dazu, die Reihenfolge der durch die letzten ineinander verschachtelten Funktionsaufrufe erstellten Umgebungen zu nennen. Weitere Befehle gibt `help` im *browser* aus, `Q (quit)` bricht das debugging ab. Soll eine Funktion nicht weiter analysiert werden, ist ihr Name an `undebug(<Funktionsname>)` als Argument zu übergeben.

Als Beispiel diene die in Abschn. 17.3.7 erstellte generische `info()` Funktion.

```
> debug(info)                                # setze debug-flag
> info(sample(1:10, 10, replace=TRUE))       # Aufruf startet browser
debugging in: info(sample(1:10, 10, replace = TRUE))
debug: {
  UseMethod("info")
}

Browse[1]> n                                  # beginne Ausführung
debug: UseMethod("info")

Browse[1]> n                                  # führe 1. Befehl aus
debugging in: info.numeric(sample(1:10, 10, replace = TRUE))
debug: {
  range(x)
}

Browse[2]> ls()                               # vorhandene Objekte
[1] "x"

Browse[2]> x                                  # zeige Inhalt von x
[1] 5 8 2 4 8 9 7 9 6 2
```

```

Browse[1]> where                                # letzte Funktionsaufrufe
where 1: info.numeric(sample(1:10, 10, replace = TRUE))
where 2: info(sample(1:10, 10, replace = TRUE))

Browse[2]> c                                     # setze Funktion fort
exiting from: info.numeric(sample(1:10, 10, replace = TRUE))
exiting from: info(sample(1:10, 10, replace = TRUE))
[1] 2 9

> undebug(info)                               # entferne debug-flag

```

Mitunter ist es mühsam, durch die schrittweise Ausführung einer Funktion im browser schließlich an die Stelle zu gelangen, die z. B. im Rahmen einer Fehlersuche als mögliche Problemquelle identifiziert wurde. Bei selbst erstellten Funktionen lässt sich der Debug-Prozess abkürzen: Mit Einfügen des `browser()` Befehls vor dem zu analysierenden Abschnitt setzt man einen Haltepunkt, durch den sich beim Ausführen der Funktion der browser öffnet, sobald die bezeichnete Stelle erreicht ist. Damit i. S. eines bedingten Haltepunkts nur dann das debugging gestartet wird, wenn bestimmte Randbedingungen gelten, ist `if(<Bedingung>) { browser() }` zu verwenden (Abschn. 17.1.1).

Für eine weitere Möglichkeit, Haltepunkte in selbst geschriebenen Skriptdateien zu setzen, vgl. `?setBreakpoint` sowie insbesondere die grafische Umsetzung innerhalb einer Entwicklungs-Umgebung wie RStudio. Bei fremden Funktionen dient die Option `options(error=recover)` dazu, beim Auftreten eines Fehlers direkt in den browser zu wechseln. Danach sollte die Option mit `options(error=NULL)` wieder zurückgesetzt werden.

`trace(<Funktion>, tracer=browser, at=<Position>, ...)` erlaubt es, bei nicht selbst erstellten Funktionen Haltepunkte an einer beliebigen Stelle zu setzen. Dafür muss man sich zunächst den Quelltext des Funktionsrumpfes mit `as.list(body(<Funktion>))` als Liste anzeigen lassen. Jeder Funktionsblock ist dabei ein Listenelement. Der numerische Index des Funktionsblocks, bei dem man den Debug-Prozess beginnen möchte, ist an `trace()` für das Argument `at` zu übergeben. Mit `untrace(<Funktion>)` wird der Haltepunkt wieder entfernt.

```

> as.list(body(median.default))
[[1]]
`-` 

[[2]]
if (is.factor(x) || is.data.frame(x)) stop("need numeric data")

[[3]]
if (length(names(x))) names(x) <- NULL
# ...

# setze Haltepunkt bei Block 3
> trace(median.default, tracer=browser, at=3)
Tracing function "median.default" in package "stats"
[1] "median.default"

```

```

> median.default(rnorm(100))
Tracing median.default(rnorm(100)) step 4
Called from: eval(expr, p)
Browse[1]> n
debug: if (length(names(x))) names(x) <- NULL

Browse[2]> c                      # setze Funktionsaufruf fort
[1] -0.0518763

> untrace(median.default)
Untracing function "median.default" in package "stats"

```

17.5 Effizienz von Auswertungen steigern

Bei komplexeren Analysen und großen Datensätzen beginnt die Frage an Bedeutung zu gewinnen, wie die Geschwindigkeit der Auswertungsschritte erhöht werden kann. Der folgende Abschnitt stellt mögliche Ansätze dafür vor.

17.5.1 Grundlegende Empfehlungen

Bevor generelle Empfehlungen ausgesprochen werden, sei an die klassische Warnung vor einer verfrühten Optimierung erinnert: Wichtiger als besonders schnelle Berechnungen ist zunächst immer ihre Richtigkeit. Zudem setzt Optimierung eine eingehende Diagnostik voraus: Nur wenn bekannt ist, welche Einzelschritte einer Funktion besonders viel Zeit benötigen, weiß man, wo sinnvollerweise Zeit einzusparen ist. Hier hilft die Funktion `system.time(<Befehl>)`, die ausgibt, wieviel Rechenzeit die Abarbeitung eines Befehls benötigt hat.¹⁶ Weitere Hinweise zum *profiling* finden sich unter `?Rprof`, die Ergebnisse lassen sich mit dem Paket `profvis` ([Chang, Luraschi & Mastny, 2023](#)) visualisieren. Die Speichernutzung eines Objekts erfährt man durch `object.size(<Objekt>)`.

```

# Zeit, um (400x400)-Zufallsmatrix zu invertieren
> system.time(solve(matrix(sample(1:100, 400^2, replace=TRUE),
+                               nrow=400)))
User  System  verstrichen
0.06    0.00      0.09

# Speicherverbrauch double-Gleitkommazahlen: N*8 byte + overhead
> xDbl <- rnorm(10000, 0, 10)                      # Gleitkommazahlen
> object.size(xDbl)
80048 bytes

```

¹⁶Erweiterungen liefern die Pakete `bench` ([Hester, 2020](#)) und `microbenchmark` ([Mersmann, Beleites, Hurling, Friedman & Ulrich, 2023](#)).

```
# Speicherverbrauch ganze Zahlen (integer): N*4 byte + overhead
> object.size(as.integer(xDbl))
40048 bytes
```

Bei der Analyse extrem großer Datenmengen besteht gegenwärtig das Problem, dass R Datensätze zur Bearbeitung im Arbeitsspeicher vorhalten muss, was die Größe von praktisch auswertbaren Datensätzen einschränkt (s. [?Memory](#)). Für Ansätze, diese Einschränkung zu umgehen s. den Abschnitt *High-Performance and Parallel Computing* der CRAN Task Views ([Eddelbuettel, 2020](#)). Er nennt auch Mittel zur besseren Ausnutzung paralleler Rechnerarchitekturen (Abschn. [17.5.2](#)) sowie Möglichkeiten, besonders zeitkritische Auswertungsschritte in kompilierte Sprachen wie C++ auszulagern ([Eddelbuettel, 2013](#)). Generell bieten folgende Ratschläge Potential, die Geschwindigkeit von Analysen zu erhöhen:

1. Ein großzügig dimensionierter Hauptspeicher samt angepasster maximaler Speichernutzung (s. [?Memory](#)) ist generell sinnvoll.
2. Objekte schrittweise zu vergrößern, ist aufgrund der dafür notwendigen internen Kopiervorgänge ineffizient. Objekte sollten bereits mit der Größe angelegt werden, die sie später benötigen. Dabei ist der später benötigte Datentyp zu wählen – fügt man etwa einer vorher angelegten Matrix aus logischen Werten später eine Zahl hinzu, muss die gesamte Matrix per Kopie in einen numerischen Datentyp konvertiert werden (Abschn. [1.4.5](#)).
3. Bei großen Datensätzen ist nach Möglichkeiten zu suchen, nur mit Teilmengen der Daten zu arbeiten und ggf. eine Datenbank als Speicherort zu nutzen. Über eine lokale Datenbankanbindung können Daten evtl. schneller als aus einer Datei eingelesen werden. Für Daten in Textform aus sehr großen Dateien arbeitet `fread()` aus dem Paket `data.table` deutlich schneller als `read.table()`. Eine Alternative ist das Apache Arrow Format, das vom Paket `arrow` ([Richardson et al., 2024](#)) unterstützt wird.
4. Generell ist die Leistungsfähigkeit beim Umgang mit großen Datenmengen eine Stärke des Pakets `data.table`. Manche Aufgaben, die mit dem Basisumfang von R nur sehr langsam, oder mit dem zur Verfügung stehenden Arbeitsspeicher gar nicht bearbeitet werden können, lassen sich mit `data.table` gut lösen.
5. Ganzzahlige Werte können mit `as.integer()` als solche gespeichert werden und benötigen dadurch weniger Speicher als die normalen Gleitkommazahlen doppelter Genauigkeit (Abschn. [1.4.5](#)).
6. Werden sehr große Objekte nicht mehr benötigt, sollten sie mit `rm()` entfernt werden. Dabei ist auch an das automatisch erzeugte Objekt `.Last.value` zu denken. Der Speicher wird mit der nächsten *garbage collection* freigegeben, die man manuell mit `gc()` anstoßen kann.
7. Objekte ohne Namensattribut (z. B. im Fall von Vektoren oder Matrizen) werden schneller als solche mit Namen verarbeitet. Zudem bieten Funktionen wie `lapply()`, `sapply()` oder `mapply()` an, Namensattribute mit dem Argument `USE.NAMES=FALSE` unberücksichtigt zu lassen.
8. Matrizen sind effizienter zu verarbeiten als Datensätze.

9. Bei vielen Auswertungen kommen intern Matrix-Operationen zum Einsatz (z. B. Singulärwertzerlegung, Abschn. 12.1), die bei sehr großen Datenmengen die Auswertungszeit entscheidend bestimmen können. Diese Rechnungen können von einer für den im Computer verwendeten Prozessor optimierten Version der *BLAS*-Bibliothek profitieren (*basic linear algebra subprograms*, s. Frage 8.2 in [Ripley et al., 2024](#)).
10. Schleifen sind in R als interpretierter und nicht kompilierter Sprache ein eher ineffizientes Sprachkonstrukt. Nach Möglichkeit sollte deshalb die meist schnellere und für R typische vektorwertige Formulierung von Rechnungen gewählt werden ([Ligges & Fox, 2008](#)). Weiter ist darauf zu achten, innerhalb einer oft durchlaufenen Schleife nur unbedingt notwendige Berechnungen durchzuführen. Objekte zu initialisieren sowie vorher feststehende Ergebnisse zu berechnen, sollte dagegen vorher außerhalb der Schleife geschehen. Datensätze sind in Schleifen besonders ineffizient, weil R bei jedem Durchlauf die Integrität eines Datensatzes prüft, etwa bzgl. der richtigen Anzahl von Beobachtungen in jeder Variable. Dies ist bei etwa bei Listen nicht der Fall.
11. Während `apply()` in seiner Verwendung parallelisiert scheint, beruht es tatsächlich auf R-Schleifen. Dagegen greift `lapply()` (und damit `sapply()`) intern auf kompilierten Code zurück, was für einen Geschwindigkeitsvorteil sorgen kann.

17.5.2 Auswertungen parallelisieren

Auch wenn auf einem Computer mehrere Prozessoren oder zumindest logische Rechenkerne (*cores*) zur Verfügung stehen, nutzt R für Berechnungen im Normalfall nur einen davon. Man kann jedoch dafür sorgen, dass umfangreiche Auswertungen auf mehrere Kerne verteilt parallel laufen. Dadurch kann die insgesamt aufgewendete Verarbeitungszeit deutlich sinken. Der im Folgenden vorgestellte Ablauf verwendet Funktionen des im Basisumfang von R enthaltenen Pakets `parallel`. Besonders für *high performance computing* Umgebungen ist das ergänzende Paket `parallelly` ([Bengtsson, 2023](#)) empfehlenswert.

Demonstriert werden soll hier der einfachste Fall paralleler Verarbeitung, bei dem es sich um *embarassingly parallel* Aufgaben handelt. Bei diesen ist offensichtlich, wie sie sich in einzelne, unabhängig voneinander ausführbare Teilaufgaben strukturieren lassen. Dies ist etwa der Fall, wenn beim bootstrap (Abschn. 11.1) oder bei der Kreuzvalidierung (Abschn. 13.1) dasselbe Regressionsmodell immer wieder für jeweils andere Daten angepasst werden muss.

1. Zunächst ist mit `detectCores(logical=TRUE)` zu ermitteln, wie viele logische Kerne im Prinzip genutzt werden können. Mit `logical=FALSE` erfährt man die Anzahl tatsächlicher Prozessoren, wobei das Ergebnis abhängig vom Betriebssystem sein kann. Es ist deshalb zu empfehlen, das Ergebnis von `detectCores()` mit unabhängigen Informationen zur verfügbaren Hardware des Computers zu validieren.
2. `makeCluster(<Anzahl>)` erzeugt die gewünschte Anzahl an Unterprozessen (*worker*). Jedem worker wird typischerweise ein Rechenkern zugewiesen. Das Ergebnis `<cluster>` ist eine Liste, die in allen weiteren Verarbeitungsschritten die Kommunikation mit den workern ermöglicht. In jedem worker wird eine unabhängige Instanz von R ausgeführt, so dass sie sich als *cluster* von getrennten Computern betrachten lassen. Wenn R alle

Kerne eines Computers verwendet, kann dies die Systemlast so stark erhöhen, dass sich der Computer in der Zwischenzeit nur noch schlecht für andere Tätigkeiten nutzen lässt.

3. Da die worker unabhängig vom Hauptprozess sind, muss ihre jeweilige R-session so vorbereitet werden, dass sie die Auswertung durchführen kann. Insbesondere müssen mit `clusterEvalQ(<cluster>, <Befehl>)` ggf. notwendige Zusatzpakete geladen werden. Außerdem sind mit `clusterExport(<cluster>, c("<Objekt1>", ...))` die Namen der Objekte zu nennen, die vom Hauptprozess in die jeweilige R-session der worker kopiert werden müssen. Relevant sind dabei alle Objekte, auf die die im worker später ausgeführte Funktion zugreift.
4. Die von einem worker jeweils in einem Schritt zu verarbeitenden Daten müssen als Komponente einer Liste vorliegen, wie sie etwa `lapply()` oder `split()` aus einem Datensatz erzeugt. Es kann mehr solcher Komponenten als verwendete Rechenkerne geben. Die Komponenten werden dann nach und nach an die worker verteilt und dort verarbeitet. Alternativ kann man mit `clusterExport()` zunächst den gesamten Datensatz allen workern zur Verfügung stellen und mit `splitIndices(<# Indizes>, <# Komponenten>)` seine Indizes gleichmäßig auf die Komponenten einer Liste verteilen. Die in den Workern ausgeführte Funktion erhält dann nicht jeweils eine Teilmenge der Daten selbst, sondern nur eine Teilmenge der Indizes zum gesamten Datensatz.
5. `parLapply(<cluster>, X=<Liste>, fun=<Funktion>, ...)` wendet wie `lapply()` die mit `fun` definierte Funktion auf jede Komponente der Liste `X` an und sammelt die jeweiligen Ergebnisse als Komponenten einer Liste. Benötigt `fun` weitere Argumente, können diese an Stelle der `...` übergeben werden. Die Besonderheit von `parLapply()` besteht darin, dass die Komponenten von `X` an die worker des bezeichneten clusters verteilt und dort jeweils parallel verarbeitet werden. Gegebenenfalls kann man die von `parLapply()` ausgegebene Liste mit `unlist()` in einen Vektor oder mit `do.call(rbind.data.frame, <-> <Liste>)` in einen Datensatz konvertieren.
6. Schließlich beendet `stopCluster(<cluster>)` die Prozesse der worker.

Als Beispiel soll die k -fache Kreuzvalidierung (13.1.1) einer ordinalen Regression (Abschn. 8.2.1) dienen. Dabei werden die dafür notwendigen k Modellanpassungen parallel durchgeführt. Aus jeder Anpassung wird das odds ratio der ersten Kovariate gespeichert, ebenso die Rate der korrekten Klassifikation der vorhergesagten Kategorien als Maß für die Vorhersagegüte in der Teststichprobe.

```
> N      <- 1000                                # Daten simulieren
> X1    <- rnorm(N, 175, 7)                      # Prädiktor 1
> X2    <- rnorm(N, 30, 8)                       # Prädiktor 2
> Ycont <- 0.5*X1 - 0.3*X2 + 10 + rnorm(N, 0, 6)  # stetige Zielgröße

# ordinale Zielgröße
> Yord <- cut(Ycont, breaks=quantile(Ycont), include.lowest=TRUE,
+               labels=c("--", "-", "+", "++"), ordered=TRUE)

# parallel ausgeführte Kreuzvalidierung
> library(parallel)                            # für detectCores(), ...
```

```

> datOrd <- data.frame(X1, X2, Yord)           # Datensatz
> k      <- detectCores() - 1                  # Anzahl verwendeter Kerne
> cl     <- makeCluster(k)                     # cluster erzeugen
> stuff  <- clusterEvalQ(cl, library(VGAM))   # Paket in workern laden
> clusterExport(cl, "datOrd")                  # Daten in worker kopieren

# numerische Indizes Teststichprobe -> Rate korrekter Klassifikation
> getPerf <- function(idx) {
+   datTst <- datOrd[ idx, , drop=FALSE]    # Teststichprobe
+   datTrn <- datOrd[-idx, , drop=FALSE]    # Trainingsstichprobe
+   # Regression in Trainingsstichprobe anpassen
+   fit <- vglm(Yord ~ X1 + X2, family=propodds, data=datTrn)
+   # vorhergesagte Gruppenwahrscheinlichkeiten in Teststichprobe
+   Phat <- predict(fit, newdata=datTst, type="response")
+   # vorhergesagte Kategorien in Teststichprobe
+   catHat <- levels(datOrd$Yord)[max.col(Phat)]
+   # beobachtete vs. vorhergesagte Kategorien in Teststichprobe
+   cTab <- xtabs(~ Yord + catHat, data=datTst)
+   cbind(OR=exp(VGAM::coef(fit))["X1"],   # Odds Ratio X1
+         CCR=sum(diag(cTab)) / sum(cTab)) # Rate korrekte Klassif.
+ }

# Kreuzvalidierung: partitioniere Daten in k zufällige Gruppen
> tstGrp <- sample(seq_len(N), size=N, replace=FALSE) %% k
> foldsL <- split(seq_along(tstGrp), tstGrp) # Liste k Teststichproben
> outL   <- parLapply(cl, foldsL, getPerf)    # getPerf() parallel
> stopCluster(cl)                           # worker beenden
> do.call(rbind.data.frame, outL)            # Ergebnisse aller worker
      OR_X1      CCR
X1  1.144516  0.3263403
X11 1.158220  0.2217036
X12 1.156725  0.3488915
X13 1.149187  0.3325554

```

Bei der parallelen Verarbeitung nach obigem Schema sind folgende Aspekte zu beachten:

- Bei der Verwendung von Zufallszahlen in workern sind Ergebnisse bei gleichem seed nur dann reproduzierbar, wenn vor dem Aufruf von `set.seed()` im Hauptprozess zunächst mit `RNGkind("L'Ecuyer-CMRG")` explizit ein passender Generator ausgewählt wird (Abschn. 2.4, Fußnote 11). Details erläutert `vignette("parallel")`.
- Die Speichernutzung kann deutlich höher als bei seriellen Arbeitsschritten ausfallen. Nicht ausreichender Hauptspeicher kann zu zufällig auftretenden und schwer diagnostizierbaren Fehlern führen. Es empfiehlt sich daher, die Speichernutzung vorher genau zu analysieren und an den verfügbaren Hauptspeicher anzupassen.
- Typischerweise nimmt die beanspruchte Zeitdauer nicht proportional mit der Anzahl der eingesetzten Rechenkerne ab. Eine Ursache dafür ist, dass die Kommunikation zwischen

Hauptprozess und workern *overhead* erzeugt und ihrerseits Zeit beansprucht. Hauptprozess und worker kommunizieren u.a. dann, wenn ein worker die übergebenen Daten bearbeitet hat und neue Daten erhält. Parallelisierung lohnt sich erst dann, wenn der pro worker jeweils zu erledigende Arbeitsschritt so umfassend ist, dass die durch overhead verlorene Zeit nicht stark ins Gewicht fällt. Nach Möglichkeit sollten deshalb die jeweils unabhängigen Arbeitsschritte in größere *batches* gebündelt werden.

Unter Unix-artigen Betriebssystemen wie Linux und MacOS gibt es als weiteren Weg zur Parallelisierung eine effiziente Möglichkeit, den Hauptprozess mit der darin ausgeführten R-session per *fork* zu kopieren. Auf dieser Möglichkeit basiert die Funktion `mclapply()`. Sie vereinfacht den skizzierten Ablauf, indem sie die Schritte 2, 3, 5 und 6 in einem Aufruf vereint. `mclapply()` hat den Vorteil, dass die erstellten Unterprozesse automatisch dieselben Speicherinhalte und Ressourcen wie der Hauptprozess besitzen. Deshalb müssen mit dieser Methode die R-sessions in den workern nicht manuell für die Datenauswertung vorbereitet werden, wenn dies schon im Hauptprozess erledigt wurde.¹⁷

```
# Beispiel unter Linux / MacOS statt parLapply(), aber ohne
# makeCluster() -> clusterEvalQ() -> clusterExport() -> stopCluster()
> library(VGAM)
> outL <- mclapply(foldsL, getPerf, mc.cores=k)
```

Manche R Funktionen bieten direkt Unterstützung für parallele Ausführung, etwa `boot()` aus dem Paket `boot` mit den Optionen `parallel` und `ncpus` (Abschn. 11.1.1). Gibt man im Aufruf von `install.packages()` für die Option `Ncpus` die Anzahl zu verwendender Rechenkerne an, wird die Installation von Zusatzpaketen parallel ausgeführt. Ist die globale Option `options(Ncpus=<Anzahl>)` gesetzt, greifen `install.packages()` und damit auch `update.packages()` automatisch darauf zu. Weiterhin gibt es die Möglichkeit, eine Variante der BLAS Systembibliothek für lineare Algebra zu verwenden, die automatisch parallel arbeitet (Abschn. 17.5.1, Punkt 9).

¹⁷Allerdings sorgen die Besonderheiten der `fork` Technik für mögliche Probleme, etwa wenn Verbindungen zum Lesen und Schreiben von Daten parallel verwendet werden. Auch sind nicht alle Zusatzpakete für die `fork` Technik geeignet. Zudem sollte R dann nicht innerhalb von RStudio oder einer anderen grafischen Entwicklungsumgebung laufen. Details erläutert `?mclapply`.

Literaturverzeichnis

- Adler, D. & Murdoch, D. (2020). rgl: 3D visualization device system (OpenGL) [Software]. URL <https://CRAN.R-project.org/package=rgl> (R package version 0.103.5)
- Agresti, A. (2015). *Foundations of linear and generalized linear models*. New York, NY: Wiley.
- Agresti, A. (2018). *An introduction to categorical data analysis* (3. Aufl.). New York, NY: Wiley.
- Aiken, L. S. & West, S. G. (1991). *Multiple regression: Testing and interpreting interactions*. Thousand Oaks, CA: Sage.
- Allaire, J. & Dervieux, C. (2024). quarto: R interface to ‘Quarto’ markdown publishing system [Software]. Boston, MA. URL <https://CRAN.R-project.org/package=quarto> (Version 1.4.4)
- Allaire, J. J., Ushey, K. & Tang, Y. (2020). reticulate: Interface to ‘Python’ [Software]. URL <https://CRAN.R-project.org/package=reticulate> (R package version 1.18)
- Allaire, J. J., Xie, Y., McPherson, J., Luraschi, J., Ushey, K., Atkins, A., ... Iannone, R. (2020). rmarkdown: Dynamic documents for R [Software]. URL <https://rmarkdown.rstudio.com/> (R package version 2.6)
- Allignol, A. & Latouche, A. (2020). CRAN task view: Survival analysis. URL <https://CRAN.R-project.org/view=Survival> (Version 2020-03-14)
- Allison, P. D. (2003). Convergence problems in logistic regression. In M. Altman, J. Gill & M. P. McDonald (Hrsg.), *Numerical issues in statistical computing for the social scientist* (S. 238–252). Hoboken, NJ: Wiley.
- Allison, P. D. (2008). Convergence failures in logistic regression. *SAS Global Forum, 360*, 1–11. URL <https://support.sas.com/resources/papers/proceedings/pdfs/sgf2008/360-2008.pdf>
- Arnold, J. B. (2019). ggthemes: Extra themes, scales and geoms for ‘ggplot2’ [Software]. URL <https://CRAN.R-project.org/package=ggthemes> (R package version 4.2.0)
- Backhaus, K., Erichson, B., Plinke, W. & Weiber, R. (2018). *Multivariate Analysemethoden* (15. Aufl.). Heidelberg: Springer Gabler.
- Backhaus, K., Erichson, B. & Weiber, R. (2015). *Fortgeschrittene Multivariate Analysemethoden* (3. Aufl.). Heidelberg: Springer Gabler.
- Balasubramanian, N., Johnson, S. G., Hahn, T., Bouvier, A. & Kieu, K. (2020). cubature: Adaptive multivariate integration over hypercubes [Software]. URL <https://CRAN.R-project.org/package=cubature> (R package version 2.0.4.1)
- Barthelme, S. (2020). imager: Image processing library based on ‘CImg’ [Software]. URL <https://CRAN.R-project.org/package=imager> (R package version 0.42.1)
- Bates, D. (2004). Least squares calculations in R. *R News*, 4 (1), 17–20. URL <https://CRAN.R-project.org/doc/Rnews/>
- Bates, D. & Maechler, M. (2019). Matrix: Sparse and dense matrix classes and methods [Software]. URL <https://CRAN.R-project.org/package=Matrix> (R package version 1.2-18)

- Bates, D., Maechler, M., Bolker, B. & Walker, S. (2015). Fitting linear mixed-effects models using lme4. *Journal of Statistical Software*, 67 (1), 1–48. URL <https://www.jstatsoft.org/article/view/v067i01/>
- Becker, R. A., Chambers, J. M. & Wilks, A. R. (1988). *The new S language: A programming environment for data analysis and graphics*. Pacific Grove, CA: Wadsworth & Brooks-/Cole.
- Ben-Shachar, M., Makowski, D., Lüdecke, D., Patil, I., Wiernik, B., Thériault, M., ... Waggoner, P. (2024). effectsize: Indices of effect size and standardized parameters [Software]. URL <https://CRAN.R-project.org/package=effectsize> (R package version 0.8.8)
- Bender, R., Augustin, T. & Blettner, M. (2005). Generating survival times to simulate Cox proportional hazards models. *Statistics in Medicine*, 24, 1713–1723.
- Bengtsson, H. (2023). parallelly: Enhancing the ‘parallel’ package [Software]. URL <https://CRAN.R-project.org/package=parallelly> (Version 1.37.1)
- Bernaards, C. A. & Jennrich, R. I. (2005). Gradient projection algorithms and software for arbitrary rotation criteria in factor analysis. *Educational and Psychological Measurement*, 65 (5), 676–696.
- Beyersmann, J., Allignol, A. & Schumacher, M. (2012). *Competing risks and multistate models with R*. New York, NY: Springer.
- Blair, R. & Higgins, J. (1978). Tests of hypotheses for unbalanced factorial designs under various regression/coding method combinations. *Educational and Psychological Measurement*, 38, 621–631.
- Bliese, P. (2016). multilevel: Multilevel functions [Software]. URL <https://CRAN.R-project.org/package=multilevel> (R package version 2.6)
- Blischak, J. & Hill, A. (2020). CRAN task view: Reproducible research. URL <https://CRAN.R-project.org/view=ReproducibleResearch> (Version 2020-12-16)
- Bloomfield, V. A. (2014). *Using R for numerical analysis in science and engineering*. Boca Raton, FL: Chapman & Hall/CRC.
- Boettiger, C. & Eddelbuettel, D. (2017). An introduction to Rocker: Docker containers for R. *The R Journal*, 9, 527–536. URL <https://journal.r-project.org/archive/2017/RJ-2017-065/>
- Boker, S., Neale, M., Maes, H., Wilde, M., Spiegel, M., Brick, T., ... Fox, J. (2011). OpenMx: An open source extended structural equation modeling framework. *Psychometrika*, 66 (2), 306–317. URL <https://openmx.ssri.psu.edu/>
- Bolker, B., Piaskowski, J., Tanaka, E., Alday, P. & Viechtbauer, W. (2024). CRAN task view: Mixed, multilevel, and hierarchical models in R. URL <https://CRAN.R-project.org/view=MixedModels> (Version 2024-05-08)
- Borchers, H. W. (2019a). CRAN task view: Numerical mathematics. URL <https://CRAN.R-project.org/view=NumericalMathematics> (Version 2020-01-17)
- Borchers, H. W. (2019b). pracma: Practical numerical math functions [Software]. URL <https://CRAN.R-project.org/package=pracma> (R package version 2.2.9)
- Bortz, J., Lienert, G. A. & Boehnke, K. (2010). *Verteilungsfreie Methoden in der Biostatistik* (3. Aufl.). Heidelberg: Springer.
- Brilleman, S. (2021). simsurv: Simulate survival data [Software]. URL <https://CRAN.R-project.org/package=simsurv> (R package version 1.0.0)
- Bronstein, I. N. & Semendjajew, K. A. (2012). *Springer Taschenbuch der Mathematik* (3. Aufl.). Berlin: Springer.

- Büning, H. & Trenkler, G. (1994). *Nichtparametrische statistische Methoden* (2. Aufl.). Berlin: Walter de Gruyter.
- Bürkner, P.-C. (2020). brms: Bayesian regression models using Stan [Software]. URL <https://CRAN.R-project.org/package=brms> (R package version 2.14.4)
- Canty, A. & Ripley, B. D. (2020). boot: Bootstrap R (S-Plus) functions [Software]. URL <https://CRAN.R-project.org/package=boot> (R package version 1.3-25)
- Carr, D., Lewin-Koh, N. & Maechler, M. (2020). hexbin: Hexagonal binning routines [Software]. URL <https://CRAN.R-project.org/package=hexbin> (R package version 1.28.1)
- Chamberlain, S., Leeper, T., Mair, P., Ram, K. & Gandrud, C. (2020). CRAN task view: Web technologies and services. URL <https://CRAN.R-project.org/view=WebTechnologies> (Version 2020-11-12)
- Chambers, J. M. (2016). *Extending R*. Boca Raton, FL: Chapman & Hall/CRC.
- Champely, S. & De Rosario, H. (2020). pwr: Basic functions for power analysis [Software]. URL <https://CRAN.R-project.org/package=pwr> (R package version 1.3-0)
- Chang, W. (2018). *R Graphics Cookbook* (2. Aufl.). Sebastopol, CA: O'Reilly. URL <https://r-graphics.org/>
- Chang, W., Cheng, J., Allaire, J. J., Xie, Y. & McPherson, J. (2020). shiny: Web application framework for R [Software]. URL <https://CRAN.R-project.org/package=shiny> (R package version 1.5.0)
- Chang, W., Luraschi, J. & Mastny, T. (2023). profvis: Interactive visualizations for profiling R code [Software]. URL <https://CRAN.R-project.org/package=profvis> (R package version 0.3.8)
- Chihara, L. & Hesterberg, T. (2022). *Mathematical statistics with resampling and R* (3. Aufl.). Hoboken, NJ: Wiley. URL <https://github.com/lchihara/MathStatsResamplingR>
- Christensen, R. (2020). *Plane answers to complex questions: The theory of linear models* (5. Aufl.). New York, NY: Springer.
- Circle Systems. (2019). Stat/Transfer [Software]. URL <https://www.stattransfer.com/> (Version 15)
- Clarke, E. & Sherrill-Mix, S. (2017). ggbeeswarm: Categorical scatter (violin point) plots [Software]. URL <https://CRAN.R-project.org/package=ggbeeswarm> (R package version 0.6.0)
- Cowlishaw, M. F. (2008). *Decimal arithmetic FAQ Part 1: general questions*. URL <http://speleotrove.com/decimal/decifaq1.html>
- Cribari-Neto, F. & Zeileis, A. (2010). Beta regression in R. *Journal of Statistical Software*, 34 (2), 1–24. doi: 10.18637/jss.v034.i02
- Csárdi, G. & Hester, J. (2024). pak: Another approach to package installation [Software]. URL <https://CRAN.R-project.org/package=pak> (R package version 0.7.2)
- Dalgaard, P. (2007). New functions for multivariate analysis. *R News*, 7 (2), 2–7. URL <https://CRAN.R-project.org/doc/Rnews/>
- Davison, A. C. & Hinkley, D. V. (1997). *Bootstrap methods and their applications*. Cambridge, UK: Cambridge University Press.
- De Cock, B., Nieboer, D., Van Calster, B., Steyerberg, E. & Vergouwe, Y. (2024). CalibrationCurves: Calibration performance [Software]. URL <https://CRAN.R-project.org/package=CalibrationCurves> (R package version 2.0.3)
- Delignette-Muller, M. L. & Dutang, C. (2015). fitdistrplus: An R package for fitting distributions. *Journal of Statistical Software*, 64 (4), 1–34. URL <https://www.jstatsoft.org/v64/i04/>

- Dowle, M. & Srinivasan, A. (2020). *data.table*: Extension of *data.frame* [Software]. URL <https://CRAN.R-project.org/package=data.table> (R package version 1.13.6)
- Dutang, C. & Kiener, P. (2020). *CRAN task view: Probability distributions*. URL <https://CRAN.R-project.org/view=Distributions> (Version 2020-01-26)
- Eddelbuettel, D. (2013). *Seamless R and C++ integration with Rcpp*. New York, NY: Springer. URL <http://www.rcpp.org/>
- Eddelbuettel, D. (2020). *CRAN task view: High-performance and parallel computing with R*. URL <https://CRAN.R-project.org/view=HighPerformanceComputing> (Version 2020-01-17)
- Eid, M., Gollwitzer, M. & Schmitt, M. (2017). *Statistik und Forschungsmethoden* (5. Aufl.). Weinheim: Beltz.
- Eklund, A. (2016). *beeswarm*: The bee swarm plot, an alternative to stripchart [Software]. URL <https://CRAN.R-project.org/package=beeswarm> (R package version 0.2.3)
- Enamorado, T., Fifield, B. & Imai, K. (2018). *fastLink*: Fast probabilistic record linkage with missing data [Software]. URL <https://CRAN.R-project.org/package=fastLink> (R package version 0.5.0)
- Fagerland, M. W. (2024). *contingencytables*: Statistical analysis of contingency tables [Software]. URL <https://CRAN.R-project.org/package=contingencytables> (R package version 3.0.0)
- Fagerland, M. W., Lydersen, S. & Laake, P. (2017). *Statistical analysis of contingency tables*. Boca Raton, FL: Chapman & Hall/CRC. URL <https://contingencytables.com/>
- Faraway, J. J. (2014). *Linear models with R* (2. Aufl.). Boca Raton, FL: Chapman & Hall/CRC. URL <https://julianfaraway.github.io/faraway/LMR/>
- Faraway, J. J. (2016). *Extending the linear model with R: Generalized linear, mixed effects and nonparametric regression models* (2. Aufl.). Boca Raton, FL: Chapman & Hall/CRC. URL <https://julianfaraway.github.io/faraway/ELM/>
- Faul, F., Erdfelder, E., Lang, A.-G. & 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 <https://www.gpower.hhu.de/>
- Fay, M. P. (2020). *exact2x2*: Exact tests and confidence intervals for 2x2 tables [Software]. URL <https://CRAN.R-project.org/package=exact2x2> (R package version 1.6.9)
- Fieller, N. (2017). *Basics of matrix algebra for statistics with R*. Boca Raton, FL: Chapman & Hall/CRC.
- Filzmoser, P., Fritz, H. & Kalcher, K. (2018). *pcaPP*: Robust PCA by projection pursuit [Software]. URL <https://CRAN.R-project.org/package=pcaPP> (R package version 1.9-73)
- Filzmoser, P. & Gschwandtner, M. (2018). *mvoutlier*: Multivariate outlier detection based on robust methods [Software]. URL <https://CRAN.R-project.org/package=mvoutlier> (R package version 2.0.9)
- Finley, A., Banerjee, S. & Hjelle, Ø. (2024). *MBA*: Multilevel b-spline approximation [Software]. URL <https://CRAN.R-project.org/package=MBA> (R package version 1.1-6)
- Fischer, G. & Springborn, B. (2020). *Lineare Algebra: Eine Einführung für Studienanfänger* (19. Aufl.). Wiesbaden: Springer Spektrum.
- Fox, J., Friendly, M. & Weisberg, S. (2013). Hypothesis tests for multivariate linear models using the car package. *The R Journal*, 5 (1), 39–52. URL <https://journal.r-project.org/archive/2013-1/>

- Fox, J. & Monette, G. (2024). cv: Cross-validating regression models [Software]. URL <https://CRAN.R-project.org/package=cv> (R package version 2.0.0)
- Fox, J., Nie, Z. & Byrnes, J. (2020). sem: Structural equation models [Software]. URL <https://CRAN.R-project.org/package=sem> (R package version 3.1-11)
- Fox, J. & Weisberg, S. (2019). *An R Companion to applied regression* (3. Aufl.). Thousand Oaks, CA: Sage. URL <https://www.john-fox.ca/Companion/>
- Fox, J. & Weisberg, S. (2020). car: Companion to applied regression [Software]. URL <https://CRAN.R-project.org/package=car> (R package version 3.0-10)
- Fox, J., Weisberg, S. & Price, B. (2020). carData: Companion to applied regression data sets [Software]. URL <https://CRAN.R-project.org/package=carData> (R package version 3.0-4)
- Frick, H., Chow, F., Kuhn, M., Mahone, M., Silge, J. & Wickham, H. (2024). rsample: General resampling infrastructure [Software]. URL <https://CRAN.R-project.org/package=rsample> (R package version 1.2.1)
- Friedman, J., Hastie, T. & Tibshirani, R. (2010). Regularization paths for generalized linear models via coordinate descent. *Journal of Statistical Software*, 33 (1), 1–22. URL <https://www.jstatsoft.org/v33/i01/>
- Gabry, J. & Goodrich, B. (2020). rstanarm: Bayesian applied regression modeling via Stan [Software]. URL <https://CRAN.R-project.org/package=rstanarm> (R package version 2.21.1)
- Gandrud, C. (2020). *Reproducible research with R & RStudio* (3. Aufl.). Boca Raton, FL: Chapman & Hall/CRC. URL <https://github.com/christophergandrud/Rep-Res-Book>
- Gebhardt, A., Bivand, R. & Sinclair, D. (2024). interp: Interpolation methods [Software]. URL <https://CRAN.R-project.org/package=interp> (R package version 1.1-6)
- Gentleman, R. C., Carey, V. J., Bates, D. M., Bolstad, B., Dettling, M., Dudoit, S., ... Zhang, J. (2004). Bioconductor: Open software development for computational biology and bioinformatics. *Genome Biology*, 5, R80. URL <https://genomebiology.biomedcentral.com/articles/10.1186/gb-2004-5-10-r80>
- Genz, A. & Bretz, F. (2009). *Computation of multivariate normal and t probabilities. Lecture notes in statistics Vol. 195*. Heidelberg: Springer.
- Genz, A., Bretz, F., Miwa, T., Mi, X. & Hothorn, T. (2020). mvtnorm: Multivariate normal and t distributions [Software]. URL <https://CRAN.R-project.org/package=mvtnorm> (R package version 1.1-1)
- Gilbert, P. & Varadhan, R. (2019). numDeriv: Accurate numerical derivatives [Software]. URL <https://CRAN.R-project.org/package=numDeriv> (R package version 2016.8-1.1)
- Gillespie, C. & Lovelace, R. (2017). *Efficient R programming*. Sebastopol, CA: O'Reilly. URL <https://csgillespie.github.io/efficientR/>
- Gohel, D. (2020). flextable: Functions for tabular reporting [Software]. URL <https://CRAN.R-project.org/package=flextable> (R package version 0.5.9)
- Goldberg, D. (1991). What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23 (1), 5–48. URL <https://www.validlab.com/goldberg/paper.pdf>
- Goldfeld, K. & Wujciak-Jens, J. (2024). simstudy: Simulation of study data [Software]. URL <https://CRAN.R-project.org/package=simstudy> (R package version 0.8.1)
- Good, P. I. (2004). *Permutation, parametric, and bootstrap tests of hypotheses* (3. Aufl.). New York, NY: Springer.

- Goulet, V., Dutang, C., Maechler, M., Firth, D., Shapira, M. & Stadelmann, M. (2020). expm: Matrix exponential [Software]. URL <https://CRAN.R-project.org/package=expm> (R package version 0.999-5)
- Goyvaerts, J. & Levithan, S. (2012). *Regular expressions cookbook*. Sebastopol, CA: O'Reilly. URL <https://www.regular-expressions.info/>
- Grolemund, G. & Wickham, H. (2011). Dates and times made easy with lubridate. *Journal of Statistical Software*, 40 (3), 1–25. URL <https://www.jstatsoft.org/v40/i03/>
- Gross, J. & Ligges, U. (2015). nortest: Tests for normality [Software]. URL <https://CRAN.R-project.org/package=nortest> (R package version 1.0-4)
- Grothendieck, G. & Petzoldt, T. (2004). Date and time classes in R. *R News*, 4 (1), 29–32. URL <https://CRAN.R-project.org/doc/Rnews/>
- Grün, B., Kosmidis, I. & Zeileis, A. (2012). Extended beta regression in R: Shaken, stirred, mixed, and partitioned. *Journal of Statistical Software*, 48 (11), 1–25. doi: 10.18637/jss.v048.i11
- Harrell Jr, F. E. (2015). *Regression modeling strategies* (2. Aufl.). New York, NY: Springer. URL <https://hbiostat.org/doc/rms/book/>
- Harrell Jr, F. E. (2020a). Hmisc: Harrell miscellaneous [Software]. URL <https://CRAN.R-project.org/package=Hmisc> (R package version 4.4-2)
- Harrell Jr, F. E. (2020b). rms: Regression modeling strategies [Software]. URL <https://CRAN.R-project.org/package=rms> (R package version 6.1-0)
- Harrer, M., Cuijpers, P., Furukawa, T. & Ebert, D. (2021). *Doing meta-analysis with R: A hands-on guide*. Boca Raton, FL: Chapman Hall/CRC. URL https://bookdown.org/MathiasHarrer/Doing_Meta_Analysis_in_R/
- Hartig, F. (2022). DHARMa: Residual diagnostics for hierarchical (multi-level / mixed) regression models [Software]. URL <https://CRAN.R-project.org/package=DHARMa> (R package version 0.4.6)
- Hastie, T., Tibshirani, R. & Friedman, J. (2009). *The elements of statistical learning: Data mining, inference, and prediction* (2. Aufl.). New York, NY: Springer. URL <https://hastie.su.domains/ElemStatLearn/>
- Heinze, G. & Ploner, M. (2020). logistf: Firth's bias reduced logistic regression [Software]. URL <https://CRAN.R-project.org/package=logistf> (R package version 1.24)
- Hendrickx, J. (2019). perturb: Tools for evaluating collinearity [Software]. URL <https://CRAN.R-project.org/package=perturb> (R package version 2.10)
- Hester, J. (2020). bench: High precision timing of R expressions [Software]. URL <https://CRAN.R-project.org/package=bench> (R package version 1.1.1)
- Hester, J., Csárdi, G., Wickham, H., Chang, W., Morgan, M. & Tenenbaum, D. (2020). remotes: R package installation from remote repositories, including 'GitHub' [Software]. URL <https://CRAN.R-project.org/package=remotes> (R package version 2.2.0)
- Hester, J. & Wickham, H. (2020). odbc: Connect to ODBC compatible databases (using the DBI interface) [Software]. URL <https://CRAN.R-project.org/package=odbc> (R package version 1.2.2)
- Hewson, P. (2020). CRAN task view: Multivariate statistics. URL <https://CRAN.R-project.org/view=Multivariate> (Version 2020-03-04)
- Hoffman, D. D. (2000). *Visual intelligence: How we create what we see*. New York, NY: W. W. Norton & Company.
- Hofmann, M., Gatu, C., Kontoghiorghe, E. J., Colubi, A. & Zeileis, A. (2020). lmSubsets: Exact variable-subset selection in linear regression for R. *Journal of Statistical Software*,

- 93, 1–21. URL <https://www.jstatsoft.org/v93/i03/>
- Højsgaard, S., Halekoh, U. & Yan, J. (2006). The R package geepack for generalized estimating equations. *Journal of Statistical Software*, 15 (2), 1–11. URL <https://www.jstatsoft.org/v15/i02/>
- Honaker, J., King, G. & Blackwell, M. (2011). Amelia II: A program for missing data. *Journal of Statistical Software*, 45 (7), 1–47. URL <https://www.jstatsoft.org/v45/i07/>
- Hornik, K. & R Core Team. (2025). The R FAQ [Software-Handbuch]. URL <https://cran.r-project.org/doc/FAQ/R-FAQ.html>
- Hosmer Jr, D. W., Lemeshow, S. & May, S. (2008). *Applied survival analysis: Regression modeling of time to event data* (2. Aufl.). New York, NY: Wiley.
- Hothorn, T. (2020). CRAN task view: Machine learning & statistical learning. URL <https://CRAN.R-project.org/view=MachineLearning> (Version 2020-01-30)
- Hothorn, T., Bretz, F. & Westfall, P. (2008). Simultaneous inference in general parametric models. *Biometrical Journal*, 50 (3), 346–363.
- Hothorn, T. & Everitt, B. S. (2014). *A handbook of statistical analysis using R* (3. Aufl.). Boca Raton, FL: Chapman & Hall/CRC.
- Hothorn, T. & Everitt, B. S. (2020). HSAUR3: A handbook of statistical analyses using R (3rd edition) [Software]. URL <https://CRAN.R-project.org/package=HSAUR3> (R package version 1.0-10)
- Hothorn, T., Hornik, K., van de Wiel, M. A. & Zeileis, A. (2008). Implementing a class of permutation tests: The coin package. *Journal of Statistical Software*, 28 (8), 1–23. URL <https://www.jstatsoft.org/v28/i08/>
- Huber, W., Carey, V. J., Gentleman, R., Anders, S., Carlson, M., Carvalho, B. S., ... Morgan, M. (2015). Orchestrating high-throughput genomic analysis with Bioconductor. *Nature Methods*, 12, 115–121.
- Hugh-Jones, D. & Possenriede, D. (2024). santoku: A versatile cutting tool [Software]. URL <https://CRAN.R-project.org/package=santoku> (R package version 1.0)
- Hyndman, R. J. (2020). CRAN task view: Time series analysis. URL <https://CRAN.R-project.org/view=TimeSeries> (Version 2020-12-20)
- Hyndman, R. J. & Athanasopoulos, G. (2019). *Forecasting: Principles and practice* (3. Aufl.). Melbourne, Australia: OTexts. URL <https://otexts.com/fpp3/>
- Iannone, R., Cheng, J., Schloerke, B., Hughes, E., Lauer, A., Seo, J., ... Roy, O. (2024). gt: Easily create presentation-ready display tables [Software]. URL <https://CRAN.R-project.org/package=gt> (R package version 0.11.0)
- Ihaka, R. & 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., Fisher, J. C., Stauffer, R., Wilke, C. O., McWhite, C. D. & Zeileis, A. (2020). colorspace: Color space manipulation [Software]. URL <https://CRAN.R-project.org/package=colorspace> (R package version 2.0-0)
- James, G., Witten, D., Hastie, T. & Tibshirani, R. (2021). *An introduction to statistical learning with applications in R* (2. Aufl.). New York, NY: Springer. URL <https://statlearning.com/>
- Jombart, T., Rolland, M. & Gruson, H. (2024). CRAN task view: Epidemiology. URL <https://CRAN.R-project.org/view=Epidemiology> (Version 2024-06-20)
- Kelley, K. (2020). MBESS: The MBESS R package [Software]. URL <https://CRAN.R-project.org/package=MBESS> (R package version 4.8.0)

- Kirk, R. E. (2013). *Experimental design: Procedures for the social sciences* (4. Aufl.). Thousand Oaks, CA: SAGE. URL <https://studysites.sagepub.com/kirk/>
- Klein, J. P. & Moeschberger, M. L. (2003). *Survival analysis: Techniques for censored and truncated data* (2. Aufl.). New York, NY: Springer.
- Kleinman, K. & Horton, N. J. (2014). *SAS and R: Data management, statistical analysis, and graphics* (2. Aufl.). Boca Raton, FL: Chapman Hall/CRC. URL <https://nhorton.people.amherst.edu/sasr2/>
- Kluyver, T., Ragan-Kelley, B., Pérez, F., Granger, B., Bussonnier, M., Frederic, J., ... Willing, C. (2016). Jupyter notebooks: A publishing format for reproducible computational workflows. In F. Loizides & B. Schmidt (Hrsg.), *Positioning and power in academic publishing: Players, agents and agendas* (S. 87–90). IOS Press. URL <https://jupyter.org/>
- Koenker, R., Portnoy, S., Tian, P., Melly, B., Zeileis, A., Grosjean, P., ... BD, R. (2024). quantreg: Quantile regression [Software]. URL <https://CRAN.R-project.org/package=quantreg> (R package version 5.98)
- Kolde, R. (2019). pheatmap: Pretty heatmaps [Software]. URL <https://CRAN.R-project.org/package=pheatmap> (R package version 1.0.12)
- Korkmaz, S., Goksuluk, D. & Zarasiz, G. (2021). MVN: Multivariate normality tests [Software]. URL <https://CRAN.R-project.org/package=MVN> (R package version 5.9)
- Kruschke, J. K. (2015). *Doing Bayesian data analysis: A tutorial with R, JAGS, and Stan* (2. Aufl.). Amsterdam: Academic Press. URL <https://sites.google.com/site/doingbayesiandataanalysis/>
- Leisch, F. & Gruen, B. (2020). CRAN task view: Cluster analysis & finite mixture models. URL <https://CRAN.R-project.org/view=Cluster> (Version 2020-01-17)
- Lenth, R. (2020). emmeans: Estimated marginal means, aka least-squares means [Software]. URL <https://CRAN.R-project.org/package=emmeans> (R package version 1.5.3)
- Li, C. (2019). JuliaCall: Seamless integration between R and ‘Julia’ [Software]. URL <https://CRAN.R-project.org/package=JuliaCall> (R package version 0.17.1)
- Ligges, U. (2002). R help desk: Automation of mathematical annotation in plots. *R News*, 2 (3), 32–34. URL <https://CRAN.R-project.org/doc/Rnews/>
- Ligges, U. (2003). R help desk: Package management. *R News*, 3 (3), 37–39. URL <https://CRAN.R-project.org/doc/Rnews/>
- Ligges, U. & Fox, J. (2008). R help desk: How can I avoid this loop or make it faster? *R News*, 8 (1), 46–50. URL <https://CRAN.R-project.org/doc/Rnews/>
- Love, J., Dropmann, D. & Selker, R. (2024). jamovi: Sats. Open. Now [Software]. URL <https://www.jamovi.org/> (Version 2.5.4)
- Maechler, M. (2020). CRAN task view: Robust statistical methods. URL <https://CRAN.R-project.org/view=Robust> (Version 2020-03-11)
- Maindonald, J. & Braun, W. J. (2020). DAAG: Data analysis and graphics data and functions [Software]. URL <https://CRAN.R-project.org/package=DAAG> (R package version 1.24)
- Maindonald, J., Braun, W. J. & Andrews, J. L. (2024). *A practical guide to data analysis using R: An example-based approach*. Cambridge, UK: Cambridge University Press. URL <https://github.com/jhmaindonald/PGRcode>
- Mair, P. (20120). CRAN task view: Psychometric models and methods. URL <https://CRAN.R-project.org/view=Psychometrics> (Version 2020-11-15)

- Makowski, D., Wiernik, B., Patil, I., Lüdecke, D., Ben-Shachar, M., Thériault, R., ... Rabe, M. (2024). correlation: Methods for correlation analysis [Software]. URL <https://CRAN.R-project.org/package=correlation> (R package version 0.8.5)
- Maxwell, S. E., Delaney, H. D. & Kelley, K. (2017). *Designing experiments and analyzing data: A model comparison perspective* (3. Aufl.). Mahwah, NJ: Lawrence Erlbaum.
- Mazerolle, M. J. (2020). AICcmodavg: Model selection and multimodel inference based on (Q)AIC(c) [Software]. URL <https://CRAN.R-project.org/package=AICcmodavg> (R package version 2.3-1)
- McElreath, R. (2020). *Statistical rethinking: A Bayesian course with examples in R and Stan* (2. Aufl.). Boca Raton, FL: Chapman & Hall/CRC. URL <https://xcelab.net/rm/>
- McFarlane, J. (2024). pandoc: A universal document converter [Software]. URL <https://pandoc.org/> (Version 3.3)
- Meier, L. (2022). *ANOVA and mixed models: A short introduction using R*. Boca Raton, FL: Chapman & Hall/CRC. URL <https://people.math.ethz.ch/~meier/teaching/anova/>
- Mersmann, O., Beleites, C., Hurling, R., Friedman, A. & Ulrich, J. (2023). microbenchmark: Accurate timing functions [Software]. URL <https://CRAN.R-project.org/package=microbenchmark> (R package version 1.4.10)
- Meyer, D. & Hornik, K. (2009). Generalized and customizable sets in R. *Journal of Statistical Software*, 31 (2), 1–27. URL <https://www.jstatsoft.org/v31/i02/>
- Meyer, D., Zeileis, A. & Hornik, K. (2020). vcd: Visualizing categorical data [Software]. URL <https://CRAN.R-project.org/package=vcd> (R package version 1.4-8)
- Meyer, F. & Perrier, V. (2024). esquisse: Explore and visualize your data interactively [Software]. URL <https://CRAN.R-project.org/package=esquisse> (R package version 2.0.0)
- Meyer, S. & Held, L. (2014). Power-law models for infectious disease spread. *The Annals of Applied Statistics*, 8 (3), 1612–1639.
- Microsoft. (2024). Microsoft Visual Studio Code [Software]. URL <https://code.visualstudio.com/docs/languages/r> (Version 1.90.2)
- Miller, A. J. (2002). *Subset selection in regression* (2. Aufl.). Boca Raton, FL: Chapman & Hall/CRC.
- Morgan-Wall, T. (2020). rayshader: Create and visualize hillshaded maps from elevation matrices [Software]. URL <https://CRAN.R-project.org/package=rayshader> (R package version 0.13.7)
- Morgan-Wall, T. (2024). rayrender: Build and raytrace 3D scenes [Software]. URL <https://CRAN.R-project.org/package=rayrender> (R package version 0.32.2)
- Muenchen, R. A. (2011). *R for SAS and SPSS users* (2. Aufl.). New York, NY: Springer. URL <http://r4stats.com/>
- Muenchen, R. A. (2018). *Why R is hard to learn*. URL <http://r4stats.com/articles/why-r-is-hard-to-learn/>
- Müller, K., Wickham, H., James, D. A. & Falcon, S. (2020). RSQLite: SQLite interface for R [Software]. URL <https://CRAN.R-project.org/package=RSQLite> (R package version 2.2.0)
- Murrell, P. (2018). *R graphics* (3. Aufl.). Boca Raton, FL: Chapman & Hall/CRC. URL <https://www.stat.auckland.ac.nz/~paul/RG3e/>
- Nash, J. C. (2014a). *Nonlinear parameter optimization using R tools*. Chichester, UK: Wiley.

- Nash, J. C. (2014b). On best practice optimization methods in R. *Journal of Statistical Software*, 60 (2), 1–14. URL <https://www.jstatsoft.org/v60/i02/>
- Nash, J. C. & Ravi, V. (2011). Unifying optimization algorithms to aid software system users: optimx for R. *Journal of Statistical Software*, 43 (9), 1–14. URL <https://www.jstatsoft.org/v43/i09/>
- Neth, H., Gaisbauer, F., Gradwohl, N. & Gaissmaier, W. (2019). riskyr: Rendering risk literacy more transparent [Software]. URL <https://CRAN.R-project.org/package=riskyr> (R package version 0.2.0)
- Nieuwenhuis, R., te Grotenhuis, M. & Pelzer, B. (2017). wec: Weighted effect coding for observational data with wec. *The R Journal*, 9, 477–485.
- Oksanen, J., Blanchet, F. G., Friendly, M., Kindt, R., Legendre, P., Minchin, P. R., ... Wagner, H. (2019). vegan: Community ecology package [Software]. URL <https://CRAN.R-project.org/package=vegan> (R package version 2.5-6)
- Ooi, H. (2020). glmnetUtils: Utilities for ‘glmnet’ [Software]. URL <https://CRAN.R-project.org/package=glmnetUtils> (R package version 1.1.5)
- Ooms, J. (2024). magick: Advanced graphics and image-processing in R [Software]. URL <https://CRAN.R-project.org/package=magick> (Version 2.8.3)
- OpenAnalytics BVBA. (2023). Architect [Software]. URL <https://www.getarchitect.io/downloads/> (Version 1.3)
- Park, J. H. (2020). CRAN task view: Bayesian inference. URL <https://CRAN.R-project.org/view=Bayesian> (Version 2020-12-07)
- Pebesma, E. & Bivand, R. S. (2023). Spatial data science: With applications in R. Boca Raton, FL: Chapman & Hall/CRC. URL <https://r-spatial.org/book/>
- Pedersen, T. L. (2020). patchwork: The composer of plots [Software]. URL <https://CRAN.R-project.org/package=patchwork> (R package version 1.1.1)
- Pedersen, T. L. & Robinson, D. (2020). gganimate: A grammar of animated graphics [Software]. URL <https://CRAN.R-project.org/package=gganimate> (R package version 1.0.5)
- Peters, A. & Hothorn, T. (2019). ipred: Improved predictors [Software]. URL <https://CRAN.R-project.org/package=ipred> (R package version 0.9-9)
- Pinheiro, J. C. & 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. & R Core Team. (2020). nlme: Linear and nonlinear mixed effects models [Software]. URL <https://CRAN.R-project.org/package=nlme> (R package version 3.1-151)
- Plate, T. & Heiberger, R. (2016). abind: Combine multi-dimensional arrays [Software]. URL <https://CRAN.R-project.org/package=abind> (R package version 1.4-5)
- Ploner, M. & Heinze, G. (2018). coxphf: Cox regression with Firth's penalized likelihood [Software]. URL <https://CRAN.R-project.org/package=coxphf> (R package version 1.13)
- Posit PBC. (2021). RStudio: Integrated development environment for R [Software]. Boston, MA. URL <https://posit.co/products/open-source/rstudio/> (Version 1.4.1100)
- Posit PBC. (2024). Positron: A next-generation data science IDE [Software]. Boston, MA. URL <https://github.com/posit-dev/positron/> (Version 2024.07.0-27)
- R Consortium. (2024). R Consortium [Software-Handbuch]. Vienna, Austria. URL <https://www.r-consortium.org/>
- R Core Team. (2020a). R: A language and environment for statistical computing [Software-Handbuch]. Vienna, Austria. URL <https://www.r-project.org/>

- R Core Team. (2020b). R: Data import/export [Software-Handbuch]. Vienna, Austria. URL <https://CRAN.R-project.org/doc/manuals/R-data.html>
- R Core Team. (2020c). R: Installation and administration [Software-Handbuch]. Vienna, Austria. URL <https://CRAN.R-project.org/doc/manuals/R-admin.html>
- R Core Team. (2020d). Writing R extensions [Software-Handbuch]. Vienna, Austria. URL <https://cran.r-project.org/doc/manuals/R-exts.html>
- R Foundation. (2024). R Foundation [Software-Handbuch]. Vienna, Austria. URL <https://www.r-project.org/foundation/>
- R Foundation for Statistical Computing. (2018). R: Regulatory compliance and validation issues: A guidance document for the use of R in regulated clinical trial environments [Software-Handbuch]. Vienna, Austria. URL <https://www.r-project.org/doc/R-FDA.pdf>
- R Special Interest Group on Databases, Wickham, H. & Müller, K. (2019). DBI: R database interface [Software]. URL <https://CRAN.R-project.org/package=DBI> (R package version 1.1.0)
- Rabe, M. M., Vasishth, S., Hohenstein, S., Kliegl, R. & Schad, D. J. (2023). hypr: Hypothesis matrix translation [Software]. URL <https://CRAN.R-project.org/package=hypr> (R package version 0.2.8)
- Revelle, W. (2020). psych: Procedures for psychological, psychometric, and personality research [Software]. URL <https://CRAN.R-project.org/package=psych> (R package version 2.0.12)
- Richardson, N., Cook, I., Crane, N., Dunnington, D., Franois, R., Keane, J., ... Wujciak-Jens, J. (2024). arrow: Integration to Apache Arrow [Software]. URL <https://CRAN.R-project.org/package=arrow> (R package version 16.1.0)
- Rigby, R. A., Stasinopoulos, M. D., Heller, G. Z. & de Bastiani, F. (2019). *Distributions for modeling location, scale, and shape: Using GAMLSS in R*. Boca Raton, FL: Chapman & Hall/CRC.
- Ripley, B. D., Murdoch, D. & Kalibera, T. (2024). *R for Windows FAQ*. URL <https://CRAN.R-project.org/bin/windows/base/rw-FAQ.html>
- Ritz, C. & Streibig, J. C. (2009). *Nonlinear regression with R*. New York, NY: Springer.
- Robin, X., Turck, N., Hainard, A., Tiberti, N., Lisacek, F., Sanchez, J.-C. & Müller, M. (2011). pROC: an open-source package for R and S+ to analyze and compare ROC curves. *BMC Bioinformatics*, 12, 77. URL <https://web.expasy.org/pROC/>
- Rodrigues, B. & Baumann, P. (2024). rix: Reproducible environments with Nix [Software]. URL <https://github.com/b-rodrigues/rix/> (R package version 0.9.1)
- Rosseel, Y. (2012). lavaan: An R package for structural equation modeling. *Journal of Statistical Software*, 48 (2), 1–36. URL <https://www.jstatsoft.org/v48/i02/>
- Rousseeuw, P. J., Croux, C., Todorov, V., Ruckstuhl, A., Salibian-Barrera, M., Verbeke, T. & Maechler, M. (2020). robustbase: Basic robust statistics [Software]. URL <https://CRAN.R-project.org/package=robustbase> (R package version 0.93-6)
- Sabanes Bove, D., Li, L., Dedic, J., Kelkhoff, D., Kunzmann, K., Lang, B. M., ... Sjoberg, D. D. (2024). mmrm: Mixed models for repeated measures [Software]. URL <https://CRAN.R-project.org/package=mmrm> (R package version 0.3.12)
- Sarkar, D. (2008). *Lattice: Multivariate data visualization with R*. New York, NY: Springer. URL <https://lmdvr.r-forge.r-project.org>
- Schaubberger, P. & Walker, A. (2020). openxlsx: Read, write and edit XLSX files [Software]. URL <https://CRAN.R-project.org/package=openxlsx> (R package version 4.2.3)

- Schwarzer, G., Carpenter, J. R. & Rücker, G. (2015). *Meta-analysis with R*. New York, NY: Springer. URL <https://www.uniklinik-freiburg.de/imbi/stud-le/meta-analysis-with-r.html>
- Schwendinger, F. & Borchers, H. W. (2019). CRAN task view: Optimization and mathematical programming. URL <https://CRAN.R-project.org/view=Optimization> (Version 2024-07-05)
- Shumway, R. H. & Stoffer, D. S. (2016). *Time series analysis and its applications* (4. Aufl.). New York, NY: Springer. URL <https://github.com/nickpoison/tsa4>
- Shumway, R. H. & Stoffer, D. S. (2019). *Time series: A data analysis approach using R*. Boca Raton, FL: Chapman & Hall/CRC. URL <https://github.com/nickpoison/tsda/>
- Sievert, C. (2020). *Interactive web-based data visualization with R, plotly, and shiny*. Boca Raton, FL: Chapman & Hall/CRC. URL <https://plotly-r.com/>
- Sievert, C., Parmer, C., Hocking, T., Chamberlain, S., Ram, K., Corvellec, M. & Despouy, P. (2020). plotly: Create interactive web graphics via plotly's JavaScript graphing library [Software]. URL <https://CRAN.R-project.org/package=plotly> (R package version 4.9.2)
- Signorell, A. (2020). DescTools: Tools for descriptive statistics. URL <https://CRAN.R-project.org/package=DescTools> (R package version 0.99.39)
- Simpson, G. L. (2019). permute: Functions for generating restricted permutations of data [Software]. URL <https://CRAN.R-project.org/package=permute> (R package version 0.9-5)
- Slowikowski, K. (2024). ggrepel: Automatically position non-overlapping text labels with 'ggplot2' [Software]. URL <https://CRAN.R-project.org/package=ggrepel> (R package version 0.9.5)
- Smyth, G. (2020). statmod: Statistical modeling [Software]. URL <https://CRAN.R-project.org/package=statmod> (R package version 1.4.34)
- Soetaert, K. (2019). rootSolve: Nonlinear root finding, equilibrium and steady-state analysis of ordinary differential equations [Software]. URL <https://CRAN.R-project.org/package=rootSolve> (R package version 1.8.2)
- Soetaert, K. & Petzoldt, T. (2020). CRAN task view: Differential equations. URL <https://CRAN.R-project.org/view=DifferentialEquations> (Version 2020-01-12)
- Strang, G. (2003). *Lineare Algebra*. Berlin: Springer.
- Tang, Y. & Balamuta, J. J. (2023). CRAN task view: Databases with R. URL <https://CRAN.R-project.org/view=DifferentialEquations> (Version 2023-02-23)
- Templ, M. (2020). CRAN task view: Official statistics & survey methodology. URL <https://CRAN.R-project.org/view=OfficialStatistics> (Version 2020-12-20)
- Therneau, T. (2020). survival: Survival analysis [Software]. URL <https://CRAN.R-project.org/package=survival> (R package version 3.1-11)
- Tierney, N., Cook, D., McBain, M. & Fay, C. (2020). naniar: Data structures, summaries, and visualisations for missing data [Software]. URL <https://CRAN.R-project.org/package=naniar> (R package version 0.6.0)
- Tingley, D., Yamamoto, T., Keele, L., Imai, K., Trinh, M. & Wong, W. (2019). mediation: R package for causal mediation analysis [Software]. URL <https://CRAN.R-project.org/package=mediation> (R package version 4.5.0)
- Unwin, A. (2015). *Graphical data analysis with R*. Boca Raton, FL: Chapman & Hall/CRC. URL <http://www.gradaanwr.net/>

- Urbanek, S. (2019). rJava: Low-level R to Java interface [Software]. URL <https://CRAN.R-project.org/package=rJava> (R package version 0.9-11)
- Urbanek, S. & Horner, J. (2020). Cairo: R graphics device using cairo graphics library for creating high-quality bitmap (PNG, JPEG, TIFF), vector (PDF, SVG, PostScript) and display (X11 and Win32) output [Software]. URL <https://CRAN.R-project.org/package=Cairo> (R package version 1.5-11)
- Ushey, K. (2020). renv: Project environments [Software]. URL <https://CRAN.R-project.org/package=renv> (R package version 0.12.3)
- van Buuren, S. (2018). *Flexible imputation of missing data* (2. Aufl.). Boca Raton, FL: Chapman & Hall/CRC. URL <https://stefvanbuuren.name/fimd/>
- van Buuren, S. & Groothuis-Oudshoorn, K. (2011). MICE: Multivariate imputation by chained equations in R. *Journal of Statistical Software*, 45 (3), 1–67. URL <https://www.jstatsoft.org/v45/i03/>
- van der Loo, M. P. J. (2014). The stringdist package for approximate string matching. *The R Journal*, 6, 111–122. URL <https://CRAN.R-project.org/package=stringdist>
- Venables, W. N. (2018). codingMatrices: Alternative factor coding matrices for linear model formulae [Software]. URL <https://CRAN.R-project.org/package=codingMatrices> (R package version 0.3.2)
- Venables, W. N. & Ripley, B. D. (2002). *Modern applied statistics with S* (4. Aufl.). New York, NY: Springer. URL <https://www.stats.ox.ac.uk/pub/MASS4/>
- Venables, W. N., Smith, D. M., Gentleman, R., Ihaka, R., Maechler, M. & R Core Team. (2020). An introduction to R [Software-Handbuch]. Vienna, Austria. URL <https://CRAN.R-project.org/doc/manuals/R-intro.html> (Version 2020-10-10)
- Wahlbrink, S. (2024). Eclipse plug-in for R: StatET [Software]. URL <https://projects.eclipse.org/projects/science.statet> (Version 4.10.0)
- Wand, M. (2019). KernSmooth: Functions for kernel smoothing for Wand & Jones (1995) [Software]. URL <https://CRAN.R-project.org/package=KernSmooth> (R package version 2.23-16)
- Wellek, S. (2010). *Statistical hypotheses of equivalence and noninferiority* (2. Aufl.). Boca Raton, FL: Chapman & Hall/CRC.
- West, B. T., Welch, K. B. & Gałecki, A. T. (2022). *Linear mixed models: A practical guide using statistical software* (3. Aufl.). Boca Raton, FL: Chapman & Hall/CRC. URL <https://public.websites.umich.edu/~bwest/almmussp.html>
- Wickens, T. (2015). *The geometry of multivariate statistics*. New York, NY: Psychology Press.
- Wickham, H. (2019a). Advanced R (2. Aufl.). Boca Raton, FL: Chapman & Hall/CRC. URL <https://adv-r.hadley.nz/>
- Wickham, H. (2019b). stringr: Simple, consistent wrappers for common string operations [Software]. URL <https://CRAN.R-project.org/package=stringr> (R package version 1.4.0)
- Wickham, H. (2020a).forcats: Tools for working with categorical variables (factors) [Software]. URL <https://CRAN.R-project.org/package=forcats> (R package version 0.5.0)
- Wickham, H. (2020b). *Mastering Shiny*. Sebastopol, CA: O'Reilly. URL <https://mastering-shiny.org/>
- Wickham, H. (2023). *R packages* (2. Aufl.). Sebastopol, CA: O'Reilly. URL <https://r-pkgs.org/>
- Wickham, H. & Bryan, J. (2019). readxl: Read Excel files [Software]. URL <https://CRAN.R-project.org/package=readxl> (R package version 1.3.1)

- Wickham, H., Francois, R., Henry, L. & Müller, K. (2020). dplyr: A grammar of data manipulation [Software]. URL <https://CRAN.R-project.org/package=dplyr> (R package version 1.0.2)
- Wickham, H. & Grolemund, G. (2023). *R for data science* (2. Aufl.). Sebastopol, CA: O'Reilly. URL <https://r4ds.hadley.nz/>
- Wickham, H. & Henry, L. (2020). tidyverse: Easily tidy data with spread and gather functions [Software]. URL <https://CRAN.R-project.org/package=tidyr> (R package version 1.1.2)
- Wickham, H., Hester, J. & Francois, R. (2020). readr: Read tabular data [Software]. URL <https://CRAN.R-project.org/package=readr> (R package version 1.4.0)
- Wickham, H. & Miller, E. (2020). haven: Import SPSS, Stata and SAS files [Software]. URL <https://CRAN.R-project.org/package=haven> (R package version 2.3.1)
- Wickham, H. & Sievert, C. (2024). *ggplot2: Elegant graphics for data analysis* (3. Aufl.). New York, NY: Springer. URL <https://ggplot2-book.org/>
- Wilke, C. O. (2019). *Fundamentals of data visualization*. Sebastopol, CA: O'Reilly. URL <https://clauswilke.com/dataviz/>
- Wilkinson, G. N. & Rogers, C. E. (1973). Symbolic description of factorial models for analysis of variance. *Applied Statistics*, 22, 392–399.
- Wilkinson, L. (2005). *The grammar of graphics* (2. Aufl.). New York, NY: Springer.
- Wirtz, M. & Caspar, F. (2002). *Beurteilerübereinstimmung und Beurteilerreliabilität*. Göttingen: Hogrefe.
- Wood, S. N. (2017). *Generalized additive models: An introduction with R* (2. Aufl.). Boca Raton, FL: Chapman & Hall/CRC.
- Xie, Y. (2015). *Dynamic documents with R and knitr* (2. Aufl.). Boca Raton, FL: Chapman & Hall/CRC.
- Xie, Y. (2019). TinyTeX: A lightweight, cross-platform, and easy-to-maintain LaTeX distribution based on TeX Live. *TUGboat* (1), 30–32. URL <http://tug.org/TUGboat/Contents/contents40-1.html>
- Xie, Y. (2020a). knitr: A general-purpose package for dynamic report generation in R [Software]. URL <https://CRAN.R-project.org/package=knitr> (R package version 1.30)
- Xie, Y. (2020b). tinytex: Helper functions to install and maintain ‘TeX Live’, and compile ‘LaTeX’ documents [Software]. URL <https://CRAN.R-project.org/package=tinytex> (R package version 0.20)
- Xie, Y., Allaire, J. J. & Grolemund, G. (2018). *R markdown: The definitive guide*. Boca Raton, FL: Chapman & Hall/CRC. URL <https://bookdown.org/yihui/rmarkdown/>
- Xie, Y., Dervieux, C. & Riederer, E. (2020). *R markdown cookbook*. Boca Raton, FL: Chapman & Hall/CRC. URL <https://bookdown.org/yihui/rmarkdown-cookbook/>
- Yan, J. & Fine, J. P. (2004). Estimating equations for association structures. *Statistics in Medicine*, 23 (6), 859–874.
- Yee, T. W. (2010). The VGAM package for categorical data analysis. *Journal of Statistical Software*, 32 (10), 1–34. URL <https://www.jstatsoft.org/v32/i10/>
- Zeileis, A. (2004). Econometric computing with HC and HAC covariance matrix estimators. *Journal of Statistical Software*, 11 (10), 1–17. URL <https://www.jstatsoft.org/v11/i10/>
- Zeileis, A., Fisher, J., Hornik, K., Ihaka, R., McWhite, C., Murrell, P., ... Wilke, C. (2019). colorspace: A toolbox for manipulating and assessing colors and palettes. *arXiv.org E-Print Archive*, 1903.06490. URL <https://arxiv.org/abs/1903.06490>

Literaturverzeichnis

- Zeileis, A. & Hothorn, T. (2002). Diagnostic checking in regression relationships. *R News*, 2 (3), 7–10. URL <https://CRAN.R-project.org/doc/Rnews/>
- Zeileis, A., Kleiber, C. & Jackman, S. (2008). Regression models for count data in R. *Journal of Statistical Software*, 27 (8), 1–25. URL <https://www.jstatsoft.org/v27/i08/>
- Zelterman, D. (2015). *Applied multivariate statistics with R*. New York, NY: Springer.
- Zhang, S. & Cook, D. (2024). *CRAN task view: Dynamic visualizations and interactive graphics*. URL <https://CRAN.R-project.org/view=DynamicVisualizations> (Version 2024-09-25)

Index

- (2×2)-Kontingenztafel, 425
Ableitung, 665
Abstand, 489
AIC, 235, 347
AICc, 235
allgemeines lineares Modell, 528
 Codiermatrix, 531
 Codierschema, 531
 Designmatrix, 222, 528
 Hat-Matrix, 498, 541
 Kontrastmatrix, 532
 lineare Hypothesen, 542
 Modellvergleiche, 542, 548
 Parameterschätzungen, 540
 parametrische Funktionen, 542
 Teststatistiken, 549
 α -Adjustierung, 280, 283, 284, 307, 308, 333, 451
 α -Fehler, 222
Anderson-Darling-Test, 265
Anführungszeichen, 26, 100, 126
anonyme Funktion, *siehe* Funktion
anpassen, *siehe* Optionen
Ansari-Bradley-Test, 443
Anzahl, *siehe* Vektor
Arbeitsverzeichnis, *siehe* Verzeichnis
area under the curve (AUC), 427
Arithmetik, 13
array, 95
Assoziativität, 15
Attribut, 23
Ausführungsreihenfolge, *siehe* Assoziativität
äußeres Produkt, 82
Auswahl, *siehe* Daten
AV, 222

Balkendiagramm, *siehe* Grafik
Batch-Modus, 206
Bayes Datenanalyse, 221
beenden, 11

Befehlshistorie, 12
Befehlsskript, *siehe* Skript
Beobachterübereinstimmung, *siehe* Inter-Rater-Übereinstimmung
 β -Fehler, 222
Bhapkar-Test, 461
BIC, 235
Binomialtest, 410, 447
 power, 334
Binomialverteilung, *siehe* Verteilung
Bland-Altman Diagramm, 440
Bootstrap-Verfahren, *siehe* Resampling-Verfahren, 474
Boschloo-Test, 422
Bowker-Test, 456
Box-Cox-Transformation, *siehe* Regression
Boxplot, *siehe* Grafik
Bradley-Terry-Modelle, 362
Brier-Score, 563
Brown-Forsythe-Test, 268

CART-Modelle, 524
Chancenverhältnis, 426
 χ^2 -Test
 Auftretenswahrscheinlichkeiten, 421
 feste Verteilung, 415
 Gleichheit von Verteilungen, 420
 Normalverteilung, 417
 Unabhängigkeit, 419
 Verteilungsklasse, 417
 χ^2 -Verteilung, *siehe* Verteilung
Cholesky-Zerlegung, *siehe* Matrix
Clusteranalyse, 524
Cochran-Mantel-Hänszel-Test, 423
Cochran-Q-Test, 455
Cohens d , 270, 272, 273
Cohens κ , *siehe* Inter-Rater-Übereinstimmung
connection, 141
Cox proportional hazards Modell, *siehe* Survival-Analse
Cramérs V , 431

INDEX

- CRAN, 18
- d, siehe* Cohens *d*
- Dateien
- auflisten, 152
 - manipulieren, 154
 - Pfad, Name, Endung, 154
- Daten
- Ausreißer, 247, 604, 616
 - auswählen, 52
 - diskretisieren, 59, 68
 - doppelte Werte, 41, 161
 - eingeben, 139
 - ersetzen, 56
 - Extremwerte, 247
 - fehlende Werte, 15, 117, 123, 161
 - ausschließen, 120
 - codieren, 117
 - ersetzen, 123
 - fallweiser Ausschluss, 121
 - identifizieren, 118
 - Indexvektor, 40
 - multiple Imputation, 123
 - paarweiser Ausschluss, 122
 - recodieren, 119
 - sortieren, 122
 - zählen, 119
- Fishers Z-Transformation, 230
- getrennt nach Gruppen auswerten, 80
- Import / Export, 139
- andere Programme, 143
 - Datenbank, 147
 - Konsole, 150
 - online, 141
 - R-Format, 142
 - Textformat, 139, 141, 143
 - Zwischenablage, 140, 142
- interpolieren, 657
- Kennwerte, 70
- Median-Split, 69
- Qualität, 117, 161
- Rangtransformation, 55
- recodieren, 56, 63
- skalieren, 55
- speichern, 139
- Viewer, 105
- wiederholen, 48
- winsorisieren, 74
- z-transformieren, 54*
- zentrieren, 54
- Zufallsauswahl, 52
- Datenbank, *siehe* Daten
- Datensatz, 103
- Aufbau, 104, 170, 172
 - Dimensionierung, 105
 - Dokumentation, 21
 - doppelte Werte, 161
 - Elemente auswählen und verändern, 106
 - erstellen, 104
 - fehlende Werte, 161
 - Funktionen anwenden, 176, 179, 181
 - indizieren, 106
 - Long-Format, 172, 172, 287, 309, 317
 - Pakete, 21, 104
 - sortieren, 164
 - Suchpfad, 108
 - teilen, 165
 - Teilmengen auswählen, 158
 - umformen, 170, 172
 - Umwandlung in Datensatz, 106
 - Variablen hinzufügen, 163
 - Variablen löschen, 163
 - Variablennamen, 108
 - verbinden, 166
 - Wide-Format, 172, 172, 315
- Datentyp, 23, 26
- mischen, 35, 83, 97, 103
 - umwandeln, 27
- Delta-Methode, 666
- deskriptive Kennwerte, *siehe* Daten
- Determinante, *siehe* Matrix
- Determinationskoeffizient, *siehe* Regression
- Devianz, *siehe* Regression
- Diagramm, *siehe* Grafik
- Differenzen, 70
- differenzieren, 665
- Dimensionierung, *siehe* Datensatz, Matrix
- Diskriminanzanalyse, 523
- Distanzmaß, *siehe* Abstand
- Diversitätsindex, 77
- Dokumentation, *siehe* Literatur
- download, 4, 5
- Dummy-Codierung, *siehe* Varianzanalyse

INDEX

- eager evaluation, 680
Editor, 5, 206
Effektcodierung, *siehe* Varianzanalyse
Effektstärke, *siehe* t-Test, Varianzanalyse
Effizienz, 691
Eigenwerte, -vektoren, *siehe* Matrix
Eingabe, *siehe* Daten
Einstellungen, 11
ENTWEDER-ODER, 28
Environment, *siehe* Umgebung
Ereigniszeiten, *siehe* Survival-Analise
Escape-Sequenz, 26, 127
 $\hat{\eta}^2$ (Effektstärke), 278, 290, 296, 312, 318, 331
Excel, 143
Exponentialfunktion, 14
Extremwerte, 71

 f (Effektstärke), 278
 F -Maß, *siehe* (2×2)-Kontingenztafel
 F -Verteilung, *siehe* Verteilung
Faktor, 59
Faktorenanalyse, 506
Fakultät, 71
Fallunterscheidung, 672
Fallzahlberechnung, 334
FALSCH, 28
Faltung, 604
FAQ, 5
Farben, *siehe* Grafik
fehlende Werte, *siehe* Daten
Fehlerbalken, *siehe* Grafik
Fehlermeldung, 17
Fisher-Pitman Test, 478
Fishers exakter Test
 Gleichheit von Verteilungen, 424
 Unabhängigkeit, 422
Fleiss' κ , *siehe* Inter-Rater-Übereinstimmung
Fligner-Killeen-Test, 268
format string, 126, 134
Fourier-Transformation, 604
Friedman-Test, 453
Funktion
 anonyme, 684
 Argumente, 16, 679
 aufrufen, 16, 684
 debuggen, 687
erstellen, 679
generische, 685
Geschwindigkeit, 691
getrennt nach Gruppen anwenden, 80
Kopf, 679
Methode, 685
profilen, 691
Quelltext, 687
Rumpf, 680
Rückgabewert, 684
unbenannte, 684
Funktionsgraph, *siehe* Grafik

 Γ -Funktion, 71
GEE (generalized estimating equations), 261
gemischte Modelle, 261
Genauigkeit, *siehe* Zahlen
generische Funktion, *siehe* Funktion
geordnete Paare, 82
Geschwindigkeit, *siehe* Effizienz
Gleichverteilung, *siehe* Verteilung
Globbing-Muster, *siehe* Zeichenketten
GLS Modelle, 261
Glättung, 661
Goodman und Kruskals γ , 431
Grafik
 3D, 618
 3D Gitter-Diagramm, 620
 3D Streudiagramm, 620
 Achsen, 578, 598
 Ausgabe, 567
 Balkendiagramm, 579
 bedingte Diagramme, 643
 Beschriftung, 596–598
 Bildbearbeitung, 604
 Boxplot, 608
 clipping, 574, 587
 Datenpunkte identifizieren, 571
 Datenpunktssymbole, 575
 device, 567
 unterteilen, 625
 verwalten, 567–569
 dotchart, 584
 Einstellungen, 573
 Elemente hinzufügen, 585
 Farben, 576
 Fehlerbalken, 599

INDEX

- Fenster öffnen, 567
Formate, 569
formatieren, 573
Funktionsgraph, 595
gemeinsame Verteilung, 614
Gerade, 589
`ggplot2`, 632
Gitter, 589
Glätter, 660
heatmap, 624
High-Level-Funktion, 567, 567, 585, 629
Histogramm, 604
Höhenlinien, 618, 619
interaktive, 620
interpolieren, 658
konvexe Hülle, 616
Koordinaten identifizieren, 586
Koordinatensysteme, 587
Kreisdiagramm, 613
kumulierte Häufigkeiten, 613
Legende, 596
Linien, 589
Liniendiagramm, 570, 572
Liniensegmente, 590
Linientypen, 575
Low-Level-Funktion, 567, 585
mathematische Formeln, 597
mehrere Grafiken in einem device, 625
multivariate Daten, 618
Optionen, 573
Pfeile, 591
Polygone, 593
Punktdiagramm, 584
Punkte, 588
Quantil-Quantil-Diagramm, 610
Rahmen, 592
Rastergrafik, 602
Rechtecke, 592
Regionen, 567
Ränder, 567
Säulendiagramm, 579
Scree-Plot, 505
Seitenverhältnis, 590
speichern, 569
Stamm-Blatt-Diagramm, 606
Streudiagramm, 570, 572
Streudiagramm-Matrix, 622
Streuungsellipse, 614
stripchart, 609
Symbole, 597
Säulendiagramm, 580
Text, 596, 597
Titel, 596
Transparenz, 578
grafische Benutzeroberfläche, 5
Groß- und Kleinschreibung, 24
GUI, *siehe* grafische Benutzeroberfläche
 H_0, H_1 , 221
Harrels C , 394
Häufigkeiten
 absolute, 110
 bedingte relative, 113
 Iterationen, 112
 Kreuztabelle, 112
 kumulierte relative, 111, 116
 Randkennwerte, 115
 relative, 111
Hauptkomponentenanalyse, 499
herunterladen, *siehe* download
Hesse-Matrix, 667
Hilfe
 eingebautes Hilfesystem, 16
 Internet, 5
 Literatur, 6
 Zusatzpakete, 21
Histogramm, *siehe* Grafik
Hodges-Lehmann-Schätzer, 74, 449, 450
Homepage, 4, 5
Hotelling-Lawley-Spur, 521, 549
Hotellings T^2
 eine Stichprobe, 516
 zwei abhängige Stichproben, 519
 zwei unabhängige Stichproben, 518
Huber- M -Schätzer, 75
ICC, *siehe* Inter-Rater-Übereinstimmung
Imputation, *siehe* Daten
Indexvektor, *siehe* Vektor
indizieren, *siehe* Datensatz, Matrix, Vektor
Installation, 4
Integration, 664
Inter-Rater-Übereinstimmung, 433
 r_{WG} -Koeffizienten, 433

INDEX

- Cohens κ , 434, 436
Fleiss' κ , 435
Intra-Klassen-Korrelation, 439
Kendalls W , 437
prozentuale Übereinstimmung, 433
Interquartilabstand, 79
Intra-Klassen-Korrelation, *siehe* Inter-Rater-Übereinstimmung
Iterationen, *siehe* Häufigkeiten
Iterationslängentest, *siehe* Runs-Test

Jonckheere-Terpstra-Test, 453

Karten, *siehe* räumliche Statistik
Kendalls τ , 429
Kendalls W , *siehe* Inter-Rater-Übereinstimmung
Kern, *siehe* Matrix
Kerndichteschätzer, 661
Klammern, 15
Klasse, 22
Klassifikationsverfahren, 351, 524
Kolmogorov-Smirnov-Test
 Anpassungstest, 413
 Gleichheit von Verteilungen, 445
 Lilliefors-Schranken, 265
 Normalverteiltheit, 413
Kombinatorik, 43
Kommentar, 8, 17, 207, 673
komplexe Zahlen, *siehe* Zahlen
Kondition κ , *siehe* Matrix
Konfidenzintervall, 222
Konfusionsmatrix, 425
Konkordanz, 394
Konsole, 7
Konstanten, 13
Kontingenzkoeffizient, 431
Kontingenztafel, *siehe* Häufigkeiten
Kontraste, *siehe* Varianzanalyse
Kontrollstrukturen, 672
konvexe Hülle, *siehe* Grafik
Korrelation, 78
 kanonische, 78
 Kendalls τ , 429
 Korrelationsmatrix, 93
 multiple Korrelation, 235
 Partialkorrelation, 78, 262
 polychorische, 78
polyseriale, 78
Rangkorrelation, 429
Semipartialkorrelation, 79, 263
Spearmans ρ , 429
Test, 229
Kovarianz, 78
 Kovarianzmatrix, 93
 Rangkovarianz, 429
robuste Schätzer, 80
unkorrigierte, 78, 94
Kovarianzanalyse, 326
 Effektstärke, 331
 Einzelvergleiche (Kontraste), 332, 333
Kreuztabelle, *siehe* Häufigkeiten
Kreuzvalidierung, 560
 k -fache, 560
 Kreuzvalidierungsfehler, 560
 leave-one-out, 562
 verallgemeinerte, 563
Kruskal-Wallis- H -Test, 452
kumulierte relative Häufigkeiten, *siehe* Häufigkeiten
kumulierte Summe, *siehe* Summe
kumulierte Produkt, *siehe* Produkt
Kurtosis, *siehe* Wölbung

Länge
 Norm, 488
 Vektor, 23
 Zeichenkette, 124
lazy evaluation, 680
leere Menge, 15
Levene-Test, 267
Lilliefors-Test, *siehe* Kolmogorov-Smirnov-Test
lineare Algebra, *siehe* Matrix
lineare Gleichungssysteme lösen, 488
lineare Strukturgleichungsmodelle, 506
Liste, 97
Literatur, 5
logische Operatoren, *siehe* Vergleiche
logische Vergleiche, *siehe* Vergleiche
logische Werte, *siehe* Wahrheitswerte
logistische Regression, *siehe* Regression
Logit-Funktion, 343
Long-Format, *siehe* Datensatz

INDEX

- Mahalanobisdistanz, 491
Mahalanobistransformation, 490
Mailing-Liste, 5
Mann-Whitney-*U*-Test, 451
Mantel-Hänszel-Test, *siehe* Survival-Analyse
markdown, *siehe* R-Dokumente
Matrix
 Addition, 485
 Algebra, 485
 Cholesky-Zerlegung, 495
 Datentypen, 83
 Determinante, 492
 Diagonalelemente, 89
 diagonalisieren, 494
 Diagonalmatrix, 485
 Dimensionierung, 84
 Dreiecksmatrix, 85
 Eigenwerte, -vektoren, 493, 499
 Einheitsmatrix, 485
 Elemente auswählen und verändern, 86,
 88
 erstellen, 83, 89
 Funktionen anwenden, 92
 Hadamard-Produkt, 485
 indizieren, 86, 88
 Inverse, 488
 Kern, 488
 Kondition κ , 255, 493
 Kronecker-Produkt, 485
 mit Kennwert verrechnen, 92
 Multiplikation, 485
 Norm, 489
 orthogonale Projektion, 496
 Pseudoinverse, 237, 488, 498
 QR-Zerlegung, 492, 495
 Randkennwerte, 91
 Rang, 492
 Singulärwertzerlegung, 495, 504
 Skalarprodukt, 485
 sortieren, 90
 Spaltennorm, 488
 Spektralzerlegung, 494
 Spur, 492
 SSP-Matrix, 486
 transponieren, 484
 umwandeln, 85
 Umwandlung in Matrix, 84
verbinden, 89
Wurzel, 494
Zeilen- und Spaltenindex, 85
zentrieren, 92, 486
Matthews-Korrelationskoeffizient, 431
Mauchly-Test, 292, 294
Maximum, 71
McNemar-Test, 458
Median, 73
 Pseudo-Median, 74, 449
Mediationsanalyse, *siehe* Regression
Mengen, 41, 42
Methode, *siehe* Funktion
Minimum, 71
Mittelwert, 72
 geometrischer, 73
 gestutzter, 74
 harmonischer, 73
 winsorierter, 74
Mittelwertstabelle, 81
Modalwert, 73
Modellformel, 222
Modus, *siehe* Datentyp
Mood-Test, 442
Multicore, 693
multidimensionale Skalierung, 512
Multinomiale Regression, *siehe* Regression
multiple Imputation, *siehe* Daten
multivariate Verfahren, 484
multivariate *z*-Transformation, 490
nonparametrische Methoden, 409
Norm, 488
Normalengleichungen, 496
Normalverteilung, *siehe* Verteilung
Notation, 3
Objekt, 22
 Größe, 691
 Gültigkeit, 680
 label, 23
 Lebensdauer, 680
 lokales, 680
 löschen, 26
 Maskierung, 24
 Namenskonflikte, 24
 Objektnamen, 23, 25

INDEX

- verstecktes, 25
Objektorientierung, 685
odds ratio, 346, 426
ODER, 28, 37, 38
 $\hat{\omega}^2$ (Effektstärke), 278
OpenOffice, 143
Operatoren, 13
Optimierung, 668, *siehe* Effizienz
Optionen, 11
Ordinale Regression, *siehe* Regression
Ordner, *siehe* Verzeichnis

Page-Trend-Test, 455
Pakete, 18
Parallelisieren, 693
Partialkorrelation, *siehe* Korrelation
Permutation, *siehe* Kombinatorik
Permutationstest, *siehe* Resampling-Verfahren
 ϕ -Koeffizient, 431
Pillai-Bartlett-Spur, 521, 549
Poisson-Regression, *siehe* Regression
positiver Vorhersagewert, *siehe* (2×2)-Kontingenztafel
power, 222, 334
Probit-Regression, *siehe* Regression
Produkt, 71
programmieren, 672
Projektion, *siehe* Matrix
Prompt, 7
Protokoll, 124
Prozentrang, 75, 116
Prävalenz, *siehe* (2×2)-Kontingenztafel
Präzision, *siehe* (2×2)-Kontingenztafel
Pseudo-Median, 74, 449
Pseudoinverse, *siehe* Matrix

 Q -Koeffizient, *siehe* Yules Q -Koeffizient
 Q -Test, *siehe* Cochran- Q -Test
 QR -Zerlegung, *siehe* Matrix
Quadratsummen-Typen, *siehe* Varianzanalyse
Quantil, 75
Quartil, 75
Quarto, *siehe* R-Dokumente

R-Dokumente, 207, 214
R-Markdown, *siehe* R-Dokumente
 r_{WG} -Koeffizienten, 433
Rang, *siehe* Matrix

range, 72
Rangkorrelation, *siehe* Korrelation
Rangplatz, *siehe* Daten
Rate der korrekten Klassifikation, *siehe* (2×2)-Kontingenztafel
recall, *siehe* (2×2)-Kontingenztafel
recodieren, *siehe* Daten
record linkage, 216
recycling, *siehe* Vektor
Regression
bedingte logistische, 344
Beta-Regression, 262
Box-Cox-Transformation, 253
Determinationskoeffizient R^2 , 234
Devianz, 347
Einfluss, 249
elastic net, 258
grafische Darstellung, 232, 237
Hat-Matrix, 237, 248
Hauck-Donner-Phänomen, 353
Hebelwert, 248
hierarchische, 239
Homoskedastizität, 252, 540
Hurdle-Regression, 374
Interaktion, 242
Kalibrierung, 350
Konfidenzintervall, 235, 245
Kreuzvalidierung, *siehe* Kreuzvalidierung
LASSO, 258
lineare, 230, 236
logistische, 344
 Konvergenz, 355
Mediation, 233
Modell verändern, 238
Modellvergleich, 239
Moderation, 242
Multikollinearität, 254
multinomiale, 362
multivariate, 514, 528, 550
negative Binomial-Regression, 370
nichtlineare, 261
ordinale, 356
penalisierte, 257
Poisson-Regression, 367
polynomiale, 261
PRESS-Residuen, 562
Probit-Regression, 354

INDEX

- psueduo- R^2 , 348
Quantilregression, 262
Regressionsanalyse, 233, 550
Regressionsdiagnostik, 247
Residuen, 232, 347
Ridge-Regression, 257
robuste, 256
Sobel-Test, 233
standardisierte, 231
Stepwise-Verfahren, 239
Toleranzintervall, 245
Varianzinflationsfaktor, 254
verallgemeinerte additive Modelle, 261
Verteilungsregression, 262
Voraussetzungen, 247
zero-inflated Modelle, 372
zero-truncated Modelle, 374
regulärer Ausdruck, *siehe* Zeichenketten
Relevanz, *siehe* (2×2)-Kontingenztafel
Reproduzierbarkeit, 217
Resampling-Verfahren, 462
 Bootstraping
 Konfidenzintervalle, 468, 475
 Nonparametrisch, 462
 Parametrisch, 474
 stratifiziert, 468
 Bootstrapping
 case resampling, 469
 model-based resampling, 471
 wild resampling, 473
 Konfidenzintervalle, 466
 Permutationstests, 477
robuste Verfahren
 Ausreißer identifizieren, 248
 Diskriminanzanalyse, 524
 Hauptkomponentenanalyse, 500
 Kovarianzschätzer, 80
 Regression, 256
 Streuungsmaße, 79
 zentrale Tendenz, 74
ROC-Kurve, 427
Roys Maximalwurzel, 521, 549
RStudio, 5, 7
Runs-Test, 411
räumliche Statistik, 6
Rückgabewert, *siehe* Funktion
S, 1
S3-, S4-Paradigma, 685
SAS, 147
Säulendiagramm, *siehe* Grafik
Schiefe, 77
Schleifen, 675
Schlüsselwörter, 24
Scoping-Regeln, *siehe* Objekte
Scree-Plot, *siehe* Grafik
Semipartialkorrelation, *siehe* Korrelation
Sensitivität, *siehe* (2×2)-Kontingenztafel
Shannon-Index, 77
Shapiro-Wilk-Test, 265
Sign-Test, *siehe* Vorzeichen-Test
Signalverarbeitung, 604
Simulation, 104, 676
Singulärwertzerlegung, *siehe* Matrix
Skalarprodukt, *siehe* Matrix
Skript, 206
Sobel-Test, *siehe* Regression
Somers' d , 394, 431
sortieren, *siehe* Datensatz, Matrix, Vektor
Spannweite, *siehe* range
Spearmans ρ , 429
speichern, *siehe* workspace
Spektralzerlegung, *siehe* Matrix
Spezifität, *siehe* (2×2)-Kontingenztafel
Splines, 658
splines, 224, 261
SPSS, 145, 356
starten, 7
Stata, 147
Stichprobengröße, 334
Streudiagramm, *siehe* Grafik
Streuung, 76
Streuungsellipse, *siehe* Grafik
Stuart-Maxwell-Test, 460
Stufenfunktion, 116
Suchpfad, 12, *siehe* Datensatz
Summe, 70
Summenscore, 56
support vector machines, 524
Survival-Analyse, 381
 Cox proportional hazards Modell, 392
 Kaplan-Meier, 388
 Log-Rank-Test, 390
 Mantel-Hänszel-Test, 390

INDEX

- parametrische Modelle, 403
wiederkehrende Ereignisse, 386
zeitabhängige Prädiktoren, 387
Zählprozess-Darstellung, 385
Sweave, *siehe* R-Dokumente
- t*-Test
Effektstärke, 270, 272, 273
eine Stichprobe, 269
power, 336, 337
Stichprobengröße, 337
zwei abhängige Stichproben, 273
zwei unabhängige Stichproben, 270
- t*-Verteilung, *siehe* Verteilung
Tabellenkalkulation, 143
Task Views, 18
Tastaturkürzel, 11, 206
Teststärke, *siehe* power
Text, *siehe* Grafik, Zeichenkette
Tjur Diskriminationsindex, 349
transponieren, *siehe* Matrix
Treatment-Kontraste, *siehe* Varianzanalyse
trigonometrische Funktionen, 14
Tschuprows *T*, 431
Tukey Mean Difference Plot, 440
- Uhrzeit, 135
Umgebung, 12, 680
unbenannte Funktion, *siehe* Funktion
UND, 28, 37, 38
UV, 222
- van der Waerden-Test, 478
- Variable
kategoriale, *siehe* Faktor
latente, 506
neue aus bestehenden bilden, 56
- Varianz, 76
unkorrigierte, 76, 94
winsorisierte, 80
- Varianzanalyse
assoziierte einfaktorielle, 304, 306
bedingte Haupteffekte, 302
Blockeffekt, 287, 309
dreifaktorielle
abzählige Gruppen (RBF-*pqr*), 311
Split-Plot (SPF-*p · qr*), 323
Split-Plot (SPF-*pq · r*), 324
- unabhängige Gruppen (CRF-*pqr*), 295
Dummy-Codierung, 300, 532
Effektcodierung, 300, 533
Effektstärke, 278, 290, 296, 312, 318
einfaktorielle
abhängige Gruppen (RB-*p*), 286, 292, 520
unabhängige Gruppen (CR-*p*), 274, 276, 277
fester Effekt, 287
Helmert-Codierung, 534
Kontraste
a-priori, 279, 303
paarweise *t*-Tests, 284
post-hoc (Scheffé), 283, 307
Tukey HSD, 285, 308
zweifaktoriell, 307
multivariate
einfaktorielle, 521, 530, 543, 552
zweifaktorielle, 522, 536, 543, 556
orthogonales Design, 295, 298
power, 339
Quadratsummen-Typen, 240, 298, 543
Random-Faktor, 287, 309, 317
simple effects, 302
Stichprobengröße, 339
Treatment-Kontraste, 532
Voraussetzungen, 278
Zirkularität, 287, 290, 314, 319
 ε -Korrektur, 291, 292, 293, 314, 320
Mauchly-Test, 292, 294
zweifaktorielle
abhängige Gruppen (RBF-*pq*), 309
Split-Plot (SPF-*p · q*), 316
unabhängige Gruppen (CRF-*pq*), 294
- Varianzhomogenität
F-Test, 266
Ansari-Bradley-Test, 443
Brown-Forsythe-Test, 268
Fligner-Killeen-Test, 268
Levene-Test, 267
Mood-Test, 442
- Vektor, 23, 31
Anzahl Elemente, 32
Bedingungen prüfen, 37
Datentypen, 35
Elemente auslassen, 33

INDEX

- Elemente auswählen, 32
- Elemente benennen, 35
- Elemente ersetzen, 56
- Elemente löschen, 36
- Elemente ändern, 34
- elementweise Berechnung, 53
- erstellen, 31
- Fälle zählen, 39
- Indexvektor
 - logisch, 39
 - numerisch, 33, 41
- indizieren, 32
- Kreuzprodukt, 485
- Länge, 32
- Norm, 488
- recycling, 54
- Reihenfolge, 50, 51
- Skalarprodukt, *siehe* Matrix
- sortieren, 51
- unbenannt, 33
- Vergleiche, 37
- verkürzen, 36
- verlängern, 34
- zusammenfügen, 32
- zyklische Verlängerung, 54
- Verallgemeinerte Schätzgleichungen, 261
- verallgemeinertes lineares Modell, 343
- Vergleiche, 28
- Verteilung
 - F -Verteilung, 49, 225
 - χ^2 -Verteilung, 49, 225
 - t -Verteilung, 49, 225
 - Binomialverteilung, 49, 225
 - Dichtefunktion, 225
 - Dichtefunktion schätzen, 606
 - Gleichverteilung, 49, 225
 - Hypergeometrische Verteilung, 225
 - kritischer Wert, 221, 227
 - Nonzentralitätspараметр, 336, 338, 340
 - Normalverteilung, 49, 225
 - χ^2 -Test, 417
 - Anderson-Darling-Test, 265
 - Kolmogorov-Smirnov-Test, 413
 - Lilliefors-Test, 265
 - Shapiro-Wilk-Test, 265
 - p -Wert, 221, 227
 - Poisson-Verteilung, 225
- Quantilfunktion, 227
- Verteilungsfunktion, 226
- Wahrscheinlichkeitsfunktion, 225
- Wilcoxon-Vorzeichen-Rang-Verteilung, 225
- verteilungsfreie Methoden, *siehe* nonparametrische Methoden
- Vertrauensintervall, *siehe* Konfidenzintervall
- Verzeichnis
 - Arbeitsverzeichnis, 11
 - Pfadangabe, 152
 - Verzweigung, 672
 - Vorzeichen-Test, 447
 - Vuong-Test, 373
- WAHR, 28
- Wahrheitswerte, 26, 28
- Wald-Wolfowitz-Test, 413
- Wide-Format, *siehe* Datensatz
- Wilcoxon-Test
 - abhängige Stichproben, 451
 - Rangsummen-Test, 449
 - Vorzeichen-Rang-Test, 448
- Wilcoxon-Vorzeichen-Rang-Verteilung, *siehe* Verteilung
- Wilks' Λ , 521, 549
- Winkelfunktionen, *siehe* trigonometrische Funktionen
- Wölbung, 77
- workspace, 12
- Yules Q -Koeffizient, 426
- Zahlen
 - Betrag, 14
 - Dezimalstellen, 14
 - Dezimalteil, 14
 - Dezimaltrennzeichen, 14, 144
 - e, 14
 - Exponentialschreibweise, 14
 - Fakultät, 14
 - ganze, 27
 - Genauigkeit, 29
 - Gleitkommazahlen, 27, 29
 - Grundrechenarten, 13
 - i, 15
 - komplexe, 15, 26
 - Logarithmus, 14
 - π , 14

INDEX

- Quadratwurzel, 14
- reelle, 27
- runden, 14
- unendlich, 14
- Vorzeichen, 14
- Zahlenfolgen erstellen, 46
- Zeichenketten, 26
 - ausführen, 133
 - Globbing-Muster, 132
 - Groß- / Kleinbuchstaben, 127
 - Länge, 124
 - nach Muster erstellen, 125
 - Platzhalter, 132
 - regulärer Ausdruck, 130
 - umkehren, 127
 - Umwandlung in Faktor, 140
 - Umwandlung in Zeichenketten, 124
 - verbinden, 127
 - wildcards, 132
 - Zeichenfolgen ersetzen, 132
 - Zeichenfolgen extrahieren, 130
 - Zeichenfolgen finden, 129
 - Zeilenumbruch, 127
 - zerlegen, 128
- Zeit, *siehe* Uhrzeit
- Zeitreihen, 6
- Zirkularität, *siehe* Varianzanalyse
- Zufallsauswahl, *siehe* Daten
- Zufallseffekt, *siehe* Varianzanalyse
- Zufallsvariablen, 225
- Zufallszahlen, 46, 48, 49
- zufällige Reihenfolge, 51
- Zusammenhangsmaße, 431
 - ϕ -Koeffizient, 431
 - r_{WG} -Koeffizienten, 433
 - Cramér's V , 431
 - Goodman und Kruskals γ , 431
 - Kontingenzkoeffizient, 431
 - Korrelation, 78
 - Matthews-Korrelationskoeffizient, 431
 - Rangkorrelation, 429
 - Somers' d , 394, 431
 - Tschuprows T , 431
- Zusatzpakete, *siehe* Pakete
- Zuweisung, 24
- Zwischenablage, 140, 142, 144
- zyklische Verlängerung, *siehe* Vektor
- Überlebenszeiten, *siehe* Survival-Analise

R-Funktionen, Klassen und Schlüsselwörter

| | |
|----------------------|---|
| !, 28 | %%, 14 |
| !=, 28 | %~%, 485 |
| ", ', 26, 126 | %in%, 43, 160 |
| \$, 100, 106 | %o%, 82 |
| &, &&, 28, 37, 38 | %x%, 485 |
| () , 25, 679 | \(), 679 |
| *, 14, 223 | \, 127 |
| +, 10, 14, 223 | \, 26 |
| -, 14, 223 | , , 28, 37, 38 |
| ., 25, 224 | ~, 14, 223 |
| ..., 680 | ~, 24 |
| .First(), 12, 679 | {}, 673, 675, 680 |
| .GlobalEnv, 12 | ~, 223 |
| .Last(), 12 | 1, -1, 223 |
| .Last.value, 25 | abbreviate(), 127 |
| .Machine, 27, 31 | abind(), 95 |
| .Platform, 217 | abline(), 232, 589 |
| .RData, 13 | abs(), 14 |
| .Rhistory, 13 | acf(), 252 |
| .Rprofile, 12 | acos(), 14 |
| .libPaths(), 19 | across(), 201, 202 |
| /, 14, 223 | ad.test(), 265 |
| :, 46, 223 | add1(), 239 |
| ::, 21, 687 | addmargins(), 115 |
| :::, 687 | addNA(), 64, 110 |
| ;, 8 | adist(), 129 |
| <, 28 | adjustcolor(), 578 |
| <-, 24 | aggregate(), 178 |
| <=, 28 | Agree(), 433 |
| =, 24 | agrep(), 129 |
| ==, 28 | AIC(), 235, 347 |
| >, 7, 28 | aictab(), 235 |
| >=, 28 | all(), 39 |
| ?, 16 | all.equal(), 29, 38 |
| [[]], 98, 106 | all_of(), 184 |
| [] , 32, 86, 96, 107 | AllDuplicated(), 41 |
| #, 8, 17, 207 | annotate(), 645 |
| %, 126 | Anova(), 292, 300, 315, 321, 323, 325, 327, |
| %*%, 485 | 521 |
| %/%, 14 | |

anova(), 241, 277, 293, 327, 353, 360, 365,
 395, 516
 ansari.test(), 443
 any(), 39
 any_of(), 184
 anyNA(), 119
 aov(), 276, 295, 309, 317, 323, 324
 aperm(), 96
 append(), 34
 apply(), 92
 approx(), 657
 approxfun(), 658
 apropos(), 17
 aq.plot(), 248
 Arg(), 15
 args(), 16
 arrange(), 189
 array(), 95
 arrayInd(), 89
 arrows(), 591
 as.<Datentyp>(), 27
 as.<Klasse>(), 22
 as.data.frame(), 106
 as.Date(), 134
 as.list(), 106
 as.matrix(), 106
 as.POSIXct(), 135
 as.POSIXlt(), 135
 as.raster(), 602
 as_factor(), 146
 as_gtable(), 643
 asin(), 14
 assign(), 26
 Assocs(), 432
 atan(), 14
 atan2(), 14
 attach(), 109
 attr(), 23
 attributes(), 23
 ave(), 80
 axis(), 598

 barplot(), 579
 basehaz(), 397
 basename(), 154
 bcPower(), 253
 beeswarm(), 609

 bind_cols(), 191
 bind_rows(), 166, 190
 binom.test(), 410
 BinomCI(), 410
 biplot(), 505
 bkde2d(), 662
 boot(), 463, 474, 696
 boot.ci(), 466, 476, 477
 boot.array(), 465
 box(), 592
 boxplot(), 608
 bptest(), 252
 brat(), 362
 break, 677
 browser(), 690
 bs(), 224, 261

 C(), 536
 c(), 31
 Cairo(), 569
 cancor(), 78
 capture.output(), 124, 687
 case_match(), 57
 case_when(), 189
 cast(), 173
 cat(), 127
 cbind(), 89, 163, 166
 cdplot(), 344, 582
 ceiling(), 14
 character, 26
 chisq.test(), 416, 419
 chol(), 495
 choose(), 44
 choose.files(), 152
 chull(), 616
 class(), 22
 clogit(), 344
 close.screen(), 631
 clusterEvalQ(), 694
 clusterExport(), 694
 cmdscale(), 512
 cnvrt.coords(), 587
 coalesce(), 189
 coef(), 232, 254, 346, 540
 coefest(), 257, 370
 CohenKappa(), 434, 436
 cohens_d(), 270, 272, 273

col(), 85
 col2rgb(), 577
 colMeans(), 91
 colorRamp(), 578
 colorRampPalette(), 578
 colors(), 576
 colSums(), 91
 combn(), 44
 comment(), 23
 complete.cases(), 162
 complex, 26
 confidenceEllipse(), 615
 confint(), 235, 281, 346, 359, 365
 conflicts(), 24
 Conj(), 15
 ConoverTest(), 453
 contains(), 184
 contour(), 618
 contr.sum(), 534
 contr.treatment(), 532
 contr.wec(), 534
 contrasts(), 536
 convertColor(), 578
 convertToDate(), 145
 convert.ToDateTime(), 145
 convolve(), 604
 coord_cartesian(), 650
 coord_fixed(), 650
 coord_flip(), 650
 cor(), 78, 93, 429
 cor.test(), 229, 430
 cor2cov(), 94
 cos(), 14
 count(), 203
 cov(), 78, 93, 429
 cov.wt(), 94
 cov2cor(), 94, 487
 covMcd(), 80
 covOGK(), 80
 cox.zph(), 398
 coxnet(), 403
 coxph(), 393
 cross(), 485
 crossprod(), 485
 cumprod(), 71
 cumsum(), 70
 curve(), 595
 cut(), 68
 cv.glm(), 560
 cv.glmnet(), 258
 D(), 665
 data(), 21
 data.frame(), 104
 data.matrix(), 106
 Date, 134
 date(), 135
 db/Befehl/(), 148
 dbinom(), 226
 dchisq(), 226
 debug(), 689
 delayedAssign(), 680
 demo(), 567, 622
 density(), 606, 661
 deparse(), 133
 Desc(), 70
 desc(), 190
 det(), 492
 detach(), 21, 109
 detectCores(), 693
 dev.cur(), 568
 dev.list(), 568
 dev.new(), 568
 dev.next(), 568
 dev.off(), 569
 dev.prev(), 568
 dev.set(), 568
 deviance(), 232, 347
 df(), 226
 diag(), 89, 485
 diff(), 70
 difftime(), 137
 dim(), 84, 105
 dimnames(), 108
 dir(), 206
 dir.create(), 154
 dir.exists(), 154
 dirname(), 154
 dist(), 489
 distinct(), 187
 dmvnorm(), 619
 dnorm(), 226
 do.call(), 177, 180, 694
 dotchart(), 584

```

double, 27
dperm(), 479
dpill(), 660
drop, 87, 107
drop1(), 239, 300, 353
droplevels(), 64, 159
dt(), 226
dummy.coef(), 540
dump(), 143
DunnTest(), 453
duplicated(), 41, 161
durbinWatsonTest(), 252

ecdf(), 116, 613
eigen(), 493
ellipse(), 615
else, 673
emmeans(), 332
emmip(), 244
emtrends(), 243
ends_with(), 184
endsWith(), 129
Entropy(), 77
erase.screen(), 631
ErrBars(), 600
Error(), 288, 309, 317
eta_squared(), 278, 290, 296, 312, 318
eval(), 133
everything(), 190
example(), 17
exists(), 24
exp(), 14
expand.grid(), 45, 68
expression(), 597
extractAIC(), 235

F, siehe FALSE
fa.parallel(), 510
facet_grid(), 643
facet_wrap(), 643
factanal(), 507
factor(), 60
factorial(), 14, 71
FALSE, 28
fct_collapse(), 64
fct_drop(), 64
fct_expand(), 63

fct_na_level_to_value(), 65
fct_na_value_to_level(), 65
fct_recode(), 63
fct_relevel(), 67
fft(), 604
file.choose(), 152
file.copy(), 154
file.create(), 154
file.exists(), 154
file.remove(), 154
file.rename(), 154
file_ext(), 154
file_path_sans_ext(), 154
filled.contour(), 619
filter(), 186, 200
find.package(), 19
fisher.test(), 422, 424
FisherZ(), 230
FisherZInv(), 230
fitdistr(), 668
fitted(), 232, 349
fligner.test(), 268
floor(), 14
for(), 675
format(), 134, 136
formatC(), 124
formula, 222
fread(), 142, 692
friedman.test(), 453
ftable(), 114
full_join(), 192
function(), 679
fwrite(), 142

gam(), 261
gamma(), 71
gauss.quad(), 665
gc(), 692
geom_bar(), 635, 636
geom_beeswarm(), 640
geom_boxplot(), 635, 639
geom_col(), 635, 637
geom_density(), 635, 639
geom_errorbar(), 645
geom_function(), 645
geom_hex(), 648
geom_histogram(), 635, 638

```

| | |
|----------------------------|-----------------------------|
| geom_hline(), 644 | hcl(), 578 |
| geom_line(), 635 | head(), 105 |
| geom_linerange(), 645 | heatmap(), 624 |
| geom_point(), 635 | help(), 16 |
| geom_polygon(), 644 | help.search(), 17 |
| geom_qq(), 641 | help.start(), 16 |
| geom_qq_line(), 641 | hessian(), 667 |
| geom_rect(), 644 | hexbin(), 617 |
| geom_ribbon(), 645 | hist(), 605 |
| geom_rug(), 645 | hist3d(), 620 |
| geom_segment(), 644 | histbackback(), 605 |
| geom_smooth(), 645 | history(), 12 |
| geom_text(), 645 | Hmean(), 73 |
| geom_violin(), 640 | HodgesLehmann(), 74 |
| geom_vline(), 644 | hsv(), 578 |
| get(), 25, 687 | huberM(), 75 |
| getAnywhere(), 687 | hurdle(), 374 |
| getOption(), 12 | I(), 104, 224 |
| getS3method(), 687 | ICC(), 439 |
| getwd(), 11 | identical(), 29 |
| ggplot(), 633 | identify(), 571 |
| ggsave(), 634 | identity(), 82 |
| GiniMd(), 80 | if(), 672 |
| ginv(), 488 | if_all(), 200 |
| gl(), 68 | if_any(), 200 |
| glht(), 279, 304, 322, 332 | if_else(), 189 |
| glm(), 344, 367 | ifelse(), 58 |
| glm.diag(), 349 | Im(), 15 |
| glm.nb(), 371 | Inf, 14 |
| glmnet(), 258 | influence.measures(), 249 |
| glob2rx(), 132, 153 | influenceIndexPlot(), 250 |
| Gmean(), 73 | influencePlot(), 250 |
| GoodmanKruskalGamma(), 431 | inherits(), 22, 683 |
| grad(), 665 | inner_join(), 191, 193 |
| graphics.off(), 569 | install.packages(), 18 |
| gray(), 578 | install_github(), 19 |
| gregexpr(), 131 | installed.packages(), 20 |
| grep(), 130 | integer, 27 |
| grepl(), 130 | integrate(), 664 |
| grepv(), 130 | interaction(), 62, 175, 295 |
| grid(), 589 | interaction.plot(), 296 |
| group_by(), 195, 202 | intersect(), 42 |
| gsub(), 132 | inverse.rle(), 112 |
| guides(), 652 | invisible(), 684 |
| hasName(), 99, 157 | IQR(), 79 |
| hatvalues(), 248 | is.Datentyp(), 27 |

| | |
|---|-------------------------------|
| is.<Speicherart>(), 27 | lm.ridge(), 257 |
| is.element(), 43, 160 | lmrob(), 256 |
| is.finite(), 15 | load(), 142 |
| is.infinite(), 15 | loadedNamespaces(), 21 |
| is.na(), 15, 118 | loadhistory(), 13 |
| is.nan(), 15 | locator(), 586 |
| is.null(), 15 | locpoly(), 660 |
| isFALSE(), 29 | loess(), 660 |
| ISOdate(), 136 | loess.smooth(), 660 |
| isTRUE(), 29, 38 | log(), log2(), log10(), 14 |
| jitter(), 605, 617 | logical(), 26 |
| joint_tests(), 302 | logLik(), 347 |
| JonckheereTerpstraTest(), 453 | loglm(), 376 |
| jpeg(), 569, 634 | logm(), 485 |
| kable(), 214 | logrank_test(), 390 |
| kappa(), 255, 493 | lower.tri(), 85 |
| KappaM(), 435 | ls(), 12, 25 |
| KendallW(), 437 | ls.str(), 25 |
| knit(), 209 | ltsReg(), 256 |
| knots(), 117 | mad(), 79 |
| kronecker(), 485 | mahalanobis(), 491 |
| kruskal.test(), 452 | make.names(), 139, 143 |
| ks.test(), 414, 445 | makeCluster(), 693 |
| Kurt(), 77 | Manova(), 523 |
| labs(), 644, 649, 652 | manova(), 515, 518, 521, 522 |
| lapply(), 177, 179 | mantelhaen.test(), 423 |
| layout(), 626 | mapply(), 181 |
| layout.show(), 627 | match(), 129 |
| lbl_test(), 433 | match.arg(), 681 |
| lcm(), 627 | matches(), 184 |
| lda(), 524 | matlines(), 589 |
| left_join(), 191 | matplot(), 572 |
| legend(), 596 | matpoints(), 588 |
| length(), 23, 32, 97, 105 | matrix(), 83 |
| lengths(), 97 | mauchly.test(), 294, 315, 321 |
| LETTERS, letters, 32 | max(), 71 |
| levels(), 63 | max.col(), 361 |
| leveneTest(), 267 | mclapply(), 696 |
| library(), 20 | mcnemar.test(), 457, 458 |
| lillie.test(), 265 | mean(), 72 |
| lines(), 589 | MeanAD(), 80 |
| list(), 97 | median(), 73 |
| list.files(), 152 | melt(), 173 |
| lm(), 231, 236, 277, 300, 515, 516, 523 | merge(), 167 |
| lm.influence(), 251 | methods(), 685 |
| | mh_test(), 460 |

```

min(), 71
missing(), 681
mle(), 669
Mod(), 15
Mode(), 73
mode(), 23, 26
model.frame(), 232
model.matrix(), 222, 232, 529
model.tables(), 276, 296
monoMDS(), 512
months(), 136
mood.test(), 442
mosaic(), 378
mosaicplot(), 582
mtext(), 597
mutate(), 188, 201

n(), 188, 197
n2mfrow(), 629
n_distinct(), 187
NA, 15, 117
na.exclude(), 231
na.fail(), 123
na.omit(), 120, 121, 123, 162
na_if(), 187
names(), 35, 99, 108
NaN, 15
nchar(), 124
ncol(), 84, 105
next, 677
nlevels(), 61
nls(), 261
noquote(), 127
norm(), 489
normal_test(), 478
nrow(), 84, 105
ns(), 224, 261
NULL, 15
Null(), 488
numeric, 26
numeric(), 31

object.size(), 691
OddsRatio(), 426
odTest(), 371
omega_squared(), 278
on.exit(), 684

oneway.test(), 274
oneway_test(), 478
optim(), 669
optimize(), 669
options(), 12, 14, 123, 234, 535, 683, 688
order(), 51, 90, 164
ordered(), 65
outer(), 82

PageTest(), 455
pairs(), 622
pairwise.t.test(), 284
pairwise.wilcox.test(), 451
palette(), 577
par(), 574, 628
parLapply(), 694
parse(), 133
paste(), 125
path.expand(), 11
rbinom(), 226
pbirthday(), 225
pchisq(), 226
pcout(), 248
pdf(), 569, 634
PearsonTest(), 417
PercentRank(), 75
Permn(), 45, 482
persp(), 620
persp3d(), 620
pf(), 226
pi, 14
pie(), 613
pivot_longer(), 194
pivot_wider(), 195
plogis(), 343
plot(), 116, 250, 286, 570
plot.design(), 277, 296
plot.new(), 585
plot3d(), 620
PlotFdist(), 605
pmatch(), 129
pmax(), 72
pmin(), 72
pnorm(), 226
points(), 588
poly(), 224, 261
polygon(), 593

```

| | | | |
|--------------------|--------------------------------|-------------------|-----------------|
| polypath() | , 593 | rasterImage() | , 602 |
| polyroot() | , 664 | rbind() | , 89, 166 |
| position_dodge() | , 647 | rbinom() | , 49 |
| position_fill() | , 647 | rchisq() | , 49 |
| position_jitter() | , 647 | rcorr() | , 230 |
| position_stack() | , 647 | Re() | , 15 |
| POSIXct | , 135 | read.table() | , 139 |
| POSIXlt | , 135 | read_dta() | , 147 |
| PostHocTest() | , 283, 307 | read_sas() | , 147 |
| power.anova.test() | , 339 | read_sav() | , 146 |
| power.t.test() | , 337 | read_xlsx() | , 144 |
| powerTransform() | , 253 | readLines() | , 152 |
| ppoints() | , 612 | readRDS() | , 142 |
| prcomp() | , 499 | rect() | , 592 |
| predict() | , 245, 349, 361, 366, 401, 525 | reframe() | , 198 |
| princomp() | , 499, 504 | regexpr() | , 131 |
| print() | , 25 | regmatches() | , 131 |
| prod() | , 71 | relevel() | , 66 |
| prop.test() | , 421 | relocate() | , 190 |
| prop.trend.test() | , 422 | RelRisk() | , 427 |
| proportions() | , 111 | remove.packages() | , 19 |
| PseudoR2() | , 348 | rename() | , 184 |
| pt() | , 226 | rename_with() | , 198 |
| pull() | , 187 | render() | , 209 |
| q() | , 11 | reorder() | , 66 |
| qbinom() | , 227 | rep() | , 48 |
| qchisq() | , 227 | repeat | , 678 |
| qda() | , 523 | replace() | , 56 |
| qf() | , 227 | replicate() | , 678 |
| qlogis() | , 343 | require() | , 20 |
| Qn() | , 80 | reshape() | , 173 |
| qnorm() | , 227 | residualPlots() | , 250 |
| qperm() | , 479 | residuals() | , 232, 347, 398 |
| qqline() | , 613 | return() | , 684 |
| qqnorm() | , 613 | rev() | , 50 |
| qqplot() | , 611 | rf() | , 49 |
| qr() | , 492, 495 | rgb() | , 577 |
| qt() | , 227 | rgb2hsv() | , 578 |
| quantile() | , 75, 389 | right_join() | , 192 |
| quarters() | , 136 | rle() | , 112 |
| quartz() | , 568 | rm() | , 26 |
| r.test() | , 230 | rmvnorm() | , 368, 490 |
| R.Version() | , 217 | RNGkind() | , 219 |
| range() | , 71 | rnorm() | , 49 |
| rank() | , 55 | roc() | , 427 |
| | | round() | , 14 |
| | | row() | , 85 |

```

rowMeans(), 91
rownames(), 108
rowSums(), 91
rperm(), 479
Rprofile.site, 11
rstandard(), 252, 562
rstudent(), 252
rt(), 49
rug(), 605
runif(), 49
RunsTest(), 412

sample(), 48
sandwich(), 257
sapply(), 176, 179
save(), 142
savehistory(), 13
saveRDS(), 142
scale(), 54
scale_color_continuous_sequential(), 654
scale_colour_gradient(), 654
scale_colour_grey(), 654
scale_colour_hue(), 654
scale_fill_discrete_qualitative(), 654
scale_x_continuous(), 649
scale_x_date(), 649
scaleTau2(), 80
scan(), 150, 151
scatter3d(), 237
ScheffeTest(), 283, 307
screen(), 630
sd(), 76
search(), 12
segments(), 590
select(), 184, 185, 258
seq(), 47
seq_along(), 47
seq_len(), 46
sessionInfo(), 217
set.seed(), 46, 219
setdiff(), 43
setequal(), 42
setNames(), 36, 99, 157
setRepositories(), 19
setwd(), 11
shapiro.test(), 265
shell(), 7, 207

showMethods(), 685, 687
sign(), 14
signif(), 14
SignTest(), 447
simulate(), 476
sin(), 14
sink(), 124
Skew(), 77
slice(), 186
slice_head(), 186
slice_sample(), 187
slice_tail(), 186
smooth.spline(), 659
smoothScatter(), 606, 617
Sn(), 80
sobel(), 233
some(), 52
SomersDelta(), 431
sort(), 51
sort_by(), 164
source(), 143, 206
spineplot(), 582
spline(), 659
splinefun(), 659
split(), 165
split.screen(), 629
splitIndices(), 694
sprintf(), 126
sqrt(), 14
sqrtm(), 494
stack(), 170
stars(), 619
starts_with(), 184
startsWith(), 129
stem(), 606
step(), 239
stop(), 682
stopCluster(), 694
stopifnot(), 682
str(), 23, 106
stringsAsFactors, 45
stripchart(), 609
strptime(), 135
strrep(), 127
StrRev(), 127
strsplit(), 128
strtrim(), 127

```

```

structure(), 23
sub(), 132
subset(), 158
substring(), 130
sum(), 70
summarise(), 197, 202
summary(), 61, 70, 105, 114, 233, 249, 276,
    280, 293, 329, 352, 502, 521
sunflowerplot(), 619
support(), 479
Surv(), 383
survdiff(), 390
survfit(), 388, 395
survreg(), 405
survSplit(), 386
svd(), 495
sweep(), 92
switch(), 674
symbols(), 619
symmetry_test(), 455, 457, 459
Sys.Date(), 134
Sys.getenv(), 4
Sys.getlocale(), 28, 134
Sys.glob(), 153
Sys.info(), 217
Sys.setlocale(), 28, 134
Sys.time(), 135
system.time(), 691

T, siehe TRUE
t(), 484
t.test(), 269, 270, 273
table(), 110
tail(), 105
tan(), 14
tapply(), 80
terms(), 224
test(), 243
text(), 596
textConnection(), 141
theme(), 656
theme_bw(), 656
theme_classic(), 656
theme_minimal(), 656
title(), 596
tmd(), 440
tolower(), 127

toString(), 124
toupper(), 127
trace(), 690
traceback(), 688
transform(), 163
trimws(), 127
TRUE, 28
trunc(), 14
try(), 683
tryCatch(), 683
TschuprowT(), 432
TukeyHSD(), 285, 308
typeof(), 27

unclass(), 61
undebug(), 689
ungroup(), 196
union(), 42
unique(), 42, 161
uniroot(), 662
unlist(), 103
uname(), 36
unstack(), 171
Untable(), 115
untrace(), 690
update(), 238, 348
update.packages(), 19
upper.tri(), 85
UseMethod(), 685

vapply(), 179
var(), 76
var.test(), 266
vcov(), 232
vcovHC(), 257, 370
vector(), 31
vglm(), 358, 364, 367, 371–373
View(), 105
vif(), 255
vignette(), 21
VSS(), 510
vuong(), 373

warning(), 682
weekdays(), 136
weighted.mean(), 72
where(), 185
which(), 41, 89

```

which.max(), 71
which.min(), 71
while(), 677
wilcox.test(), 448, 449, 451
windows(), 567
Winsorize(), 74, 80
with(), 108
write.table(), 141
write_dta(), 147
write_sav(), 147
writeLines(), 152

x11(), 568
xlab(), 649
xor(), 28
xspline(), 659
xtabs(), 110

ylab(), 649
YuleQ(), 427

zapsmall(), 30
zeroinfl(), 372

Zusatzpakete

abind, 95
AICcmodavg, 235
Amelia, 123
arrow, 692

beeswarm, 609
bench, 691
betareg, 262
boot, 349, 463, 474, 560, 696
brms, 221, 356

Cairo, 569
CalibrationCurves, 350
car, 52, 237, 247, 250, 252, 253, 255, 267, 292, 315, 321, 323, 325, 327, 521, 523, 615
carData, 8, 104
caret, 564
codingMatrices, 530
coin, 390, 433, 455, 457, 459, 460, 478
colorspace, 577, 578, 654
contingencytables, 419
correlation, 78, 79
coxphf, 403
cubature, 665
cv, 559

DAAG, 104
data.table, 142, 173, 182, 692
DBI, 148
DescTools, 41, 45, 70, 73–75, 77, 80, 115, 128, 230, 283, 307, 348, 410, 412, 417, 426, 427, 431–437, 439, 447, 453, 455, 461, 482, 600, 605
DHARMa, 349
dplyr, 57, 59, 155, 166, 182

effectsize, 270, 272, 273, 278, 290, 296, 312, 318
emmeans, 243, 296, 302, 327, 332
esquisse, 632
exact2x2, 422, 423

expm, 485, 494
fastLink, 216
fitdistrplus, 668
flextable, 214
forcats, 62
foreign, 146

gamlss, 262
geepack, 261
ggridge, 632
ggbeeswarm, 640
ggplot2, 632
ggrepel, 645
ggthemes, 655
glmnet, 258, 403
glmnetUtils, 258
GPArotation, 507
grid, 632
gt, 214, 643

haven, 146
hexbin, 617
Hmisc, 230, 587, 605
HSAUR3, 104
hypr, 532

imager, 602, 604
interp, 659
ipred, 560, 564

KernSmooth, 660, 662
knitr, 207, 209

lattice, 441
lavaan, 506
lme4, 261
lmSubsets, 239
lmtest, 252, 257, 370, 373
logistf, 356
lubridate, 134

magick, 602, 604

MASS, **257**, 371, 376, 488, 523, 524, 668
 Matrix, **484**
 MBA, **659**
 MBESS, **94**
 mediation, **233**
 mgcv, **261**
 mice, **123**
 microbenchmark, **691**
 mmrm, **261**
 multcomp, **279**, 304, 322, 332
 multilevel, **233**
 MVN, **266**
 mvoutlier, **248**
 mvtnorm, **225**, 368, 490, 619
 naniar, **161**
 nlme, **261**
 nortest, **265**
 numDeriv, **665**
 odbc, **148**
 OpenMx, **506**
 openxlsx, **145**
 optimx, **669**
 pak, **20**
 parallel, **693**
 parallelly, **693**
 patchwork, **643**
 pcaPP, **500**
 permute, **478**
 perturb, **255**
 pheatmap, **624**
 plotly, **621**
 polyCub, **665**
 pracma, **485**
 pROC, **427**
 profvis, **691**
 pscl, **371**
 psych, **230**, 506, 507, 510
 pwr, **334**
 quantreg, **262**
 quarto, **214**
 rayrender, **622**
 rayshader, **622**
 readr, **142**
 readxl, **144**
 remotes, **19**
 renv, **218**
 rgl, **620**, 661
 riskyr, **425**
 rix, **218**
 rmarkdown, **207**, 209
 rms, **80**, 348
 robustbase, **75**, 80, 256
 rootSolve, **664**
 rsample, **463**, 560, 562
 RSQLite, **148**
 rstanarm, **221**, 356
 sandwich, **257**, 370
 santoku, **70**
 sem, **506**
 sets, **41**
 shiny, **6**
 simstudy, **104**
 simsurv, **384**
 splines, **224**, 261
 statmod, **665**
 stats4, **669**
 stringdist, **129**
 stringr, **123**
 survival, **344**, 381, 383, 386, 388, 390, 393, 397, 405
 tidyR, **194**
 tinytex, **209**
 tools, **154**
 vcd, **344**, 378
 vegan, **512**
 VGAM, **358**, 360, 362, 364, 367, 371–373
 wec, **534**