Statistik mit *R*

Einführung und Anwendung

Ingo Steinke, Toni C. Stocker

Version: Februar 2020

Vorwort

R ist eine in der Statistik beliebte Programmierumgebung, die *S* bzw. *S+* nachempfunden ist und den Nutzern kostenlos zur Verfügung steht. Das vorliegende Skript führt zunächst in die grundlegenden Funktionalitäten und Datenstrukturen von *R* ein. Darauf aufbauend liegt der inhaltliche Schwerpunkt dann darauf, wie sich ausgewählte deskriptive und induktive Analysemethoden, die typischerweise in vielen einführenden Statistik-Lehrveranstaltungen thematisiert werden, praktisch umsetzen lassen. Das Skript ist als Begleitmaterial zum Lehrbuch „Statistik – Grundlagen und Methodik“ der beiden Autoren konzipiert und auf dieses inhaltlich abgestimmt. An verschiedenen Stellen, etwa bei Formeln, Abbildungen oder Beispielen, wird darauf explizit Bezug genommen. Prinzipiell kann das Skript jedoch auch unabhängig davon als eigenständige Einführung verwendet werden.

Die an verschiedenen Stellen (insbesondere in Kapitel 3) verwendeten Datensätze finden sich als Begleitmaterial zum Lehrbuch auf der Webseite

https://www.vwl.uni-mannheim.de/rothe/lehrbuecher/

einschließlich zugehöriger Informationsdateien zur jeweiligen inhaltlichen Bedeutung.

Verbleibende inhaltliche und typografische Fehler sind mit Sicherheit vorhanden. Für Hinweise sind wir diesbezüglich sehr dankbar. Bei Bedarf werden entsprechend Korrekturen und Ergänzungen vorgenommen.

Mannheim, im Februar 2020 Ingo Steinke und Toni C. Stocker

Inhaltsverzeichnis

1 Einige Grundlagen

1.1 Grundlegendes zum Start

1.1.1 Installation

R ist eine Programmiersprache und Programmierumgebung für statistische Analysen. Es wird von einer weltweiten Entwickler- und Nutzergemeinschaft kostenlos bereitgestellt und ständig weiterentwickelt. Über die Internetadresse

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

kann *R* heruntergeladen werden. Dies ist die Homepage von *R*. Neue *R*-Versionen werden hier veröffentlicht und zusätzliche Informationen sowie Benutzerhandbücher (*Manuals*) werden hier zur Verfügung gestellt.

Zur Installation einer aktuellen Version (Version 3.6.1 wurde am 5.7.19 veröffentlicht) gehen wir zunächst auf

CRAN.

Wir wählen anschließend einen Server aus, z. B.

https://cran.uni-muenster.de/

der Universität Münster. Das Klicken auf

Download R for Windows

(gegebenenfalls (Mac)OS X oder Linux) und anschließend auf

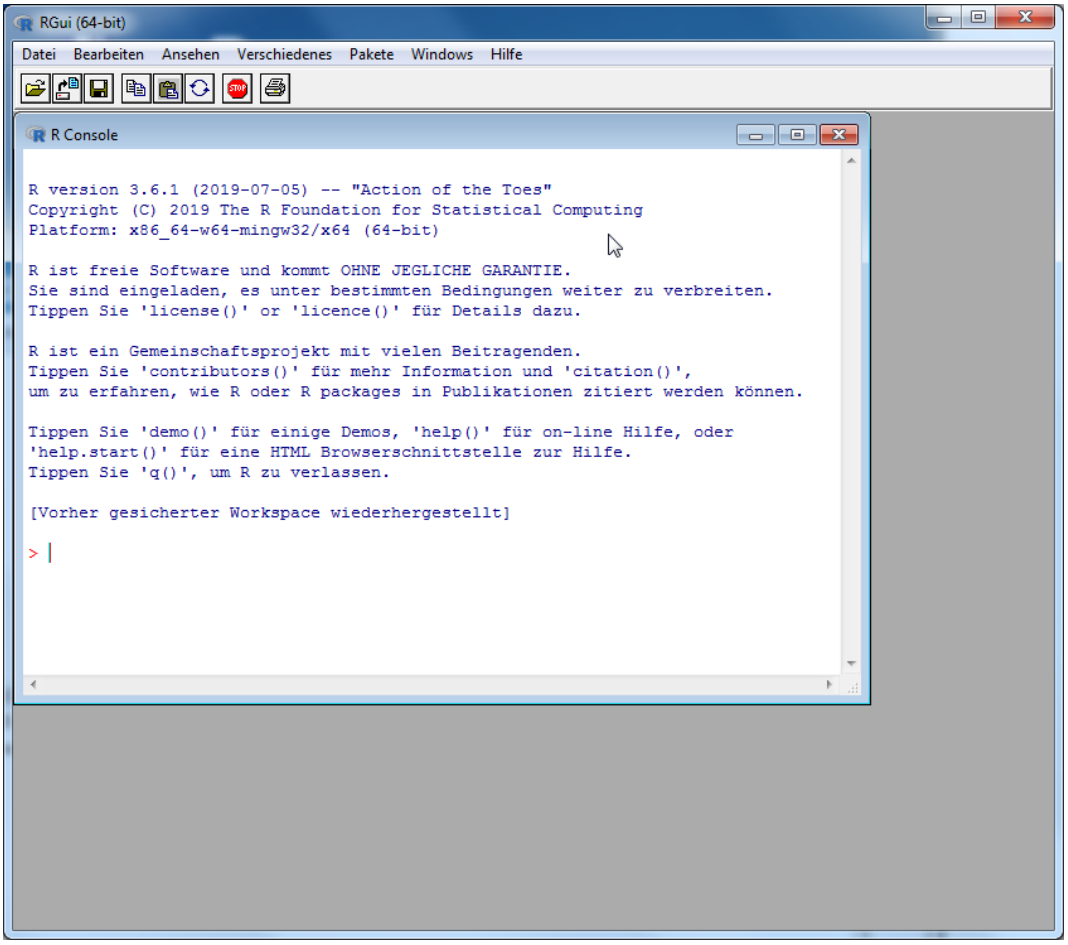
base

führt uns auf eine Seite, auf der wir die Installationsdateien von *R* finden. Wir wählen hier z. B.

Download R 3.6.1 for Windows

aus. Auf dem Desktop erscheint dann ein Icon zur Installation der aktuellen Version. Doppelklicken darauf startet den Installationsprozess. Es ist hierbei am einfachsten, den Anweisungen zu folgen und alle Voreinstellungen zu übernehmen. Am Ende erscheint ein *R*-Icon mit der aktuellen Versionsnummer auf dem Desktop als Verknüpfung. Doppelklicken darauf startet die grafische Benutzeroberfläche von *R* (vgl. Abb. 1.1.1).

Abbildung 1.1.1: *R*-Programmfenster unter Windows nach dem Start



1.1.2 Ausführung von Anweisungen

Das Programmfenster von *R* ist sehr schlicht, dafür aber auch übersichtlich. Im inneren Teil befindet sich die *R-Konsole* (auch *Command*-Fenster genannt) zur Eingabe von Anweisungen. Sämtliche Anweisungen zur Durchführung statistischer Berechnungen und zur Erzeugung von Grafiken werden wir vorerst hier eingeben. Später werden wir längere Programme in einem Texteditor schreiben, dessen Inhalt an das *Command*-Fenster übergeben wird. Mit der Menüleiste im oberen Teil des *R*-Fensters werden wir uns vorerst nicht befassen. Die Symbolleiste in der zweiten Reihe ist weitgehend selbsterklärend.

Der Fokus von *R* liegt auf der Programmierebene. Dies macht einen entscheidenden Unterschied zu verbreiteter kommerzieller Software wie bspw. *Excel* oder *SPSS* aus, wo die meisten Rechen- und Grafikoperationen menügeführt sind, d.h. mit Hilfe der grafischen Benutzeroberflächen durch das Klicken auf Menüs und Symbole durchgeführt werden können. Der Ansatz von *R*, nämlich alles über Anweisungen einzugeben, wirkt da zunächst einmal umständlich. Langfristig wird sich dieser Umstand jedoch auszahlen. Der im Fachgebiet kundige Benutzer bleibt Herr der Lage. So kann er statistische Berechnungen nach seinen in-

dividuellen Bedürfnissen durchführen und bleibt sehr flexibel in der Gestaltung von Grafiken. Für den Benutzer ist die Software keine Blackbox mehr, über deren Funktionsweise er sich nicht so recht im Klaren ist, sondern ein für ihn transparent funktionierendes Werkzeug. Alle Programme und implementierten *R*-Anweisungen sind in *R* einsehbar und können zumindest theoretisch bis ins letzte Detail nachvollzogen werden. Wenn man verstehen möchte, wie die Ergebnisse eigener statistischen Analysen zustandekommen, dann ist es hilfreich, wenn man diese (zumindest theoretisch) auch „von Hand“ nachrechnen könnte.

Geben wir im *Command*-Fenster nach dem **Bereitschaftszeichen** (`>`)

```
> 1+1
```

ein und betätigen anschließend die Enter-Taste, erhalten wir

```
[1] 2
>
```

Das Rechenergebnis 2 wird ausgegeben (was für eine Überraschung!). Die Bedeutung des vorangestellten `[1]` wird sich später noch klären (s. S. 30). Weiter erhalten wir wieder ein *Bereitschaftszeichen* für die Eingabe der nächsten Anweisung. Die Eingabe von

```
> 1+2+3
```

liefert

```
[1] 6
>
```

Jeder neuen Anweisung muss ein *Bereitschaftszeichen* vorausgehen. Das nächste *Bereitschaftszeichen* erscheint nur dann, falls die letzte Anweisung richtig, d.h. in korrekter *R*-Syntax, abgeschlossen wurde. So folgt der Eingabe von

```
> 1+2+3+
```

die Ausgabe

```
+
```

Das Pluszeichen ist hier nicht als Rechenoperator zu interpretieren, sondern als **Fortsetzungszeichen** für eine noch nicht abgeschlossene Anweisung. *R* hat in diesem Fall erkannt, dass die eingegebene Anweisung noch nicht vollständig ist. Der letzte Summand fehlt. Ergänzen wir nach dem Pluszeichen eine „4“, also

```
+ 4
```

erhalten wir das Rechenergebnis der vollständigen Anweisung

```
[1] 10
>
```

und das nächste *Bereitschaftszeichen*. Das *Fortsetzungszeichen* erscheint also immer dann, wenn *R* die Anweisungseingabe als noch nicht vollständig betrachtet. So könnte es auch sein, dass z.B. eine Klammerung noch nicht abgeschlossen wurde. Geben wir

```
> 2*(2+3
```

ein, folgt ein *Fortsetzungszeichen*, da die Klammer nicht geschlossen wurde. Ergänzen wir

```
+ )
```

nach dem *Fortsetzungszeichen*, so erhalten wir das zu erwartende Ergebnis

```
[1] 10
```

> Man kann das *Bereitschaftszeichen* auch durch Betätigen der *Esc*-Taste erzwingen (Anweisungsabbruch).

1.2 Grundlegende Funktionalitäten

1.2.1 *R* als Taschenrechner

R kann als komfortabler Taschenrechner verwendet werden. Die vier Grundrechenarten werden erwartungsgemäß mit den Operatoren

`+, -, *, /`

ausgeführt. Dezimalstellen sind über einen Punkt zu deklarieren, also

1.8 und nicht 1,8.

Es folgen ein paar Beispielrechnungen.

```
> 1.8+2
[1] 3.8

> 1.8-2
[1] -0.2

> 1.8*2
[1] 3.6

> 1.8/2
[1] 0.9
```

Dabei wird die „Punkt-vor-Strich-Regel“ automatisch beachtet. Mehrere „Punkt“-Operationen, d. h. Multiplikationen und Divisionen, werden dabei in der Reihenfolge ihres Auftretens ausgeführt. Im Rahmen mathematischer Berechnungen können nur runde Klammern verwendet werden.

```
> 2+2*3
[1] 8

> (2+2)*3
[1] 12
```

```
> 2/2*2
[1] 2

> 2/2/2
[1] 0.5
```

In den letzten beiden Beispielen wird also `(2/2)*2` bzw. `(2/2)/2` gerechnet.

Exponenten werden über

`^`

deklariert, also mit Hilfe der *Dach-Taste*. Die Zahl 2^3 wird also als `2^3` geschrieben. Beachten Sie, dass dieses Symbol meistens erst nach der Eingabe der ersten Zahl für den Exponenten erscheint.

```
> 2^3
[1] 8

> 8^(1/3)
[1] 2

> 3^2
[1] 9

> 9^0.5
[1] 3
```

Potenzausdrücke werden vor Multiplikation und Addition ausgewertet. Treten mehrere Potenzen auf, werden diese von rechts nach links berechnet.

```
> 2^2*2+2
[1] 10

> 2^3^2
[1] 512
```

`2^3^2` wird also gemäß `2^(3^2)` berechnet.

```
> 2^(2*((0.2+0.3)*(1+2)))+4
[1] 12
```

Das letzte Beispiel zeigt, dass bei Mehrfachklammern die Übersichtlichkeit u.U. leidet, da nur runde Klammern verwendet werden dürfen.

1.2.2 Funktionen und Zuweisungen

Funktionen

Funktionen in *R* haben einen *Funktionsbezeichner* und ein oder mehrere *Argumente*, die dem Funktionsbezeichner in runden Klammern folgen. Eine Funktion liefert i.d.R. einen Funktionswert als Ergebnis. Manche *R*-Funktionen haben aber auch keinen Funktionswert.

Häufig verwendete mathematische Funktionen sind in *R* implementiert. Eine Auswahl finden Sie in nachfolgender Tabelle.

R-Funktion	Funktion
`abs`(...)	Absolutbetrag einer Zahl
`round`(...,m)	Rundung auf m Nachkommastellen
`sqrt`(...)	Wurzelfunktion (Quadratwurzel)
`exp`(...)	Exponentialfunktion
`log`(...)	natürlicher Logarithmus
log(...,10)	Logarithmus mit Basis 10
`sin`(...)	Sinusfunktion
`cos`(...)	Cosinusfunktion
`tan`(...)	Tangensfunktion

Es folgen ein paar einfache Beispielrechnungen.

```
> sqrt(2)
[1] 1.414214

> exp(1)
[1] 2.718282

> log(7.389056)
[1] 2

> log(exp(3))
[1] 3

> log(100,10)
[1] 2

> abs(1.8-2)
[1] 0.2
```

Beim Runden gibt man als zweites *Argument* der Funktion `round` die Anzahl der darzustellenden Nachkommastellen an.

```
> round(sqrt(2),4)
[1] 1.4142
```

Die Kreiszahl π (3.141593...) erhält man durch Eingabe von `pi`. Die Euler'sche Zahl e erhält man über `exp(1)`, also über e^1.

```
> pi
[1] 3.141593

> sin(pi/2)
[1] 1

> cos(pi)
[1] -1
```

```
> sin(pi)
[1] 1.224606e-16

> tan(pi)
[1] -1.224606e-16
```

Es ist zu beachten, dass die Ausgabe

```
    1.224606e-16
```

als

$$1.224606 \cdot 10^{-16}$$

zu interpretieren ist. Wie wir gesehen haben, erhielten wir das Ergebnis aus `sin(pi)`. In der Praxis verbergen sich hinter solchen Ausgaben häufig „exakte Nullen“, d.h. das Rechenergebnis müsste (theoretisch) exakt 0 sein ($\sin(\pi) = 0$). Da aber R nur mit einer begrenzten Anzahl von Nachkommastellen arbeiten kann, wird ein numerisch ungenaues Ergebnis ausgegeben.

Komplexe Zahlen werden mit Hilfe der imaginären Einheit `1i` dargestellt. `2i` bzw. `3i` stehen dann bspw. für $2 \cdot i$ bzw. $3 \cdot i$. Auf diese Weise sind auch Berechnungen für komplexe Zahlen möglich.

```
> 1i+(3+2i)
[1] 3+3i

> (1+1i)*(3+2i)
[1] 1+5i

> exp(1+1i)
[1] 1.468694+2.287355i

> abs(1+1i)
[1] 1.414214
```

Die Funktion `abs` berechnet die *Norm* $\sqrt{x^2 + y^2}$ für eine komplexe Zahl $x + y \cdot i$.

Zuweisungen

Durch ein Gleichheitszeichen können wir **Zuweisungen** vornehmen. Darunter versteht man die Speicherung eines bestimmten Wertes (auch mehrerer Werte oder größerer Datenmengen) unter einem vorgegebenen (Objekt-)*Namen*.

```
> x=0.5
> x
[1] 0.5
```

In diesem Fall wird unter dem Namen `x` (wir könnten auch von dem **Objekt** oder der **Variable** `x` sprechen) zunächst der Wert 0.5 gespeichert. Der Wert von `x` kann durch Ein-

gabe von `x` abgefragt werden. Mithilfe des Objektnamens `x` können nun weitere Rechnungen durchgeführt werden. Dabei wird der gerade aktuell gespeicherte Wert von `x` verwendet.

```
> x+1
[1] 1.5
> x=1
> x
[1] 1
```

Bei einer neuen Wertzuweisung (hier 1) wird der alte Wert (0.5) überschrieben und geht somit verloren.

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

Beachten Sie, dass der vorhandene Wert von `x` ohne Warnung überschrieben wurde. In der Regel ist das ein Segen. Bei Unachtsamkeit kann das aber auch zum Fluch werden.

```
> pi=2
> sin(pi)
[1] 0.9092974
```

In diesem Beispiel wurde die Variable `pi` überschrieben. Sollte `pi` später in der Bedeutung von π verwendet werden sollen, ohne den richtigen Wert wieder herzustellen, wird das voraussichtlich zu Fehlern im Rechenergebnis führen.

Alternative Zuweisungsmöglichkeiten verwenden die folgende Syntax:

```
> x<-2
> 3->y
> x+y
[1] 5
```

Hier wird `x` der Wert 2 und `y` der Wert 3 zugewiesen. Man kann *mehrere R-Anweisungen* auch mit Semikolon getrennt in eine Zeile schreiben. Sie werden dann nacheinander ausgeführt. Man hätte hier also auch schreiben können:

```
> x=2; y=3; x+y
[1] 5
```

`<-` ist als Zuweisungsoperator sehr verbreitet. In diesem Skript werden i.d.R. mit Gleichheitszeichen (`=`) Zuweisungen vorgenommen.

Funktionsargumente

In Abschnitt 1.2.2 haben wir schon erste Funktionen in *R* kennengelernt. Funktionen spielen in *R* eine herausragende Bedeutung, weil nahezu alle Anweisungen in Form von Funktionen angegeben werden.

Schauen wir uns einmal die allgemeine Syntax der Berechnung des Logarithmus an:

```
log(x, base = exp(1))
```

Diese Syntax kann man in der Hilfe von R nachschlagen, s. Abschnitt 1.2.4 unten.

Die `log( )`-Funktion besitzt zwei Argumente, `x` und `base`. Für den Wert `x` soll der Logarithmus zur Basis `base` berechnet werden.

```
> log(8,2)
[1] 3
```

Der Logarithmus von 8 zur Basis 2 ist 3. Alternativ kann man auch schreiben

```
> log(x=8,base=2)
[1] 3
```

Man *kann* die Bezeichner für die Argumente also explizit angeben. Man muss es aber nicht, wenn man den Quelltext einfacher halten möchte.

```
> log(base=2,x=8)
[1] 3
```

Außerdem kann man unter Verwendung der Argumentbezeichner die Reihenfolge der Argumente auch vertauschen. Das ist bei vielen anderen Programmiersprachen nicht erlaubt. Natürlich liefert aber

```
> log(2,8)
[1] 0.3333333
```

nicht das gleiche wie `log(8,2)`, weil R hier davon ausgeht, dass der Logarithmus von 2 zur Basis 8 zu berechnen ist. Die Angabe der Argumentbezeichner kann hilfreich sein, um sicher zu stellen, dass man bei Funktionen mit mehreren Argumenten diese richtig zuordnet. Diese Argumentbezeichner lassen sich auch abkürzen:

```
> log(b=2,x=8)
[1] 3
```

`b` passt hier nur zum Argument `base` und wird von R richtig interpretiert. Auf diese Weise kann man Schreibweisen ggf. verkürzen. Falls keine oder keine eindeutige Zuordung zu einem Argument möglich ist, dann erhält man eine Fehlermeldung.

```
> log(8,basis=2)
Error in log(8, basis = 2) : unused argument (basis = 2)
```

Das zweite Argument `base` ist ein **optionales Argument**. Das erkennt man an der Schreibweise `base = exp(1)`. Wenn das zweite Argument, `base`, nicht angegeben wird, dann wird der **Vorgabewert**, hier `exp(1)`$= e^1$, verwendet.

```
> log(8)
[1] 2.079442
```

Hier wird das Argument `base` auf den Standardwert `exp(1)` gesetzt, d.h. es wird der Logarithmus von 8 zur Basis $e = \exp(1)$ berechnet. Optionale Parameter erlauben eine große Flexibilität von Funktionen, ohne ihre Anwendung unnötig zu verkomplizieren. Die Logarithmus-Funktion wird z.B. in den meisten Fällen zur Basis e verwendet. Falls doch einmal ein Logarithmus zu einer anderen Basis berechnet werden muss, dann und nur dann verwendet man den optionalen Parameter `base`.

1.2.3 Sichern und Löschen von Objekten

Durch Eingabe von

```
> ls()
```

(*list*) erhalten wir als Ausgabe die Namen aller Objekte, die wir durch Zuweisungen erzeugt haben. In unserem Fall erhalten wir:

```
[1] "pi" "x"  "y"
```

Nehmen wir eine weitere Zuweisung vor, z.B.

```
> z=4
> ls()
```

liefert die `ls( )`-Anweisung

```
[1] "pi" "x"  "y"  "z"
```

Nehmen wir nun folgende Zuweisung mit anschließender Abfrage vor:

```
> v=c(1,2,3)
> v
```

so erhalten wir

```
[1] 1 2 3
```

Das Objekt `v` enthält mehrere Werte, nämlich 1, 2 und 3. Der `ls()`-Befehl liefert jetzt

```
[1] "pi" "v"  "x"  "y"  "z"
```

Wir sehen, dass wir anhand der Objektnamen keine Unterscheidung zwischen den *Objekttypen* vornehmen können. So ist nicht ersichtlich, dass z. B. unter `x` und `y` jeweils einen Wert und unter `v` drei Werte gespeichert sind. Übrigens werden wir später `x` und `y` als **Skalare** und `z` als **Vektor** bezeichnen. Vektoren werden in Abschnitt 2.2, S.30ff, näher besprochen.

Angenommen, wir möchten nun *R* beenden. Wir klicken auf das *Kreuz* rechts oben im *Command*- oder *R*-Fenster. Daraufhin folgt die Abfrage *„Workspace sichern?“*. Wird diese Frage mit *„Nein“* beantwortet, so gehen alle Objekte verloren, die während der aktuellen Sitzung erzeugt wurden. Beim nächsten Start von *R* sind nur noch die Objekte der letzten vorgenommenen Sicherung vorhanden.

Wir beenden *R*, indem wir auf das *Kreuz* klicken oder in der Menüleiste *„Datei“* und anschließend *„Beenden“* wählen. Der Workspace wird gesichert, indem die Abfrage mit *„Ja“* beantwortet wird. Wir starten *R* erneut. Der `ls()`-Befehl liefert:

```
> ls()
[1] "pi" "v"  "x"  "y"  "z"
```

Es sind noch alle Objekte vorhanden. Wir nehmen eine neue Zuweisung vor.

```
> w=3.4
```

Wir fragen den Wert von `w` ab und listen alle vorhandenen Objekte auf.

```
> w
[1] 3.4
> ls()
[1] "pi" "v"  "w"  "x"  "y"  "z"
```

Wir verlassen *R* dieses Mal, ohne den Workspace zu sichern, und starten *R* erneut. Nun erhalten wir

```
> w
Fehler: Objekt 'w' nicht gefunden
> ls()
[1] "pi" "v"  "x"  "y"  "z"
```

Da `w` nach Beendigung der letzten Sitzung mit dem Workspace nicht gespeichert wurde, ist *R* das Objekt jetzt nicht mehr bekannt.

Das Ganze funktioniert in gewissem Sinne auch umgekehrt. Angenommen, wir möchten das Objekt `y` entfernen. Dazu verwenden wir den `rm`( )-Befehl (*remove*). Zur Entfernung von `y` geben wir

```
> rm(y)
```

ein. Listen wir alle Objekte, so erhalten wir jetzt:

```
> ls()
[1] "pi" "v"  "x"  "z"
```

Das Objekt `y` ist nicht mehr vorhanden. Wir verlassen nun *R* – wiederum ohne Sicherung – und starten es erneut. Was wird nun der `ls()`-Befehl ausgeben? Wir erhalten:

```
[1] "pi" "v"  "x"  "y"  "z"
```

Das ist der Stand der letzten Sicherung. Wir entfernen jetzt `pi`.

```
> rm(pi)
> ls()
[1] "v" "x" "y" "z"
> pi
[1] 3.141593
```

Die von uns erzeugte Variable bzw. das Objekt `pi` wurde entfernt und die ursprüngliche Definition von `pi` als Kreiszahl π erfreulicherweise wieder hergestellt.

Um alle Objekte zu entfernen, geben wir die Anweisung

```
> rm(list=ls())
```

ein. Dadurch werden alle Objekte der aktuellen Sitzung gelöscht. Wir verlassen nun *R* und nehmen eine Sicherung des *Workspace* vor, indem die Abfrage mit „Ja“ beantwortet wird. Wir starten *R* erneut und geben den `ls()`-Befehl ein.

```
> ls()
character(0)
```

Dies bedeutet, dass keine Objekte vorhanden sind.

1.2.4 Hilfesystem

Ein unverzichtbarer und integraler Bestandteil von *R* ist das Hilfesystem. Dieses liefert umfangreiche und miteinander verlinkte Hilfedateien zu allen Funktionen, die *R* enthält. Dies sei an einem Beispiel kurz illustriert.

Wir weisen zunächst einem Datenvektor `x` die Zahlen 3, 2, 5, 6 und 4 zu.

```
> x=c(3,2,5,6,4)
```

Die Summe dieser 5 Zahlen berechnet sich über die Funktion `sum( )`, angewendet auf `x`:

```
> sum(x)
[1] 20
```

Die Summe ist 20. Die Hilfedatei zum Befehl `sum( )` wird über

```
> ?sum
```

aufgerufen. Alternativ kann man auch `help(sum)` eingeben. Es öffnet sich daraufhin ein neues Fenster mit entsprechendem Hilfetext (vgl. Abb 1.2.1).

Der Aufbau der Hilfereferenz folgt bei den meisten Funktionen bzw. Anweisungen dem gleichen Grundmuster. Dieses umfasst die Abschnitte

Description, Usage, Arguments, Details, Value, References.

Nach einer allgemeinen inhaltlichen Beschreibung (*Description*) folgt die Syntax (*Usage*) für die Anwendung der Funktion. Hieraus geht schon meist hervor, welche (optionalen) Argumente der Befehl zulässt und welche Voreinstellungen evtl. für bestimmte Argumente schon getroffen wurden. Diese können aber meistens noch detaillierter im Abschnitt *Arguments* nachgelesen werden. In unserem Fall verlangt der Befehl zunächst die Eingabe eines Datenvektors. Danach folgt das Argument `na.rm`. Das `na` steht für fehlende Werte (*not available*), das `rm` für Entfernen (*remove*). Für dieses Argument wurde bereits eine Voreinstellung getroffen, nämlich `FALSE`. Dies bedeutet, dass im Falle von fehlenden Werten innerhalb des Datenvektors, diese nicht (`FALSE`) entfernt werden. Angenommen, wir haben 5 Personen zu ihrem Gewicht befragt. Eine Person hat dabei die Antwort verweigert. Wir geben nun die einzelnen Werte in *R* ein:

```
> Gewicht=c(72, 91, NA, 66, 53)
```

Fehlende Werte werden häufig mit `NA` (*Not Available*) gekennzeichnet. Berechnen wir nun die Summe über

```
> sum(Gewicht)
```

so erhalten wir als Ergebnis

```
[1] NA
```

Die Rechenoperation kann folglich unter Einschluss des fehlenden Wertes nicht sinnvoll durchgeführt werden.

Abbildung 1.2.1: *R*-Hilfe zur *R*-Funktion `sum`

sum {base} R Documentation

Sum of Vector Elements

Description

sum returns the sum of all the values present in its arguments.

Usage

```
sum(..., na.rm = FALSE)
```

Arguments

...
numeric or complex or logical vectors.

na.rm
logical. Should missing values (including NaN) be removed?

Details

This is a generic function: methods can be defined for it directly or via the Summary group generic. For this to work properly, the arguments ... should be unnamed, and dispatch is on the first argument.

If na.rm is FALSE an NA or NaN value in any of the arguments will cause a value of NA or NaN to be returned, otherwise NA and NaN values are ignored.

Logical true values are regarded as one, false values as zero. For historical reasons, NULL is accepted and treated as if it were integer(0).

Loss of accuracy can occur when summing values of different signs: this can even occur for sufficiently long integer inputs if the partial sums would cause integer overflow. Where possible extended-precision accumulators are used, but this is platform-dependent.

Value

The sum. If all of ... are of type integer or logical, then the sum is integer, and in that case the result will be NA (with a warning) if integer overflow occurs. Otherwise it is a length-one numeric or complex vector.

NB: the sum of an empty set is zero, by definition.

S4 methods

This is part of the S4 Summary group generic. Methods for it must use the signature x, ..., na.rm.

'plotmath' for the use of sum in plot annotation.

References

Häufig erscheint es dann sinnvoll, die fehlenden Werte zu ignorieren (zu „entfernen“) und die Berechnung nur mit den verwertbaren Daten durchzuführen. Dies bewerkstelligt man über

```
> sum(Gewicht,na.rm=T)
```

Der logische Wert `T` steht abkürzend für den Ausdruck `TRUE` („wahr“). Fehlende Werte sollen also entfernt (genauer: ignoriert) werden. Folgerichtig lautet jetzt das Ergebnis:

```
[1] 282
```

Man beachte, dass der Datenvektor `Gewicht` unverändert bleibt. Das „Entfernen“ bezieht sich nur temporär auf die Rechenoperation, bewirkt aber keinen Eingriff in den Datenvektor selbst.

Es bleibt zu vermerken, dass nicht alle Argumente (in diesem Fall zwei) bei der Eingabe spezifiziert werden müssen. Da in diesem Fall das zweite Argument die Voreinstellung `FALSE`

besitzt, wird diese automatisch bei jeder Befehlsanwendung übernommen, sofern sie nicht anderweitig spezifiziert wurde. Dies kann dann, wie in unserem Fall passiert, bei Vorliegen fehlender Werte zu einem nicht verwertbaren Rechenergebnis führen. Im Abschnitt *Details* wird häufig etwas detaillierter auf den genauen Berechnungsvorgang eingegangen. Im Falle der Summierungsfunktion gibt es da natürlich nicht viel zu vermerken. Ebenso werden weitere befehlsspezifische Eigenheiten aufgeführt.

Im Abschnitt *Value* wird dann beschrieben, was als Ergebnis ausgegeben wird. Im Falle des Befehls `sum( )` ist dies natürlich die Summe von Zahlen, sofern es sich um einen numerischen Datenvektor handelt. Häufig besitzen *R*-Funktionen auch mehrere Rückgabeergebnisse. Diese werden dann in einer *Liste* zusammengefasst, s. dazu auch Abschnitt 2.4, S. 42ff.

Gelegentlich ergibt sich das praktische Problem, dass man nicht weiß, ob es zur Berechnung einer speziellen Statistik schon eine vorgefertigte Funktion in *R* gibt oder nicht. Eine Möglichkeit, die *R* bietet, ist die Funktion `help.search( )`, bei der nach einem bestimmten Begriff gesucht werden kann. Taucht dieser Begriff als Funktionsname (auch ähnliche Namen werden akzeptiert) oder im Zusammenhang einer mehr oder weniger verwandten Funktion auf, so wird diese in einer Ergebnisliste aufgeführt. Bei Eingabe von

```
> help.search("median")
```

werden im Ergebnisfenster Befehle aufgeführt, die mit dem Schlüsselwort *median* in Zusammenhang stehen. Es sei an dieser Stelle erwähnt, dass Anführungszeichen stets hochgestellt sein müssen, wie in obigem Beispiel aufgeführt. So führt beispielsweise die Eingabe

```
> help.search(,,median'')
```

zu einer Fehlermeldung.

> Anführungszeichen innerhalb der R-Syntax müssen stets hochgestellt sein, z.B. `help.search("median")`.

Wir möchten nun eine Boxplot-Darstellung für unseren Datenvektor *Gewicht* erzeugen. Es wäre naheliegend davon auszugehen, dass die entsprechende *R*-Funktion `boxplot( )` lautet. Tatsächlich liefert die Eingabe

```
> ?boxplot
```

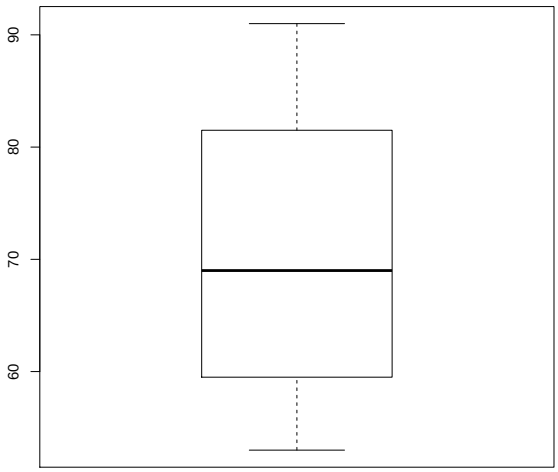
einen Hilfetext zum gewünschten Befehl. Geben wir nun

```
> boxplot(Gewicht)
```

ein, so erhalten wir eine Boxplot-Darstellung unseres Datenvektors, s. Abb. 1.2.2. Dabei haben wir alle Voreinstellungen übernommen. Aus dem Abschnitt *Details* des Hilfetextes geht außerdem hervor, dass fehlende Werte automatisch ignoriert werden.

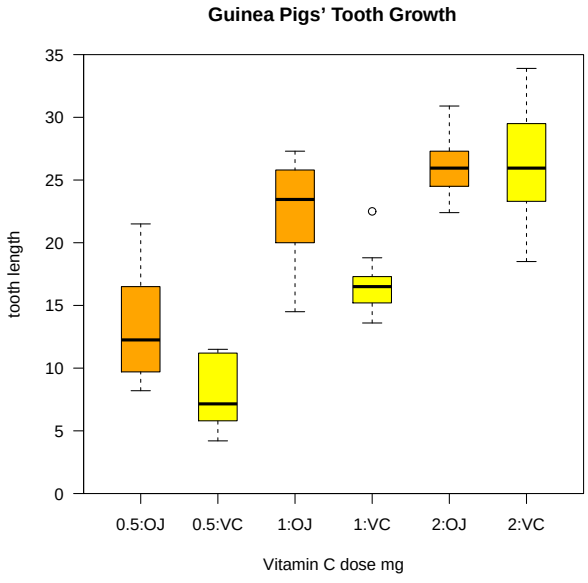
Am unteren Ende jeder Hilfedatei sind häufig Beispiele aufgeführt. So auch im Falle des `boxplot( )`-Befehls. Diese sind häufig Anregungen, in welcher Weise der entsprechende Befehl genutzt bzw. variiert werden kann. Es empfiehlt sich, die einzelnen Beispiele mit der Maus

Abbildung 1.2.2: Boxplot



zu markieren und ins *Command*-Fenster hinein zu kopieren. Der Boxplot in Abbildung 1.2.3 wurde durch ein solches Vorgehen erzeugt.

Abbildung 1.2.3: Boxplot



Beim Aufrufen einer Hilfedatei von *R* wird diese standardmäßig in einem Internetbrowser dargestellt. Mit dem nachstehenden Kommando wird die Hilfedatei als zusätzliches Fenster in der *R*-Entwicklungsumgebung dargestellt.

```
> options(help_type="text")
```

Eine weitere Möglichkeit, Hilfe zu *R*-Funktionen oder *R*-Themen zu erhalten, ist natürlich auch das themenspezifische Recherchieren in der umfangreichen Online-Hilfe (*Documentati-*

on) auf der *R*-Homepage bzw. in den Handbüchern von *R*, auf die man auch über die Menüleiste (*Hilfe*) Zugriff hat. Bei komplexeren Themen lohnt es sich auch häufig, ein Schlagwort (in Englisch!) zusammen mit „in R“ , z.B. „Boxplot in R“, in einer Suchmaschine einzugeben. Meistens wird man rasch fündig hinsichtlich vorhandener Befehle, Variationsmöglichkeiten und häufiger Fragen und Probleme zu dem Thema.

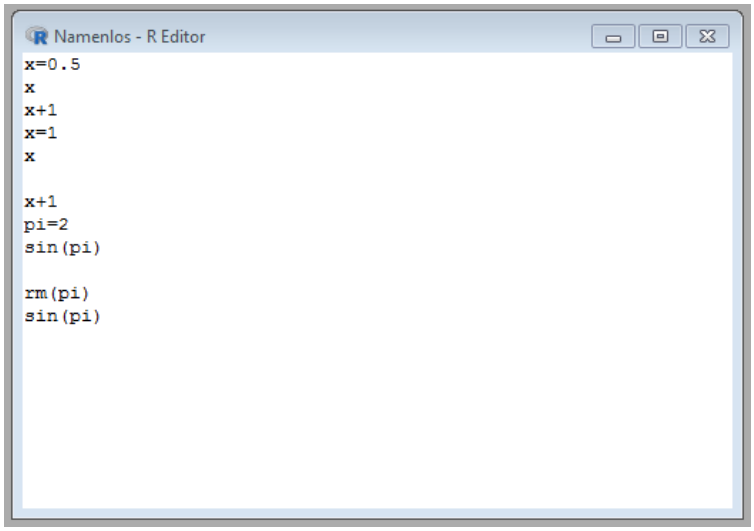
1.2.5 *R*-Skripte mit *RGui*

Bei umfangreicheren Berechnungen mit *R* ist es sinnvoll, die *R*-Programmzeilen in eine eigene *R*-Skriptdatei zu schreiben. Ein *R*-Skript ist nichts weiter als eine Textdatei, die eine zeilenweise Auflistung von *R*-Anweisungen enthält und üblicherweise die Endung *.R* hat. Um ein *R*-Skript in der mit *R* mitgelieferten Benutzeroberfläche *RGui* zu erstellen, gehen wir über das Menü

Datei/Neues Skript

In die Skriptdatei können dann eine Reihe von *R*-Anweisungen eingetragen werden. Das Ergebnis könnte wie in Abbildung 1.2.4 aussehen. Das sind einige von den Anweisungen, die wir in den Abschnitten 1.2.2 bis 1.2.3 kennengelernt haben.

Abbildung 1.2.4: *R*-Skript in *RGui*



Mit

Strg+R

kann man die aktuelle Skript-Zeile oder mehrere im Skript markierte Zeilen ausführen lassen. Das Skript kann man mittels *Datei/Speichern unter ...* speichern (z.B. unter dem Namen *erstesSkript.R*) und dann bei aktiviertem Konsolen-Fenster über

Datei/Lese R Code ein ...

ausführen. Dabei werden *alle* Anweisungen, die im Skript stehen, ausgeführt. Alternativ kann man die `source( )`-Funktion verwenden.

```
> source("C:/RData/erstesSkript.R")
```

Möchte man *R*-Code kommentieren, dann kann man das mit dem `#`-Symbol machen. Der Text, der in der gleichen Zeile hinter dem #-Symbol steht, wird als Kommentar interpretiert.

```
> 1+1 # eine sehr einfache Rechnung
[1] 2
```

Bei größeren Projekten fallen Daten an, es werden Abbildungen erstellt und die *R*-Hilfe wird aufgerufen. Unter Umständen kann es dann auch sinnvoll sein, Programme auf mehrere Dateien aufzuteilen.

1.3 Ergänzungen

1.3.1 Erweiterung von *R* mit Paketen

Wenn wir zum Befehl `mean( )` den Hilfetext aufrufen

```
> ?mean
```

steht am linken oberen Ende des Textes *mean* {*base*}. Rufen wir den Hilfetext zum Befehl `hist()` auf, steht an entsprechender Stelle {*graphics*}. Die in den geschweiften Klammern angegebenen Namen sind nichts anderes als die Namen der *Packages*, zu denen diese Befehle gehören. *Packages* sind Befehlspakete zu einem bestimmten Themengebiet. So enthält das Paket *graphics* wichtige Befehle zur Erstellung von Grafiken. Das Paket *base*, das Basispaket also, enthält sehr grundlegende Befehle wie z.B. `mean()`, `matrix()` oder `table()`, die für statistische Analysen quasi unverzichtbar sind. Es gibt Hunderte von Paketen, die mittlerweile für *R* zur Verfügung stehen. Diese können von der *R*-Homepage bezogen werden. Manche Pakete sind auf ein spezielles statistisches Problem zugeschnitten und enthalten nur wenige Befehle, andere Pakete enthalten hunderte von Befehlen zu einem bestimmten Themenbereich. Beim Starten von *R* werden standardmäßig nur einige wenige Pakete *geladen*, d.h. es können auch dann nur die Funktionen aus diesen Paketen benutzt werden. Befehle in Paketen, die nicht geladen sind, können nicht verwendet werden.

Welche Pakete augenblicklich geladen sind, kann über die Eingabe von

```
> search()
```

festgestellt werden. Dies führt beispielsweise zur Ausgabe

```
[1] ".GlobalEnv"        "package:stats"     "package:graphics"
[4] "package:grDevices" "package:utils"     "package:datasets"
[7] "package:methods"   "Autoloads"         "package:base"
```

Wir sehen, dass im Beispiel acht Pakete geladen wurden, darunter z.B. auch die Pakete *base* und *graphics*. Die erste Komponente dieser Ausgabe *.GlobalEnv* steht für kein Paket, sondern

für den *Workspace*. Im *Workspace* sind die Objekte enthalten, die der Benutzer generiert hat, z.B. Vektoren, Matrizen oder eigene Funktionen. Zur Konstruktion von Funktionen werden wir später im Kapitel 5 noch kommen. Die angegebenen Pakete sind diejenigen, die automatisch beim Start geladen werden. Jedoch sind dies längst nicht alle Pakete, die beim Herunterladen von *R* mitgeliefert wurden. Welche Pakete weiter unmittelbar zur Verfügung stehen, lässt sich mit Hilfe der Eingabe

```
> library()
```

feststellen. Es öffnet sich ein Fenster mit weiteren Paketnamen und kurzen inhaltlichen Beschreibungen dazu. Alternativ kann man auch einen Blick in den Programmordner von *R* werfen. Dort findet sich ein Ordner *library*, in dem alle Pakete enthalten sind. Welche Funktionen, statistische und nichtstatistische Verfahren mit den einzelnen Paketen genutzt werden können, lässt sich über den Aufruf der Paket-Dokumentation feststellen. Im Falle des Paketes *base* erfolgt dies über

```
> library(help=base)
```

im Falle des Pakets *graphics* entsprechend über

```
> library(help=graphics)
```

Sollen aus dem *library*-Ordner enthaltene Pakete genutzt werden, welche nicht automatisch beim Starten von *R* geladen werden, müssen diese mit der `library( )`-Funktion durch Angabe des betreffenden Paketnamens geladen werden. Beispielsweise wird das Paket *MASS* über

```
> library(MASS)
```

geladen. Die Pakete im *library*-Ordner sind jedoch längst nicht alle für *R* verfügbaren Pakete. Deren Anzahl ist viel größer. Wenn Sie einen direkten Internetanschluss haben, können Sie in der Menüleiste *Pakete* und danach

Installiere Paket(e)

auswählen. Sie werden zunächst aufgefordert, einen der weltweit für *R* zur Verfügung gestellten Server auszuwählen. Wählen Sie den zu ihrem Standort nächsten Server aus, z.B. *Germany (Münster)*. Nach Auswahl eines Servers erhalten Sie eine Liste mit zahlreichen Paketen, die Sie sich durch schlichtes Anklicken einfach herunterladen können. Das Paket wird automatisch in den *library*-Ordner des Programmordners von *R* gestellt. Es ist denkbar, dass die in dem Ordner enthaltenen Befehle wiederum Befehle benötigen, die nicht automatisch in *R* enthalten sind. Dann werden davon betroffene weitere Pakete automatisch mit heruntergeladen. Wenn Sie also nur ein Paket herunterladen wollen, heißt dies noch lange nicht, dass es bei diesem einen Paket auch bleibt. Dies ist übrigens auch der Vorteil einer Online-Paket-Installation. Sie müssen nicht mitüberlegen, welche weiteren Pakete Sie möglicherweise noch benötigen.

Angenommen, wir möchten die Funktion `EDA( )` (*Exploratory Data Analysis*) aus dem Packet *BSDA* (*Basic Statistics and Data Analysis*) verwenden. Informationen zur Funktion und Syntax erhalten wir durch

```
> help("EDA",package="BSDA")
```

Die Funktion generiert zu einem Datenvektor verschiedene Abbildungen wie ein Histogramm und einen Boxplot. Zunächst muss das Paket installiert werden. Das kann man wie oben beschrieben oder mit der Funktion `install.packages( )` machen.

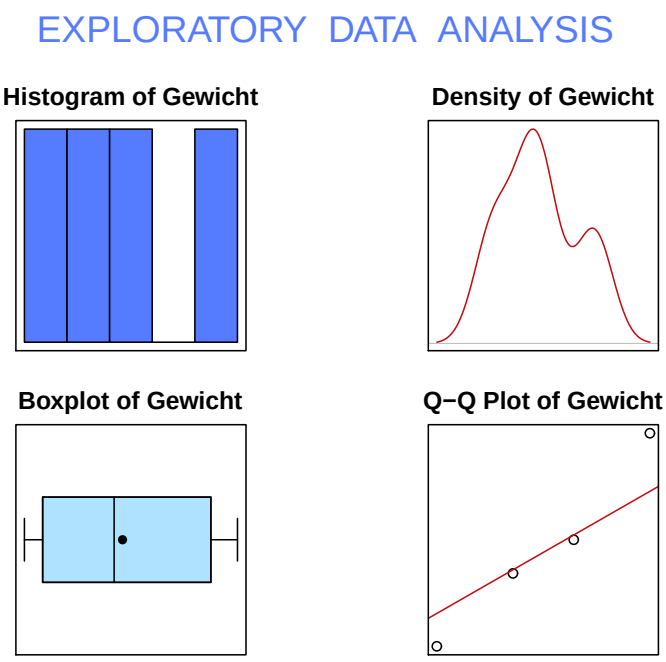
```
> install.packages("BSDA")
```

Anschließend müssen wir das Paket laden und können dann die Funktion anwenden.

```
> library(BSDA)
> EDA(Gewicht)
```

Das Ergebnis ist in Abbildung 1.3.1 dargestellt.

Abbildung 1.3.1: Anwendung der EDA( )-Funktion



Falls man auf dem Arbeitsrechner keinen Internetanschluss hat, kann man sich die Pakete über die *R*-Homepage herunterladen, vgl. Abschnitt 1.1.1. Auf der Seite, auf der man auch *R* herunterladen kann, findet man einen Link *Packages*, der zu allen verfügbaren Paketen führt. Nach dem Herunterladen des Pakets muss dieses in den *library*-Ordner von *R* kopiert werden. Möglicherweise ist zuvor eine Dekomprimierung der heruntergeladenen Datei nötig. Aus der Paketdokumentation geht dann in der Regel noch hervor, welche weiteren Pakete möglicherweise noch benötigt werden. Wird es versäumt, alle benötigten Pakete herunterzuladen, kann es bei gewissen Befehlsaufrufen zu Fehlermeldungen kommen.

R ist über die *Paket-Architektur* sehr ausbaufähig ist. Für sehr viele statistische Problemstellungen existieren ein oder mehrere Pakete. Zu beachten ist, dass die inhaltliche Prüfung der Korrektheit der implementierten Software weitestgehend in der Verantwortung der Autoren liegt und damit ohne eine persönliche Prüfung nicht garantiert werden kann, dass

das angewandte statistische Verfahren anerkannt oder die Implementierung korrekt ist. Ggf. sollte man sich über die Autoren und über zugehörige Fachartikel informieren und die Implementierung mit Beispielrechnungen oder Simulationen prüfen.

Zusammenfassend halten wir fest, dass die Funktionalität von *R* mit Paketen erweitert werden kann, die zu bestimmten Themengebieten Funktionen und Daten bereitstellen. Einige Pakete werden automatisch schon beim Start von *R* geladen, einige können während einer Sitzung aus dem *library*-Ordner hinzu geladen werden und sehr viele weitere Pakete sind über die *R*-Homepage verfügbar. Die letzteren können von dort heruntergeladen oder auf bequeme Art online installiert werden.

1.3.2 Oberflächen zu *R*

RStudio

Die mit *R* mitgelieferte Benutzeroberfläche *RGui* erlaubt es, *R*-Funktionen anzuwenden und *R*-Skripte zu erstellen und auszuführen. Es gibt aber komfortablere Oberflächen. Z.B. wird jeder, der schon einmal programmiert hat, eine Syntaxhervorhebung im Programm-Code zu schätzen wissen. Häufig unterstützen Editoren auch dabei, dafür zu sorgen, dass die Anzahl der öffnenden Klammern der Anzahl der schließenden Klammern entspricht. So etwas wird z. B. von *RGui* nicht bereitgestellt.

Eine sehr populäre Benutzerumgebung ist *RStudio*. Die zugehörige Internetseite ist

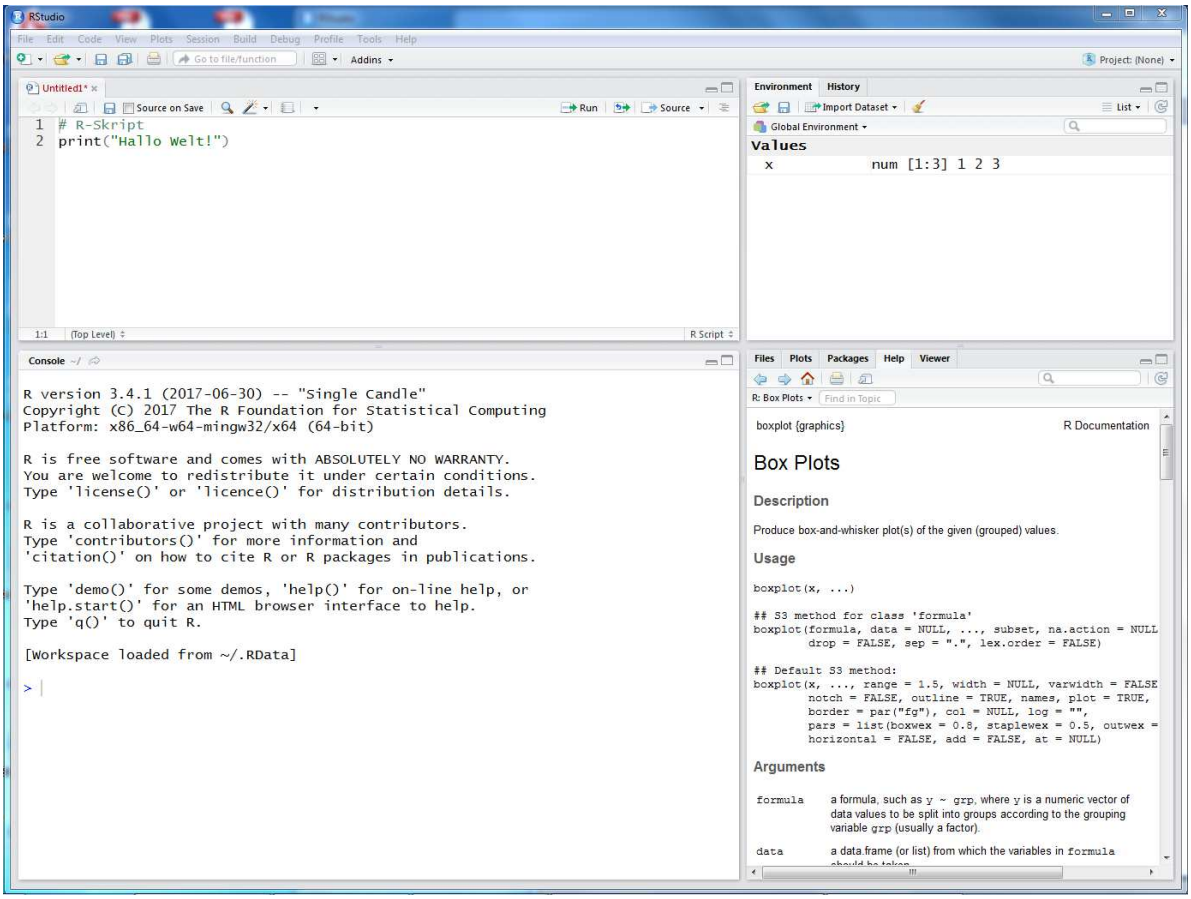
www.rstudio.com

Dort findet sich ein Download-Link zu einer aktuellen Version von *RStudio*. Einzelnutzerlizenzen werden in der *Open Source Edition* als *AGPLv3* (*Affero General Public License v3*) kostenlos zur Verfügung gestellt.

Bei der Installation folgen wir den Installationsanweisungen. Abbildung 1.3.2 zeigt die Oberfläche von *RStudio*. Typisch ist die Einteilung in vier Fenster. Unten links findet sich ein Konsolenfenster, wie es auch aus *RGui* bekannt ist. Hier können alle *R*-Berechnungen auch schrittweise so durchgeführt werden, wie es bisher besprochen wurde. Oben links ist eine *R*-Skript-Datei eingeblendet. Diese kann man als ganzes oder abschnittsweise ausführen lassen. Nützlich ist hier insbesondere die Syntaxhervorhebung, die das Lesen von *R*-Skripten erleichtert. Wenn man mit mehreren *R*-Dateien arbeitet und diese geöffnet hat, kann man über einen Reiter auf die Dateien zugreifen. Im unteren rechten Fenster kann man sich z.B. Hilfetexte (*Help*) oder grafische Darstellungen (*Plots*) anzeigen lassen. Über den zugehörigen Reiter wählt man die entsprechende Anzeigeoption. Im Fenster oben rechts kann man sich die aktuellen Werte von Variablen anzeigen lassen. Über das Menü kann man Einstellungen vornehmen, Daten importieren oder exportieren oder auch Pakete installieren.

Weitere populäre Editoren sind z.B. *Tinn-R*, *Emacs* oder *Notepad++* mit *NppToR*.

Abbildung 1.3.2: Oberfläche von *RStudio*



R Commander

Der *R Commander* bietet eine grafische Oberfläche, die es vor allem erleichtern soll, statistische Analysen mit *R* durchzuführen, ohne allzu große Programmierkenntnisse zu besitzen. Wie wir in den nachfolgenden Kapiteln sehen werden, muss man in *R* Anweisungen in Form von Funktionen aufrufen, um bspw. Grafiken zu erstellen und Tests durchzuführen. Mit der *R Commander*-Oberfläche kann man viele einfache Grafiken und Analysen dadurch erstellen, indem man geeignete Menüpunkte auswählt und gewisse Einstellungen in Eingabefenstern vornimmt (durch „Klicken“). Das entspricht der Vorgehensweise in vielen kommerziellen Statistik-Programmen wie z.B. *SPSS* oder auch *Excel*.

Zur Installation von *R Commander* muss man das Paket *Rcmdr* installieren

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

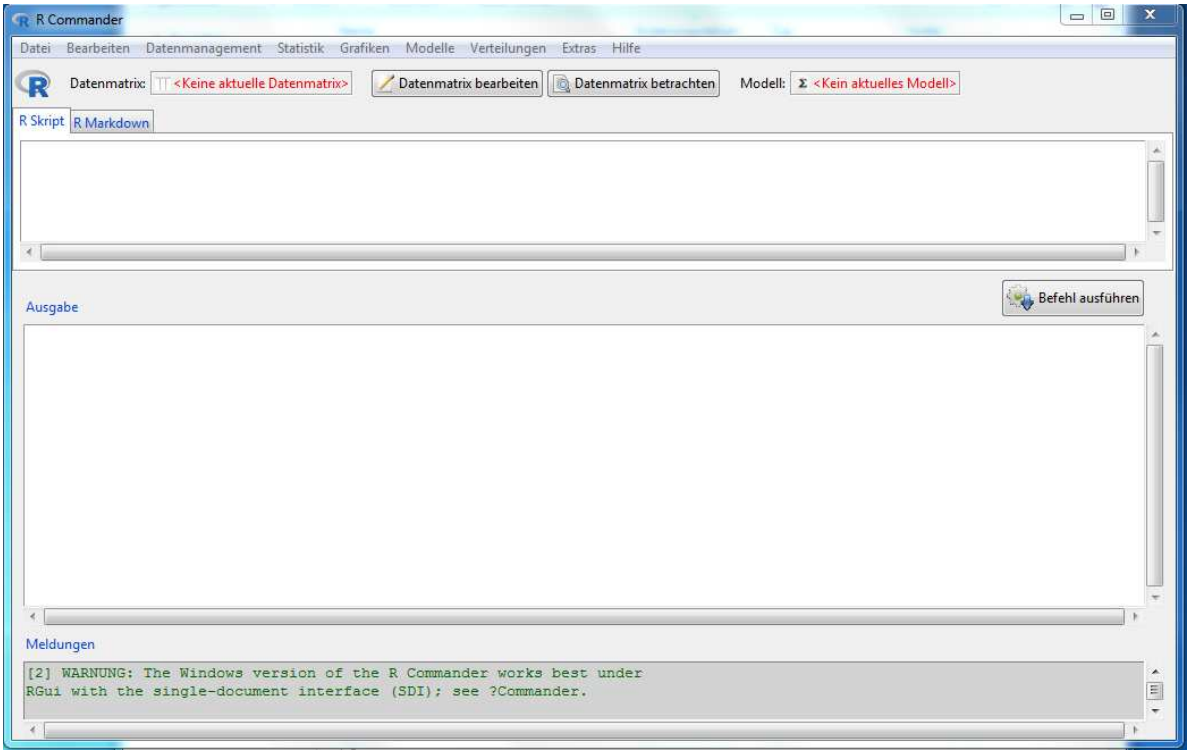
Dann braucht man das Paket nur noch aufzurufen:

```
> library("Rcmdr")
```

Das kann u.U. eine Installation von weiteren Paketen erfordern und kann beim ersten Mal

einige Minuten dauern. Das Aufrufen des Pakets startet dann die *R Commander*-Oberfläche. Abbildung 1.3.3 zeigt die Oberfläche nach dem Start.

Abbildung 1.3.3: Oberfläche vom *R Commander*



1.3.3 *R* als Programmiersprache

R ist eine Programmiersprache, die primär für die Anwendung im Bereich Statistik ausgelegt ist. Es gibt zahlreiche andere, kommerzielle Statistikprogramme wie *SPSS*, *SAS* oder *Stata*. Diese haben oft eine komfortablere grafische Oberfläche, die es einfacher macht, auch ohne Programmierkenntnisse statistische Analysen durchzuführen. Einen Schritt in diese Richtung macht *R* mit der *R Commander*-Oberfläche. Häufig umfassen die kommerziellen Statistikprogramme auch Programmiersprachen, die die Erstellung von Skripten ermöglichen. Das ist insbesondere bei *SAS* und *Stata* der Fall. *R* bietet hier aber deutlich vielfältigere Programmiermöglichkeiten und es ist auch ohne Probleme möglich, *R* auf nichtstatistische Fragestellungen anzuwenden. Außerdem ist *R* natürlich kostenlos.

Auf der anderen Seiten gibt es Programmiersprachen wie *C*, *C++*, *Java* etc. zur allgemeineren Anwendungsprogrammierung, die aber nicht ihren Fokus im Bereich Statistik haben. Der Vorteil von *R* ist hier vor allem die sehr umfangreiche Bibiliothek von statistischen Funktionen, die in anderen Programmiersprachen nur eingeschränkt zur Verfügung stehen. Weitere Vorteile von *R* sind das relativ einfache Erstellen von insbesondere im statistischen Bereich relevanten Abbildungen und die native Unterstützung von Vektoren und Matrizen

als Datenstrukturen – auch das ist sehr nützlich für statistische Anwendungen. Für Programmiereinsteiger ist es von Vorteil, dass keine genaue Typspezifikation in *R* für Variablen notwendig ist. Das vereinfacht den Quellcode deutlich, kann aber auch zu nicht leicht auffindbaren Fehlern führen. Ein großer Nachteil von *R* gegenüber Programmiersprachen wie bspw. *C* ist letztendlich die vergleichsweise niedrige Geschwindigkeit von *R*. Um die Geschwindigkeit von *R*-Programmen zu verbessern, ist es möglich, ggf. Quelltext aus *C* in *R* zu importieren.

1.3.4 Literaturhinweise

Möchte man sich neben dieser Einführung weitergehend mit *R* vertraut machen, gibt es verschiedene Möglichkeiten: Interessiert man sich für eine spezielle Thematik, kann man Informationen dazu einfach im Internet suchen. Von einer Reihe von Autoren findet man solche Informationen als kurze Blog-Beiträge, umfangreichere Einführungen oder Youtube-Videos. Einen Überblick über die Syntax und verschiedene Anwendungsmöglichkeiten von *R* erhält man bspw. über die englischsprachige Website *www.statmethods.net*, die man auch unter dem Schlagwort *Quick-R* findet. Möchte man sich lieber in ein Thema mit einem Buch einarbeiten, dann wird man feststellen, dass es zum Thema *R* sehr viel Literatur gibt. Zahlreiche Lehrbücher (insbesondere englischsprachige) sind bereits auf der *R* -Homepage aufgelistet. Im Folgenden soll nur eine sehr kleine Auswahl der zur Verfügung stehenden Literatur erwähnt werden.

Für erste Einführungen in *R* eignen sich zum Beispiel Wollschläger [2016] und Luhmann [2015]. Hier wird der Umgang mit *R* beschrieben und die wichtigsten Funktionen eingeführt. Außerdem wird vorgestellt, wie elementare und nicht mehr ganz so elementare statistische Verfahren mit *R* realisiert werden können.

Es gibt aber auch zahlreiche Bücher zu fortgeschrittenen und spezielleren statistischen Themen. In Wickham und Grolemund [2016] werden u.a. fortgeschrittene Methoden zur Erstellung von Grafiken mit *ggplot2* und Datentransformationen mit *dplyr* vorgestellt. In Fox [2008] findet man eine ausführlichere Besprechung der linearen Regressionsmodelle und verallgemeinerten linearen Regressionsmodelle. Anwendungsmöglichkeiten von *R* im Bereich der multivariaten Statistik findet man z.B. in Everitt und Hothorn [2011]. In Toomey [2014] werden u.a. Data Mining, Datenvisualisierung und Machine Learning besprochen. Über die Anwendung von Bayesscher Statistik mit *R* kann man sich in Albert [2009] oder Cowles [2013] informieren.

Liegt das Interesse mehr bei der Programmierung mit *R*, so kann man sich hier z.B. Ligges [2007] oder Grolemund [2014] näher anschauen. Der Fokus liegt hier im besseren Verständnis der *R*-Objekte und in dem Schreiben und der Fehlerkorrektur in *R*-Code. In Wickham [2015] wird die Struktur und Erstellung von *R*-Paketen näher besprochen. In diesen Büchern findet man auch Informationen, wie man *C*-Code in *R* implementiert, um so die Programmausführung zu beschleunigen.

2 Wichtige Datenstrukturen und deren Arithmetik

2.1 Datentypen

Die Daten in *R* werden in verschiedenen Datentypen gespeichert. Die meisten Daten werden mit Zahlen ausgedrückt und in einem numerischen Fließkommaformat (*double*) gespeichert.

```
> x=2
> typeof(x)
[1] "double"
```

Mit der Funktion `typeof( )` kann man sich den Typ der Daten anzeigen lassen. Mit numerischen Daten kann man die üblichen Berechnungen durchführen, die wir schon in Abschnitt 1.2.1, S.9, besprochen haben.

Ein weiterer wichtiger Datentyp ist die Zeichenkette. Hier werden Buchstaben bzw. Worte in einem Objekt gespeichert.

```
> a="a"
> typeof(a)
[1] "character"
```

Die Zeichenkette, die einer Variable zugewiesen wird, ist dabei in Anführungszeichen `" "` zu setzen. Der Datentyp selbst wird *character* genannt. Dieser Typ kann auch verwendet werden, um die verschiedenen Ausprägungen nominal skalierter Merkmalen zuzuweisen. Arithmetische Operationen machen hier dann natürlich keinen Sinn mehr.

```
> y="2"
> y
[1] "2"
> x+y
```

liefert die Fehlermeldung

```
Error in x + y : non-numeric argument to binary operator
```

Hier enthält `x` den numerischen Wert 2, während `y` ein String ist, dem der Wert `"2"` zugewiesen wurde.

Der obige Fall ist unproblematisch, da anhand der Anführungszeichen klar erkennbar ist, welche Variablen numerisch und welche nichtnumerisch sind, d.h. mit welchen gerechnet bzw. nicht gerechnet werden kann. Es kommt häufig vor (insbesondere bei fehlerhaftem Einlesen von Daten aus externen Dateien), dass Zahlen zwar ohne Anführungszeichen angezeigt werden, mit diesen aber dennoch nicht gerechnet werden kann, da *R* diese nicht als Zahlen, sondern als Zeichenketten interpretiert. Auf diesen Punkt wird im Abschnitt 2.7, S. 59ff, noch einmal eingegangen. Insofern ist es wichtig zu wissen, welche Daten von *R* als Zahlen interpretiert werden und welche nicht.

Weitere Typen sind die Wahrheitswerte `TRUE` und `FALSE`, die bei logischen Ausdrücken und Bedingungen auftreten können. Sie können auch durch `T` und `F` abgekürzt werden, sofern diese Bezeichnungen nicht schon an anderer Stelle verwendet werden.

```
> typeof(TRUE)
[1] "logical"
> typeof(T)
[1] "logical"
```

Im Abschnitt 1.2.1 wurde bereits erwähnt, dass in *R* auch komplexe Zahlen implementiert sind.

```
> typeof(1+2i)
[1] "complex"
```

2.2 Vektoren

Vektoren speichern mehrere Objekte des gleichen Datentyps unter einem Namen. Ein einzelnes Objekt bezeichnet man dann im Gegensatz zum Vektor auch als **Skalar**.

Konstruktion von Vektoren

Mit der Funktion `c( )` (*combine*) kann man mehrere Objekte des gleichen Datentyps zu einem Vektor *zusammensetzen*.

```
> v1=c(1,2,3)
> v1
[1] 1 2 3
> s1=0.8; s2=2.4
> v2=c(s1,s2)
> v2
[1]  0.8  2.4
```

Ein Vektor ist in diesem Beispiel sozusagen eine „Kombination“ mehrerer Zahlen. Bei der Ausgabe tauchen zuvor eingegebene Objektnamen nicht mehr auf. `v1` und `v2` sind jeweils Vektoren der Länge 3 bzw. 2, die mit Hilfe von drei bzw. zwei Zahlen konstruiert wurden.

Man kann auch Vektoren mit Skalaren oder Vektoren mit Vektoren zusammenfügen.

```
> v3=c(v2,4,5)
> v3
[1]  0.8  2.4 4.0  5.0
> v4=c(c(4,2,3),v3,c(3,8),s3,1,2,c(v2,v1,v3))
> v4
```

Wir erhalten:

```
 [1]  4.0  2.0  3.0  0.8  2.4  4.0  5.0  3.0  8.0 -3.4  1.0
[12]  2.0  0.8  2.4  1.0  2.0  3.0  0.8  2.4  4.0  5.0
```

Die Ausgabe des Vektors `v4` passte nicht mehr in eine Zeile. Hier sehen wir auch, welche Bedeutung die Zahl in eckigen Klammern in der Ausgabe hat. Der erste Eintrag von `v4` ist 4.0. Der 12-te Eintrag von `v4` passte nicht mehr in die 1. Zeile und wird in der 2. Zeile dargestellt. Dass es sich um den 12. Eintrag handelt, wird mittels `[12]` am Zeilenbeginn

vermerkt. Wie viele Einträge eines Vektors in einer Zeile angezeigt werden, hängt von der Länge der Zahlen und der Breite des *R*-Kommandofensters ab.

Die **Länge eines Vektors**, d.h. die Anzahl der Komponenten des Vektors, kann man sich mit der Funktion `length( )` angeben lassen. Einen Skalar kann man auch als Vektor der Länge 1 auffassen.

```
> length(v4)
[1] 20
> length(2.4)
[1] 1
```

Es lassen sich auch aus Zeichenketten oder anderen Datentypen Vektoren bilden.

```
> Geschlecht=c("m","w","m","w","m")
> Geschlecht
[1] "m" "w" "m" "w" "m"
> Name=c("Anton","Berta","Caesar","Dora","Emil")
> Name
[1] "Anton"  "Berta"  "Caesar" "Dora"   "Emil"
> length(Name)
[1] 5
> v2s=c("s1","s2")
> v2s
[1] "s1" "s2"
```

Die Länge eines Vektors von Zeichenketten ist erneut die Anzahl der dort kombinierten Zeichenketten und nicht etwa die Anzahl der Buchstaben. Man beachte, dass es wichtig ist, Zeichenketten mit hochgestellten Anführungszeichen zu kennzeichnen. `v2` und `v2s` unterscheiden sich bei ihrer Definition nur durch die Anführungszeichen. Sie liefern jedoch unterschiedliche Ergebnisse.

Setzt man Vektoren aus Objekten unterschiedlicher Typen zusammen, versucht *R* einen Typ zu finden, den man allen Objekten zuweisen kann und bildet dann ohne Fehler- oder Warnmeldung einen Vektor.

```
> c(1,"2")
[1] "1" "2"
>   c(3.4,TRUE,FALSE)
[1] 3.4 1.0 0.0
>   c(1,1i)
[1] 1+0i 0+1i
```

Eine Kombination aus Zahl und Zeichenkette wird in einen Vektor von Zeichenketten umgewandelt. Die Wahrheitswerte `TRUE` bzw. `FALSE` werden intern auch als 1 bzw. 0 aufgefasst.

```
> TRUE+FALSE*1i
[1] 1+0i
```

Erzeugung spezieller Vektoren

In der Praxis kommt es häufig vor, dass die Werte eines Vektors einem ganz bestimmten Grundschema folgen sollen. Beispielsweise folgen einzelne Werte aufeinander in fest vorgegebenen Abständen oder sie wiederholen sich ständig. Hier sind die Funktionen

`seq( )` und `rep( )` und der Doppelpunkt als Operator, also `:`,

von großem Nutzen, da diese eine manuelle Eingabe der Werte überflüssig machen bzw. stark verkürzen.

Mit der Funktion `seq( )` (*sequence*) kann man Sequenzen von Zahlen generieren.

```
> seq(from=2,to=30,by=4)
[1]  2  6 10 14 18 22 26 30
```

Beginnend bei dem Startwert *from* werden die Zahlen jeweils um die Schrittweite *by* erhöht, bis der Endwert *to* erreicht, aber noch nicht überschritten ist. Die etwas kürzere Syntax ist (vgl. dazu auch Abschnitt 1.2.2, S. 13, zu Funktionsargumenten):

```
> seq(2,30,4)
[1]  2  6 10 14 18 22 26 30
> seq(2,30,10)
[1]  2 12 22
```

Da *from*, *to* und *by* jeweils das erste, zweite bzw. dritte Argument der Funktion `seq( )` sind, kann man die entsprechenden Bezeichner auch weglassen. Im letzten Beispiel wird der Wert $22 + 10 = 32$ nicht mehr angegeben, da er den Endwert 30 überschreitet. Ein negativer Wert für *by* ist auch zulässig.

```
> seq(10,3,-2)
[1] 10  8  6  4
```

Mit der Option *length.out*, kurz *le*, anstelle von *by*, kann man einen Vektor generieren, dessen erstes Element *from* ist, dessen letztes Element *to* ist, der insgesamt *length.out* Elemente umfasst und dessen benachbarte Elemente den gleichen Abstand haben; sie sind damit äquidistant.

```
> seq(from=2,to=30,length.out=4)
[1]  2.00000 11.33333 20.66667 30.00000
> seq(2,30,le=4)
[1]  2.00000 11.33333 20.66667 30.00000
```

Mit dem Operator *Doppelpunkt* `:` lassen sich Sequenzen mit Schrittweite 1 (bzw. -1) erzeugen. Links des Operators steht der Startwert und rechts davon der maximale (bzw. minimale) Endwert. Passt der Endwert wiederum nicht in die Schrittfolge der erzeugten Sequenz, wird beim nächstkleineren (bzw. nächstgrößeren) Wert davor abgebrochen.

```
> 1:4
[1] 1 2 3 4
> 2:7
[1] 2 3 4 5 6 7
```

```
> 10:3
[1] 10  9  8  7  6  5  4  3
> 1.5:4.8
[1] 1.5 2.5 3.5 4.5
```

Die Funktion `rep`(x,times) (*repeat*) erzeugt einen Vektor, in dem das Objekt x *times* Mal wiederholt wird.

```
> rep(x=1,times=3)
[1] 1 1 1
> rep(2,4)
[1] 2 2 2 2
```

Das Objekt x kann dabei auch ein Vektor sein.

```
> rep(c(1,2),4)
[1] 1 2 1 2 1 2 1 2
```

Den mit diesen Funktionen erzeugten Vektoren können natürlich auch Objektnamen zugewiesen werden.

```
> x=seq(2,30,4)
> x
[1]  2  6 10 14 18 22 26 30
```

Zugriff auf Vektorkomponenten

Die einzelnen Komponenten eines Vektors `x` sind durchnummeriert von 1 bis n, wobei n die Gesamtanzahl der Komponenten ist. Mithilfe von eckigen Klammern

`x[c(...)]`

kann gezielt auf bestimmte Teile eines Vektors zugegriffen werden. Dabei steht innerhalb der Klammern ein Vektor (oder ein Skalar) ganzzahliger Werte, der angibt, welche Komponenten gemäß der Nummerierung von Interesse sind. Man spricht in diesem Zusammenhang auch von einem ***Indexvekor***.

```
> Name = c("Anton","Berta","Caesar","Dora","Emil")
> length(Name)
[1] 5
> Name[1]
[1] "Anton"
> Name[2]
[1] "Berta"
> Name[c(1,2)]
[1] "Anton" "Berta"
> Name[1:3]
[1] "Anton"  "Berta"  "Caesar"
```

Ein negatives Vorzeichen vor dem Index oder dem Indexvektor bedeutet, dass auf alle Komponenten mit Ausnahme der angegebenen Indizes zugegriffen werden soll.

```
> Name[-2]
[1] "Anton"  "Caesar" "Dora"   "Emil"
> Name[-c(1,2)]
[1] "Caesar" "Dora"   "Emil"
```

Man kann den Kompenten von Vektoren auch Bezeichner zuordnen. Das wird z.B. beim linearen Regressionmodell zur Bezeichnung der Koeffizienten gemacht, vgl. Abschnitt 3.1.4, S.76f. Einem Vektor können über die Funktion `names`(x) Bezeichnungen zugewiesen werden.

```
> beta = c(1.2,4.8)
> names(beta)=c("beta0","beta1")
> beta
beta0 beta1
  1.2   4.8
```

Die Bezeichner können auch zum Zugriff auf die entsprechende Vektorkomponente verwendet werden. Mit `unname`(x) werden die Komponentenbezeichner gelöscht.

```
> beta["beta1"]
beta1
  4.8
> unname(beta)
[1] 1.2 4.8
```

Rechnen mit Vektoren

Beim Rechnen mit Vektoren ist grundsätzlich zu beachten, dass ein Vektor üblicherweise entweder mit einem Skalar oder mit einem Vektor gleicher Länge verknüpft werden kann. Im ersten Fall wird jede einzelne Komponente mit dem Skalar auf die gleiche Weise verknüpft. Beispielsweise wird zu jeder Komponente der gleiche Wert addiert.

```
> c(1,2,3)+1   # 1+1, 2+1, 3+1
[1] 2 3 4
> c(1,2,3)*2   # 1*2, 2*2, 3*2
[1] 2 4 6
> c(1,2,3)/2   # 1/2, 2/2, 3/2
[1] 0.5 1.0 1.5
```

Das Kommentarsymbol # (vgl. S. 22) wurde hier verwendet, um deutlich zu machen, wie hier gerechnet wird.

Bei einer Verknüpfung eines Vektors mit einem Vektor gemäß eines vorgegebenen Operators wird diese **komponentenweise** durchgeführt. Beispielsweise werden alle gleichrangigen Komponenten jeweils addiert.

```
> c(1,2,3)+c(3,2,1)   # 1+3, 2+2, 3+1
[1] 4 4 4
> c(1,2,3)*c(3,2,1)   # 1*3, 2*2, 3*1
[1] 3 4 3
> c(1,2,3)/c(3,2,1)   # 1/3, 2/2, 3/1
[1] 0.3333333 1.0000000 3.0000000
```

Diese komponentenweise Verknüpfung gilt für alle Grundrechenarten.

Werden zwei Vektoren ungleicher Länge, die jeweils über mindestens 2 Komponenten verfügen, miteinander verknüpft, wird in der Regel eine Warnmeldung ausgegeben. Es wird aber dennoch ein Rechenergebnis ermittelt. Betrachten wir dazu

```
> c(1,2,3,4,5)+c(1,2,3)
```

mit der Ausgabe

```
[1] 2 4 6 5 7
```

und der Warnmeldung

```
Warning message:
In c(1, 2, 3, 4, 5) + c(1, 2, 3) :
longer object length is not a multiple of shorter object length
```

Wie kommt dieses Ergebnis zustande? *R* verfährt hier nach dem Prinzip der sog. **zyklischen Verlängerung**. Darunter versteht man das Prinzip, dass bei zwei unterschiedlich langen Objekten, das kürzere Objekt bis auf die Länge des längeren Objekts verlängert wird. Dabei werden die bereits vorhandenen Komponenten am Ende des Vektors einfach nochmals angefügt. Dies wird so oft wiederholt, bis sich die Längen der beiden Vektoren entsprechen. Im vorliegenden Beispiel wurden dem Vektor `c(1,2,3)` also nochmals die Zahlen 1 und 2 angefügt, so dass im Endeffekt die beiden Vektoren `c(1,2,3,4,5)` und `c(1,2,3,1,2)` addiert wurden. Streng genommen wird auch bei der Verknüpfung eines Vektors mit einem Skalar nicht anders verfahren, nur dass eine Warnmeldung ausbleibt. So ließe sich die Rechnung

```
> c(1,2,3)+1
```

auch ausführen über

```
> c(1,2,3)+c(1,1,1)
```

In *R* implementierte Funktion erlauben i.d.R. auch die Verwendung von Vektoren anstelle von einzelnen Zahlenwerten als Argument. Den Ergebnisvektor erhält man dann, in dem man die Funktion auf jede Komponente des Argumentes anwendet.

```
> sin(c(1,2,3))   # sin(1), sin(2), sin(3)
[1] 0.8414710 0.9092974 0.1411200
> exp(1:3)        # exp(1), exp(2), exp(3)
[1]  2.718282  7.389056 20.085537
```

Im zweiten Beispiel wird der Vektor (e^1, e^2, e^3) berechnet.

2.3 Matrizen

Im folgenden Abschnitt beschäftigen wir uns damit, wie Matrizen in *R* konstruiert werden und wie mit diesen gearbeitet und gerechnet werden kann.

Konstruktion von Matrizen

Der grundlegende Befehl zur Konstruktion von Matrizen lautet

`matrix(data,nrow=1,ncol=1,byrow=FALSE)`.

Meist genügt die Eingabe von 3 Argumenten:

1. ein Datenvektor `data`, aus dem eine Matrix konstruiert werden soll,
2. die Anzahl `nrow` von Zeilen,
3. die Anzahl `ncol` von Spalten.

Die Einträge des Datenvektors werden benutzt, um die Einträge der Matrix **spaltenweise** aufzufüllen.

```
> v=1:12
> m1=matrix(v,3,4)
> m1
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> m2=matrix(v,4,3)
> m2
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
```

Die Einträge von `v` werden benutzt, um zunächst die 1. Spalte, dann die 2. Spalte usw. zu befüllen.

Mit der Option `byrow=TRUE` kann auch eine **zeilenweise** Auffüllung der Matrix erfolgen.

```
> matrix(v,3,4,byrow=T)
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
```

In diesem Fall wird erst die 1. Zeile, dann die 2. Zeile usw. mit den Einträgen des Datenvektors aufgefüllt.

In der Regel sollte das Produkt von Zeilen- und Spaltenanzahl gerade der Vektorlänge entsprechen. Passen mehr Zahlen in die Matrix als der Vektor Komponenten aufweist, wird wiederum nach dem Prinzip der **zyklischen Verlängerung** verfahren (mit oder ohne Warnmeldung). Umfasst der Vektor mehr Komponenten als in der Matrix Platz haben, wird an entsprechender Stelle einfach abgebrochen.

```
> matrix(v,4,4)
     [,1] [,2] [,3] [,4]
[1,]    1    5    9    1
[2,]    2    6   10    2
[3,]    3    7   11    3
[4,]    4    8   12    4

> matrix(v,5,5)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    6   11    4    9
[2,]    2    7   12    5   10
[3,]    3    8    1    6   11
[4,]    4    9    2    7   12
[5,]    5   10    3    8    1
Warning message:
In matrix(v, 5, 5) :
  Datenlänge [12] ist kein Teiler oder Vielfaches der
    Anzahl der Zeilen [5]
> matrix(v,2,2)
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

Mit `cbind( )` kann man eine Matrix um eine oder mehrere Spalten ergänzen. Das *c* in *cbind* steht hierbei für *column*, also Spalte.

```
> m3=cbind(m1,c(13,14,15))
> m3
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    4    7   10   13
[2,]    2    5    8   11   14
[3,]    3    6    9   12   15

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

Mit `rbind( )` wird einer Matrix eine oder mehrere Zeilen hinzugefügt. Das *r* in *rbind* steht hierbei für *row*, also Zeile.

```
> m4=rbind(m2,c(5,9,13))
> m4
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
[5,]    5    9   13

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

Zugriff auf Matrixkomponenten

Wie bei Vektoren erfolgt auch bei Matrizen der Zugriff auf einzelne Komponenten über eckige Klammern `[ ]`. Dabei sind zwei Argumente anzugeben:

1. ein Skalar oder ein Vektor für die interessierenden Zeilen,
2. ein Skalar oder ein Vektor für die interessierenden Spalten.

```
> m1
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> m1[1,1]
[1] 1
> m1[2,3]
[1] 8
```

Mittels `m1[i,j]` kann man also auf das Element der Matrix `m1` zugreifen, welches in der i-ten Zeile und j-ten Spalte steht.

Wie bei Vektoren können hierbei i und j für einzelne Indizes oder Vektoren von Indizes stehen.

```
> m1[c(1,3),2:3]
     [,1] [,2]
[1,]    4    7
[2,]    6    9
```

Wir erhalten hier die Teilmatrix von `m1`, die sich aus den Zeilen 1 und 3 und den Spalten 2 und 3 zusammensetzt. Wird kein Index für die Zeilen- oder Spaltenposition angegeben, verwendet R dort standardmäßig alle Zeilen bzw. Spalten. D.h.

```
> m1[1,]
[1]  1  4  7 10
```

liefert die 1. Zeile von `m1` und damit das gleiche wie

```
> m1[1,1:4]
[1]  1  4  7 10
```

Die 2. Spalte erhalten wir durch:

```
> m1[,2]
[1] 4 5 6
```

Es ist zu beachten, dass Zeilen- und Spaltenvektoren jeweils als Vektor-Objekte und nicht etwa als Matriz-Objekte ausgegeben werden. Die Spalte 2 und 4 erhalten wir durch:

```
> m1[,c(2,4)]
     [,1] [,2]
[1,]    4   10
[2,]    5   11
[3,]    6   12
```

Folgerichtig liefert auch

```
> m1[,]
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

die gesamte Matrix. Mittels

```
> m1[,-4]
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

erhält man die Matrix, die sich ergibt, wenn man die 4. Spalte aus `m1` streicht.

Mit der Funktion `dim( )` lässt sich das **Format** bzw. die *Dimension* einer Matrix ausgeben.

```
> dim(m1)
[1] 3 4
```

`m1` ist eine Matrix, die über 3 Zeilen und 4 Spalten verfügt.

```
> dim(c(1,1))
NULL
```

Angewandt auf einen Vektor oder einen Skalar liefert `dim( )` kein Ergebnis.

> Die Dimensionen einer Matrix (Zeilen- bzw. Spaltenanzahl) lassen sich mit dem `dim()`-Befehl ermitteln.

Rechnen mit Matrizen

Zuerst generieren wir ein paar Matrizen und Vektoren für die Beispielrechnungen.

```
> m1=matrix(c(1,4,3,0),2,2)
> m2=matrix(c(1,2,3,2),2,2)
> m3=matrix(1:6,3,2)
> v1= 1:4; v2= 1:3
> m1
     [,1] [,2]
[1,]    1    3
[2,]    4    0
> m2
     [,1] [,2]
[1,]    1    3
[2,]    2    2
```

Die Grundrechenarten Addition (+), Multiplikation (*), Substraktion (–) und Division (/) sind auch auf Matrizen anwendbar. Dabei kann eine Matrix mit

- einer Matrix *gleichen Formats* oder

- einem Skalar

addiert, multipliziert, subtrahiert bzw. dividiert werden.

```
> m1+m2
     [,1] [,2]
[1,]    2    6
[2,]    6    2
> m1*m2
     [,1] [,2]
[1,]    1    9
[2,]    8    0
```

Bei Verknüpfungen mit Matrizen mit gleichem Format werden die Operationen **komponentenweise** wie bei Vektoren ausgeführt, d.h. der Eintrag der 1. Matrix in der i-ten Zeile und j-ten Spalte wird mit dem entsprechenden Eintrag der 2. Matrix in der i-ten Zeile und j-ten Spalte addiert, multipliziert etc. Im letzten Beispiel wird also

$$\begin{pmatrix} 1 \cdot 1 & 3 \cdot 3 \\ 4 \cdot 2 & 0 \cdot 2 \end{pmatrix}$$

gerechnet. Analog ist

```
> m1/m2
     [,1] [,2]
[1,]    1    1
[2,]    2    0
```

Sind die Formate der Matrizen nicht identisch, gibt es eine Fehlermeldung.

```
> m1+m3
Fehler in m1 + m3 : nicht passende Arrays
```

Diese komponentenweise Ausführung der Operationen trifft, wie wir oben gesehen haben, auch auf die Multiplikation zu. Für die Matrixmultiplikation, die aus der Mathematik bekannt ist, gibt es eine eigene Symbolik:

%*%

Das Produktsymbol wird also zwischen zwei Prozentzeichen gesetzt, um den Unterschied zur komponentenweisen Multiplikation * zu kennzeichnen.

```
> m1%*%m2
     [,1] [,2]
[1,]    7    9
[2,]    4   12
```

Das Ergebnis des Beispiels errechnet sich dann über

$$\begin{pmatrix} 1 & 3 \\ 4 & 0 \end{pmatrix} \begin{pmatrix} 1 & 3 \\ 2 & 2 \end{pmatrix} = \begin{pmatrix} 1 \cdot 1 + 3 \cdot 2 & 1 \cdot 3 + 3 \cdot 2 \\ 4 \cdot 1 + 0 \cdot 2 & 4 \cdot 3 + 0 \cdot 2 \end{pmatrix} = \begin{pmatrix} 7 & 9 \\ 4 & 12 \end{pmatrix}.$$

Der Grundgedanke ist:

„Jede Zeile ist mit jeder Spalte zu multiplizieren und aufzuaddieren“.

Es ist wichtig, diese eigentliche Matrixmultiplikation von der komponentenweisen Multiplikation mit dem Operator `*` zu unterscheiden!

Die Grundrechenarten können auch auf eine Matrix und einen Skalar angewandt werden. In diesem Fall wird die Operation mit dem Skalar auf jede Komponente der Matrix angewandt.

```
> m1*2
     [,1] [,2]
[1,]    2    6
[2,]    8    0
> m1+1
     [,1] [,2]
[1,]    2    4
[2,]    5    1
```

Während das Ergebnis bei der Multiplikation `m1*2` keine Überraschung darstellt, weicht die *R*-Syntax bei der Addition (oder Substraktion) von der mathematischen Notation ab, in der `m1+1` gar nicht definiert wäre.

```
> m1/2
     [,1] [,2]
[1,]  0.5  1.5
[2,]  2.0  0.0
```

Eine weitere Operation, die auf Matrizen angewandt werden kann, ist die **Transposition** mit Hilfe der Funktion `t( )`, d.h. das Vertauschen von Zeilen und Spalten.

```
> t(m1)
     [,1] [,2]
[1,]    1    4
[2,]    3    0
```

Mit dem Befehl `solve( )` wird die inverse Matrix berechnet.

```
> solve(m1)
           [,1]        [,2]
[1,] 0.0000000  0.25000000
[2,] 0.3333333 -0.08333333
```

Wir erinnern uns, dass die Multiplikation (im mathematischen Sinn!) einer Matrix mit ihrer Inversen die Einheitsmatrix ergibt.

```
> m1%*%solve(m1)
     [,1]          [,2]
[1,]    1 1.387779e-17
[2,]    0 1.000000e+00
```

Mit den Anmerkungen aus Abschnitt 1.2.2 (vgl. S.12) wissen wir nun auch, dass im vorliegenden Beispiel die Matrixkomponente

```
    1.387779e-17
```

als „exakte Null“ zu interpretieren ist.

> Matrizenmultiplikationen (im mathematischen Sinn) werden mit dem Operator `%*%` durchgeführt und nicht etwa mit `*`.

2.4 Listen

Eine Liste ist in *R* eine Zusammenfassung von mehreren Objekten – möglicherweise unterschiedlichen Typs – zu einem neuen Objekt. Sie stellt damit eine sehr flexible Objektform dar. Dies bedeutet, dass sich innerhalb dieser Datenstruktur Objekte unterschiedlicher Gattung und Skalierung (numerisch und nichtnumerisch) organisieren lassen. Diese unterschiedlichen Objekte bilden dann die *Komponenten* der Liste. Im Gegensatz dazu lassen sich mittels Vektoren nur Objekte gleichen Typs zusammenfassen. Listen werden häufig als Funktionsergebnis verwendet, da Funktionen in *R* jeweils nur ein Objekt als Rückgabewert haben dürfen (vgl. Abschnitt 5.2.2, S.158ff).

Konstruktion von Listen

Die grundlegende Anweisung zur Konstruktion von Listen lautet

```
list()
```

Die Argumente sind *R*-Objekte, die zu einem Listenobjekt zusammengefasst werden sollen.

```
> v=c("Hallo","Tag","Hi","Guten Tag")
> m=matrix(1:9,3,3)
> v
[1] "Hallo"     "Tag"       "Hi"        "Guten Tag"
> m
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> objekte=list(v,m)
> objekte
[[1]]
[1] "Hallo"     "Tag"       "Hi"        "Guten Tag"

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

Im vorliegenden Fall werden unter dem Namen `objekte` ein Vektor und eine Matrix abgespeichert. Bei der Ausgabe werden diese Teilobjekte deutlich getrennt als Listenkomponenten aufgeführt. Die Komponenten einer Liste erkennt man an der eckigen Doppelklammer, innerhalb derer die laufende Nummer der Komponente steht. Im Beispiel bildet der Vektor die 1. Komponente und die Matrix die 2. Komponente der Liste.

Zugriff auf Listenkomponenten

Bei Listen erfolgt der Zugriff auf einzelne Komponenten über eckige Doppelklammern.

`[[ ]]`

Als Argument ist der Index (oder ein Indexvektor) für die interessierenden Komponenten anzugeben.

```
> objekte[[1]]
[1] "Hallo"     "Tag"       "Hi"         "Guten Tag"
> objekte[[2]]
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

Mit `objekte[[i]]` wird die i-te Komponente abgerufen.

Die Anzahl der Komponenten einer Liste kann über `length( )` abgefragt werden.

```
> length(objekte)
[1] 2
```

Ist eine Komponente der Liste bspw. ein Vektor oder eine Matrix, so kann auf diese wiederum entsprechend ihrer Objektform zugegriffen werden. Der Zugriff erfolgt hierarchisch. Im vorliegenden Beispiel wird mit der Eingabe

```
> objekte[[1]][2]
[1] "Tag"
```

auf die 2. Komponente des Vektors, welcher die 1. Listenkomponente bildet, zugegriffen. Mit der Eingabe

```
> objekte[[2]][,2]
[1] 4 5 6
```

wird auf die 2. Spalte der Matrix, welche die 2. Listenkomponente bildet, zugegriffen.

Gerade bei Listen mit sehr vielen Komponenten ist es ratsam, den einzelnen Komponenten **Namen** zu geben, die über den Inhalt der Komponente sinnvoll Auskunft geben. Die Namen einer bestehenden Liste können über den Befehl `names( )` abgefragt werden.

```
> names(objekte)
NULL
```

Im aktuellen Beispiel wurden keine Namen zugewiesen und es wird `NULL` ausgegeben. Für Namenszuweisungen kann in der Definition der Liste in den Argumenten die Syntax *name=objekt* verwendet werden.

```
> objekte1=list(Begrüßungen=v,Zahlenmatrix=m)
```

Die erste Komponente hat jetzt den Namen *Begrüßungen* und die zweite *Zahlenmatrix*.

```
> objekte1
$Begrüßungen
[1] "Hallo"     "Tag"      "Hi"        "Guten Tag"

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

> names(objekte1)
[1] "Begrüßungen" "Zahlenmatrix"
```

Falls den Komponenten Namen zugewiesen wurden, kann auch alternativ über

`Listenname$Komponentenname`

auf einzelne Listenkomponenten zugegriffen werden.

```
> objekte1$Begrüßungen
[1] "Hallo"     "Tag"      "Hi"        "Guten Tag"

> objekte1$Zahlenmatrix
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

Eine Namenszuweisung kann auch über die `names( )`-Funktion erfolgen.

```
> names(objekte)=c("Begrüßungen","Zahlenmatrix")
> objekte$Begrüßungen
[1] "Hallo"     "Tag"       "Hi"         "Guten Tag"
```

Listen als Funktionsergebnis

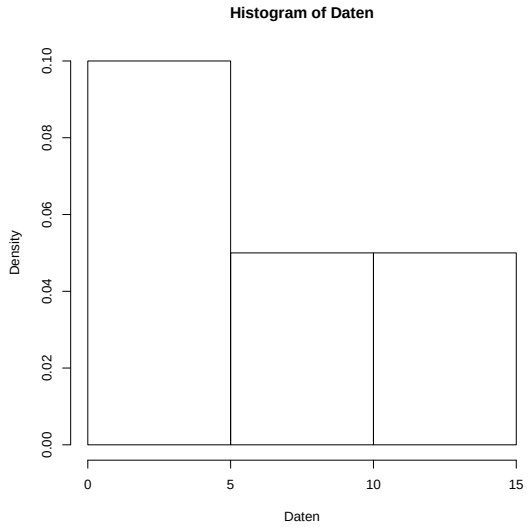
Viele Funktionen in *R* geben ein Listen-Objekt als Funktionsergebnis aus. Betrachten wir dazu als Beispiel die `hist( )`-Funktion, mit der sich *Histogramme* erstellen lassen.

```
> Daten=c(1,5,3,8,9,4,11,15)
> hist(Daten)
> ?hist
> hist(Daten,prob=T)
```

In Abbildung 2.4.1 ist das so gezeichnete Histogramm abgebildet. Wie aus dem Hilfetext zum Befehl `hist()` hervorgeht, ist zur Erstellung eines Histogramms zusätzlich die Option `prob=T` zu wählen. Die Klasseneinteilung wird automatisch von *R* vorgenommen. Sie ließe sich aber auch über das Argument `breaks=c(...)` mit Eingabe eines Vektors mit Klassengrenzen steuern.

Weiter geht aus dem Hilfetext hervor (Abschnitt *Value*), dass das Ergebnis der Funktion `hist()` eine Liste ist, welche die Komponenten

Abbildung 2.4.1: Histogramm



`breaks`, `counts`, `density` usw.

umfasst. Das ist bei der Anwendung zunächst nicht ersichtlich. Die `hist( )`-Funktion ist nämlich so voreingestellt, dass standardmäßig nur die Grafik erstellt wird. Auf die Ausgabe der Listenkomponenten wird verzichtet. Wie kann nun aber auf die vollständige Ergebnisliste zugegriffen werden? Dies lässt sich bspw. über eine Zuweisung der Form

```
> HistDaten=hist(Daten)
```

bewerkstelligen. Nun existiert das Objekt `HistDaten`, welches formal eine Liste ist.

```
> HistDaten
$breaks
[1]  0  5 10 15
$counts
[1] 4 2 2
$intensities
[1] 0.09999998 0.05000000 0.05000000
$density
[1] 0.09999998 0.05000000 0.05000000
$mids
[1]  2.5  7.5 12.5
$xname
[1] "Daten"
$equidist
[1] TRUE
attr(,"class")
[1] "histogram"
> length(HistDaten)
[1] 7
> names(HistDaten)
[1] "breaks"  "counts"  "intensities" "density"  "mids"
[6] "xname"       "equidist"
```

Über Eingabe von `HistDaten` erhalten wir eine vollständige Ausgabe der Liste mit allen Komponenten. Die einzelnen Komponenten sind jeweils über das $-Zeichen identifizierbar. Am Ende der Ausgabe ist noch ein Attribut `attr(,"class")` angefügt. Dies ist keine Listenkomponente. Die Bedeutung dieses Attributs ist für uns hier nicht unmittelbar relevant. Um sicher zu gehen, dass auch alle Komponenten zu sehen sind, können wir über `length( )` und `names( )` die Länge der Liste bzw. alle Komponentennamen abfragen.

> Zahlreiche Funktionen in *R* geben als Ergebnis eine Liste aus, in der die einzelnen Bestandteile einer statistischen Analyse aufgeführt werden.

2.5 Data Frames

Datensätze lassen sich i.d.R. als Zahlen und Zeichenketten auffassen, die in einem Tabellenformat angeordnet sind. Schauen wir uns dazu ein einfaches Beispiel an.

Beob.-Nr.	Name	Geschlecht	Größe	Gewicht
1	Anton	m	182	80
2	Berta	w	174	68
3	Caesar	m	189	92
4	Emil	m	165	55
5	Dora	w	180	78

In den einzelnen Zeilen dieser Matrix stehen die Beobachtungswerte, die für einen *Merkmalsträger* erhoben wurden. Im vorliegenden Beispiel korrespondiert also jede Zeile mit einer Person. Die Beobachtungswerte sind hier (ohne Nummerierung) 4-dimensional, d.h. von jeder Person wurden 4 Merkmale erhoben. In jeder Spalte stehen dann die Werte bzw. Ausprägungen eines Merkmals über alle Merkmalsträger hinweg. Aufgrund der Anordnung in einem Rechteckschema, das einer Matrix ähnelt, spricht man auch von der sogenannten **Datenmatrix**. Optional kann die Datenmatrix noch über eine Spalte mit fortlaufender Nummerierung der Beobachtungswerte verfügen. Im engeren Sinne gehört diese ebenso wie eine optionale Zeile mit den Variablennamen (1. Zeile) nicht zur eigentlichen Datenmatrix.

> Unter einem **Datensatz** verstehen wir im Wesentlichen eine *Datenmatrix*, deren Zeilen mit einzelnen Merkmalsträgern und deren Spalten mit den zugehörigen Merkmalsausprägungen korrespondieren.

Datensätze werden in *R* jedoch üblicherweise nicht im Objekttyp `matrix`, sondern im Objekttyp `data.frame` abgespeichert. Solche **Data Frames** („Datenrahmen“) weisen äußerlich zwar große Ähnlichkeiten zu Matrizen auf, sie erlauben es aber im Gegensatz zu Matrizen, auch Spaltenvektoren unterschiedlichen Typs zusammenzufassen. Im Beispiel werden Größe und Gewicht durch Zahlen, Name und Geschlecht aber über Zeichenketten ausgedrückt.

Konstruktion von Data Frames

Data Frames kann man als spezielle Form von Listen auffassen, welche die Einschränkung besitzen, dass sie nur Vektoren gleicher Länge als Komponenten enthalten dürfen. Die Vektoren selbst können, wie es auch im Zusammenhang von Datensätzen sinnvoll erscheint, sowohl numerischer als auch nichtnumerischer Natur sein.

Zunächst stellen wir noch einmal die Datenvektoren der einzelnen Merkmale bereit.

```
> rm(list=ls())
> Name = c("Anton","Berta","Caesar","Dora","Emil")
> Geschlecht=c("m","w","m","w","m")
> Größe = c(182,174,189,165,180)
> Gewicht=c(80,68,92,55,78)
```

Der grundlegende Befehl zur Konstruktion von Data Frames lautet

```
data.frame(...)
```

Danach werden die Namen der Vektoren durch Kommata getrennt eingegeben. Im vorhergehenden Beispiel wäre dies also

```
> Personen=data.frame(Name,Geschlecht,Größe,Gewicht)
> Personen
    Name Geschlecht Größe Gewicht
1  Anton          m   182      80
2  Berta          w   174      68
3 Caesar          m   189      92
4   Dora          w   165      55
5   Emil          m   180      78
```

Die Ausgabe des Data Frames erfolgt in einer für einen Datensatz sinnvollen Matrixdarstellung. Wir bemerken, dass weder die numerischen noch die nichtnumerischen Daten in Anführungszeichen gesetzt sind. Tatsächlich ist es theoretisch möglich, dass die zu den Vektoren `Größe` und `Gewicht` zugehörigen Zahlen von *R* nicht als numerische Werte interpretiert werden. Dieser Umstand würde sich dann erst bei statistischen Berechnungen mit diesen Werten ergeben. Eine einfache Möglichkeit die Skalierung innerhalb eines Data Frames festzustellen, ist die Anwendung des Befehls

```
summary(...)
```

auf den Data Frame. Der `summary( )`-Befehl lässt sich praktisch auf alle Objekte in *R* anwenden und liefert kontextbezogen, d.h. vom jeweiligen Objekttyp abhängig, sinnvolle Zusammenfassungen statistischer und technischer Details des vorliegenden Objekts. In unserem Beispiel liefert der Befehl die Ausgabe

```
> summary(Personen)
     Name   Geschlecht     Größe         Gewicht
 Anton :1   m:3         Min.   :165   Min.   :55.0
 Berta :1   w:2         1st Qu.:174   1st Qu.:68.0
 Caesar:1               Median :180   Median :78.0
 Dora  :1               Mean   :178   Mean   :74.6
```

```
 Emil   :1              3rd Qu.:182   3rd Qu.:80.0
                        Max.   :189   Max.   :92.0
```

Wir sehen, dass bei den nichtnumerischen Vektoren Häufigkeitsauszählungen über die verschiedenen Merkmalsausprägungen durchgeführt werden. Bei den numerischen Vektoren werden statistische Kennzahlen wie Minimum, Maximum und Mittelwert berechnet.

Mit der Analyse solcher Zusammenfassungen lassen sich häufig auch „Störungen“ in großen Datensätzen entdecken. Würde beispielsweise beim Merkmal `Größe` das Maximum 819 lauten, könnte man auf eine Falscheingabe (z. B. verursacht durch einen Zahlendreher) oder einen anderen (möglicherweise größeren) Fehler im Datensatz schließen.

> Mit dem Befehl `summary( )` gewinnt man einen Schnellüberblick über einen Datensatz. Außerdem lassen sich damit häufig Fehler in den Daten und „inhaltliche Ungereimtheiten“ feststellen.

Zugriff auf Komponenten eines Data Frames

Ein Data Frames Objekt weist Gemeinsamkeiten zu Listen und Matrizen auf. Wie bei Listen kann man über die Namen auf die entsprechenden Spalten des Datensatzes zugreifen, s.a. Abschnitt 2.4.

```
> names(Personen)
[1] "Name"       "Geschlecht" "Größe"      "Gewicht"

> Personen$Name
[1] Anton  Berta  Caesar Dora   Emil
Levels: Anton Berta Caesar Dora Emil

> Personen$Gewicht
[1] 80 68 92 55 78
```

Der Zugriff auf die Spalten kann auch über `[[ ]]` erfolgen.

```
> Personen[[1]]
[1] Anton  Berta  Caesar Dora   Emil
Levels: Anton Berta Caesar Dora Emil

> Personen[[4]]
[1] 80 68 92 55 78
```

Der Data Frame verhält sich wie eine Liste mit 4 Komponenten.

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

Andererseits kann man auch wie bei Matrizen auf die im Rechteckformat angeordneten Einträge zugreifen, s.a. Abschnitt 2.3.

```
> dim(Personen)
[1] 5 4
```

```
> Personen[5,4]
[1] 78
```

Der Data Frame verhält sich hier wie eine Matrix mit 5 Zeilen und 4 Spalten.

```
> Personen[1,1]
[1] Anton
Levels: Anton Berta Caesar Dora Emil
```

Bei den nichtnumerischen Merkmalen werden zusätzlich alle vorkommenden Merkmalsausprägungen angegeben.

Wie bei Matrizen kann man auf einzelne Zeilen oder Spalten zugreifen.

```
> Personen[,1]
[1] Anton  Berta  Caesar Dora   Emil
Levels: Anton Berta Caesar Dora Emil
> Personen[1,]
   Name Geschlecht Größe Gewicht
1 Anton          m   182      80
```

Wir erhalten die erste Spalte bzw. die erste Zeile des Datensatzes.

Der attach-Befehl

Um den Befehl `attach()` verstehen zu können, löschen wir zunächst die Vektoren `Name`, `Geschlecht`, `Größe` und `Gewicht` aus dem aktuellen Workspace. Dies bewerkstelligen wir mittels `rm`.

```
> rm(Name,Geschlecht,Größe,Gewicht)
```

Bei der Auflistung aller erzeugten Objekte mit dem Befehl

```
> ls()
[1] "Personen"
```

dürften diese nicht mehr auftauchen. Eine Überprüfung anhand von `Name` liefert

```
 > Name
Fehler: objekt "Name" nicht gefunden
```

Das gleiche trifft auf `Geschlecht`, `Größe` und `Gewicht` zu. Dennoch existiert natürlich weiterhin – sofern er nicht auch gelöscht wurde – der Data Frame `Personen`, der diese Vektoren beinhaltet.

Die Anwendung des Befehls `attach()` auf den Data Frame `Personen` bewirkt nun, dass auf die einzelnen Vektoren dieses Data Frames direkt über die Namen der Vektoren zugegriffen werden kann, so als ob diese als eigene Objekte im Workspace vorhanden wären.

```
> attach(Personen)
> Gewicht
[1] 80 68 92 55 78
```

Der Vorteil des Befehls `attach()` besteht also darin, dass der eher umständliche Komponentenzugriff über Listen- und Matrizensyntax entfällt. Zu beachten ist, dass ein solcher `attach`-Zugriff nur während der laufenden Sitzung möglich ist. Bei einem erneuten Starten von *R* muss auf den Data Frame erneut `attach()` angewendet werden.

```
> DS = data.frame(x=c(1,2),y=c(2.4,4.1))
> attach(DS)
> x
[1] 1 2
```

Der Data Frame `DS` wurde erzeugt und seine Komponenten, `x` und `y`, mittels `attach()` freigeschaltet.

```
> ls()
[1] "DS"       "Personen"
```

Globale Objekte sind nur die beiden Data Frames `DS` und `Personen`. Wie kommt es, dass bspw. `x` und `Gewicht` trotzdem zur Verfügung stehen?

```
> search()
 [1] ".GlobalEnv"        "DS"                "Personen"
 [4] "package:stats"     ...
```

Mit der `search( )`-Anweisung kann man sich anzeigen lassen, wo *R* nach Bezeichnern sucht. Zuerst in `".GlobalEnv"`, d.h. in der globalen Variablenumgebung. Deren Einträge kann man sich mit `ls()` anzeigen lassen. Danach wird im Data Frame `DS` und dann im Data Frame `Personen` gesucht. Die `attach()`-Anweisung bewirkt i.W. einen Eintrag der entsprechenden Data Frames in den Suchpfad von `search()`.

Rückgängig machen kann man die `attach()`-Anweisung mittels `detach( )`.

```
> detach(DS)
> x
Error: object 'x' not found
```

`detach()`, angewandt auf einen Datensatz, macht die vorher durchgeführte `attach()`-Anweisung unwirksam. `x` steht nicht mehr zur Verfügung.

Maskierung beim attach-Befehl

Was würde eigentlich passieren, wenn wir ein neues Objekt mit dem Namen `Gewicht` erzeugen? Angenommen, wir weisen Gewicht z.B. den Wert 75 zu, also

```
> Gewicht=75
```

Die Ausgabe von Größe liefert nun

```
> Gewicht
[1] 75
```

Der Wertevektor aus `Personen` steht nicht mehr zur Verfügung. Wir erzeugen einen Vektor `Farbe` und einen Data Frame `P2` und wenden `attach()` an.

```
> Farbe= c("schwarz","weiß")
> P2 = data.frame(Name=c("Frank","Heinz"),Gewicht=c(71,72),
                  Zahl=c(6,7),Farbe=c("rot","blau"))
> attach(P2)
The following object is masked _by_ .GlobalEnv:

    Farbe

The following objects are masked from Personen:

    Gewicht, Name
```

Es werden zwei Warnhinweise ausgegeben. Der erste bezieht sich auf `Farbe`.

```
> Farbe
[1] "schwarz" "weiß"
```

Der `Farbe`-Vektor, den wir zunächst definiert haben, überdeckt die `Farbe`-Komponente aus `P2`. Der zweite Warnhinweis bezieht sich auf `Gewicht` und `Name`.

```
> Gewicht
[1] 75
```

Der Wert für Gewicht überdeckt („maskiert“) die `Gewicht`-Komponente von `P2`.

```
> Name
[1] Frank Heinz
Levels: Frank Heinz
```

Die `Name`-Komponenten von `P2` überdeckt („maskiert“) diejenige von `Personen`. Schauen wir uns den Suchpfad an:

```
> search()
 [1] ".GlobalEnv"        "P2"                "Personen"
 [4] "package:stats"     ...
```

`Gewicht` und `Farbe` werden zuerst unter in der globalen Umgebung `".GlobalEnv"` gefunden und die entsprechenden Komponenten in `P2` bzw. `Personen` ignoriert. `Name` wird zuerst als Komponente von `P2` gefunden und die von `Personen` ignoriert. `Geschlecht` tritt das erste Mal in `Personen` auf. Die entsprechende Komponenten wird verwendet.

```
> Geschlecht
[1] m w m w m
Levels: m w
```

Mittels `rm()` und `detach()` können Objekte aus dem Suchpfad von `search()` entfernt und Komponenten von Data Frames wieder sichtbar gemacht werden.

```
> rm(Gewicht)
> detach(P2)
> Gewicht
[1] 80 68 92 55 78
> Name
[1] Anton  Berta  Caesar Dora   Emil
Levels: Anton Berta Caesar Dora Emil
```

Die `attach()`-Anweisung ist nützlich, um auf die Komponenten eines Data Frames auf vereinfachte Weise Zugriff zu erhalten. Man muss aber aufgrund der Maskierungseffekte aufpassen, dass man tatsächlich auch mit den gewünschten Vektoren die vorgesehenen Berechnungen durchführt.

2.6 Arbeiten mit logischen Operatoren

Neben numerischen und nichtnumerischen Daten können in *R* auch logische Werte verarbeitet werden. Darunter versteht man die beiden Werte

`TRUE` (kurz `T`) und `FALSE` (kurz `F`),

also *„Wahr“* und *„Falsch“*. Die Werte TRUE und FALSE können dabei auch verkürzt mit T und F angegeben werden. Anderweitige Abkürzungen scheiden aus.

Logische Werte werden häufig mit Vergleichsoperationen erzeugt. Dabei wird geprüft, ob einzelne Daten eine ganz bestimmte Bedingung (Eigenschaft) erfüllen oder nicht. Solche Überprüfungen lassen sich mithilfe von *Vergleichsoperatoren* durchführen. Dazu zählen

$$<,\ >,\ <=,\ >=,\ ==,\ !=,$$

also „kleiner“, „größer“, „kleiner oder gleich“, „größer oder gleich“, „gleich“ und „ungleich“. Wir betrachten dazu einige Beispiele.

```
> x=c(8,5,5,1,6,9,4)
> x<6
[1] FALSE  TRUE  TRUE  TRUE FALSE FALSE  TRUE
```

Hier werden alle Komponenten von `x` auf die Bedingung `x<6` hin geprüft. Für jeden Wert, bei dem die Bedingung erfüllt ist, wird ein `TRUE` gesetzt, sonst ein `FALSE`. Analog erfolgt die Auswertung von `x>=6`.

```
> x>=6
[1]  TRUE FALSE FALSE FALSE  TRUE  TRUE FALSE
> x==5
[1] FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE
> x!=5
[1]  TRUE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
```

Der Operator `==` prüft auf Gleichheit, der Operator `!=` auf Ungleichheit, d.h. es wird *komponentenweise* geprüft, ob ein Wert gleich 5 ist bzw. ungleich 5 ist.

```
> x-1==4*2
[1] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE
```

Arithmetische Operationen werden *vor* Vergleichen durchgeführt, d.h. hier wird `x-1==8` bzw. `x==9` geprüft.

Mit den logisichen Operatoren

`&` und `|`

können zwei logische Ausdrücke verknüft werden. `A1 & A2` (*„UND-Operation“*) liefert genau dann *wahr*, wenn die logischen Ausdrücke `A1` und `A2` beide *wahr* sind.

```
> 2<x & x<8
[1] FALSE  TRUE  TRUE FALSE  TRUE FALSE  TRUE
```

Der Wert 8 ist nicht gleichzeitig größer als 2 und kleiner als 8 (`FALSE`), dafür aber der Wert 5 (`TRUE`).

`A1 | A2` (*„ODER-Operation“*) liefert genau dann *wahr*, wenn mindestens einer der beiden logischen Ausdrücke `A1` und `A2` *wahr* ist.

```
> x<4 | x>6
[1]  TRUE FALSE FALSE  TRUE FALSE  TRUE FALSE
```

Der Wert 8 ist größer als 6 (`TRUE`), der Wert 5 ist weder größer als 6 noch kleiner als 4 (`FALSE`).

Mit logischen Werten selbst kann auch gerechnet werden. Bei arithmetischen Verknüpfungen werden die `TRUE`-Werte wie Einsen und `FALSE`-Werte wie Nullen behandelt.

```
> T+T+F+T
[1] 3
> T*T
[1] 1
```

Insbesondere in Verbindung mit dem Befehl `sum()` sind solche Operationen bei statistischen Analysen äußerst nützlich. Mit dem Befehl `sum()` lassen sich alle `TRUE`-Werte zählen.

```
> sum(x>3)
[1] 6
> sum(x<4|x>6)
[1] 3
```

Man bekommt also Aufschluss darüber, wie viele Werte eines Vektors eine bestimmte Eigenschaft erfüllen. Stellen wir uns vor, wir hätten einen Datenvektor mit den Körpergewichten von 1000 Personen. Mit Hilfe logischer Operatoren könnten wir rasch die Anteile von Personen in bestimmten Gewichtsklassen ermitteln.

• Bedingter Zugriff auf Komponenten – reflexiv und projektiv • Von großem Nutzen ist die Verwendung logischer Werte für den gezielten Zugriff auf einzelne Komponenten eines vorliegenden Datenobjekts. Demnach werden alle zu einem `TRUE` korrespondierenden Komponenten ausgegeben und alle zu einem `FALSE` korrespondierenden Komponenten unterdrückt.

```
> x = c(3,1,5,3,6,-1,0)
> x[(c(TRUE,TRUE,FALSE,TRUE,TRUE,FALSE,FALSE))]
[1] 3 1 3 6
```

Der wesentliche „Trick“ besteht nun darin, solche logischen Werte über eine logische Abfrage zu erzeugen, um unter bestimmten Bedingungen dann einen Zugriff vorzunehmen.

Häufig erzeugt man die logischen Werte dabei über logische Abfragen zu erzeugen, um unter bestimmten Bedingungen dann einen Zugriff vorzunehmen.

```
> x>2
[1]  TRUE FALSE  TRUE  TRUE  TRUE FALSE FALSE
> x[x>2]
[1] 3 5 3 6
```

In diesem Fall werden nur diejenigen Werte von `x` ausgegeben, die größer als 2 sind. Die Bedingung kann dabei auch auf einer andere abstellen, wie folgendes Beispiel demonstriert:

```
> y=c(0,0,0,0,1,1,1)
> y==1
[1] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE
> x[y==1]
[1]  6 -1  0
```

Der Vektor `y` muss dazu die gleiche Länge wie Vektor `x` besitzen. Eine solche Situation tritt bei Vorliegen eines 2-dimensionalen Datensatzes mit Beobachtungen (x_i, y_i) auf. Mit obiger Operation wird nur auf diejenigen Komponenten von `x` zugegriffen, die mit einer 1 von `y` kooperieren.

Beziehen sich Zugriff und Bedingung auf die gleiche Variable, sprechen wir von einem ***reflexiven Zugriff*** („Greife auf `x` zu, bedingt darauf, dass `x` ...„). Unterscheiden sich Zugriffs- und Bedingungsvariable, sprechen wir von einem ***projektiven Zugriff*** („Greife auf `x` zu, bedingt darauf, dass `y` ...„). Insbesondere lassen sich über projektive Zugriffe partielle statistische Analysen in Untergruppen von Datensätzen durchführen.

Wir betrachten dazu die folgenden Vektoren, die Angaben zum Namen, Geschlecht und Größe von 5 Personen machen. Solche Daten speichert man günstigerweise zusammen in einem *Data Frame*, s. Abschnitt 2.5, S.46ff.

```
> Name = c("Anton","Berta","Caesar","Dora","Emil")
> Geschlecht=c("m","w","m","w","m")
> Größe = c(182,174,189,165,180)
```

Möchte man nur die Namen der weiblichen Studenten haben, erhält man diese über

```
> Name[Geschlecht=="w"]
[1] "Berta" "Dora"
```

Möchte man nur auf die Körpergrößen der männlichen bzw. nur der weiblichen Personen zugreifen, bewerkstelligt man das über

```
> Größe[Geschlecht=="m"]
[1] 182 189 180
> Größe[Geschlecht=="w"]
[1] 174 165
```

Die jeweiligen Ausgaben lassen sich selbstverständlich weiter statistisch auswerten, wie im nachfolgenden Punkt am Beispiel von `mean()` und `table()` ausgeführt wird.

• Partialanalysen am Beispiel von `mean()` und `table()` • Mittels bedingter Zugriffe lassen sich dataillierte statistische Partialanalysen durchführen – ein enorm wichtiger und auch notwendiger Schritt im Rahmen jeder statistischen Auswertung. Wir verwenden die Datenvektoren der vorherigen Abschnitts.

Zunächst berechnen wir die Durchschnittsgrößen der Männer bzw. Frauen.

```
>   mean(Größe[Geschlecht=="m"])
[1] 183.6667
>   mean(Größe[Geschlecht=="w"])
[1] 169.5
```

Als nächstes ermitteln wir die absolute oder relative Häufigkeitsverteilung für das Geschlecht für diejenigen Personen, die größer als 170 cm sind.

```
> tab=table(Geschlecht[Größe>170])
> tab
m w
3 1
> prop.table(tab)
    m    w
0.75 0.25
```

Die *R*-Funktionen `table()` und `prop.table()` zur Ermittlung von Häufigkeitsverteilungen werden ausführlicher im Abschnitt 3.1.1 besprochen.

• Dichotomisierung über `ifelse()` • Manchmal ist es von Interesse die Ausprägungen von Variablen in bestimmter Weise zu kategorisieren. Im Falle zweier Kategorien sprechen wir von einer ***Dichotomisierung***.

Für eine Dichotomisierung eignet sich häufig der Befehl

`ifelse`(bedingung,yes,no).

bedingung ist ein Vektor von Wahrheitswerten. Das Ergebnis ist ein Vektor, dessen Komponente den Wert *yes* annimmt, wenn der entsprechende Wahrheitswert *TRUE* war und sonst den Wert *no*.

Möchte man beispielsweise eine Variable kreieren, welche nur nach kleinen ($\leq$ 175cm) und großen Personen ($>$ 175cm) unterscheidet, kann man dies bewerkstelligen über

```
> Größe = c(175,170,180,174,162)
> ifelse(Größe>=175,1,0)
[1] 1 0 1 0 0
```

Immer dann, wenn der logische Ausdruck wahr ist, wird eine 1 gesetzt und sonst eine 0. Um tatsächlich eine dichotome Variable zu erhalten, muss lediglich noch eine Zuweisung vorgenommen werden:

```
> großVSklein=ifelse(Größe>=175,1,0)
> großVSklein
[1] 1 0 1 0 0
```

Natürlich können auch andere numerische oder nichtnumerische Werte gesetzt werden, wie folgende Beispiele zeigen:

```
> ifelse(Größe>=175,2,1)
[1] 2 1 2 1 1
> ifelse(Größe>=175,"groß","klein")
[1] "groß" "klein" "groß" "klein" "klein"
```

Ein Anwendungsbeispiel für eine solche Operation ist Abbildung 3.2.53, S.119, aus Abschnitt 3.2.4, in welcher geschlechtsspezifische Symbole für das Streudiagramm verwendet wurden.

• **Ersetzen und Kategorisieren über `replace()`** • Manchmal sollen bestimmte Werte einer Variablen durch andere Werte ersetzt werden oder eine Variablen soll auf bestimmte Weise kategorisiert werden. Zur Veranschaulichung nehmen wir weiterhin den Datensatz `Studenten`, den wir bereits verwendet haben.

Für beide Fälle eignet sich häufig der Befehl

`replace`(x,bedingung,wert).

bedingung ist ein Indexvektor. Rückgabewert ist eine Kopie von Vektor x, bei dem die Werte an den Stellen des ***Indexvektors*** durch *wert* ersetzt werden.

Folgende Beispiele illustrieren das Ersetzen von Werten innerhalb von Variablen und das Kategorisieren von Variablen:

```
> Größe
[1] 175 170 180 174 162
> Größe=replace(Größe,Größe<170,0)
> Größe
[1] 175 170 180 174   0
> Größe=replace(Größe,Größe>=170 & Größe<175,1)
> Größe
[1] 175   1 180   1   0
> Größe=replace(Größe,Größe>=175,2)
> Größe
[1] 2 1 2 1 0
```

Grundsätzlich ist `replace()` auch ohne Auswertung logischer Ausdrücke anwendbar. Auf weitergehende Ausführungen hierzu sei verzichtet. Zu beachten ist, dass im vorliegenden Fall ein schrittweises Vorgehen der Form

```
> Größe = c(175,170,180,174,162)
> Größe=replace(Größe,Größe<170,"klein")
> Größe=replace(Größe,Größe>=170 & Größe<175,"mittelgroß")
```

nicht möglich ist und zu einer Fehlermeldung führt. Dies liegt daran, dass bereits bei der ersten Operation alle Einträge in Größe in `character`-Typ umgewandelt werden. Ein Vektor muss von einem einheitlichen Datentyp sein. Damit ist ein numerischer Abgleich der Form `Größe>=170` jedoch nicht mehr möglich.

Partieller Zugriff auf *Data Frames*

Gerade im Zusammenhang mit größeren Datensätzen erweisen logische Operationen nützliche Dienste, wenn es darum geht, gezielt nur auf ausgewählte Teile der Datensätze zuzugreifen. Stellen wir uns vor, wir hätten einen Datensatz mit mehreren Tausend Beobachtungen (Zeilen) und einigen Dutzend Variablen (Spalten) in einem *Data Frame* gespeichert. Eine übersichtliche Ausgabe in der *R*-Konsole wäre aufgrund des begrenzt verfügbaren Platzes schon nicht mehr möglich. In solchen Fällen kann mit Hilfe des Befehls

```
subset(x,subset,select)
```

partiell auf einen Datensatz zugegriffen werden. `x` ist der *Data Frame*, aus dem ein Teildatensatz selektiert werden soll. `subset` ist ein Vektor von logischen Werten, der angibt, welche *Zeilen* des Datensatzes ausgewählt werden sollen. Über `select` werden die Spalten des Datensatzes ausgewählt. `subset` und `select` sind einschränkende Bedingungen, die als optionale Argumente nicht verwendet werden müssen.

Wir demonstrieren dies anhand des kleinen Datensatzes aus Abschnitt 2.5, S.46ff, nach. Wir verwenden den *Data Frame* `Personen`, s.a. S.47.

```
> Personen
    Name Geschlecht Größe Gewicht
1  Anton          m   182      80
2  Berta          w   174      68
3 Caesar          m   189      92
4   Dora          w   165      55
5   Emil          m   180      78
```

Den vollständigen Datensatz erhalten wir auch mit

```
> subset(Personen)
```

Den Teildatensatz aller männlichen Personen erhalten wir folgendermaßen:

```
> subset(Personen,Geschlecht=="m")
    Name Geschlecht Größe Gewicht
1  Anton          m   182      80
3 Caesar          m   189      92
5   Emil          m   180      78
```

Das optionale Argument `select` wird hier nicht verwendet. Wir sehen, dass nur diejenigen Beobachtungswerte (Zeilen) selektiert werden, für welche eine vorgegebene Bedingung erfüllt ist. So werden im Beispiel nur die Beobachtungswerte der männlichen Personen ausgegeben. Alternativ hätte man auch schreiben können:

```
> Personen[Personen$Geschlecht=="m",]
```

Diese Art von ***bedingtem Zugriff*** wurde bereits in Abschnitt 2.6, S.53f, besprochen.

Alle Personen sollen ausgewählt werden, die größer als 180 cm sind.

```
> subset(Personen,Größe>180)
    Name Geschlecht Größe Gewicht
```

```
1  Anton           m   182      80
3 Caesar           m   189      92
```

Zusätzlich lässt sich die Ausgabe der Variablen (Spalten) noch auf eine vorgegebene Auswahl mit Hilfe der Option `select` beschränken. Dabei wird eine interessierende Variable oder ein Vektor interessierender Variablen spezifiziert.

```
> subset(Personen,Geschlecht=="m",select=Name)
    Name
1  Anton
3 Caesar
5   Emil
```

Hier werden die Namen der männlichen Personen als Ergebnis ausgeben.

```
> subset(Personen,Größe>180,select=c(Größe,Gewicht))
  Größe Gewicht
1   182      80
3   189      92
```

Es werden die Größe und das Gewicht derjenigen Personen extrahiert, die größer als 180 cm sind. Alternativ hätte man auch bspw.

```
> Personen[Personen$Größe>180,c(3,4)]
```

schreiben können.

Die Ausgabe von `subset()` selbst ist wiederum ein Data Frame. Es ist also ohne Weiteres möglich, mit solchen Teilausgaben von Datensätzen neue, d.h. insbesondere kleinere, Datensätze zu erzeugen.

```
> Männer=subset(Personen,Geschlecht=="m")
> attach(Männer)
> rm(Größe)  # falls Objekt "Größe" im Workspace
> Größe
[1] 182 189 180

> attach(Personen)
> Größe
[1] 182 174 189 165 180
```

Die letzten Eingaben des obigen Beispiels veranschaulichen was passiert, falls *Data Frames* mit identischen Vektornamen *„attached“* werden. Es wird stets der Vektor desjenigen *Data Frames* ausgegeben, auf den zuletzt der `attach()`-Befehl angewendet wurde (vgl. die Bemerkungen am Ende von Abschnitt 2.5, S. 49).

2.7 Import und Export von Daten und Objekten

Daten werden bei ihrer Erhebung häufig in eine leere Tabelle eines Textverarbeitungs- oder Tabellenkalkulationsprogramms manuell eingetragen. Daraus ergibt sich dann regelmäßig das Problem, wie die Daten nach *R* importiert werden können, um sie dort statistisch verarbeiten

und analysieren zu können. Im Folgenden betrachten wir einige ausgewählte Möglichkeiten für den Import von Daten nach *R*.

Import und Export von Daten aus Textdateien

Zum Einlesen von Text-Dateien kann die *R*-Funktion

`read.table(file,header=FALSE,dec=".",sep=" ",...)`.

verwendet werden. Dem ersten Argument `file` wird der Dateiname der einzulesenden Text-Datei zugeordnet. Der Wert ist ein String und damit in Anführungszeichen, `".."`, zu setzen. Befindet sich die Datei im aktuellen Arbeitsverzeichnis, dann reicht die Angabe des Dateinamens, ansonsten muss der Pfad mit angegeben werden, s. Beispiel 2.1 unten.

Falls die erste Zeile der Textdatei die Bezeichner der einzulesenden Variablen enthält, dann ist das optionale Argument `header=TRUE` zu setzen. Standardmäßig wird davon ausgegangen, dass ab der ersten einzulesenenden Zeile Daten auftreten und keine Variablennamen angegeben sind (`header=FALSE`). In der Textdatei sind die verschiedenen Werte mit einem speziellen Zeichen `sep` zu trennen. Standardmäßig ist das ein Leerzeichen (`sep=" "`). In *CSV*-Dateien (*comma separated values*) ist dies aber z.B. ein Komma (`sep=","`); in *CSV*-Dateien des deutschen Sprachraums ist es ein Semikolon (`sep=";"`), da Kommas bereits zum Abtrennen von Nachkommastellen in der Zahlendarstellung verwendet werden. Der optionale Parameter `dec` gibt an, mit welchem Symbol bei Fließkommazahlen Vor- und Nachkommastellen getrennt werden. Im deutschsprachigen Raum ist es üblich, die Vor- und Nachkommastellen mit einem Komma zu trennen (`dec=","`); im englischsprachigen Raum verwendet man dagegen einen Punkt (`dec="."`).

Falls es bei Anwendung der Option `header=TRUE` in der zu lesenden Datei einen Variablenbezeichner weniger gibt als Spalten für Beobachtungsdaten, dann wird die erste Spalte als Bezeichner für die Datenreihen interpretiert, vgl. auch Abbildung 2.7.2, S.61, zu Beispiel 2.2.

Mit dem Einlesen von Daten, werden die Merkmalsausprägungen in eine von zwei Kategorien eingeteilt: solche, die sich mit Zahlen ausdrücken lassen, und solche, bei denen das nicht der Fall ist. Letztere werden als Bezeichner für die Ausprägungen von qualitativen Merkmalen verwendet und auch als *Faktoren* (*factor*) bezeichnet.

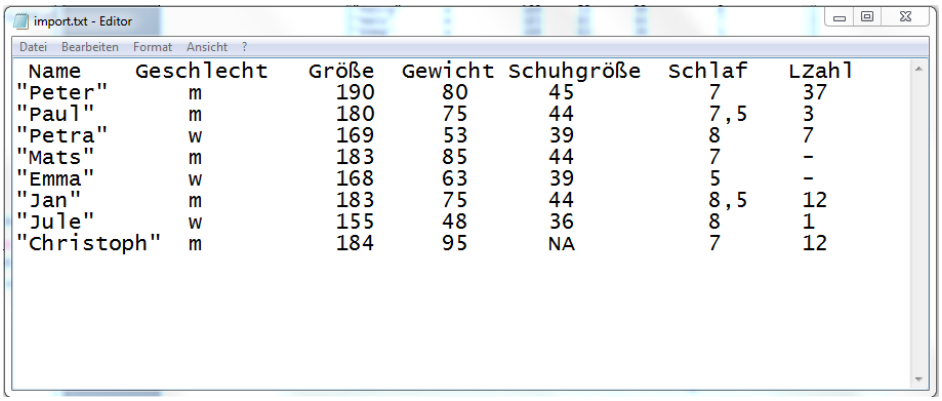
• **Beispiel 2.1** • Zur Demonstration soll die Textdatei *import.txt* aus Abbildung 2.7.1 eingelesen werden. Um die Übersicht zu erhöhen, wurden die Spalten ausgerichtet. Für das Einlesen selbst ist das aber nicht notwendig.

Mit

```
> dat= read.table(file="C:/RData/import.txt",header=T,dec=",")
```

wird in *R* ein *Data Frame* generiert, der die Variablen `Name`, `Geschlecht`, `Größe`, `Gewicht`, `Schuhgröße`, `Schlaf` und `LZahl` enthält. Dabei soll `Schlaf` für die durchschnittliche Schlaf-

Abbildung 2.7.1: Zu importierende Textdatei



dauer und `LZahl` für die Lieblingszahl stehen. Da die Fließkommazahlen im deutschen Zahlenformat angegeben sind, wurde der optionale Parameter `dec=","` gesetzt. Zum Trennen der Einträge wurde das Standardsymbol, das Leerzeichen, verwendet. Daher ist es nicht notwendig `sep=" "` beim Befehlsaufruf noch einmal anzugeben.

```
> dat
       Name Geschlecht Größe Gewicht Schuhgröße Schlaf LZahl
1     Peter          m   190      80         45   7.0    37
2      Paul          m   180      75         44   7.5     3
3     Petra          w   169      53         39   8.0     7
4      Mats          m   183      85         44   7.0     -
5      Emma          w   168      63         39   5.0     -
6       Jan          m   183      75         44   8.5    12
7      Jule          w   155      48         36   8.0     1
8 Christoph          m   184      95         NA   7.0    12
```

Die Merkmalsausprägungen von `Name` und `Geschlecht` sind Zeichenketten. Dabei ist es unerheblich, ob in der Textdatei die Zeichenketten in Anführungszeichen (`" "`) gesetzt wurden oder nicht. In *R* werden die Daten zunächst generell als Zeichenketten eingelesen. Danach versucht *R*, die Zeichenketten in Zahlen umzuwandeln. Gelingt dies für ein oder mehrere Einträge nicht, werden alle Einträge der entsprechenden Variable wie Zeichenketten bzw. als Ausprägungen von Faktoren behandelt.

Wir wählen die Variablen `Geschlecht`, `Größe`, `Schuhgröße` und `LZahl` aus und lassen uns mit `summary( )` eine Zusammenfassung darstellen.

```
> summary(subset(dat,select=c(Geschlecht,Größe,
+                             Schuhgröße,LZahl)))
 Geschlecht      Größe          Schuhgröße     LZahl
 m:5        Min.   :155.0   Min.   :36.00   - :2
 w:3        1st Qu.:168.8   1st Qu.:39.00   1 :1
            Median :181.5   Median :44.00   12:2
            Mean   :176.5   Mean   :41.57   3 :1
            3rd Qu.:183.2   3rd Qu.:44.00   37:1
            Max.   :190.0   Max.   :45.00   7 :1
                            NA's   :1
```

Die Zusammenfassung zeigt, dass die Variablen `Größe` und `Schuhgröße` bspw. als quantitativ und die Variablen `Geschlecht` und `LZahl` als qualitativ interpretiert werden. Zur Kennzeichnung, dass die Schuhgröße von Merkmalsträger *Christoph* nicht bekannt ist, wird dabei `NA` (*not available*) verwendet. In der Zusammenfassung von `Schuhgröße` ist dementsprechend ein `NA` aufgeführt. Demgegenüber ist `'-'` kein gültiges Symbol für fehlende Daten. Da `'-'` weder eine Zahl noch ein gültiges Symbol für einen fehlenden Wert ist, interpretiert *R* `'-'` als Zeichenkette. Damit wird `LZahl` insgesamt als qualitatives Merkmal aufgefasst.

```
> dat$Schuhgröße[1]+1
[1] 46
>   dat$LZahl[1]+1
[1] NA
Warning message:
In Ops.factor(dat$LZahl[1], 1) : ‘+’ not meaningful for factors
```

Dementsprechend liefert `dat$LZahl[1]+1` kein gültiges Ergebnis.

Zum Speichern von *Data Frames* in Text-Dateien kann die *R*-Funktion

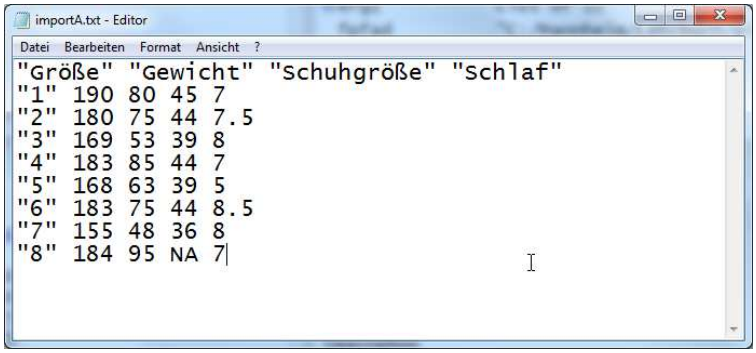
```
write.table(x, file, append=FALSE, dec=".", sep=" ",...)
```

verwendet werden. `x` ist in diesem Fall der *Data Frame*, der gespeichert werden soll, und `file` der Dateiname (ggf. mit Pfad), unter dem `x` gespeichert werden soll. Die Variablenbezeichner des *Data Frame* werden immer mitgespeichert. Das entspricht einer stets gesetzten Option `header=T`. Die optionalen Argumente `dec` und `sep` werden in gleicher Art und Weise wie bei `read.table()` verwendet. Mit `append=TRUE` kann man die Inhalte von `x` an eine bestehende Datei anhängen. Andernfalls wird die Datei einfach überschrieben.

• **Beispiel 2.2** • Angenommen, für die weitere statistische Auswertung sind nur die numerischen Variablen der Spalte 3 bis 6 relevant. Dann kann man diese folgendermaßen in einer Textdatei speichern:

```
> write.table(dat[3:6],file="C:/RData/importA.txt")
```

Abbildung 2.7.2: Exportierte Textdatei



Das Ergebnis findet sich in Abbildung 2.7.2. Man beachte, dass eine zusätzliche Spalte mit den Einträgen `"1"` bis `"8"` als Zeilenbezeichner eingeführt wurde. Wir haben also jetzt einen

Variablenbezeichner weniger als Spalten in der Datendatei. Dies wurde bei der Besprechung von `read.table()` erwähnt und wird beim Lesen entsprechend berücksichtigt.

• **Beispiel 2.3** • Vektoren und Matrizen können auch direkt in Textdateien geschrieben werden. Sie werden dann zuvor von *R* in *Data Frames* umgewandelt.

```
> A=matrix(1:20,5,4)
> write.table(A,"test.txt")
```

Die Anweisungen speichern den Inhalt der Matrix *A* in der Textdatei *test.txt*. Als generische Variablenbezeichner für die vier Spalten werden dazu `V1` bis `V4` eingeführt.

Zum Importieren und Exportieren von *CSV*-Dateien stehen die Funktionen `read.csv()`, `read.csv2()`, `write.csv()` und `write.csv2()` zur Verfügung. Im Wesentlichen sind das nur Spezialfälle der bereits besprochenen Funktionen, in denen gewisse Voreinstellungen, z.B. `header=TRUE`, gesetzt wurden. In `read.csv()` und `write.csv()` sind `dec="."` und `sep=","`, während in `read.csv2()` und `write.csv2()` `dec=","` und `sep=";"` gesetzt sind.

Import und Export von Daten aus Excel-Dateien

Es gibt verschiedene Funktionen, die es ermöglichen, auch *Excel*-Dateien direkt zu importieren. Die Funktion `read.xlsx()` soll hier als ein Beispiel kurz vorgestellt werden. Die Funktion stammt aus dem *xlsx*-Packet. Dieses ist zunächst zu installieren und zu laden, vgl. Abschnitt 1.3.1, S. 22ff.

```
> install.packages("xlsx")
> library(xlsx)
```

Zum Einlesen von *Excel*-Dateien wird die *R*-Funktion

`read.xlsx(file,sheetIndex,...)`.

verwendet. `file` steht hierbei für den Namen der Datei. Mittels `sheetIndex` kann man die *Excel*-Tabelle auswählen. `sheetIndex=1` wählt bspw. die erste Tabelle aus. Weitere Informationen zu dieser Funktionen findet man in der *R*-Hilfe.

• **Beispiel 2.4** • Gegeben sei die *Excel*-Datei aus Abbildung 2.7.3. Das sind die gleichen Daten wie im Beispiel aus dem vorherigen Abschnitt. Da Umlaute beim Einlesen zu Schwierigkeiten führen können, wurde anstelle von *Größe* hier *Goesse* etc. geschrieben.

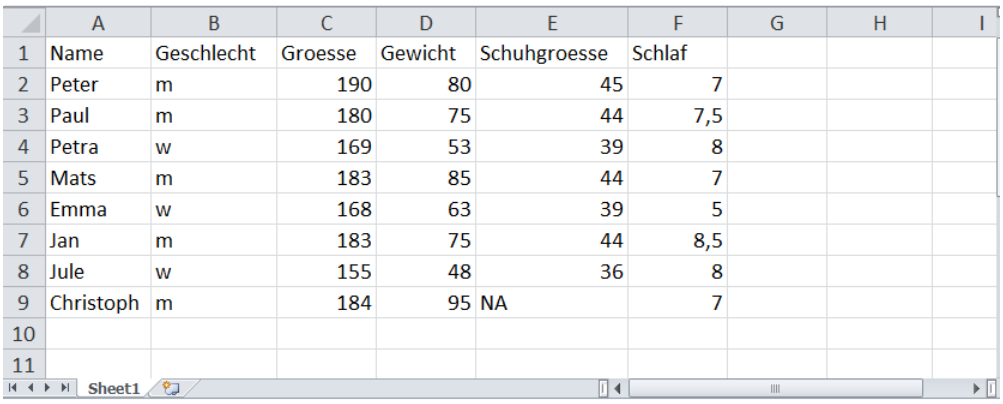
```
> datx = read.xlsx("C:/RData/import.xlsx",sheetIndex=1)
> datx
```

Wir erhalten das gleiche Ergebnis wie beim dem Einlesen der Daten aus einer Text-Datei, vgl. S. 60.

Zum Exportieren von *data.frames* in Form von *Excel*-Dateien kann die *R*-Funktion

`write.xlsx(x,file,sheetName="Sheet1",append=FALSE,...)`.

Abbildung 2.7.3: Zu importierende *Excel*-Datei



	A	B	C	D	E	F	G	H	I
1	Name	Geschlecht	Groesse	Gewicht	Schuhgroesse	Schlaf			
2	Peter	m	190	80	45	7			
3	Paul	m	180	75	44	7,5			
4	Petra	w	169	53	39	8			
5	Mats	m	183	85	44	7			
6	Emma	w	168	63	39	5			
7	Jan	m	183	75	44	8,5			
8	Jule	w	155	48	36	8			
9	Christoph	m	184	95	NA	7			
10									
11									

Sheet1

verwendet werden. `x` ist der zu exportierende *Data Frame* und `file` der Dateiname, unter dem der *Data Frame* gespeichert werden soll. Weitere Details findet man in der *R*-Hilfe.

• **Beispiel 2.5** • Zum Exportieren der Spalten 3 bis 6 von `datx` in die *Excel*-Datei *importA.xlxs* schreiben wir:

```
> write.xlsx(datx[,3:6],"importA.xlsx")
```

Import und Export von Objekten

Stellen wir uns vor, wir haben recht lange mit einer *R*-Version gearbeitet und dabei sehr viele Objekte erstellt. Diese Objekte möchten wir natürlich auch mit einer aktuelleren *R*-Version nutzen können. Wir möchten diese nicht verlieren oder nochmals neu erzeugen müssen. Es könnte auch sein, dass wir bestimmte Objekte, wie z.B. *Data Frames* oder Funktionen einem anderen *R*-Benutzer zur Verfügung stellen möchten. Wie können wir also am leichtesten *R*-Objekte austauschen, exportieren und anschließend importieren?

Zum Exportieren von *R*-Objekten kann die Funktion

```
save(...,file,...)
```

verwendet werden. `...` steht für ein oder mehrere durch Komma getrennte *R*-Objekte. Diese werden in der Datei mit Namen `file` in einem *R*-internen Format gespeichert. Der Vorteil gegenüber `write.table()` ist, dass beliebige und auch unterschiedliche Objekte in einer Datei gespeichert werden können. Mit Hilfe von

```
> save(list=ls(),file="alleObjekte")
```

werden bspw. alle aktuell im Workspace vorhandenen Objekte, s.a. `ls()`, in der Datei *alleObjekte* gespeichert.

Gelesen werden die Daten dann mit

```
load(file,...).
```

Die in der Datei `file` gespeicherten *R*-Objekte stehen dann wieder zur Verfügung.

• **Beispiel 2.6** • Zu Demonstrationszwecken werden mehrere Objekte in der Datei *tmp* gespeichert, gelöscht und dann neu geladen.

```
> x=1:10; A=matrix(1:20,5,4)
> save(dat,datx,x,A,file="tmp")
> rm(dat,datx,x,A)
> load("tmp")
> x
 [1]  1  2  3  4  5  6  7  8  9 10
```

Die Werte von `x` stehen trotz des Löschens durch `rm( )` nach dem Laden mit `load( )` wieder zur Verfügung.

3 Ausgewählte deskriptive Analysemethoden

3.1 Rechnerische Methoden

3.1.1 Ermittlung von Häufigkeitsverteilungen

• Eindimensionale Verteilungen • Die einfachste deskriptive Analysemethode besteht darin, die Häufigkeiten von Ausprägungen zu bestimmen. Die Standardfunktion hierfür ist

`table`(x)

Wir betrachten dazu als Beispiel folgende Ein- und Ausgaben:

```
> x=c(1,1,5,2,1,5,2,3,10,2)
> length(x)
[1] 10
> table(x)
x
 1  2  3  5 10
 3  3  1  2  1
> prop.table(table(x))
x
  1   2   3   5  10
0.3 0.3 0.1 0.2 0.1
```

Mit `length()` lässt sich zunächst die Länge des Datenvektors, sprich die Anzahl der Beobachtungswerte, abfragen. Über `table()` wird dann die ***absolute Häufigkeitsverteilung*** ermittelt. Das Ausgabeobjekt ist ein `table`-Objekt. Zu jeder vorkommenden Ausprägung korrespondiert die jeweilige absolute Häufigkeit ihres Auftretens. Wendet man darauf dann wiederum den Befehl

`prop.table`(tabelle)

an, erhält man die ***relative Häufigkeitsverteilung***. Es sollte hier angemerkt werden, dass die direkte Anwendung von `prop.table()` auf `x` das folgende Resultat liefert:

```
> prop.table(x)
 [1] 0.03125 0.03125 0.15625 0.06250 0.03125
 [6] 0.15625 0.06250 0.09375 0.31250 0.06250
```

Der Datenvektor würde als Vektor absoluter Häufigkeiten interpretiert werden. Da die Gesamtsumme der Vektoreinträge für das Beispiel 32 beträgt, entspräche 1 gerade dem Anteil 1/32, also 0.03125.

• Zweidimensionale Verteilungen • Der Befehl

`table`(x,y)

ist auch für zweidimensionale Auszählungen (absolute und relative) geeignet. Wir betrachten dazu als Beispiel folgende Ein- und Ausgaben:

```
> Geschlecht=c(0,0,0,0,1,1,1,1,0,0,1,1,1,1,0,0)
> Raucherstatus=c(0,1,2,0,1,2,0,0,1,1,2,0,1,0,2,0)
> table(Geschlecht,Raucherstatus)
          Raucherstatus
Geschlecht 0 1 2
         0 3 3 2
         1 4 2 2
> Tabelle=table(Geschlecht,Raucherstatus)
> prop.table(Tabelle)
          Raucherstatus
Geschlecht      0      1      2
         0 0.1875 0.1875 0.1250
         1 0.2500 0.1250 0.1250
```

Der Zwischenschritt der Zuweisung zu einem Objekt `Tabelle` ist an sich nicht notwendig, erhöht jedoch die Übersichtlichkeit.

• **Bedingte Verteilungen** • Die bedingten Häufigkeitsverteilungen einer vorgegebenen ***Kontingenztabelle***, sprich die ***Zeilen-*** und ***Spaltenverteilungen***, erhält man ebenfalls über `prop.table()`. Dazu wird

`prop.table`(tabelle,index).

auf ein bereits vorliegendes `table`-Objekt *tabelle* angewendet. Die Spezifikation des 2. Arguments *index* mit 1 deklariert die Zeilenverteilungen und mit 2 die Spaltenverteilungen.

```
> Tabelle
          Raucherstatus
Geschlecht 0 1 2
         0 3 3 2
         1 4 2 2
> prop.table(Tabelle,1)
          Raucherstatus
Geschlecht     0     1     2
         0 0.375 0.375 0.250
         1 0.500 0.250 0.250
> prop.table(Tabelle,2)
          Raucherstatus
Geschlecht         0         1         2
         0 0.4285714 0.6000000 0.5000000
         1 0.5714286 0.4000000 0.5000000
```

3.1.2 Lage- und Streuungskennwerte

• **Arithmetisches Mittel, Median und Quantile** • Mit

`mean`(x), `median`(x) und `quantile`(x,alpha,type=2)

lassen sich das arithmetische Mittel, der Median und das alpha-Quantil des Datenvektors `x` berechnen. Damit die Berechnung der Quantile mit der Lehrbuch-Konvention von Stocker und Steinke [2017] übereinstimmt, muss die Option `type=2` bei der Verwendung von

`quantile()` gewählt werden.

Anhand der folgenden Ein- und Ausgaben kann nachvollzogen werden, wie sich die wichtigsten Lagemaße mit *R* ermitteln lassen.

```
> x=c(1,1,5,2,1,5,2,3,10,2)
> length(x)
[1] 10
> sum(x)
[1] 32
> mean(x)
[1] 3.2
> median(x)
[1] 2
> sort(x)
 [1]  1  1  1  2  2  2  3  5  5 10
```

`sort`(x) sortiert die Einträge von `x`.

```
> quantile(x,0.3,type=2)
30%
1.5
> quantile(x,0.35,type=2)
35%
  2
> quantile(x,type=2)
  0%  25%  50%  75% 100%
   1    1    2    5   10
```

Ohne die Angabe eines Quantilsniveaus `alpha` werden die empirischen Quartile und der Median zu `x` bestimmt. Es können auch mehrere Quantile gleichzeitig ermittelt werden.

```
> quantile(x,probs=seq(0.1,0.9,by=0.1),type=2)
10% 20% 30% 40% 50% 60% 70% 80% 90%
1.0 1.0 2.0 2.0 2.0 2.5 5.0 5.0 7.5
```

• **Varianz und Standardabweichung** • Anhand der folgenden Ein- und Ausgaben kann nachvollzogen werden, wie sich die ***Varianz*** und ***Standardabweichung*** sowohl „von Hand“ als auch mittels der Funktionen `var()` und `sd()` berechnen lassen.

```
> x=c(1,1,5,2,1,5,2,3,10,2)
> x
 [1]  1  1  5  2  1  5  2  3 10  2
> mean((x-mean(x))^2)
[1] 7.16
> mean(x^2)-mean(x)^2
[1] 7.16
```

Dabei steht

$$\texttt{mean((x-mean(x))\^{}2)} \quad \text{für} \quad \tilde{s}^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2,$$

also für die originäre Varianzformel für Beobachtungsdaten, und

$$\texttt{mean(x\^{}2)-mean(x)\^{}2} \quad \text{für} \quad \tilde{s}^2 = \frac{1}{n}\sum_{i=1}^{n}x_i^2 - \bar{x}^2,$$

also für die entsprechende Verschiebungsformel. Man beachte hier die Besonderheiten der *R*-Arithmetik (vgl. Abschnitt 2.2, S.30ff), die eine effiziente Verrechnung von Vektoren mit Skalaren ermöglicht. So stellt sich die in einzelne Teilschritte zerlegte Berechnung nach der originären Formel wie folgt dar.

```
> x-mean(x)
 [1] -2.2 -2.2  1.8 -1.2 -2.2  1.8 -1.2 -0.2  6.8 -1.2
> (x-mean(x))^2
 [1]  4.84  4.84  3.24  1.44  4.84  3.24  1.44  0.04 46.24  1.44
> mean((x-mean(x))^2)
[1] 7.16
```

Diese entsprechen der Berechnung der einzelnen Terme

$$x_i - \bar{x}, \quad (x_i - \bar{x})^2 \quad \text{und} \quad \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2.$$

Die in *R* verfügbaren Funktionen

$$\texttt{var}(\text{x}) \quad \text{und} \quad \texttt{sd}(\text{x})$$

rechnen jeweils mit Divisor $n-1$ und ermitteln die sog. ***korrigierte Varianz*** bzw. ***korrigierte Standardabweichung*** gemäß

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2 \quad \text{bzw.} \quad s = \sqrt{s^2}.$$

Die Umrechnung von der korrigierten in die nichtkorrigierte Variante erfolgt deshalb über Multiplikation mit dem Vorfaktor

$$\frac{n-1}{n} \quad \text{bzw.} \quad \sqrt{\frac{n-1}{n}}.$$

Im Falle der Varianz ergibt sich dann

```
> var(x)
[1] 7.955556
> 9/10*var(x)
[1] 7.16
```

und im Falle der Standardabweichung:

```
> sqrt(mean(x^2)-mean(x)^2)
[1] 2.675818
> sd(x)
[1] 2.820559
> sqrt(9/10)*sd(x)
[1] 2.675818
```

• **Mittlere absolute Abweichungen** • Die mittleren absoluten Abweichungen sind im *Standardumfang* von *R* nicht implementiert. Die folgenden Ein- und Ausgaben demonstrieren, wie man sie berechnen kann.

```
> x=c(1,1,5,2,1,5,2,3,10,2)
> x
 [1]  1  1  5  2  1  5  2  3 10  2
> median(x)
[1] 2
> x-median(x)
 [1] -1 -1  3  0 -1  3  0  1  8  0
> abs(x-median(x))
 [1] 1 1 3 0 1 3 0 1 8 0
> mean(abs(x-median(x)))
[1] 1.8
> mean(abs(x-mean(x)))
[1] 2.08
```

Dabei steht

$$\texttt{mean(abs(x-median(x)))} \quad \text{für} \quad d = \frac{1}{n}\sum_{i=1}^{n} |x_i - \tilde{x}_{0.5}|,$$

die ***mittlere absolute Abweichung vom Median***, und

$$\texttt{mean(abs(x-mean(x)))} \quad \text{für} \quad d^* = \frac{1}{n}\sum_{i=1}^{n} |x_i - \bar{x}|,$$

die ***mittlere absolute Abweichung vom arithmetischen Mittel***. Hierbei bezeichnet $\tilde{x}_{0.5}$ den Median.

• **MAD** • Setzt man die Ausführungen des vorhergehenden Punktes fort, so berechnet man den ***MAD*** (***Median Absolute Deviation***) „von Hand“ wie folgt:

```
> median(abs(x-median(x)))
[1] 1
```

Das gleiche Resultat erhält man mit

```
> mad(x,constant=1)
[1] 1
```

Die *R*-Funktion `mad(x)` berechnet hingegen ohne das optionale Argument `constant=1`

```
> mad(x)
[1] 1.4826
```

Aus theoretischen Erwägungen heraus wird hier der eigentliche *MAD* noch mit der Konstanten 1.4826 multipliziert, vgl. dazu auch die Hilfe zur Funktion. Im vorliegenden Fall erhält man deshalb $1 \cdot 1.4826 = 1.4826$.

3.1.3 Berechnung von Kontingenz und Korrelation

• **Kontingenzkoeffizient nach Pearson** • Sofern die bedingten Zeilen- bzw. Spaltenverteilungen in einer Kontingenztabelle übereinstimmen, liegt keinerlei (empirische) Abhängigkeit vor. Ist das nicht der Fall, so lässt sich zur Beurteilung der Stärke der Abhängigkeit beispielsweise der *Kontingenzkoeffizient nach Pearson* berechnen. Als Beispiel betrachten wir

hierfür nachfolgende Kontingenztabelle für den Zusammenhang von Geschlecht und Rauchverhalten. Die Tabelle deckt sich mit Tabelle 5.1.9 aus Stocker und Steinke [2017].

Kontingenztabelle mit tatsächlichen und zu erwartenden Häufigkeiten sowie mit den jeweiligen Abweichungen

	Raucher	Gelegenheits-raucher	Nichtraucher	Summe
weiblich	4 (6.4) −2.4	8 (8) 0	28 (25.6) 2.4	40
männlich	12 (9.6) 2.4	12 (12) 0	36 (38.4) −2.4	60
Summe	16	20	64	100

Die Berechnung des (korrigierten) Kontingenzkoeffizienten nach Pearson für eine $(k \times l)$-Kontingenztabelle vollzieht sich in mehreren Schritten. Dabei wird zunächst der ***Koeffizient*** χ^2 berechnet, darauf basierend der ***nichtkorrigierte Kontingenzkoeffizient*** C und schließlich der ***korrigierte Kontingenzkoeffizient*** C_K:

$$\chi^2 = \sum_{i=1}^{k}\sum_{j=1}^{l} \frac{\left(n_{ij} - \frac{n_{i\bullet}n_{\bullet j}}{n}\right)^2}{\frac{n_{i\bullet}n_{\bullet j}}{n}},$$

$$C = \sqrt{\frac{\chi^2}{\chi^2 + n}}, \quad C_K = \sqrt{\frac{M}{M-1}} \cdot C \quad \text{mit } M = \min(k, l).$$

In der Standarddistribution von R gibt es tatsächlich keine Funktion, mit der sich der Kontingenzkoeffizient direkt berechnen lässt. Mithilfe der Funktion

`chisq.test()`,

mit der sich ein χ^2***-Unabhängigkeitstest*** durchführen lässt, vgl. Abschnitt 4.2.3, S.130f, kann jedoch der χ^2-Koeffizient ermittelt werden. Darauf aufbauend lässt sich dann der Kontingenzkoeffizient bestimmen.

```
> m=c(4,12,8,12,28,36)
> m=matrix(m,2,3)
> m
     [,1] [,2] [,3]
[1,]    4    8   28
[2,]   12   12   36
```

Die Daten werden als Häufigkeitsmatrix an die Funktion übergeben.

```
> chisq.test(m)

        Pearson's Chi-squared test
data:  m
X-squared = 1.875, df = 2, p-value = 0.3916
```

```
> res=chisq.test(m)
> names(res)
[1] "statistic" "parameter" "p.value" "method" "data.name"
[6] "observed" "expected"  "residuals" "stdres"
```

Das Ausgabeobjekt von `chisq.test()` ist ein `list`-Objekt, vgl. Abschnitt 2.4, welche den χ^2-Koeffizienten als Teilkomponente (`statistic`) enthält.

```
> X2=unname(res$statistic)
> X2
[1] 1.875
> n=sum(m)
> C=sqrt(X2/(X2+n))
> C
[1] 0.1356647
> CK=sqrt(2)*C
> CK
[1] 0.1918588
```

Demgemäß erhalten wir für das vorliegende Beispiel: $\chi^2 = 1.875$,

$$C = \sqrt{\frac{\chi^2}{\chi^2 + n}} \approx 0.1357, \quad C_K = \sqrt{\frac{M}{M-1}} \cdot C \approx \sqrt{\frac{2}{2-1}} \cdot 0.1357 \approx 0.1919.$$

Darüber hinaus kann man über Listensyntax z.B. auch auf die tatsächlichen und die erwarteten Zellenhäufigkeiten zugreifen.

```
> observed
     [,1] [,2] [,3]
[1,]    4    8   28
[2,]   12   12   36
> res$expected
     [,1] [,2] [,3]
[1,]  6.4    8 25.6
[2,]  9.6   12 38.4
```

• Sonderfall: Kontingenzkoeffizient für (2×2)-Tabellen • Im Falle von (2×2)-Tabellen vereinfacht sich die Berechnung des χ^2-Koeffizienten. Es gilt:

$$\chi^2 = \sum_{i=1}^{k} \sum_{j=1}^{l} \frac{\left(n_{ij} - \frac{n_{i\bullet} n_{\bullet j}}{n}\right)^2}{\frac{n_{i\bullet} n_{\bullet j}}{n}} = \frac{n(n_{11}n_{22} - n_{12}n_{21})^2}{n_{\bullet 1} n_{\bullet 2} n_{1\bullet} n_{2\bullet}}. \tag{3.1}$$

Bei der Verwendung der *R*-Funktion `chisq.test()` ist nun jedoch eine kleine Besonderheit zu beachten. Als Beispiel betrachten wir folgende Tabelle.

Kontingenztabelle mit tatsächlichen und zu erwartenden Häufigkeiten sowie mit den jeweiligen Abweichungen

	Raucher	Gelegenheitsraucher	Summe
weiblich	4	8	12
männlich	12	12	24
Summe	16	20	36

Der χ^2-Koeffizient ergibt sich hier als

$$\chi^2 = \frac{n(n_{11}n_{22} - n_{12}n_{21})^2}{n_{\bullet 1}n_{\bullet 2}n_{1\bullet}n_{2\bullet}} = \frac{36(4 \cdot 12 - 12 \cdot 8)^2}{16 \cdot 20 \cdot 12 \cdot 24} = 0.9.$$

Bei der Berechnung mit *R* über `chisq.test()` erhalten wir hingegen einen anderen Wert.

```
> m=matrix(c(4,12,8,12),2,2)
> m
     [,1] [,2]
[1,]    4    8
[2,]   12   12
> chisq.test(m)

        Pearson's Chi-squared test with Yates'
        continuity correction
data:  m
X-squared = 0.3516, df = 1, p-value = 0.5532
```

Der χ^2-Koeffizient wird mit 0.3516 ausgewiesen. Ein Blick in den Hilfetext zur Funktion verrät, dass im Falle von (2×2)-Tabellen von der sogenannten *Yates-Stetigkeitskorrektur* Gebrauch gemacht wird. Auch in der Ausgabe von `chisq.test()` findet sich hierzu ein Hinweis. Die Gründe hierfür sind wahrscheinlichkeitstheoretischer Natur und sollen hier nicht erläutert werden. Um χ^2 nach Formel (3.1) zu berechnen, muss die Option `correct=FALSE` gewählt werden.

```
> chisq.test(m,correct=FALSE)

        Pearson's Chi-squared test
data:  m
X-squared = 0.9, df = 1, p-value = 0.3428
```

• Korrelationskoeffizient nach Pearson • Zur Berechnung des Korrelationskoeffizienten nach Pearson verwendet man in *R* standardmäßig die Funktion

`cor`(x,y).

Als Beispiel betrachten wir die 10 Beobachtungspaare:

(2, 1), (4, 2), (4, 3), (5, 2), (6, 4), (8, 5), (9, 6), (10, 4), (4, 5), (7, 3).

Mit Eingabe der Daten erhält man dann

```
> x=c(2,4,4,5,6,8,9,10,4,7)
> y=c(1,2,3,2,4,5,6,4,5,3)
> cor(x,y)
[1] 0.6730033
```

Aus didaktischen Gründen sei im Folgenden auch die Berechnung „von Hand“ auf Basis der Korrelationsformel

$$r_{XY} = \frac{\tilde{s}_{XY}}{\tilde{s}_X \tilde{s}_Y} = \frac{\frac{1}{n}\sum_{i=1}^{n} x_i y_i - \bar{x}\bar{y}}{\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2 - \bar{x}^2}\sqrt{\frac{1}{n}\sum_{i=1}^{n} y_i^2 - \bar{y}^2}}$$

vorgestellt. Kovarianz und Varianzen wurden hier jeweils mit entsprechenden Verschiebungsformeln notiert. Bei einer manuellen Berechnung würde man dann etwa eine Arbeitstabelle wie folgt anlegen und daraus die wesentlichen Formelbestandteile als Spaltensummen entnehmen:

i	x_i	y_i	x_i^2	y_i^2	$x_i y_i$
1	2	1	4	1	2
2	4	2	16	4	8
3	4	3	16	9	12
4	5	2	25	4	10
5	6	4	36	16	24
6	8	5	64	25	40
7	9	6	81	36	54
8	10	4	100	16	40
9	4	5	16	25	20
10	7	3	49	9	21
Summe	59	35	407	145	231

Mit $\bar{x} = 59/10 = 5.9$ und $\bar{y} = 35/10 = 3.5$ erhalten wir

$$r_{XY} = \frac{\frac{1}{10} \cdot 231 - 5.9 \cdot .5}{\sqrt{\frac{1}{10} \cdot 407 - 5.9^2} \cdot \sqrt{\frac{1}{10} \cdot 145 - 3.5^2}} \approx \frac{2.45}{\sqrt{5.89 \cdot 2.25}} \approx 0.673.$$

In *R* sieht das Ganze ausführlich dann so aus:

```
> mean(x)
[1] 5.9
> mean(y)
[1] 3.5
> x*y
 [1]  2  8 12 10 24 40 54 40 20 21
> mean(x*y)-mean(x)*mean(y)
[1] 2.45
> covxy=mean(x*y)-mean(x)*mean(y)
> sx2=mean(x^2)-mean(x)^2
> sy2=mean(y^2)-mean(y)^2
> covxy/sqrt(sx2*sy2)
[1] 0.6730033
```

Eine alternative Möglichkeit wäre, die *R*-Funktionen `cov(x,y)` für die ***Kovarianz*** und `var()` für die Varianz oder `sd()` für die Standardabweichung zu nutzen:

```
> cov(x,y)/sqrt(var(x)*var(y))
[1] 0.6730033
> cov(x,y)/(sd(x)*sd(y))
[1] 0.6730033
```

Hierbei ist zu beachten, dass `cov()` ebenso wie `var()` und `sd()` als Divisor $n-1$ verwendet. Damit wird die korrigierte Kovarianz mit der Formel

$$s_{XY} = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})$$

berechnet. Die Umrechnung von der korrigierten in die nichtkorrigierte Variante erfolgt deshalb über Multiplikation mit dem Vorfaktor $(n-1)/n$.

```
> mean(x*y)-mean(x)*mean(y)
[1] 2.45
> cov(x,y)
[1] 2.722222
> 9/10*cov(x,y)
[1] 2.45
```

• **Korrelationskoeffizient nach Spearman** • Zur Berechnung des Korrelationskoeffizienten nach Spearman lassen sich die Befehle `rank()` und `cor()` kombinieren.

Als Beispiel betrachten wir die folgenden 4 Beobachtungspaare:

$$(1,\ 1),\ (2,\ 0.5),\ (4,\ 0.25),\ (8,\ 0.125).$$

Gemäß Abbildung 3.1.1 (linke Grafik) besteht zwischen x- und y-Werten ein streng monoton fallender Zusammenhang. Rechnet man die ursprünglichen Werte in Rangwerte mithilfe von `rank()` um, ergibt sich ein perfekter (negativer) linearer Zusammenhang. Man beachte hierzu die folgenden Ein- und Ausgaben in *R*:

```
> x=c(1,2,4,8)
> y=c(1,0.5,0.25,0.125)
> cor(x,y)
[1] -0.8434783
> rank(x)
[1] 1 2 3 4
> rank(y)
[1] 4 3 2 1
> cor(rank(x),rank(y))    # Korrelation nach Spearman
[1] -1
```

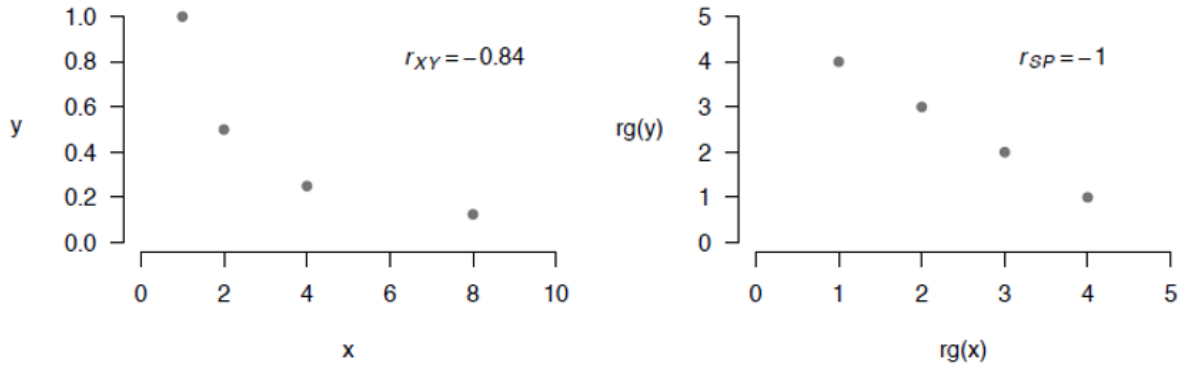
Das gleiche Ergebnis erzielt man mit `cor()` und der Option `method="spearman"`.

```
> cor(x,y,method="spearman")
[1] -1
```

Im Falle sogenannter ***Bindungen*** arbeitet `rank()` mit Bildung von Durchschnittsrängen, also genauso wie es der Lehrbuchkonvention aus Stocker und Steinke [2017] entspricht. Als Beispiel betrachte man folgende Ein- und Ausgaben:

```
> x=c(1,2,3,2,3,3,6,4,5,3)
> sort(x)
 [1] 1 2 2 3 3 3 3 4 5 6
> rank(x)
 [1]  1.0  2.5  5.5  2.5  5.5  5.5 10.0  8.0  9.0  5.5
```

Abbildung 3.1.1: Streudiagramm von Ursprungswerten und zugehörigen Rangwerten



Da der Wert 2 insgesamt 2 Mal auftritt wird anstelle der Ränge 2 und 3 der Durchschnittsrang 2.5 vergeben. Da der Wert 3 insgesamt 4 Mal auftritt wird anstelle der Ränge 4, 5, 6 und 7 der Durchschnittsrang 5.5 vergeben.

• **Kovarianz- und Korrelationsmatrizen** • Mit den Befehlen

`cov()` und `cor()`

können für zwei vorgegebene Vektoren `x` und `y` Kovarianz und Korrelation über `cov(x,y)` bzw. `cor(x,y)` berechnet werden. Die beiden Befehle lassen sich alternativ jedoch auch auf `matrix`- oder `data.frame`-Objekte anwenden. Im Folgenden sei dies anhand eines 3-dimensionalen Data Frames mit 7 Beobachtungen exemplarisch ausgeführt. Besonderheiten in der Verwendung der beiden Befehle treten im Falle fehlender Beobachtungen (NA's) auf. Hierbei gilt es dann die unterschiedlichen Verfahrensvarianten zu kennen.

```
> v1=c(5,1,4,NA,5,6,3)
> v2=c(5,NA,2,NA,NA,8,10)
> v3=c(NA,3,2,3,1,3,6)
> v123=data.frame(v1,v2,v3)
```

Der generierte Datensatz weist ingesamt nur 3 vollständige Beobachtungen (vollständig in 3-dimensionalem Sinne) auf. Dies sind die Beobachtungen Nr. 3, 6 und 7:

```
> v123
  v1 v2 v3
1  5  5 NA
2  1 NA  3
3  4  2  2
4 NA NA  3
5  5 NA  1
6  6  8  3
7  3 10  6
```

Sobald eine vorgegebene Datenmatrix auch nur einen einzigen fehlenden Wert aufweist, geben `cov()` und `cor()` lediglich Matrizen mit NA's aus.

```
> cov(v123)
   v1 v2 v3
v1 NA NA NA
v2 NA NA NA
v3 NA NA NA
```

Mit der Spezifikation `use=complete.obs` werden Kovarianz- bzw. Korrelationsmatrix alleine auf Basis der komplett vollständigen Beobachtungswerte, hier also Nr. 3, 6 und 7, berechnet.

```
> cov(v123,use="complete.obs")
           v1         v2         v3
v1  2.3333333 -0.3333333 -1.833333
v2 -0.3333333 17.3333333  7.333333
v3 -1.8333333  7.3333333  4.333333
```

Wie man anhand der Datenmatrix sieht, wären jedoch etwa zur Berechnung der Kovarianz (bzw. Korrelation) zwischen `v1` und `v2` mehr als nur 3 Beobachtungen verfügbar. Für dieses 2-dimensionale Berechnungsproblem ließen sich die Beobachtungen Nr. 1, 3, 6 und 7 nutzen. Bei der Berechnung der Kovarianz (bzw. Korrelation) zwischen `v1` und `v3` ließen sich die Beobachtungen Nr. 2, 3, 5, 6 und 7 nutzen. Möchte man also die paarweisen Kovarianzen bzw. Korrelation auf Basis der jeweils vollständigen 2-dimensionalen Beobachtungen berechnen, nutzt man die Spezifikation `use=pairwise.complete.obs`. Zu beachten ist hierbei, dass die einzelnen Kovarianzen bzw. Korrelationen dabei möglicherweise auf unterschiedlichen und unterschiedlich vielen Beobachtungswerten beruhen.

```
> cov(v123,use="pairwise.complete.obs")
      v1        v2        v3
v1  3.20 -0.500000 -1.250000
v2 -0.50 12.250000  7.333333
v3 -1.25  7.333333  2.800000
> cor(v123,use="pairwise.complete.obs")
           v1         v2         v3
v1  1.0000000 -0.1106567 -0.3473563
v2 -0.1106567  1.0000000  0.8461538
v3 -0.3473563  0.8461538  1.0000000
```

3.1.4 Lineare Regression

• Regression nach der KQ-Methode • Zur Berechnung einer ***KQ-Gerade*** verwendet man in *R* standardmäßig die Funktion

`lm`(y∼x)

(für linear model). Als Beispiel betrachten wir die 5 Beobachtungspaare:

(1, 1), (2, 2), (3, 1), (4, 3), (5, 2).

Mit Eingabe der Daten erhält man dann

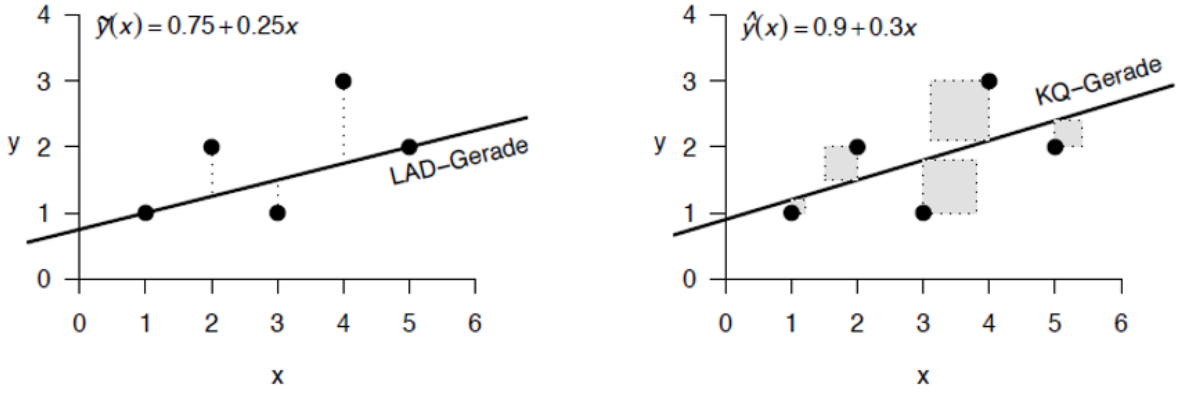
```
> x=c(1,2,3,4,5)
> y=c(1,2,1,3,2)
> lm(y~x)
```

als Ergebnis:

```
Call:
lm(formula = y ~ x)
Coefficients:
(Intercept)            x
        0.9          0.3
```

Die rechte Grafik von Abbildung 3.1.2 zeigt die errechnete KQ-Gerade. Hinweise zum Erstellen solcher Grafiken finden sich in Abschnitt 3.2.3, S.107ff.

Abbildung 3.1.2: Regression nach KQ- und LAD-Methode



Die direkte Berechnung der KQ-Koeffizienten für die Steigung $\hat{\beta}_1$ und den Achsenabschnitt $\hat{\beta}_0$ erfolgt auf Basis der Formeln

$$\hat{b}_1 = \frac{s_{XY}}{s_X^2} = \frac{\tilde{s}_{XY}}{\tilde{s}_X^2} = \frac{\frac{1}{n}\sum_{i=1}^n x_i y_i - \bar{x}\bar{y}}{\frac{1}{n}\sum_{i=1}^n x_i^2 - \bar{x}^2} \quad \text{und} \quad \hat{b}_0 = \bar{y} - \hat{b}_1 \bar{x}.$$

Kovarianz und Varianz wurden hier jeweils mit entsprechenden Verschiebungsformeln notiert. Bei einer rein händischen Berechnung würde man dann etwa eine Arbeitstabelle wie folgt anlegen und daraus die wesentlichen Formelbestandteile als Spaltensummen entnehmen:

i	x_i	y_i	$x_i y_i$	x_i^2	$\hat{y}_i$	$\hat{u}_i$	$\lvert\hat{u}_i\rvert$	$\hat{u}_i^2$
1	1	1	1	1	1.2	−0.2	0.2	0.04
2	2	2	4	4	1.5	0.5	0.5	0.25
3	3	1	3	9	1.8	−0.8	0.8	0.64
4	4	3	12	16	2.1	0.9	0.9	0.81
5	5	2	10	25	2.4	−0.4	0.4	0.16
Summe	15	9	30	55	9	0	2.8	1.9

Die letzten 4 Spalten werden hier im engeren Sinne nicht benötigt. Sie wurden zur späteren Veranschaulichung verschiedener Aspekte, wie etwa auch der Unterschiede von KQ- und LAD-Methode, mit aufgeführt. Mit $\bar{x} = 15/5 = 3$ und $\bar{y} = 9/5 = 1.8$ erhalten wir

$$\hat{b}_1 = \frac{\frac{1}{n}\sum_{i=1}^{n} x_i y_i - \bar{x}\bar{y}}{\frac{1}{n}\sum_{i=1}^{n} x_i^2 - \bar{x}^2} = \frac{30/5 - 3 \cdot 1.8}{55/5 - 3^2} = 0.3 \quad \text{und}$$
$$\hat{b}_0 = \bar{y} - \hat{b}_1 \bar{x} = 1.8 - 0.3 \cdot 3 = 0.9.$$

Gefittete Werte und Residuen ergeben sich dann durch Einsetzen der x-Werte in die KQ-Gerade

$$\hat{y}(x) = \hat{b}_0 + \hat{b}_1\, x = 0.9 + 0.3\, x$$

und anschließendes Bestimmen der Differenz zwischen tatsächlichen und gefitteten y-Werten:

$$\hat{u}_i = y_i - \hat{y}_i = y_i - \hat{y}(x_i).$$

In *R* sieht das Ganze ausführlich dann so aus:

```
> b1dach=cov(x,y)/var(x)
> b0dach=mean(y)-b1dach*mean(x)
> b0dach
[1] 0.9
> b1dach
[1] 0.3
> ydach=0.9+0.3*x
> ydach
[1] 1.2 1.5 1.8 2.1 2.4
> udach=y-ydach
> udach
[1] -0.2  0.5 -0.8  0.9 -0.4
```

Die folgenden beiden Gleichheiten der KQ-Regression

$$\sum_{i=1}^{n} \hat{y}_i = \sum_{i=1}^{n} y_i \quad \text{und} \quad \sum_{i=1}^{n} \hat{u}_i = 0$$

lassen sich für die Beispieldaten leicht überprüfen:

```
> sum(ydach)
[1] 9
> sum(y)
[1] 9
> sum(udach)
[1] 2.220446e-16
```

Hierbei ist anzumerken, dass die Ausgabe $2.220446e{-}16$ (für $2.220446 \cdot 10^{-16}$) ein numerisches Artefakt darstellt, da *R* nicht beliebig genau Nachkommastellen berechnen kann. Bequemer erhält man all diese Größen über die Verwendung der Ergebnisse, die `lm()` bereitstellt. Das Ausgabeobjekt von `lm()` ist ein `list`-Objekt, auf dessen Teilkomponenten entsprechend mit der Listensyntax zugegriffen werden kann.

```
> reslm=lm(y~x)
> names(reslm)
 [1] "coefficients"  "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"        "qr"            "df.residual"
 [9] "xlevels"       "call"          "terms"         "model"
> reslm$coefficients
(Intercept)           x
        0.9         0.3
> reslm$fitted.values
  1   2   3   4   5
1.2 1.5 1.8 2.1 2.4
> reslm$residuals
   1    2    3    4    5
-0.2  0.5 -0.8  0.9 -0.4
```

• **Regression nach der LAD-Methode** • Zur Berechnung einer ***LAD-Gerade*** kann man in *R* die Funktion

rq(y~x)

(für *quantile regression*) aus dem Paket `quantreg` verwenden Als Beispiel betrachten wir die gleichen 5 Beobachtungspaare wie im vorhergehenden Punkt.

```
> library(quantreg)
> rq(y~x)

Call:
rq(formula = y ~ x)
Coefficients:
(Intercept)           x
       0.75        0.25

Degrees of freedom: 5 total; 3 residual
```

Die linke Grafik von Abbildung 3.1.2 auf S.77 zeigt die errechnete LAD-Gerade. Hinweise zum Erstellen solcher Grafiken finden sich in Abschnitt 3.2.3, S.107ff.

Eine Berechnung „von Hand“ scheidet aus, da hierfür keine einfachen Formeln verfügbar sind. Gefittete Werte und Residuen ergeben sich dann durch Einsetzen der x-Werte in die LAD-Gerade

$$\tilde{y}(x) = \tilde{b}_0 + \tilde{b}_1\bar{x} = 0.75 + 0.25x$$

und anschließendes Bestimmen der Differenz zwischen tatsächlichen und gefitteten y-Werten.

In *R* sieht das Ganze ausführlich dann so aus:

```
> yschlange=0.75+0.25*x
> yschlange                   LAD-gefittete Werte von Hand
[1] 1.00 1.25 1.50 1.75 2.00
> uschlange=y-yschlange       LAD-Residuen von Hand
> uschlange
[1]  0.00  0.75 -0.50  1.25  0.00
```

Ähnlich wie bei `lm()` erhält man all diese Größen auch über die Verwendung der Ausgabe

von `rq()`. Das Ausgabeobjekt von `rq()` ist ein `list`-Objekt, auf dessen Teilkomponenten entsprechend mit Listensyntax zugegriffen werden kann.

```
> resrq=rq(y~x)   resrq ist jetzt eine Liste
> names(resrq)    Namen der Listenkomponenten
 [1] "coefficients"  "x"         "y"       "residuals"   "dual"
 [6] "fitted.values" "formula"   "terms"   "xlevels"     "call"
[11] "tau"           "rho"       "method" "model"
> attach(resrq)
> coefficients
(Intercept)           x
       0.75        0.25
> fitted.values
   1    2    3    4    5
1.00 1.25 1.50 1.75 2.00
> residuals
    1     2     3     4     5
 0.00  0.75 -0.50  1.25  0.00
```

Die typischen Eigenschaften einer KQ-Regression sind nun i.d.R. nicht erfüllt. Im vorliegenden Fall etwa erhalten wir:

```
> sum(fitted.values)
[1] 7.5
> sum(y)
[1] 9
> sum(residuals)
[1] 1.5
```

Somit gilt im vorliegenden Fall:

$$\sum_{i=1}^{n} \tilde{y}_i \neq \sum_{i=1}^{n} y_i \quad \text{und} \quad \sum_{i=1}^{n} \tilde{u}_i \neq 0$$

3.2 Grafische Methoden

3.2.1 Einige Grundlagen zum Erstellen von Grafiken

• **Vorbemerkung** • *R* bietet sehr umfangreiche Möglichkeiten zur Erstellung von Grafiken an. Wir beschränken uns auf die wichtigsten Funktionen, die für gewöhnliche statistische Analysen und Präsentationszwecke völlig ausreichend sein dürften. Es sei in diesem Zusammenhang auch auf die Hilfe-Dateien zu den einzelnen Grafikfunktionen mit den darin enthaltenen Beispielen verwiesen und ferner auf die ergänzenden Themen in Bezug auf Grafiken in Abschnitt 5.1.

• **Speichern von Grafiken** • Jede Grafik lässt sich bequem über Rechtsklick oder die Menüleiste in bestimmten Formaten speichern, wie z.B. *png* (Pixelgrafik) oder *eps* (Vektorgrafik, geeignet zur Einbettung in *LaTeX*). Unter Umständen erweist es sich jedoch vorteilhaft, mit den speziell dafür vorgesehenen Befehlen wie `png()` oder `eps()` zu arbeiten, wenn

man etwa an die Festlegung der Größenformate denkt. Allerdings benötigt man dann auch dafür vorgesehene Applikationen wie etwa den *Windows-Fotoanzeiger* oder den *Adobe Acrobat Reader*, um die so erzeugten Grafiken betrachten zu können. Hinweise hierzu finden sich in Abschnitt 5.1.3, S.154.

• High-Level- und Low-Level-Grafikfunktionen und ihre Parameter am Beispiel eines Streudiagramms • In *R* gibt es zwei Grundtypen von Grafikfunktionen. Dies sind zum einen Funktionen, die eine bestimmte Stammgrafik, wie z. B. ein Streudiagramm, ein Histogramm oder einen Boxplot, erzeugen. Zum anderen gibt es Funktionen, mit denen sich einer bestehenden Stammgrafik bestimmte Elemente hinzufügen lassen – wie etwa eine Achse, eine Gerade, ein Text, ein Punkt oder eine Linie. Erstere werden als ***High-Level-*** und letztere als ***Low-Level-Grafikfunktionen*** bezeichnet. Die Funktionen beider Kategorien verfügen über mehr oder weniger viele Parameter, mit denen sich Lage und Gestalt der jeweiligen Grafikelemente modifizieren lassen. Häufig sind hierbei auch vektorwertige Anwendungen möglich. Man beachte zum letztgenannten Aspekt im Folgenden beispielhaft die Anwendung der Low-Level-Funktion `points()` und den Grafikparameter `col` (Farbe).

Die folgenden Eingaben orientieren sich an Beispiel 5.2.9 aus Stocker und Steinke [2017]. Dabei wird die Lehrbuchabbildung 5.2.18 in Teilen nachempfunden.

Ein einfaches Streudiagramm erhält man mit der Funktion

`plot`(x,y),

wobei `x` bzw. `y` die Vektoren der x- bzw. y-Komponenten der darzustellenden Punkte $(x_1, y_1), \ldots, (x_n, y_n)$ sind.

```
> x=c(1,2,3,4,5)
> y=c(1,2,1,3,2)
> plot(x,y)
```

Wir erhalten Abbildung 3.2.1.

Das modifizierte Streudiagramm in Abbildung 3.2.2 erhält man über

```
> plot(x,y,                          # High-Level-Funktion
+ pch=16,                            # Ausgefüllte Punkte
+ cex=2,                             # Doppelte Größe der Punkte
+ xlim=c(0,6),ylim=c(0,4),           # Wertebereiche der Achsen
+ xlab="x-Werte",ylab="y-Werte",     # Achsenbeschriftungen
+ las=1,                  # 1: Beschriftung von x- und y-Achse horizontal
+ cex.axis=1.5,cex.lab=1.5,cex.main=1.5, # Div. Vergrößerungsfaktoren
+ main="Streudiagramm")                  # (Haupt-)Überschrift
```

Hier wurden in der Funktion `plot()` zahlreiche optionale Parameter gesetzt. Teilweise werden diese im nachfolgenden Text näher erläutert, z.B. `pch` zur Auswahl eines Symbols, das im Streudiagramm zur Darstellung der Punkte verwendet werden soll. Andere Optionen sind im Quelltext kurz erläutert (siehe Text nach #) worden. Mit `cex` kann man z.B. die Symbolgröße im Streudiagramm skalieren und mit `cex.axis`, `cex.lab` und `cex.main` kann man die Schriftgröße der Achsenbeschriftung und der Überschrift skalieren. `xlim=c(x1,x2)`

Abbildung 3.2.1: Streudiagramm ohne Modifikationen und Low-Level-Zusätze

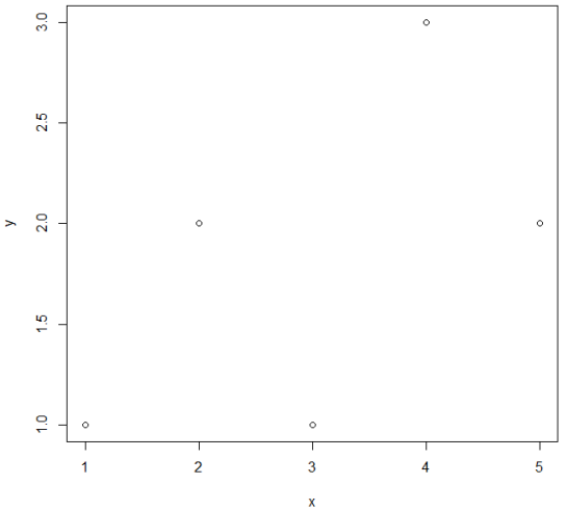
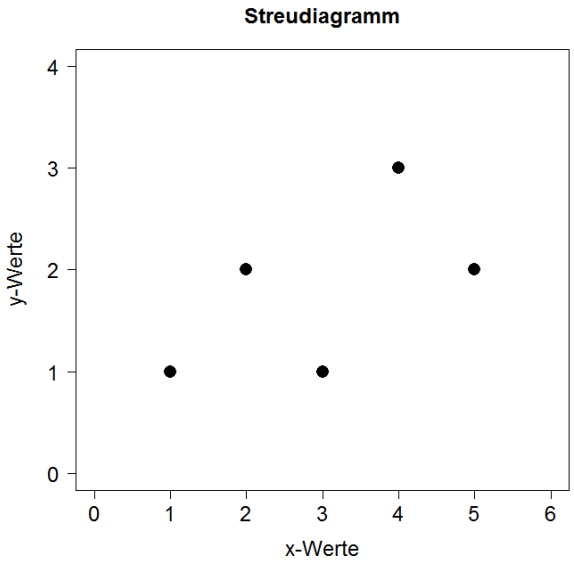


Abbildung 3.2.2: Modifiziertes Streudiagramm ohne Low-Level-Zusätze



bzw. `ylim=c(y1,y2)` legen fest, dass ein Koordinatensystem gezeichnet werden soll, deren Abzissenachse von x1 bis x2 und deren Ordinatenachse von $y1$ bis $y2$ dargestellt werden soll. Weitere Informationen findet man in der *R*-Hilfe.

Wir möchten die Darstellung weiter an unsere Bedürfnisse anpassen. Am Ende werden wir Abbildung 3.2.3 erhalten.

```
> plot(x,y,
+ pch=16,
+ cex=2,
+ xlim=c(0,6),ylim=c(0,4),
+ xlab="",ylab="",              # keine Achsenbeschriftungen
+ axes=FALSE)                   # Unterdrückung der Achsen
```

Es wird die Option `axes=FALSE` verwendet, um die stanardmäßige Achsendarstellung in Form eines Rahmen zu unterdrücken. Stattdessen werden die Koordinatenachsen mit der Funktion `axis`() konstruiert.

```
> axis(1,                        # Low-Level-Achse (1: unten)
+ cex.axis=1.5,
+ pos=0)                        # Bündigkeit an der Stelle y=0

> axis(2,                        # Low-Level-Achse (2: links)
+ cex.axis=1.5,
+ las=1,
+ pos=0)                        # Bündigkeit an der Stelle x=0
```

Mit `abline`(a,b) wird einer Gerade mit Achsenabschnitt a und Anstieg b gezeichnet. `text`(x,y,ABC) stellt an den Koordinaten (x,y) den Text ABC dar. Die Funktion wird hier zur Achsenbeschriftung verwendet.

```
> abline(a=0.9,b=0.3,           # Low-Level-Gerade mit Achsenabschnitt a
+ lwd=3)                       # und Steigung b, dreifache Linienstärke

> text(6.5,0,"x",               # Low-Level-Text an der Stelle (6.5, 0)
+ cex=1.5,
+ xpd=TRUE)                    # Position außerhalb des Achsenbereichs
> text(0,4.5,"y",cex=1.5,xpd=TRUE)
```

Mit `points`(x,y,...) kann man ein Streudiagramm in eine bestehende Abbildung zeichnen. Mithilfe von `segments`(x1,y1,x2,y2) lässt sich eine Strecke von (x1,y1) nach (x2,y2) zeichnen. In unserem Beispiel wird die Funktion `segments()` zum Darstellen horizontaler und vertikaler Hilfslinien verwendet.

```
> points(3,1.8,pch=8,cex=1.5)  # Low-Level-Punkt an der Stele (3, 1.8)
> points(c(1,2,3,4,5),c(1.2,1.5,1.8,2.1,2.4),pch=4,cex=1.5)
                               # Punkte mit x- und y-Koordinaten
> segments(0,1.8,3,1.8,        # Low-Level-Linie von (0,1.8) bis (3,1.8)
+ lty=2,                      # lty=2: Linie gestrichelt)
+ lwd=1.5)
> segments(3,0,3,1.8,lty=2,lwd=1.5)
```

Wir erhalten Abbildung 3.2.3.

Abbildung 3.2.3: Modifiziertes Streudiagramm mit Low-Level-Zusätzen

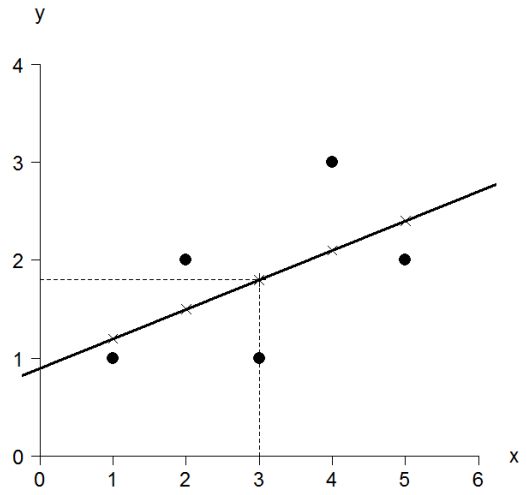
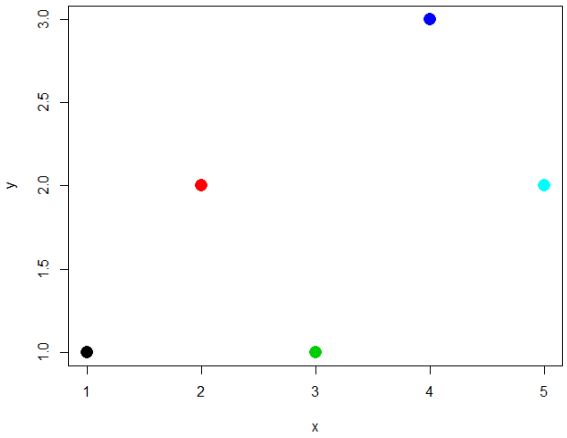


Abbildung 3.2.4: Verwendung von Farben in einem Streudiagramm



• **Farben** • Viele Grafikfunktionen bieten die Möglichkeit, für bestimmte Grafikbestandteile Farben auszuwählen. Häufig wird dies über den Grafikparameter `col` (color) festgelegt.

Abbildung 3.2.4 erhält man über

```
> x=c(1,2,3,4,5)
> y=c(1,2,1,3,2)
> plot(x,y,pch=16,cex=2,col=1:5)
```

Hier wird dem Grafikparameter `col` der Vektor (1,2,3,4,5) übergeben. Damit erhält jeder der 5 zweidimensionalen Beobachtungswerte eine andere Farbe. Die erste Beobachtung (1,1)

erhält den Farbwert 1 (schwarz), die zweite Beobachtung (2,2) den Farbwert 2 (rot) usw. Insgesamt bietet *R* in der voreingestellten Farbpalette 9 verschiedene Farben an.

Abbildung 3.2.5 erhält man über

```
> plot(0,0,
+ type="n",          # keine Darstellung der Punkte ("none ")
+ xlim=c(1,12),ylim=c(0,1),xlab="",ylab="",axes=FALSE)
> points(1:9,rep(0.5,9),pch=16,cex=5,col=0:8)
> text(1:9,1,0:8)
```

Zunächst wird eine fiktive zweidimensionale Beobachtung (0, 0) generiert. Darauf basierend wird dann ein Streudiagramm mittels `plot()` erzeugt, wobei jegliche Bestandteile optisch unterdrückt werden. In dieses leere Diagramm werden dann 9 Punkte in den 9 Standardfarben (weiß bis grau) mittels des Low-Level-Befehls `points()` eingezeichnet. Zeichnet man 12 anstelle von 9 Punkten, erhält man Abbildung 3.2.6. *R* beginnt ab der Farbe Nr. 9 wieder mit Schwarz (Farbe 1), dann kommt als nächstes wieder Rot usw.

Man beachte, dass in `text()` der Vektor der y-Koordinaten auf die Länge des Vektors der x-Koordinaten gestreckt wird, sofern er kürzer als dieser ist. Es ist hier also nicht notwendig die y-Koordinate 9 Mal zu wiederholen. Im Hilfetext zur Funktion heißt es dazu: „*If the length of x and y differs, the shorter one is recycled*“. In `points()` müssen hingegen zwei gleich lange Vektoren für x- und y-Koordinaten übergeben werden. Weiter müssen in der Funktion `text()` darzustellende Zahlenwerte (hier 0 bis 8) nicht zwingend in Anführungszeichen gesetzt bzw. nicht zwingend als Vektor des `character`-Datentyps übergeben werden.

Abbildung 3.2.5: Standardfarben

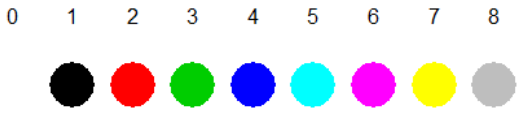


Abbildung 3.2.6: Recyclingprinzip bei Farben

• **Linientypen** • Über den Parameter `lty` (line type) sind insgesamt 6 verschiedenen Linientypen auswählbar. Verwendet wird der Parameter auf analoge Art und Weise wie `col`.

Abbildung 3.2.7 erhält man über

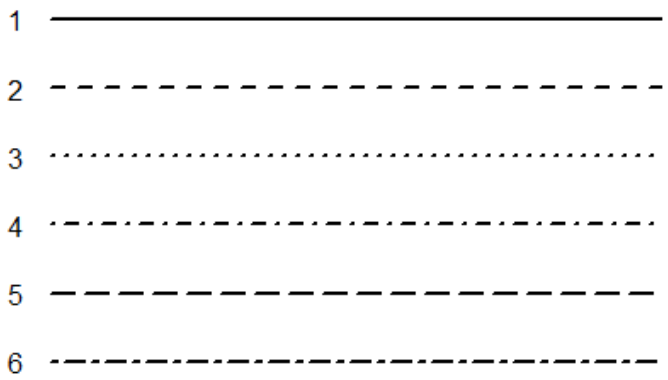
```
> plot(0,0,type="n",xlim=c(0,1),ylim=c(0,6),xlab="",ylab="",axes=FALSE)
> abline(h=6:1,lty=1:6,lwd=2)  # h: horizontale Gerade an den Stellen...
> text(-0.1,6:1,1:6,xpd=TRUE)
```

• **Symbolarten** • Die 21 verschiedenen Symbole für die Beobachtungspunkte in Streudiagrammen werden über den optionalen Parameter `pch` gesteuert.

Abbildung 3.2.8 resultiert aus

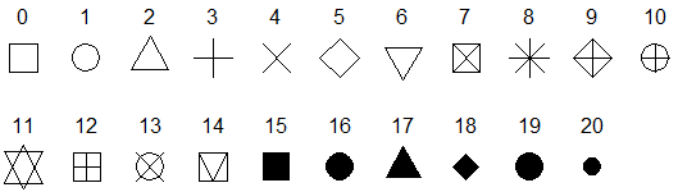
```
> plot(0,0,type="n",xlim=c(1,11),ylim=c(0,2.5),
+    xlab="",ylab="",axes=FALSE)
> points(1:11,rep(1.6,11),pch=0:10,cex=3)
> points(1:10,rep(0.6,10),pch=11:20,cex=3)
```

Abbildung 3.2.7: Standardlinientypen




```
> text(1:11,2,0:10)
> text(1:10,1,11:20)
```

Abbildung 3.2.8: Symbolarten für Beobachtungswerte



• **Globale Grafikparameter** • Über die Funktion par() lassen sich globale Grafikeigenschaften aller Grafiken für die Dauer einer laufenden *R*-Sitzung einstellen bzw. ändern.

Die Namen aller Parameter sowie deren Einstellungen lassen sich über

```
> par()
```

ausgeben. Da die Ausgabe ein `list`-Objekt ist, können die Parameternamen (die Komponentennamen dieser Liste) über `names()` abgefragt werden:

```
> names(par())
 [1] "xlog"      "ylog"      "adj"       "ann"
 [5] "ask"       "bg"        "bty"       "cex"
 [9] "cex.axis"  "cex.lab"   "cex.main"  "cex.sub"
[13] "cin"       "col"       "col.axis"  "col.lab"
[17] "col.main"  "col.sub"   "cra"       "crt"
[21] "csi"       "cxy"       "din"       "err"
[25] "family"    "fg"        "fig"       "fin"
[29] "font"      "font.axis" "font.lab"  "font.main"
[33] "font.sub"  "lab"       "las"       "lend"
[37] "lheight"   "ljoin"     "lmitre"    "lty"
[41] "lwd"       "mai"       "mar"       "mex"
[45] "mfcol"     "mfg"       "mfrow"     "mgp"
[49] "mkh"       "new"       "oma"       "omd"
```

```
[53] "omi"        "page"       "pch"        "pin"
[57] "plt"        "ps"         "pty"        "smo"
[61] "srt"        "tck"        "tcl"        "usr"
[65] "xaxp"       "xaxs"       "xaxt"       "xpd"
[69] "yaxp"       "yaxs"       "yaxt"       "ylbias"
```

Zur Bedeutung der einzelnen Parameter konsultiere man den Hilfetext:

```
> ?par
```

Um nun etwa gezielt die Einstellungen der Parameter `las` (35. Komponente) und `pch` (55. Komponente) zu erfahren, kann mittels Listensyntax (Abschnitt 2.4, S.42ff) wie folgt vorgegangen werden:

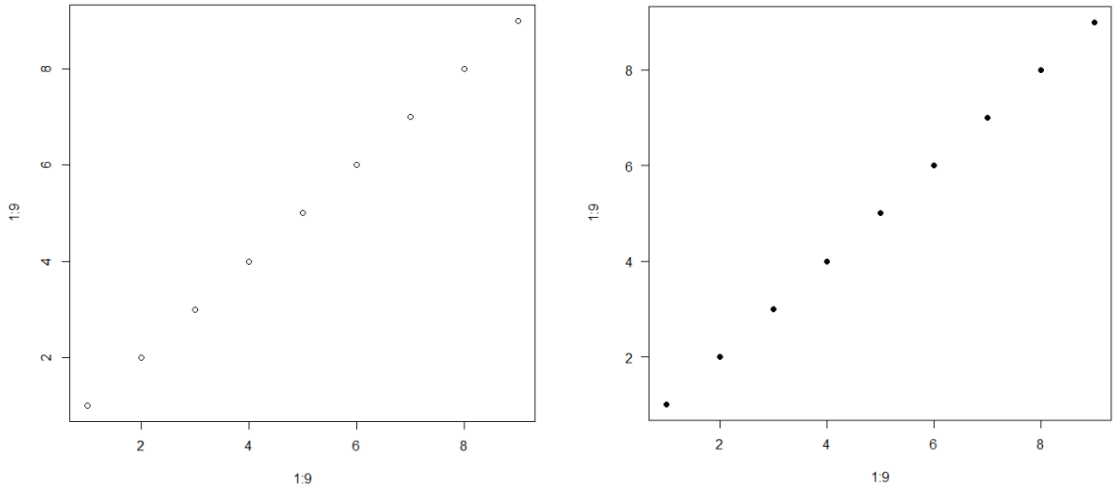
```
> par()$las
[1] 0
> par()$pch
[1] 1
> plot(1:9,1:9)
```

Die Standardeinstellungen sind hier also `las=0` und `pch=1` (Abb. 3.2.9, linke Grafik). Möchte man für die Dauer einer Sitzung nun etwa die Grafiken stets mit `las=1` (horizontale Achsenbeschriftung) und `pch=16` (ausgefüllte Punkte) arbeiten, so ließe sich dies über Eingabe von

```
> par(las=1,pch=16)
> plot(1:9,1:9)
```

bewerkstelligen (Abb. 3.2.9, rechte Grafik).

Abbildung 3.2.9: Änderung von par()-Einstellungen



Die Rücksetzung der Werte erfolgt dann über

```
> par(las=0,pch=1)
```

Wie bereits vorgestellt, sind diese beiden Parameter und viele andere meist auch als Argumente bestimmter High- oder Low-Level-Grafikfunktionen verfügbar.

• Funktionsgraphen • Ein direkter Weg, den Funktionsgraphen einer eindimensionalen Funktion zu zeichnen, besteht darin, zunächst ein Gitter von x-Werten zu erzeugen (etwa mit der Funktion `seq()`), an denen die Funktion berechnet wird (y-Werte). Anschließend erstellt man ein Streudiagramm von x- und y-Werten, wobei die Datenpunkte jedoch nicht durch Punkte dargestellt werden, sondern durch einen Linienzug verbunden werden. Im Folgenden sei dies anhand einer Parabel $y = x^2$ illustriert.

Das linke Schaubild von Abbildung 3.2.10 resultiert aus

```
> x=seq(-1,1,le=100)
> y=x^2
> plot(x,y)
```

Das mittlere Schaubild resultiert aus der Modifikation des `type`-Parameters:

```
> plot(x,y,type="l")   # l: Linie
```

Der Funktionsgraph erscheint dabei umso glatter, je dichter die x-Werte generiert werden.

Eine alternative (einfachere) Möglichkeit bietet hier die Funktion

curve(fu(x),x1,x2).

Das erste Argument ist dabei ein Ausdruck, der die Funktion in Abhängigkeit des generischen Arguments `x` berechnet. Ein Datenvektor `x` ist dazu nicht notwendig. `curve()` lässt sich sowohl als High-Level- als auch als Low-Level-Funktion verwenden.

```
> rm(x)
> curve(x^2,from=-2,to=2)                          # High-Level
> curve(2*x^2,from=-1,to=1,add=TRUE,lty=2)         # add=TRUE: Low-Level
```

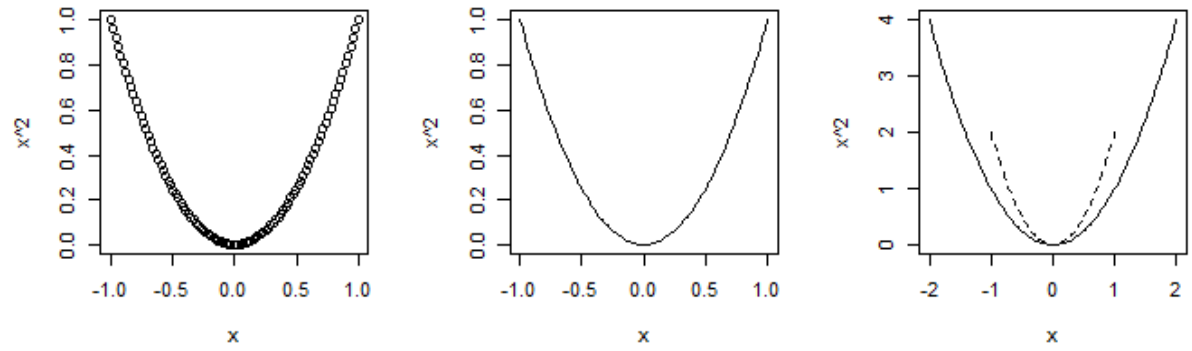
Das rechte Schaubild von Abbildung 3.2.10 zeigt als Beispiel die Zeichnung der Standardparabel im Wertebereich von -2 bis $+2$ und einer gestreckten Version ($y = 2x^2$) von -1 bis $+1$. Demonstrativ wird der Vektor `x` zunächst gelöscht, um zu zeigen, dass auf ihn verzichtet werden kann. Mit der Option `add=TRUE` wird eine Funktionsdarstellung in eine bereits bestehende Abbildung hineingezeichnet und `curve()` damit als Low-Level-Funktion verwendet. Ohne diese Option würde die alte Abbildung gelöscht und eine neue Funktionsdarstellung erzeugt werden.

Mit `plot()` und `lines()` kann man das gleiche Resultat erhalten.

```
> x=seq(-2,2,le=101)
> plot(x,x^2,type="l")        # High-Level
> x=seq(-1,1,le=101)
> lines(x,2*x^2,lty=2)        # Low-Level-Linie
```

Man beachte, dass `curve()` in der Voreinstellung zur Darstellung `n = 101` Datenpunkte generiert (vgl. `?curve`), weshalb hier der Vergleichbarkeit wegen `le=101` in `seq()` gewählt wurde.

Abbildung 3.2.10: Streudiagramm und Funktionengraph einer Parabel



• Mehrere Grafiken • Insbesondere im Zusammenhang vergleichender Gegenüberstellungen ist es oft wünschenswert, zwei oder mehr Grafiken darzustellen. Dies lässt sich etwa über globale Grafikeinstellungen der Funktionen `par()` und `mfrow()` bewerkstelligen.

Abbildung 3.2.10 zeigt eine solche Grafik als 1×3-Grafikmatrix. Am Ende des Programmcodes wird die globale Einstellung wieder auf die Erstellung einer einzigen Grafik zurückgesetzt.

```
> par(mfrow=c(1,3))           # Grafikmatrix (1 Zeile, 3 Spalten)
> x=seq(-1,1,le=100)
> plot(x,x^2)                 # 1. Grafik
> plot(x,x^2,type="l")        # 2. Grafik
> curve(x^2,from=-2,to=2)     # 3. Grafik
> curve(2*x^2,from=-1,to=1,add=TRUE,lty=2)      # Low-Level-Grafik
> par(mfrow=c(1,1))    # Rücksetzung auf Einzelgrafik (1 Zeile, 1 Spalte)
```

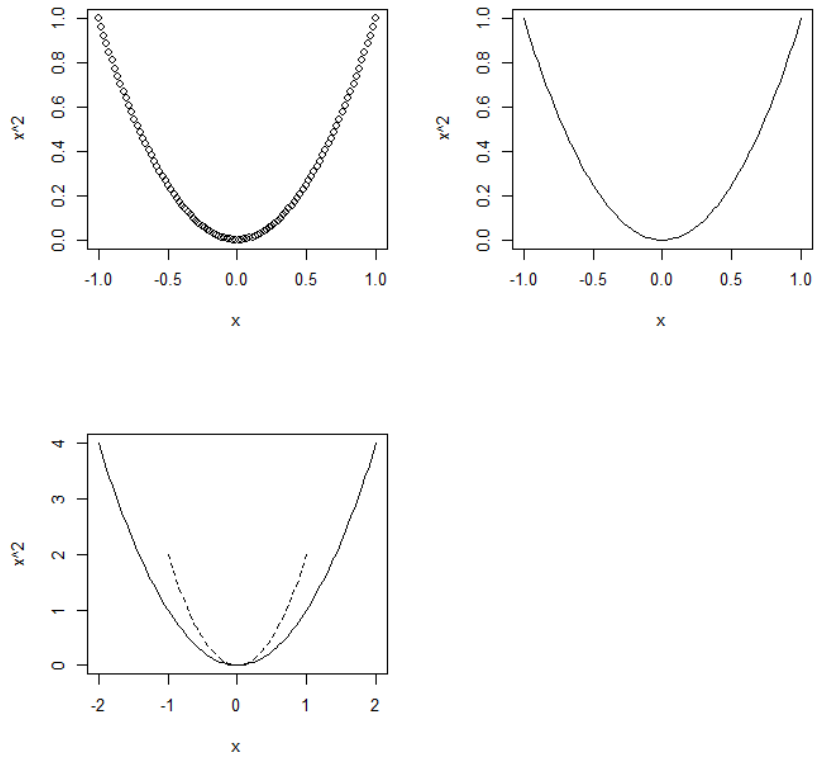
Arrangiert man die drei Schaubilder hingegen in einer 2×2-Grafikmatrix, so bleibt das letzte Grafikfeld rechts unten leer (Abb. 3.2.11).

```
> par(mfrow=c(2,2))    # Grafikmatrix (2 Zeilen, 2 Spalten)
> ...                  # s. o.
```

• Mehrere Grafikfenster • Mit jeder Ausführung einer High-Level-Grafikfunktion wird normalerweise ein Grafikfenster (*screen device*) geöffnet bzw. der Inhalt eines bereits offenen Fensters überschrieben. Möchte man mehrere Grafikfenster offen halten, so lässt sich dies mit dem Befehl `windows()` bewerkstelligen (auf einem Mac `quartz()`). Dies sei kurz anhand der vorhergehenden Grafiken von Abbildung 3.2.10 bzw. 3.2.11 demonstriert.

```
> x=seq(-1,1,le=100)
> windows()                  # 1. Fenster aktiv (2)
> plot(x,x^2)                # Grafik im 1. Fenster
> windows()                  # 2. Fenster aktiv (3)
> plot(x,x^2,type="l")       # Grafik im 2. Fenster
> dev.set(2)                 # 1. Fenster aktiv
> curve(x^3,from=-1,to=1)    # Überschreiben der Grafik im 1. Fenster
> dev.set(3)                 # 2. Fenster aktiv
> curve(2*x^2,from=-1,to=1,add=TRUE,lty=2) # Low-Level-Kurve im 2.Fenster
> dev.off(3)                 # Schließen des 2. Fensters
> dev.off(2)                 # Schließen des 1. Fensters
```

Abbildung 3.2.11: (2×2)-Grafikmatrix



Man beachte, dass die Nummerierung der Grafikfenster mit 2 beginnt. Mit dem Befehl `graphics.off()` lassen sich alle Fenster simultan schließen.

3.2.2 Darstellung kategorialer Merkmale

Übersicht

• **Vorgehensweise** • Die im Folgenden präsentierten Darstellungsmöglichkeiten für kategoriale Merkmale werden in Stocker und Steinke [2017] in den Abschnitten 3.2.1, 5.1.3 und 5.3.2 erklärt. Einige Abbildungen, die man in diesen Abschnitten finden kann, werden im Nachfolgenden nahezu exakt oder in etwas vereinfachter Form reproduziert. Die vorgestellten Grafiken basieren auf absoluten oder relativen Häufigkeiten ein- oder mehrdimensionaler Daten. Diese können, sofern sie noch nicht vorliegen, über Funktionen wie etwa `table()` oder `prop.table()` ermittelt werden. Diese Funktionen wurden im Abschnitt 3.1.1, S.65ff, vorgestellt.

• **Datenbasis** • Im Folgenden werden unter anderem die Datensätze *Straftaten*, *ASEG* und *Sternzeichen* als Datenbasis herangezogen. Man beachte hierzu die Hinweise zu den Datensätzen im Vorwort dieses Skripts.

Kreis-, Säulen und Balkendiagramme

• **Kreisdiagramme** • Kreisdiagramme können mit der Funktion

`pie`(x)

erstellt werden. `x` ist ein Vektor von postiven Werten, deren Anteile an der Gesamtsumme (z.B. 1 oder 100 (Prozent)) mit einem Kreisdiagramm veranschaulicht werden sollen.

Bei Übernahme aller Voreinstellungen erhält man ein erstes unmodifiziertes Kreisdiagramm zum Ausgang der Bundestagswahl 2013 (Abbildung 3.2.12) über

```
> Anteile=c(41.5,25.7,4.8,8.6,8.4,10.9)
> pie(Anteile)
```

Ergänzt mit Beschriftungen und passenden Farben resultiert daraus Abbildung 3.2.13.

```
> Parteien=c("CDU/CSU","SPD","FDP","DIE LINKE","GRÜNE","Sonstige")
> Buntfarben=c("black","red","yellow","darkmagenta","green","grey")
> pie(Anteile,labels=Parteien,col=Buntfarben, main="Bundestagswahl 2013")
```

Zur Verwendung von Farben bzw. Farbnamen jenseits der Standardpalette beachte man die Ausführungen in Abschnitt 5.1.1, S.151ff.

Unter Verwendung von Low-Level-Textzusätzen erhält man Abbildung 3.2.14, S.92.

```
> Graufarben=gray(seq(0.6,0.95,le=6))
> pie(Anteile,labels="",col=Graufarben)
> text(0.1,0.4,"41.5%",font=2)            # font=2: fett
> text(0.2,0.9,"CDU/CSU")
> text(-0.4,-0.15,"25.7%",font=2)
> text(-0.9,-0.4,"SPD")
> text(-0.23,-0.65,"4.8%",font=2)
> text(-0.32,-0.9,"FPD")
> text(0.08,-0.6,"8.6%",font=2)
> text(0.085,-0.95,"DIE LINKE")
> text(0.36,-0.45,"8.4%",font=2)
> text(0.65,-0.75,"GRÜNE")
> text(0.5,-0.15,"10.9%",font=2)
> text(1,-0.3,"Sonstige")
```

Das von `pie()` verwendete imaginäre Koordinatensystem weist für x- und y-Richtung jeweils den Wertebereich $[-1, 1]$ auf. Zur genauen Positionierung einzelner Textelemente kann nach dem Trial-And-Error-Prinzip verfahren werden. Dazu führe man den Programmcode am besten im Editor aus und nicht etwa im Kommandofenster, da die Zahlenwerte so leichter editierbar bleiben.

Muss die absolute oder relative Häufigkeitsverteilung erst noch ermittelt werden, bedarf es einen Vorschrittes, der mit den Funktionen `table()` oder `prop.table()` bewerkstelligt werden kann. Als Beispiel diene hierfür Abbildung 3.2.15. Diese resultiert aus

```
> x=c(0,0,0,0,0,1,1,2,2,2)
> nj=table(x)                    # absolute Häufigkeitsverteilung
```

Abbildung 3.2.12: Kreisdiagramm ohne Modifikationen und Low-Level-Zusätze

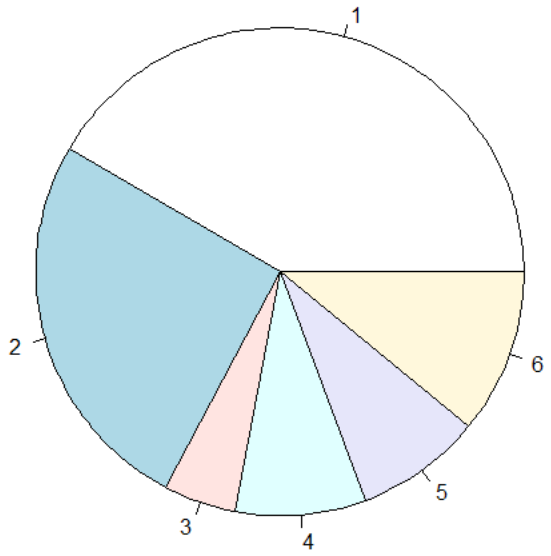
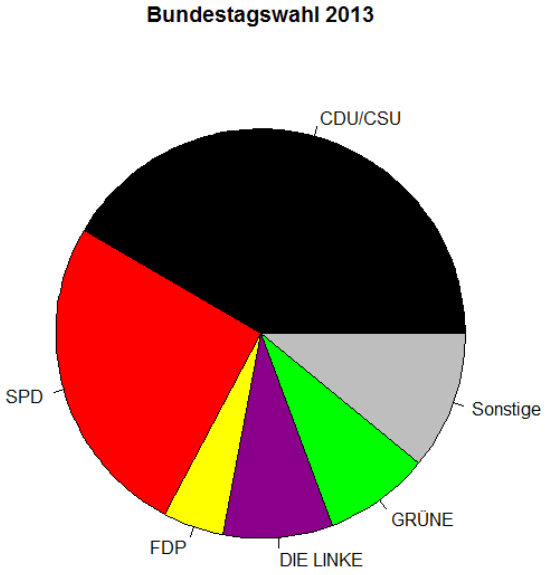


Abbildung 3.2.13: Modifiziertes Kreisdiagramm ohne Low-Level-Zusätze




```
> nj
x
0 1 2
5 2 3
> fj=prop.table(nj)              # relative Häufigkeitsverteilung
> fj                             # angewendet auf table(x) (nicht auf x!)
x
  0   1   2
0.5 0.2 0.3
> pie(nj)
> pie(fj)                        # erzeugt identische Grafik
```

• **Säulendiagramme** • Säulendiagramme können mit der Funktion

barplot(x)

erstellt werden. `x` ist der Vektor der (absoluten oder relativen) Häufigkeit für das Auftreten von gewissen Merkmalsausprägungen.

Mit den gleichen Daten wie im vorhergehenden Punkt erhält man ein unmodifiziertes Säulendiagramm zum Ausgang der Bundestagswahl (Abb. 3.2.16) über

```
> barplot(Anteile)
```

Abbildung 3.2.17 erhält man über

```
> barplot(Anteile,names.arg=Parteien,
+   ylim=c(0,50),ylab="Stimmenanteil in %",las=1,
+   cex.axis=1.5,cex.lab=1.5,col=Buntfarben,main="Bundestagswahl 2013")
```

Abbildung 3.2.18 erhält man über

Abbildung 3.2.14: Modifiziertes Kreisdiagramm mit Low-Level-Zusätzen

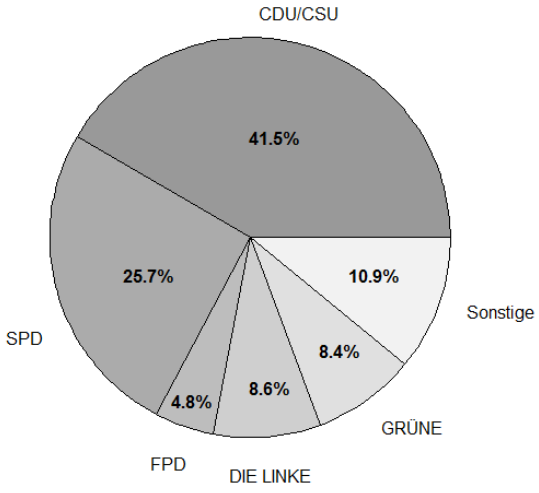
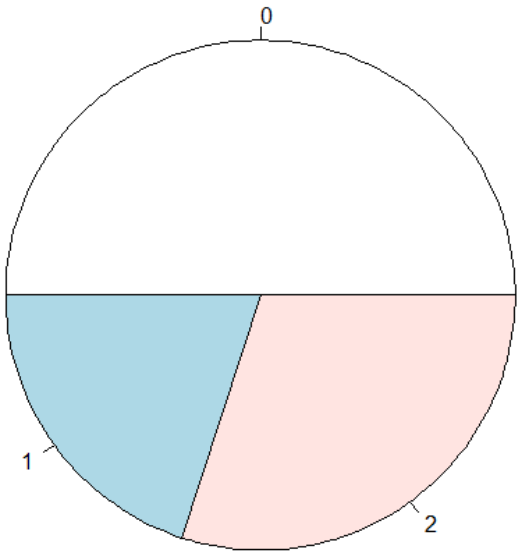


Abbildung 3.2.15: Kreisdiagramm auf Basis von `table()`




```
> barplot(Anteile,names.arg="",ylim=c(0,50),ylab="",las=2,col=gray(0.95))
> text(0,50,"Stimmenanteil in %",xpd=TRUE,pos=4,cex=1.2)
> text(seq(0.75,6.5,by=1.15),-6,Parteien,srt=45,xpd=TRUE)
> text(seq(0.75,6.5,by=1.15),Anteile+3,Anteile+3)
```

Die Option `las=2` bewirkt hier zunächst eine vertikale Beschriftung der Säulen (bei gleichzeitig horizontaler Beschriftung der y-Achse). Mit `srt=45` werden die Texte dann um 45^o rotiert. Man beachte, dass hier die Textpositionen in effizienter Weise vektormäßig eingegeben wurden. Im letzten `text`-Befehl wird das zweite Argument „`Anteile+3`“ als Vektor der y-Koordinaten zur Positionierung der Stimmenanteilswerte verwertet und das dritte („identische“) Argument „`Anteile+3`“ als Vektor des darzustellenden Textes. Da es sich hier um Zahlenwerte handelt, kann direkt der numerische Vektor mit den Anteilswerten namens `Anteile` verwendet werden. Dieser wird noch um drei Einheiten verschoben, so dass die Beschriftung mit etwas Abstand oberhalb der Balken ansetzt. Es ist nicht notwendig die Zahlenwerte im `character`-Datentyp zu übergeben. Die besten Abstände (Schrittweiten) der einzelnen Textbestandteile kann über Trial-And-Error festgelegt werden.

• **Balkendiagramme** • Wie zuvor die Säulendiagramme können auch Balkendiagramme mit der Funktion `barplot()` erstellt werden. Dazu ist lediglich die Option `horiz=TRUE` zu wählen.

So erhält man ein unmodifiziertes ***Balkendiagramm*** zum Ausgang der Bundestagswahl (Abb. 3.2.19) über

```
> barplot(Anteile,horiz=TRUE)
```

Abbildung 3.2.20 erhält man über

Abbildung 3.2.16: Säulendiagramm ohne Modifikationen und Low-Level-Zusätze

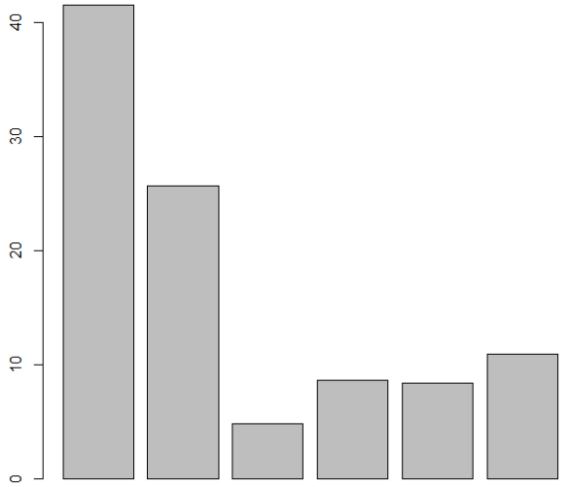


Abbildung 3.2.17: Modifiziertes Säulendiagramm ohne Low-Level-Zusätze

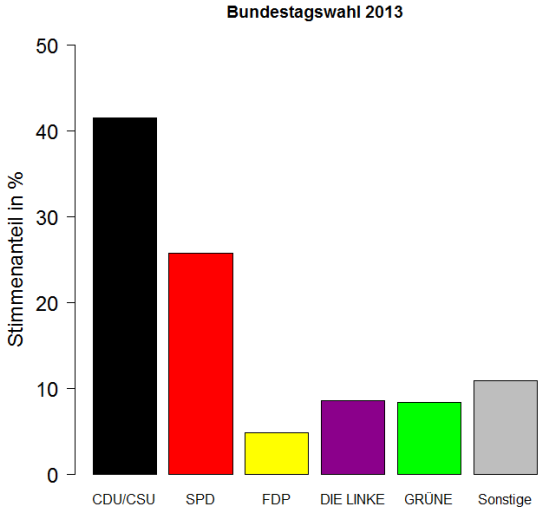
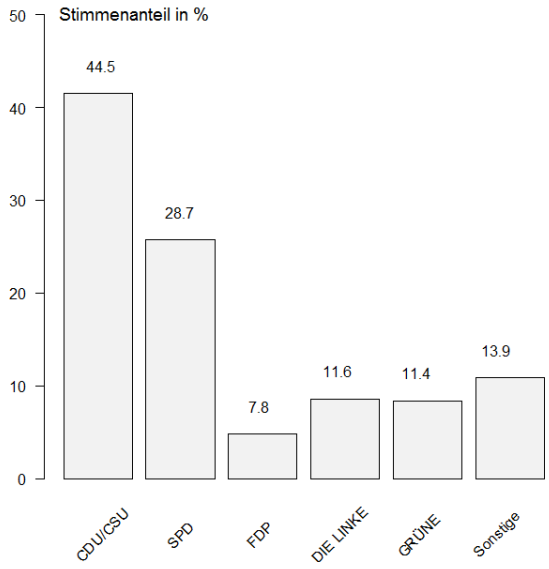


Abbildung 3.2.18: Modifiziertes Säulendiagramm mit Low-Level-Zusätzen




```
> barplot(Anteile,names.arg=Parteien,xlim=c(0,50),
+   xlab="Stimmenanteil in %",cex.axis=1.5,cex.lab=1.5,las=1,
+   col=Buntfarben,horiz=TRUE,main="Bundestagswahl 2013")
```

Wie man sieht hat die Beschriftung der Balken in Abbildung 3.2.20 stellenweise nicht ausreichend Platz. Eine Möglichkeit wäre über den Parameter `cex.names` die Beschriftung zu verkleinern. Eine andere Möglichkeit besteht darin über Low-Level-Textzusätze die Beschriftung rechts der Balken anzubringen. So erhält man Abbildung 3.2.21 über

```
> barplot(Anteile,names.arg="",xlim=c(0,50),
+   xlab="Stimmenanteil in %",las=1,cex.axis=1.5,cex.lab=1.5,
```

```
+   col=Buntfarben,horiz=TRUE,main="Bundestagswahl 2013")
> text(Anteile+5,seq(0.8,6.8,by=1.2),Parteien)
```

Möchte man partout die Beschriftung links der Balken anbringen, kann man hier über die globalen Grafikeinstellungen mittels `par()` etwas mehr Platz („weiße Fläche“) am linken Rand der Grafik schaffen. So erhält man dann Abbildung 3.2.22 über

```
> marStandard=par()$mar
> par(mar=marStandard+c(0,2,0,0))
> barplot(Anteile,names.arg=Parteien,xlim=c(0,50),
+   xlab="Stimmenanteil in %",cex.axis=1.5,cex.lab=1.5,las=1,
+   col=Buntfarben,horiz=TRUE,main="Bundestagswahl 2013")
> par(mar=marStandard)
```

Über den Parameter `mar` (für *margins*) lässt sich Platz unterhalb, links, oberhalb und rechts (in dieser Reihenfolge) der Grafik schaffen, wobei die Größe in Anzahl von Textlinien gemessen wird. Im vorliegenden Fall wird mit der Option

```
mar=marStandard+c(0,2,0,0)
```

also lediglich ein wenig mehr Platz links der Grafik, nämlich zwei Textzeilen (in diesem Fall ist eine vertikale Beschriftung anzunehmen), geschaffen. Dazu wird auf den Standardwert

```
> marStandard
[1] 5.1 4.1 4.1 2.1
```

der Vektor (0, 2, 0, 0) aufaddiert. Das bewirkt, dass lediglich der zweite Wert (für den linken Bereich) um 2 Einheiten erhöht wird. Am Ende setzt man am besten wieder auf die ursprünglichen Standardwerte zurück.

Abbildung 3.2.19: Balkendiagramm ohne Modifikationen und Low-Level-Zusätze

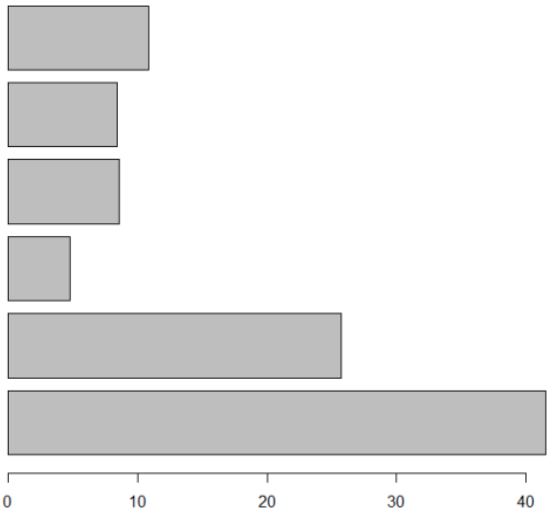


Abbildung 3.2.20: Modifiziertes Balkendiagramm mit Beschriftung links

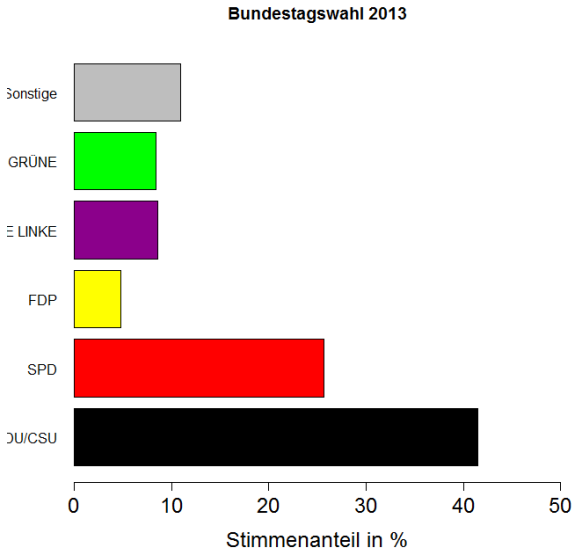


Abbildung 3.2.21: Modifiziertes Balkendiagramm mit Low-Level-Beschriftung rechts

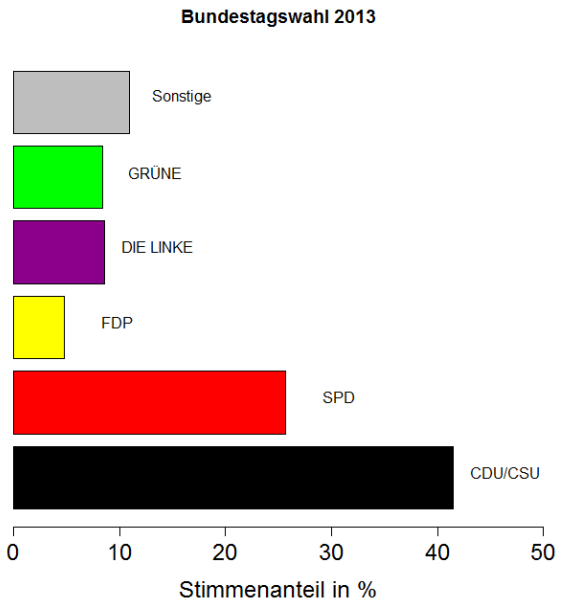
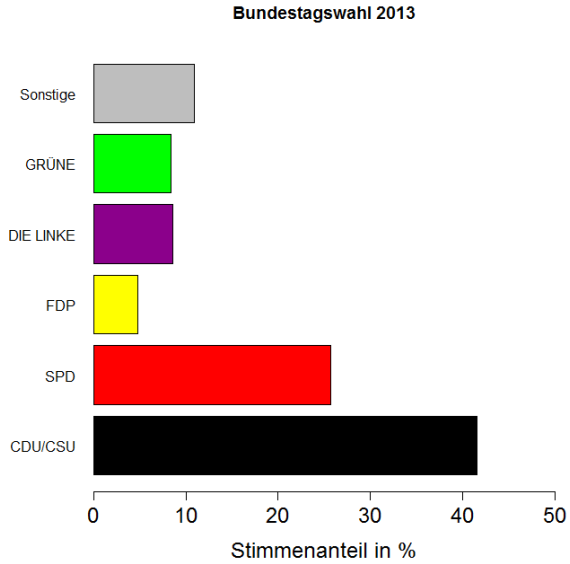


Abbildung 3.2.22: Modifiziertes Balkendiagramm mit mehr Platz für Beschriftung links



Gestapelte und gruppierte Säulendiagramme

• Gestapelte Säulendiagramme • Gestapelte Säulendiagramme basieren meist auf zweidimensionalem Datenmaterial. Zur Erstellung kann wie zuvor auf die Funktion `barplot()` zurückgegriffen werden, wobei jetzt jedoch eine Kontingenztabelle (`table`- oder `matrix`-Objekt) als Dateninput notwendig ist.

Als Beispiel wird im Folgenden die Kontingenztabelle aus Abschnitt 3.1.3 verwendet. Zunächst wird die Tabelle als Matrix-Objekt erzeugt.

```
> m=c(4,12,8,12,28,36)
> m=matrix(m,2,3)
> m
     [,1] [,2] [,3]
[1,]    4    8   28
[2,]   12   12   36
```

Ein unmodifiziertes gestapeltes Säulendiagramm (Abb. 3.2.23) erhält man über

```
> barplot(m)
```

Die Häufigkeiten werden in den Spalten der zugrunde gelegten Datenmatrix in Form gestapelter Säulen dargestellt. Dies entspricht einer Stapelung über das Y-Merkmal.

Möchte man hingegen über das X-Merkmal stapeln, wendet man die `barplot()`-Funktion auf die Transponierte von `m` an (Abb. 3.2.24):

```
> t(m)
     [,1] [,2]
[1,]    4   12
```

```
[2,]    8   12
[3,]   28   36
> barplot(t(m))
```

Abbildung 3.2.23: Gestapeltes Säulendiagramm ohne Modifikationen und Low-Level-Zusätze gestapelt über Y

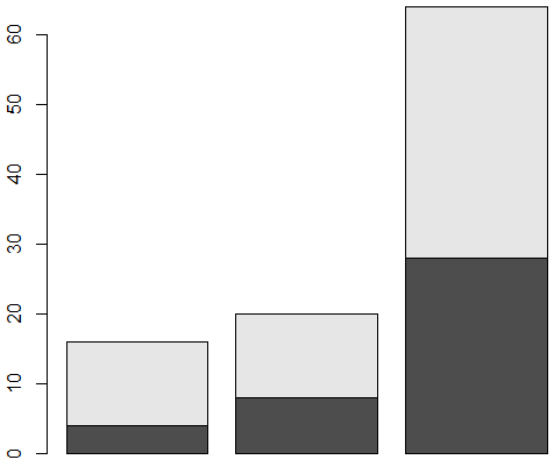
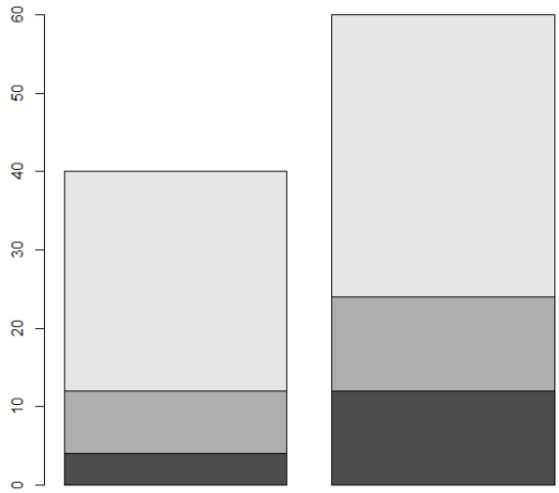


Abbildung 3.2.24: Gestapeltes Säulendiagramm ohne Modifikationen und Low-Level-Zusätze gestapelt über X



Mit diversen Optionen von `barplot()` sowie der Low-Level-Funktion `legend()` erhält man Abbildung 3.2.25 über

```
> barplot(m,
+   names.arg=c("Raucher","Gelegenheitsraucher","Nichtraucher"),
+   ylim=c(0,70),las=1,col=c("grey50","grey95"),
+   main="Geschlecht nach Raucherstatus")
> legend(0.5,60,legend=c("weiblich","männlich"),
+   fill=c("grey50","grey95"))
```

• Gruppierte Säulendiagramme • Auch gruppierte Säulendiagramme können mittels `barplot()` erstellt werden. Der erforderliche Dateninput erfolgt wie beim gestapelten Säulendiagramm. Die Säulen werden jedoch nicht gestapelt sondern nebeneinander gruppiert. Hierzu muss lediglich die Option `beside=TRUE` gewählt werden.

Ein unmodifiziertes gruppiertes Säulendiagramm (Abb. 3.2.26) mit Gruppierung nach Y erhält man über

```
> barplot(m,beside=TRUE)
```

Möchte man nach X gruppieren, wendet man die `barplot()`-Funktion wiederum auf die Transponierte von m an (Abb. 3.2.27):

```
> barplot(t(m),beside=TRUE)
```

Mit diversen Optionen von `barplot()` sowie der Low-Level-Funktion `legend()` erhält man Abbildung 3.2.28 über

Abbildung 3.2.25: Modifiziertes gestapeltes Säulendiagramm mit Low-Level-Legende gestapelt über Y (Raucherstatus)

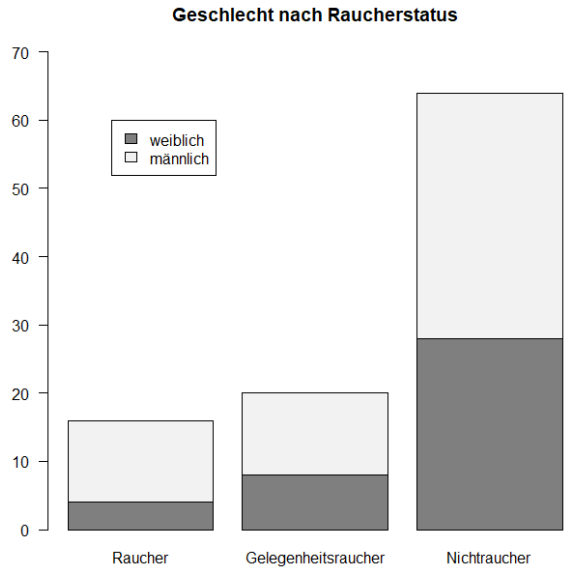
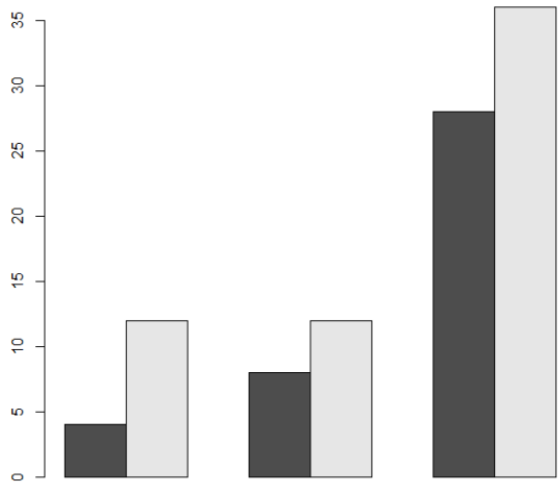


Abbildung 3.2.26: Gruppiertes Säulendiagramm ohne Modifikationen und Low-Level-Zusätze gruppiert nach Y




```
>  barplot(t(m),
+    names.arg=c("weiblich","männlich"),
+    las=1,col=c("grey35","grey65","grey95"),
+    main="Raucherstatus nach Geschlecht")
>  legend(0.5,60,
+    legend=c("Raucher","Gelegenheitsraucher","Nichtraucher"),
+    fill=c("grey35","grey65","grey95"))
```

Man beachte, dass die Säulen in der Voreinstellung an sich bereits in Grautönen dargestellt werden. Die Spezifikation des Farbvektors in `barplot()` über den Parameter `col` erscheint dennoch notwendig, um so eine exakte Übereinstimmung mit den Farben der Legende zu erzielen. Im vorliegenden Fall wird zusätzlich noch eine stärkere Kontrastierung der Grautöne darüber erreicht.

Segmentierte Säulen- und Balkendiagramme

• **Ermittlung bedingter Verteilungen** • Segmentierte Säulen- und Balkendiagramme stellen die bedingten Verteilungen (Zeilen oder Spalten) einer Kontingenztabelle dar. In *R* lassen sich diese durch Kombination von `table()` und `prop.table()` erstellen (vgl. Abschnitt 3.1.1). Auf das Ergebnis wird die Funktion `barplot()` angewendet.

• **Segmentierte Säulendiagramme** • Betrachten wir nochmals die Kontingenztabelle aus dem vorhergehenden Abschnitt

```
> m
     [,1] [,2] [,3]
[1,]    4    8   28
```

Abbildung 3.2.27: Gruppiertes Säulendiagramm ohne Modifikationen und Low-Level-Zusätze gruppiert nach X

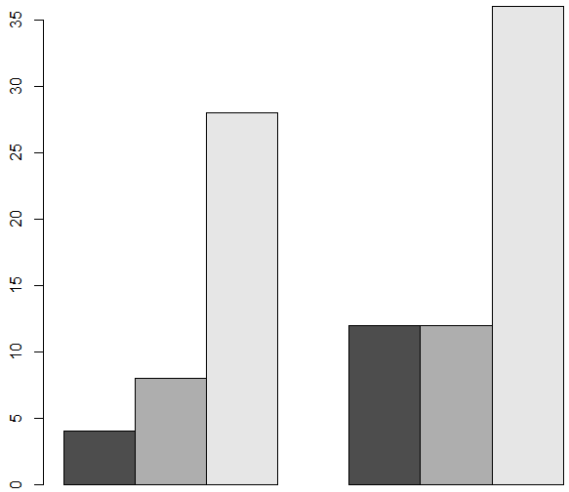
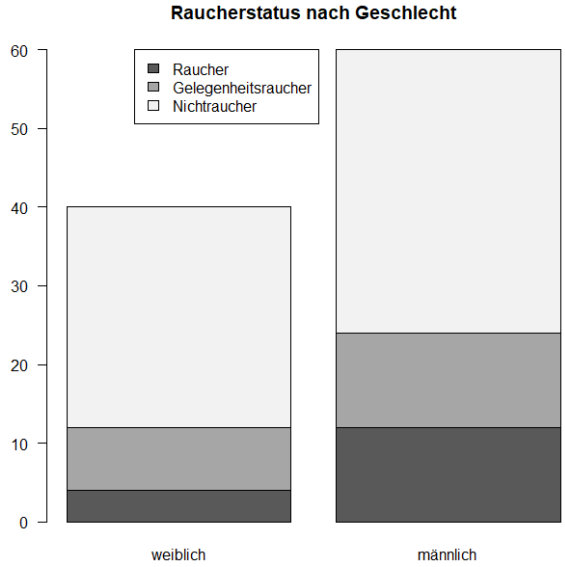


Abbildung 3.2.28: Modifiziertes gruppiertes Säulendiagramm mit Low-Level-Legende gruppiert nach X (Geschlecht)




```
[2,]   12   12   36
```

Die Spaltenverteilungen (bedingte Verteilung von X unter Y) erhält man über

```
> prop.table(m,2)
     [,1] [,2]   [,3]
[1,] 0.25  0.4 0.4375
[2,] 0.75  0.6 0.5625
```

bzw. in Prozentpunkten ausgedrückt dann über

```
> 100*prop.table(m,2)
     [,1] [,2]  [,3]
[1,]   25   40 43.75
[2,]   75   60 56.25
```

Ein unmodifiziertes ***segmentiertes Säulendiagramm*** mit X bedingt auf Y (Abb. 3.2.29) erhält man dann folglich über

```
> barplot(100*prop.table(m,2))
```

Die relativen Häufigkeiten wurden hier noch in Prozentpunkte umgerechnet.

Möchte man stattdessen die Zeilenverteilungen (bedingte Verteilung von Y unter X) darstellen, so müssen zunächst die Zeilenverteilungen von `m` ermittelt und Tabelle anschließend transponiert werden, da `barplot()` immer nur die Spalten einer vorgegebenen Datenmatrix abbildet.

```
> prop.table(m,1)
     [,1] [,2] [,3]
```

```
[1,]  0.1  0.2  0.7
[2,]  0.2  0.2  0.6
> t(prop.table(m,1))
     [,1] [,2]
[1,]  0.1  0.2
[2,]  0.2  0.2
[3,]  0.7  0.6
> 100*t(prop.table(m,1))
     [,1] [,2]
[1,]   10   20
[2,]   20   20
[3,]   70   60
```

Ein unmodifiziertes ***segmentiertes Säulendiagramm*** mit Y bedingt auf X (Abb. 3.2.30) erhält man dann über

```
> barplot(100*t(prop.table(m,1)))
```

Abbildung 3.2.29: Segmentiertes Säulendiagramm ohne Modifikationen und Low-Level-Zusätze X bedingt auf Y

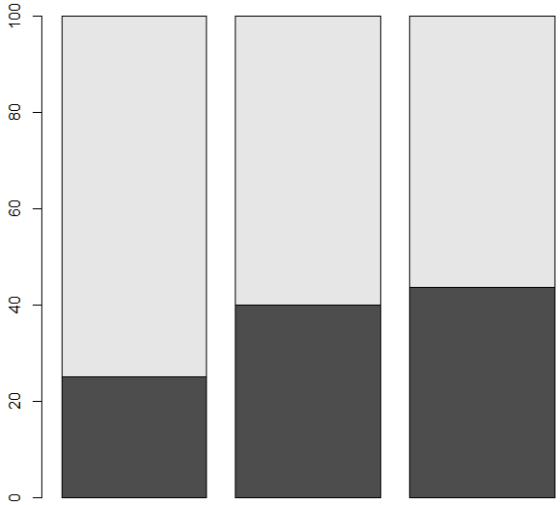
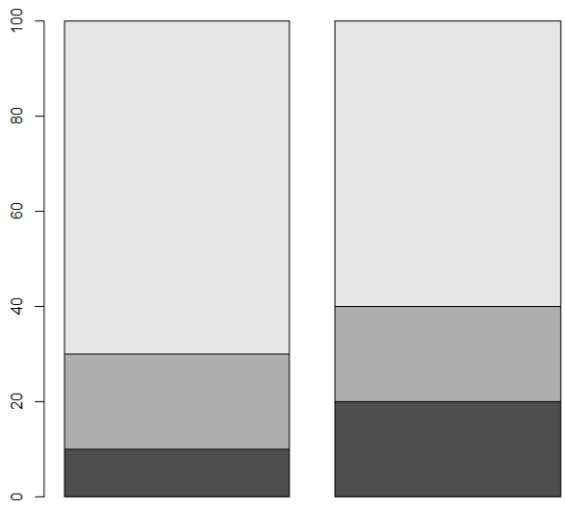


Abbildung 3.2.30: Segmentiertes Säulendiagramm ohne Modifikationen und Low-Level-Zusätze Y bedingt auf X



• **Segmentierte Balkendiagramme** • Für die Balkendarstellung muss in `barplot()` lediglich die Option `horiz=TRUE` gewählt werden.

Ein unmodifiziertes segmentiertes Balkendiagramm mit X bedingt auf Y (Abb. 3.2.31) erhält man über

```
> barplot(100*prop.table(m,2),horiz=TRUE)
```

Ein unmodifiziertes segmentiertes Balkendiagramm mit Y bedingt auf X (Abb. 3.2.32) erhält man über

```
> barplot(100*t(prop.table(m,1)),horiz=TRUE)
```

Abbildung 3.2.31: Segmentiertes Balkendiagramm ohne Modifikationen und Low-Level-Zusätze X bedingt auf Y

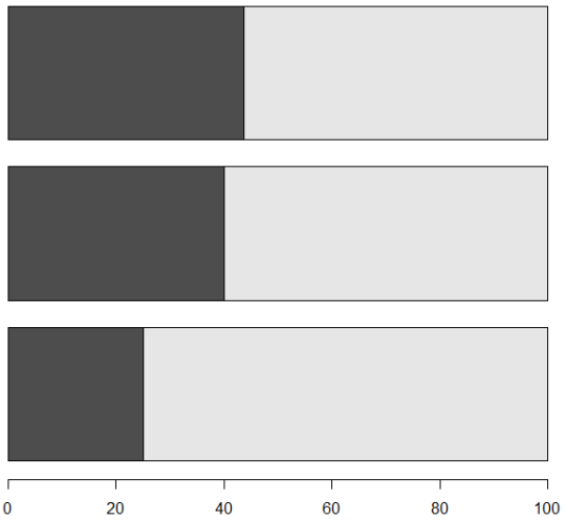
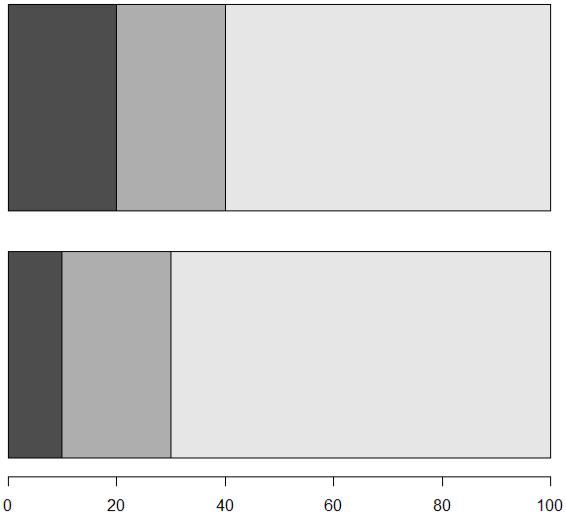


Abbildung 3.2.32: Segmentiertes Balkendiagramm ohne Modifikationen und Low-Level-Zusätze X bedingt auf Y



• **Modifikationen und Zusätze** • Über Modifikationen innerhalb von `barplot()` und Low-Level-Zusätze wie etwa `legend()` lassen sich die Grafiken entsprechend ansprechender gestalten. Man beachte hierzu die Beispiele aus den vorhergehenden Abschnitten.

Assoziationsplots

Assoziationsplots stellen die über die ***Pearson-Residuen*** gemessenen „Abweichungen von der Unabhängigkeit“ innerhalb einer zweidimensionalen Kontingenztabelle dar. Dadurch werden Unterschiede der bedingten Verteilungen leichter ersichtlich. Sie lassen sich mittels der im *R*-Paket `vcd` enthaltenen Funktion

`assoc`(m)

erzeugen. Das betreffende Paket muss also, sofern noch nicht vorhanden, zuvor erst installiert werden (vgl. Abschnitt 1.3.1).

Erneut betrachten wir als Beispiel-Daten die Kontingenztabelle aus den vorhergehenden Abschnitten:

```
> m
     [,1] [,2] [,3]
[1,]    4    8   28
[2,]   12   12   36
```

Ein darauf basierender unmodifizierter Assoziationsplot (Abb. 3.2.33) ergibt sich aus

```
> library(vcd)      # Laden des Pakets
> assoc(m)
```

Die Funktion `assoc()` erlaubt zahlreiche Modifikationen. Außerdem sind wie immer Low-Level-Zusätze möglich. Im Folgenden wird demonstriert wie man den Kategorien der beiden Variablen der Datenmatrix Namen zuordnen und darüber dann den Assoziationsplot gezielt mit sinnvollen Beschriftungen versehen kann (Abb. 3.2.34).

```
> dim(m)
[1] 2 3
> dimnames(m)
NULL
> dimnames(m)=list(
+   Geschlecht=c("weiblich","männlich"),
+   Raucherstatus=c("Raucher","Gelegenheitsraucher","Nichtraucher"))
> m
           Raucherstatus
Geschlecht Raucher Gelegenheitsraucher Nichtraucher
  weiblich       4                   8           28
  männlich      12                  12           36
> dimnames(m)
$Geschlecht
[1] "weiblich" "männlich"
$Raucherstatus
[1] "Raucher"             "Gelegenheitsraucher" "Nichtraucher"
> assoc(m)
```

Abbildung 3.2.33: Assoziationsplots (automatische Beschriftung)

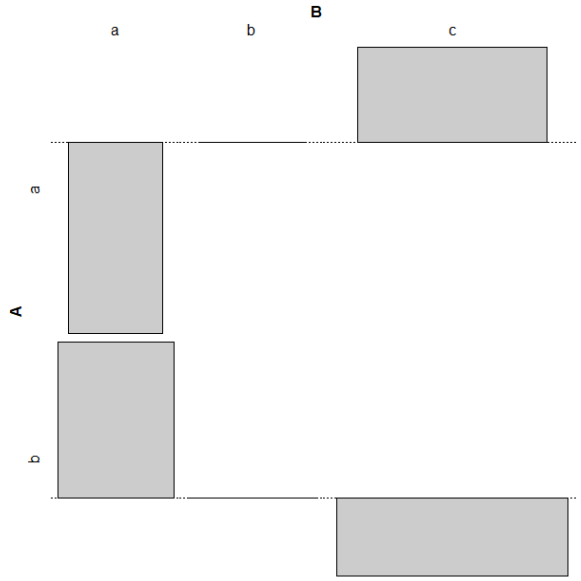
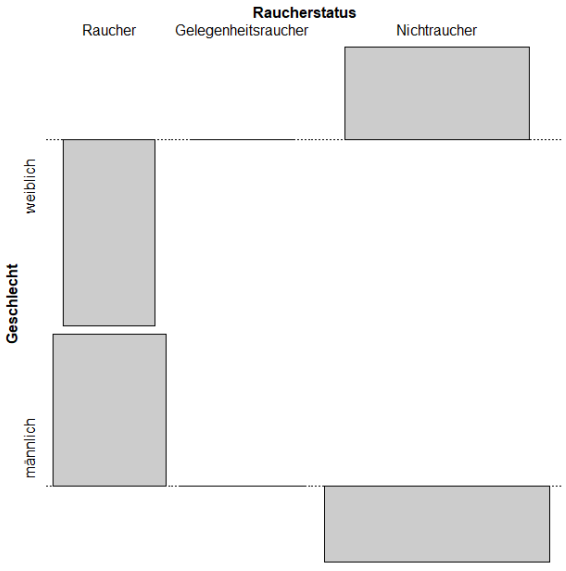


Abbildung 3.2.34: Assoziationsplots (gezielte Beschriftung)



Mosaikplots

• **Maßgebliche Funktion** • Mosaikplots sind eine Spezialform segmentierter Säulen- oder Balkendiagramme, bei denen die Flächen der Säulen der gemeinsamen Verteilung von zwei

oder mehr Merkmalen entsprechen. Sie lassen sich mit der im *R*-Paket `vcd` enthaltenen Funktion

`mosaic`(m)

erstellen. Das betreffende Paket muss also, sofern noch nicht vorhanden, zuvor erst installiert werden (siehe Abschnitt 1.3.1). Zwar gibt es auch die in der Standarddistribution von *R* bereits enthaltene Funktion `mosaicplot`(), diese produziert nach unseren Erfahrungen jedoch optisch nicht ganz so ansprechende Grafiken wie `mosaic()`.

• Zweidimensionale Mosaikplots • Zur Veranschaulichung ***zweidimensionaler Mosaikplots*** verwenden wir wieder die Kontingenztabelle der vorhergehenden Abschnitte. Die zuvorige Benennung der Kategorien über `dimnames()` behalten wir bei, d. h.

```
> m
          Raucherstatus
Geschlecht Raucher Gelegenheitsraucher Nichtraucher
  weiblich       4                   8           28
  männlich      12                  12           36
```

Einen unmodifizierten zweidimensionalen Mosaikplot (Abbildung 3.2.35, linke Grafik) mit Y bedingt auf X erhält man über

```
> library(vcd)     # Laden des Pakets, sofern noch nicht erfolgt
> mosaic(m)
```

Die Aufreihung der einzelnen Segmente erfolgt horizontal im Sinne von Balken bedingt auf dem „Zeilenmerkmal“ (hier Geschlecht) der zugrundeliegenden Tabelle.

Möchte man stattdessen eine vertikale Ausrichtung im Sinne von Säulen (Abb. 3.2.35, rechte Grafik), erhält man dies über

```
> mosaic(m,direction="v")
```

Möchte man Balken bzw. Säulen hingegen auf dem anderen Spaltenmerkmal (hier Raucherstatus) bedingen, so verwendet man die Transponierte von `m`. Unmodifizierte ***zweidimensionale Mosaikplots*** mit X bedingt auf Y (Abbildung 3.2.36) erhält man dann über

```
> mosaic(t(m))
```

bzw.

```
> mosaic(t(m),direction="v")
```

Über Modifikationen innerhalb von `mosaic()` und Low-Level-Zusätze lassen sich die Grafiken ansprechender gestalten und so etwa auch die Beschriftungen, die teils ineinander geschoben sind, lesbarer gestalten.

• Höherdimensionale Mosaikplots • Mit `mosaic()` lassen sich auch Mosaikplots für höherdimensionale Daten erstellen.

Als Beispiel wird im Folgenden der Datensatz Straftaten verwendet. Die Variablennamen des Data Frames erhält man über

Abbildung 3.2.35: Zweidimensionaler Mosaikplot ohne Modifikationen (Balken bzw. Säulen bedingen auf Geschlecht)

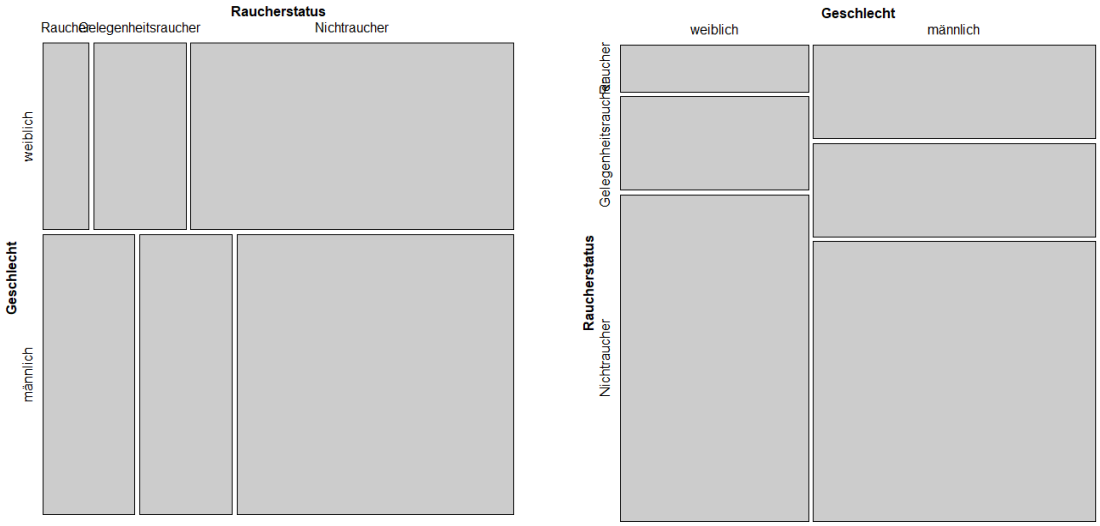
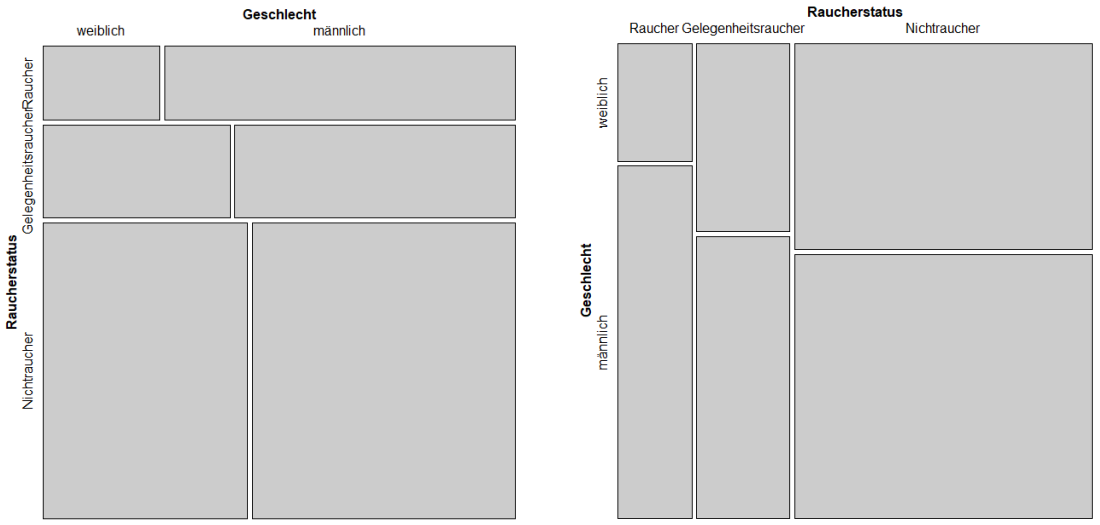


Abbildung 3.2.36: Zweidimensionaler Mosaikplot ohne Modifikationen (Balken bzw. Säulen bedingen auf Raucherstatus)




```
> names(Straftaten)
[1] "Delikt"       "Geschlecht"   "Alter"        "Nationalität"

          Eine 3-dimensionale Kontingenztabelle erhält man über
> attach(Staftaten)
> table(Nationalitaet,Delikt,Geschlecht)
, , Geschlecht = m

             Delikt
Nationalitaet     BM     BU     DU      K     PA      S      Ü     VA
           A   10128  17277  29884  14989   8596  27732  19477  18693
```

```
            D  38055  56172  72760  51139  36645 113319  42384  68326

, , Geschlecht = w

             Delikt
Nationalitaet    BM    BU    DU     K    PA     S     Ü    VA
            A   644  5355 10712  1322  1012  3356  3658  4107
            D  4717 27773 25614  5673  4761 22281  9607 17733
```

Auf dieser Tabelle basierend resultiert der ***3-dimensionale Mosaikplot*** (Abb. 3.2.37) dann aus

```
> Tabelle3D=table(Nationalitaet,Delikt,Geschlecht)
> mosaic(Tabelle3D)
```

Die Balken korrespondieren hier mit der Nationalität und die in den Balken enthaltenen Säulen mit der Deliktart. Als letzte Segmentierung verbleibt Geschlecht. Insgesamt entspricht dies der Reihenfolge der Tabellenerzeugung über `table()`. Die „optimale Reihenfolge“ der Variablen hängt davon ab, welche Erkenntnisse am deutlichsten herausgestellt werden sollen.

Der unmodifizierte ***4-dimensionale Mosaikplot*** in Abbildung 3.2.38 basiert auf dem Datensatz `ASEG`. Man erhält ihn über

Abbildung 3.2.37: 3-dimensionaler Mosaikplot ohne Modifikationen – Datensatz Straftaten

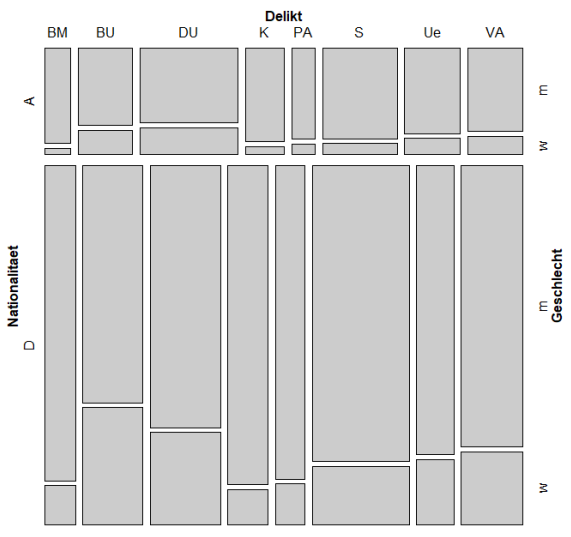
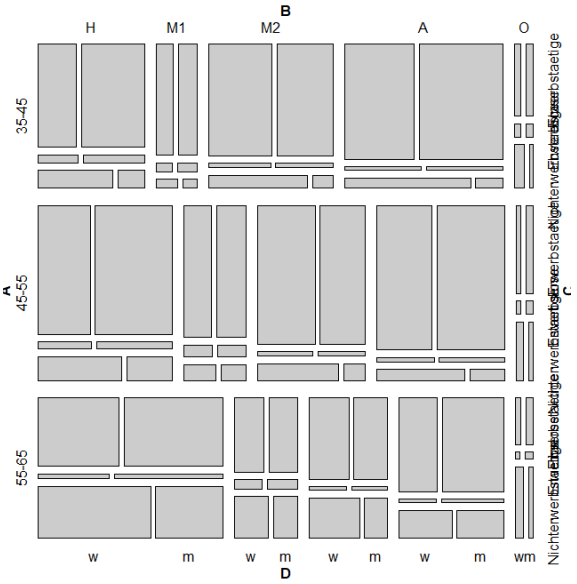


Abbildung 3.2.38: 4-dimensionaler Mosaikplot ohne Modifikationen – Datensatz `ASEG`




```
> mosaic(ASEG)
```

Er basiert auf der 4-dimensionalen Kontingenztabelle ASEG, d. h. auf

```
> ASEG
, , Erwerbstaetige, w

         H  M1   M2    A   O
35-45  784 415 1407 1565 108
45-55 1298 748 1551 1554 117
55-65 1083 444  652  727  67

, , Erwerbslose, w

       H M1 M2  A  O
35-45 63 31 53 50 20
45-55 73 70 48 45 17
55-65 64 52 25 26  8

, , Nichterwerbstaetige, w

         H  M1  M2   A   O
35-45  263  40 248 255 110
45-55  393 107 268 196 120
55-65 1141 291 412 294 151


, , Erwerbstaetige, m

         H  M1   M2    A   O
35-45 1305 445 1238 1905 142
45-55 1915 789 1212 1896 161
55-65 1324 447  546 1158  86

, , Erwerbslose, m

        H M1 M2  A  O
35-45  96 37 49 50 25
45-55 103 69 41 51 25
55-65  98 57 24 43 15

, , Nichterwerbstaetige, m

        H  M1  M2   A  O
35-45  96  27  51  56 40
45-55 211  82  69  76 65
55-65 685 208 185 261 84
```

Die Hauptbalken korrespondieren hier mit Alter und die in diesen enthaltenen Säulen mit der Schulbildung. Die in den Säulen wiederum enthaltenen Unterbalken korrespondieren mit dem Erwerbsstatus. Als letzte Segmentierung verbleibt Geschlecht. Wie man sieht, treten beim Erwerbsstatus Platzprobleme in Bezug auf die Beschriftung am rechten Rand auf.

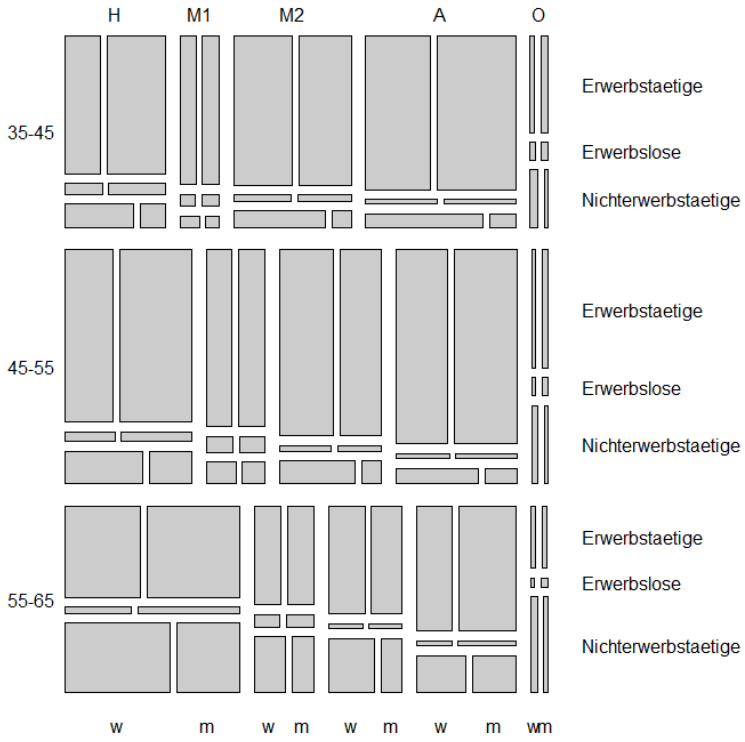
Die deutlich ansprechendere Darstellung von Abbildung 3.2.39 erhält man über

```
> mosaic(ASEG,main="",las=1,gp_labels=gpar(cex=1),
+   labeling_args=list(offset_labels=c(0,.5,.5,.5),
+   pos_labels=c("center","center","center","center"),
+   just_labels=c("center","left","center","center"),varnames=FALSE,
```

```
+    rot_labels=c(0,0,0,0)),margins=c(0,10,3,3))
```

Auf die Kommentierung der zahlreichen Parameter und Optionswahlen sei aufgrund sonst ausufernder Länge verzichtet.

Abbildung 3.2.39: 4-dimensionaler Mosaikplot mit besserer Lesbarkeit – Datensatz `ASEG`



Spineplots

Spineplots sind im Grunde eine Mischung zwischen segmentierten Säulendiagrammen und zweidimensionalen Mosaikplots. Zwischen den einzelnen Segmenten der Säulen entstehen jedoch keine Lücken. Außerdem befindet sich am Rand in der Regel eine Skalenachse. Sie lassen sich mit der Standardfunktion `plot`() erstellen.

Die Abbildungen 3.2.40 und 3.2.41 basieren jeweils auf dem Datensatz *Sternzeichen*, der nur die beiden Variablen Glaube und SternWahr und 206 Beobachtungen enthält.

```
> dim(Sternzeichen)
[1] 206   2
> names(Sternzeichen)
[1] "Glaube"    "SternWahr„
```

Den ersten nur leicht modifizierten Spineplot erhält man über

```
> attach(Sternzeichen)
> plot(as.factor(Glaube)~as.factor(SternWahr),
+  xlab="Sternzeichen",ylab="Glaube",main="")
```

den zweiten über

```
> plot(as.factor(SternWahr)~as.factor(Glaube),
+   xlab="Glaube",ylab="Sternzeichen",main="")
```

Im Wesentlichen werden die beiden numerischen Vektoren Glaube und SternWahr nur in den Datentyp ***Factor*** mittels `as.factor()` umgewandelt und erklärt, auf welchem Merkmal bedingt werden soll.

Abbildung 3.2.40: Spineplot - Säulen gemäß Einstellung zu Sternzeichen

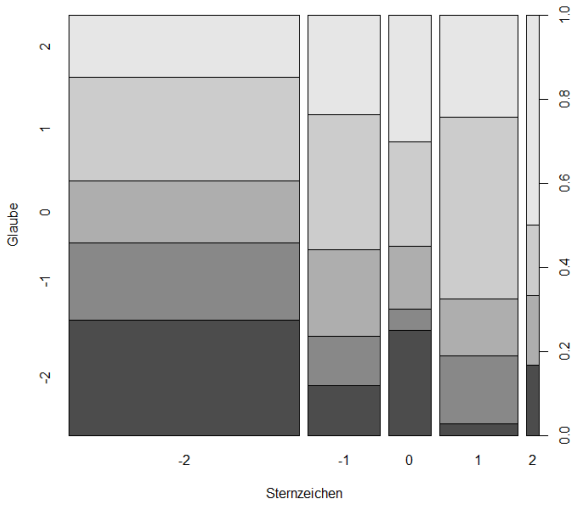
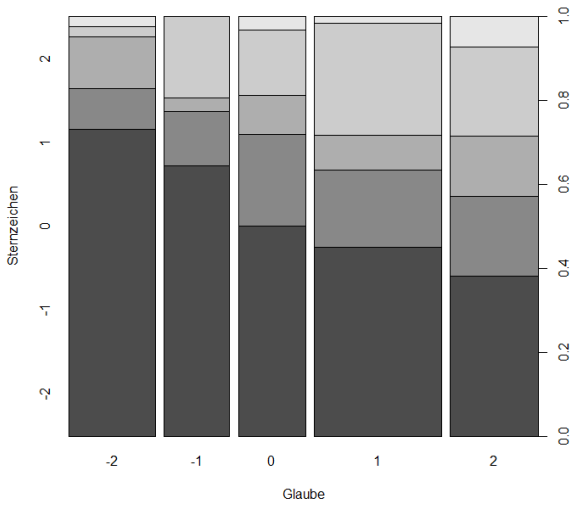


Abbildung 3.2.41: Spineplot - Säulen gemäß Einstellung zu Glauben



3.2.3 Darstellung metrischer Merkmale

Übersicht

• **Vorgehensweise** • Die im Folgenden präsentierten Darstellungsmöglichkeiten für metrische Merkmale werden in Stocker und Steinke [2017] in den Abschnitten 3.2.2, 3.2.3, 5.2.1 und 5.2.3 erklärt. Einige in diesen Abschnitten auftretende Abbildungen werden im Nachfolgenden nahezu exakt oder in etwas vereinfachter Form reproduziert.

• **Datenbasis** • Die vorgestellten Grafiken basieren alle auf Daten mit metrisch skalierten Variablen. Im Folgenden werden unter anderem die Datensätze `Deutschland` und `Studenten2` als Datenbasis herangezogen. Man beachte hierzu die Hinweise zu den Datensätzen im Vorwort.

Stamm-Blatt-Diagramme

Zur Erstellung von Stamm-Blatt-Diagrammen bietet sich die Funktion

stem(x)

an. Da der Output als Text im Command-Fenster erscheint, ist sie keine Grafikfunktion im engeren Sinne.

Als erstes Beispiel wird der Datensatz `Deutschland` verwendet, der unter anderem die Variable `Arbeitslosigkeit` enthält.

```
> dim(Deutschland)
[1] 396   4
> names(Deutschland)
[1] "Bevoelkerungsdichte" "Wanderungssaldo"     "Insolvenzen"
[4] "Arbeitslosigkeit"
```

Ein unmodifiziertes ***Stamm-Blatt-Diagramm*** erhält man dann über

```
> stem(Deutschland$Arbeitslosigkeit)

  The decimal point is at the |

   1 | 4
   2 | 1111122334456667777778888888888899999999
   3 | 000011111222222222333333455555555666666777777888889999
   4 | 00000000111111111122223333333444444555556667777788899999999999
   5 | 000000000111123334444445555556667778888888999999
   6 | 00000011112222223333444445555566666778889 99
   7 | 0000011122233344455556666777888899999
   8 | 00001112222244555666
   9 | 00122223344444556667789 9
  10 | 001222333344456666677779
  11 | 00001255668
  12 | 00111224457889
  13 | 0011223677
  14 | 12358
  15 | 1
  16 | 37
```

Über die Parameter `scale` und `width` lassen sich Höhe und Breite des „Baumes“ beeinflussen. Man probiere es selbst aus. Meist werden jedoch mit den in der Funktion voreingestellten Optionen `scale=1` und `width=80` bereits recht vernünftige Ergebnisse erzielt.

Histogramme

Histogramme können über die Standardfunktion

hist(x)

erstellt werden.

Basierend auf dem Datensatz `Deutschland` erhält man für Arbeitslosigkeit ein unmodifiziertes Histogramm (Abb. 3.2.42) über

```
> attach(Deutschland)
```

```
> hist(Arbeitslosigkeit)
```

Abbildung 3.2.43 resultiert aus

```
> hist(Arbeitslosigkeit,prob=TRUE,
+   xlab="Arbeitslosigkeit in %",ylab="Häufigkeitsdichte",
+   xlim=c(0,20),ylim=c(0,0.15),las=1,
+   main="Verteilung der Arbeitslosigkeit")
```

Die Umstellung der voreingestellten Option `prob=FALSE` auf `prob=TRUE`, bewirkt, dass die Häufigkeitsdichte anstelle der absoluten Klassenhäufigkeit auf der y-Achse dargestellt wird.

Abbildung 3.2.42: Histogramm mit absoluten Klassenhäufigkeiten - ohne Modifikationen

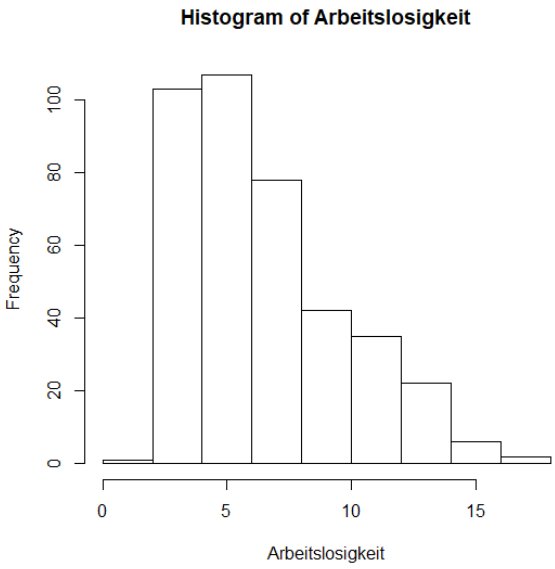
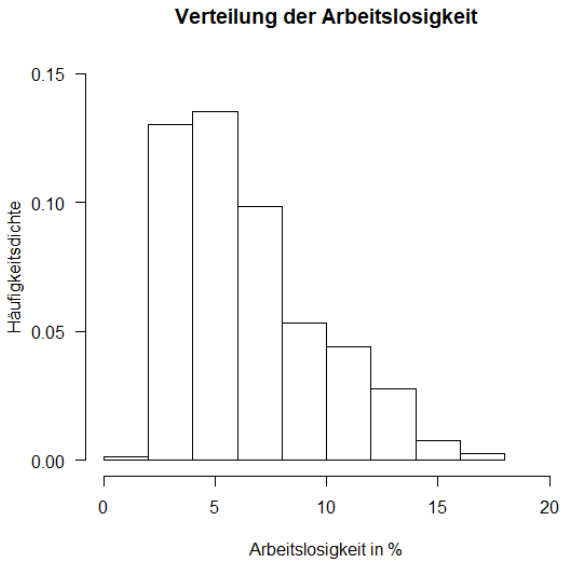


Abbildung 3.2.43: Modifiziertes Histogramm als Häufigkeitsdichte



Einige Grafikfunktionen erzeugen nicht nur eine Grafik als Output, sondern geben auch Datenobjekte in Form von Vektoren oder Listen aus, die eine weitergehende und über die Abbildung hinausgehende Analyse und Bearbeitung ermöglichen. An die entsprechenden Ausgabeergebnisse gelangt man über Zuweisungen. Im Falle von `hist()` wird ein `list`-Objekt (vgl. Abschnitt 2.4) erzeugt. Die Komponentennamen lassen sich in diesem Fall dann über `names()` erfragen. Der Zugriff erfolgt dann über Listensyntax bzw. `attach()`. Als Beispiel betrachten wir folgende Ausführungen:

```
> reshist=hist(Arbeitslosigkeit,plot=FALSE)
> names(reshist)
[1] "breaks"   "counts"   "density"  "mids"     "xname"    "equidist"
> attach(reshist)
> breaks
 [1]  0  2  4  6  8 10 12 14 16 18
> counts
[1]   1 103 107  78  42  35  22   6   2
> density
```

```
[1] 0.001262626 0.130050505 0.135101010 0.098484848
[5] 0.053030303 0.044191919 0.027777778 0.007575758
[9] 0.002525253
> mids
[1]  1  3  5  7  9 11 13 15 17
> xname
[1] "Arbeitslosigkeit"
> equidist
[1] TRUE
```

Der Vektor `breaks` (als Teilkomponente von `reshist`) enthält die Klassengrenzen, `counts` die absoluten Klassenhäufigkeiten, `density` die Klassendichte und `mids` die Klassenmitten. Mit `xname` wird der Name der Variablen ausgegeben, deren Verteilung von Interesse ist. Die logische Ausgabe von `equidist` klärt, ob die Klassengrenzen äquidistant sind oder nicht.

Über den Parameter `breaks` können im Befehl `hist()` alternative Klassengrenzen gewählt werden. Abbildung 3.2.44 resultiert aus

```
> par(mfrow=c(2,2))
> hist(Arbeitslosigkeit,prob=TRUE,ylim=c(0,0.2))
> hist(Arbeitslosigkeit,prob=TRUE,ylim=c(0,0.2),breaks=0:18)
> hist(Arbeitslosigkeit,prob=TRUE,ylim=c(0,0.2),breaks=c(0,5,10,15,20))
> hist(Arbeitslosigkeit,prob=TRUE,ylim=c(0,0.2),breaks=c(0,6,12,18))
> par(mfrow=c(1,1))
```

Meist werden mit der von `hist()` gewählten Klasseneinteilung jedoch schon recht vernünftige Ergebnisse erzielt.

Boxplots

Boxplots können mit dem Befehl

boxplot(x)

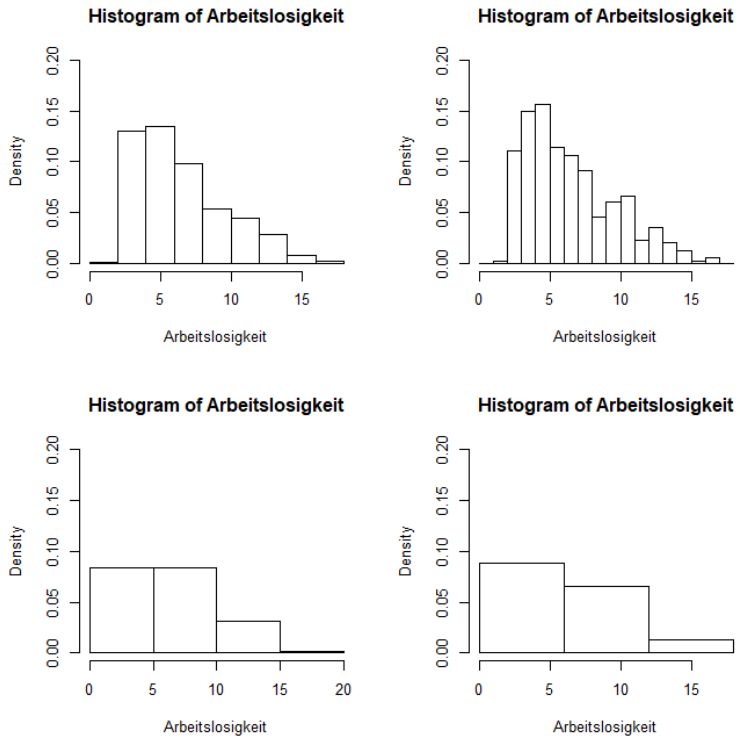
auf unterschiedliche Weise erzeugt werden. So lässt sich der Befehl auf ein oder mehrere Vektoren, auf ganze Data Frames (Matrizen) bzw. Teile davon oder auf eine Formelstruktur via „∼“ anwenden. Letzteres ist in erster Linie bei der Darstellung gemischt skalierter Fälle gefragt. Hierzu beachte man die Ausführungen und Beispiele in Abschnitt 3.2.4, S.115ff. Die anderen Möglichkeiten werden im Folgenden anhand der Datensätze `Deutschland` und `Aktienindizes` demonstriert.

Den unmodifizierten einzelnen Boxplot für die Verteilung von `Arbeitslosigkeit` (Abb. 3.2.45, linke Grafik) erhält man über

```
> attach(Deutschland)
> boxplot(Arbeitslosigkeit)
```

Den modifizierten Boxplot in horizontaler Ausrichtung (Abb. 3.2.45, rechte Grafik) erhält man über

Abbildung 3.2.44: Histogramme mit unterschiedlichen Klassengrenzen




```
> boxplot(Arbeitslosigkeit,horizontal=TRUE)
```

Angewendet auf den gesamten Data Frame erhält man den unmodifizierten multiplen Boxplot von Abbildung 3.2.46 über

```
> boxplot(Deutschland)
```

Die Erstellung eines solchen könnte etwa im Zuge eines ersten sog. Datenscreenings sinnvoll sein, mit dem Auffälligkeiten oder mögliche Fehler frühzeitig identifiziert werden sollen.

Selektiver erhält man den horizontal ausgerichteten multiplen Boxplot von Abbildung 3.2.47 entweder über

```
> boxplot(Deutschland[,2:3],horizontal=TRUE)
```

oder (äquivalent) über

```
> boxplot(Wanderungssaldo,Insolvenzen,horizontal=TRUE,
+   names=c("Wanderungssaldo","Insolvenzen"))
```

In letzterem Fall sind zusätzlich noch die Variablennamen zu spezifizieren, da die einzelnen Boxplots im Schaubild ansonsten standardmäßig lediglich durchnummeriert werden.

Streudiagramme

Gewöhnliche Streudiagramme werden meist mit der „Universalfunktion“

Abbildung 3.2.45: Einzelner Boxplot ohne Modifikationen - vertikale und horizontale Ausrichtung

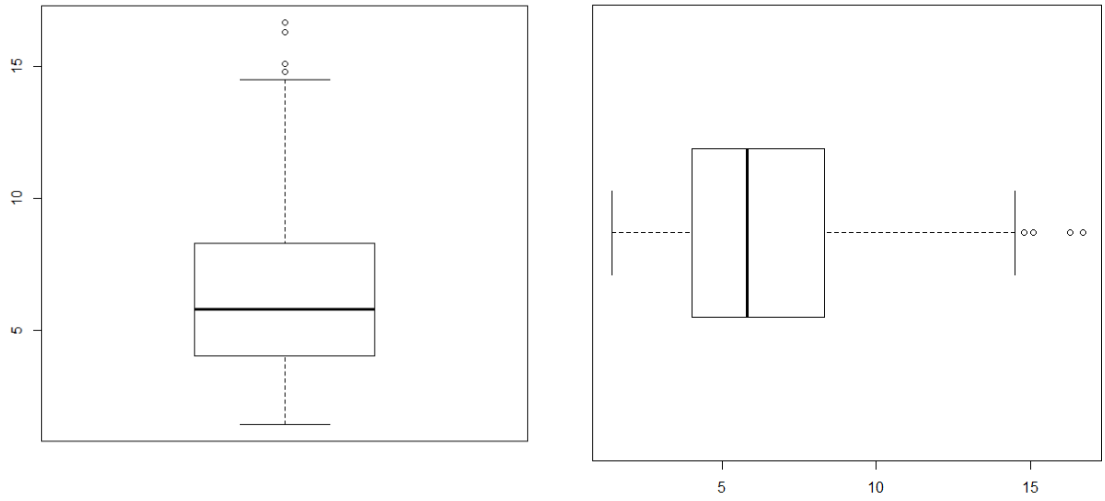


Abbildung 3.2.46: Multipler Boxplot für einen gesamten Datensatz ohne Modifikationen

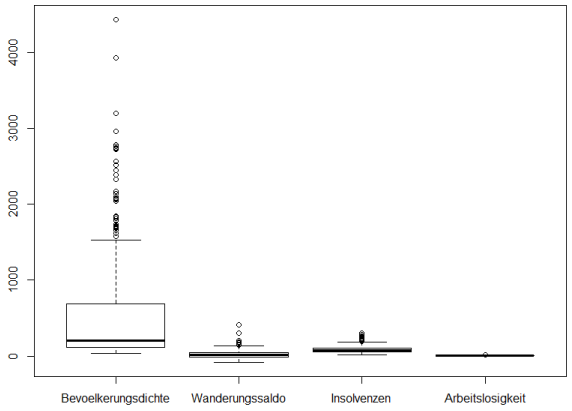
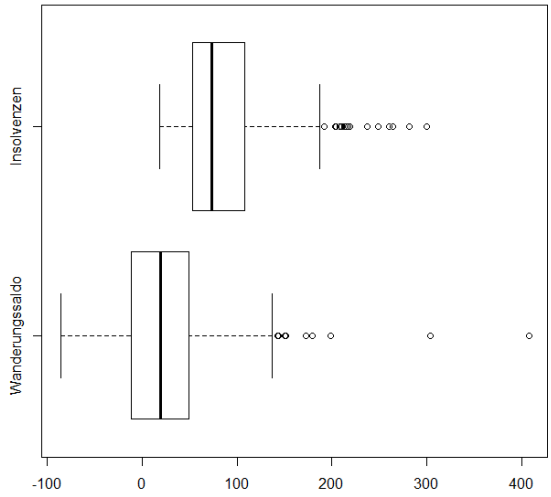


Abbildung 3.2.47: Multipler Boxplot für eine Selektion von Variablen



`plot`(x,y)

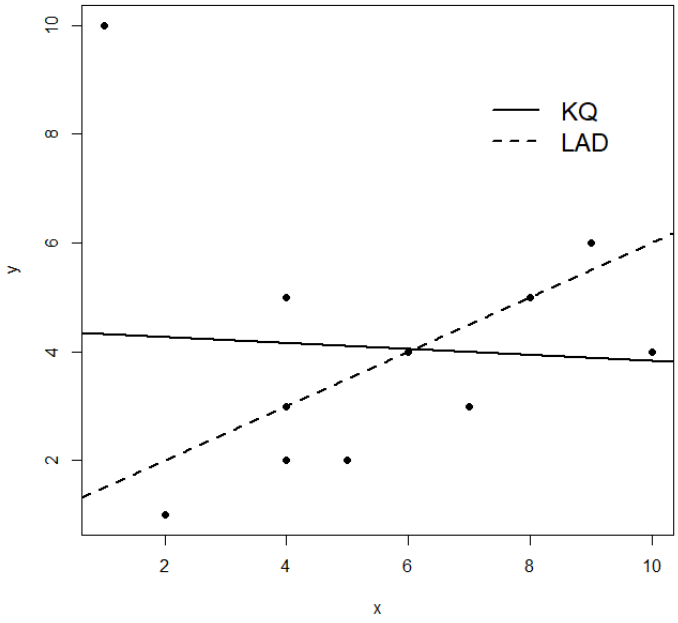
erstellt. Ausführungen zu den vielfältigen Möglichkeiten, wie Streudiagramme durch verschiedene Optionen und Low-Level-Elemente modifiziert bzw. ergänzt werden können, finden sich im einführenden Abschnitt 3.2.1, S.80ff. In Ergänzung hierzu sollen nachfolgend verschiedene Techniken vorgestellt werden, mit denen sich in bestehende Streudiagramme Regressionsgeraden einfügen lassen - ein praktisches Problem, das relativ häufig im Rahmen empirischer Analysen auftritt. Weitere Spezialfälle von Streudiagrammen finden sich später dann noch in Abschnitt 3.2.4.

Streudiagramme mit Regressionsgeraden

• **Maßgebliche Funktionen** • Da in Stocker und Steinke [2017] lediglich die KQ-Methode (L_2-Regression) und die LAD-Methode (L_1-Regression) thematisiert werden, sollen im Folgenden auch nur diese beiden Regressionstechniken ausgeführt werden. Die maßgeblichen Regressionsfunktionen, die wir hier verwenden, sind `lm`() (engl. *linear model*) für die KQ-Regression und `rq`() (engl. *quantile regression*) aus dem *R*-Paket `quantreg` für die LAD-Regression. Die beiden Funktionen wurden in Abschnitt 3.1.4 ausführlicher vorgestellt.

Auf den Regressionsergebnissen basierend lassen sich entsprechende Regressionsgeraden dann über Low-Level-Funktionen wie etwa `abline`() oder `segments`() einzeichnen. Erstere nutzt Achsenabschnitt und Steigung der Geraden und lässt sich diesbezüglich auf verschiedene Weise verwenden. Häufig werden über die Funktionsargumente `a` und `b` (Achsenabschnitt bzw. Steigung) der einzuzeichnenden Gerade spezifiziert. Kompakter lässt sich dies aber auch über das Vektorargument `coef` bewerkstelligen. Die wohl einfachste Methode besteht darin, der Funktion `abline()` den gesamten Regressionsoutput zu übergeben. In der Funktion `segments()` müssen die Koordinaten von Start- und Endpunkt der einzuzeichnenden Geraden spezifiziert werden, was den Vorteil mit sich bringt, die Länge der Gerade innerhalb des Schaubildes bestimmen zu können. Die verschiedenen Möglichkeiten werden nachfolgend anhand konkreter Beispiele vorgestellt.

Abbildung 3.2.48: Modifiziertes Streudiagramm mit KQ- und LAD-Gerade via `abline()`



• **Gerade via `abline()`** • Inklusive der Dateneingabe erhält man Abbildung 3.2.48 über

```
> x=c(2,4,4,5,6,8,9,10,4,7,1)
> y=c(1,2,3,2,4,5,6,4,5,3,10)
> plot(x,y,pch=16)
```

```
> lm_out=lm(y~x)
> library(quantreg)
> rq_out=rq(y~x)
> abline(lm_out,lwd=2)
> abline(rq_out,lty=2,lwd=2)
> legend(7,9,lty=c(1,2),lwd=c(2,2),legend=c("KQ","LAD"),cex=1.5,bty="n")
```

Man beachte, dass `lm()` und `rq()` `list`-Objekte erzeugen, in denen die Regressionskoeffizienten jeweils in der Komponente `coefficients` enthalten sind (vgl. Abschnitt 3.1.4). Die Funktion `abline()` extrahiert dann automatisch die benötigten Koeffizienten aus den jeweiligen `list`-Objekten.

• **Gerade via** `segments()` • Mit Hilfe der Funktion `segments()` können Anfang und Ende einer Geraden festgelegt werden.

Unter Zugrundelegung von `x` und `y` aus dem vorhergehenden Quellcode erhalten wir Abbildung 3.2.49 aus

```
>  plot(x,y,xlim=c(-2,12),ylim=c(0,12),pch=16)
>  lm_out=lm(y~x)
>  lm_out

Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
    4.39189     -0.05518
> 4.39189-0.05518*0           # Prognose des y-Wertes für x=0
[1] 4.39189
> 4.39189-0.05518*11          # Prognose des y-Wertes für x=11
[1] 3.78491
> segments(0,4.39189,11,3.78391,lwd=2)
```

Zum Einzeichnen der KQ-Gerade werden in diesem Fall die durch die Gerade prognostizierten y-Werte zunächst an den x-Stellen 0 und 11 berechnet. Dies geschieht durch Einsetzen in die ermittelte Geradengleichung. Die Start- und Endpunktkoordinaten ergeben sich dabei als

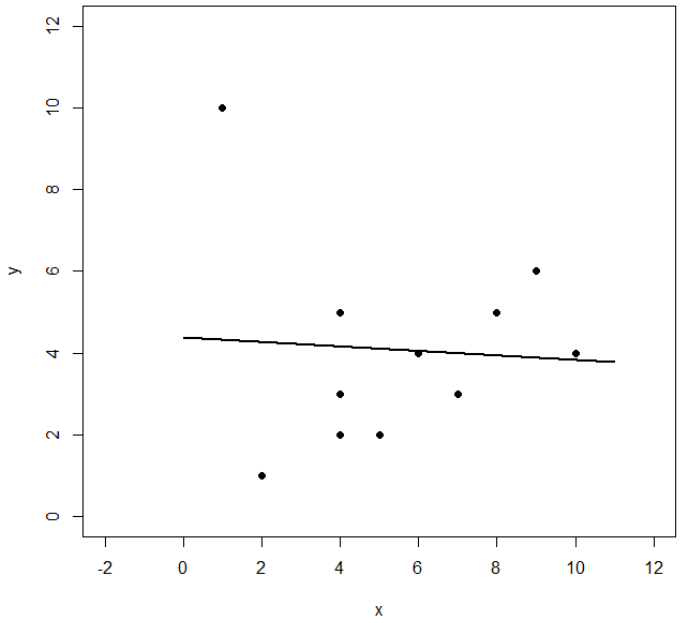
$$(0,\ 4.39189) \text{ und } (11,\ 3.78391).$$

Diese Koordinaten werden anschließend der Funktion `segments()` übergeben.

Hätte man in diesem Fall die Funktion `abline()` verwendet, würde die Gerade außerhalb des Wertebereichs der x-Achse beginnen und enden, was optisch etwas unschön ist. Man bilde sich selbst ein Urteil durch Eingabe von

```
> abline(lm_out)
```

Abbildung 3.2.49: Modifiziertes Streudiagramm mit KQ-Gerade via `segments()`



Streudiagramm-Matrizen

Streudiagramm-Matrizen lassen sich über die Funktion `pairs`(m) erstellen. `m` ist hierbei ein Data Frame ode eine Datenmatrix.

Eine Anwendung von `pairs()` sei im Folgenden anhand des Datensatzes `Deutschland` demonstriert. Abbildung 3.2.50 erhält man über

```
pairs(Deutschland)
```

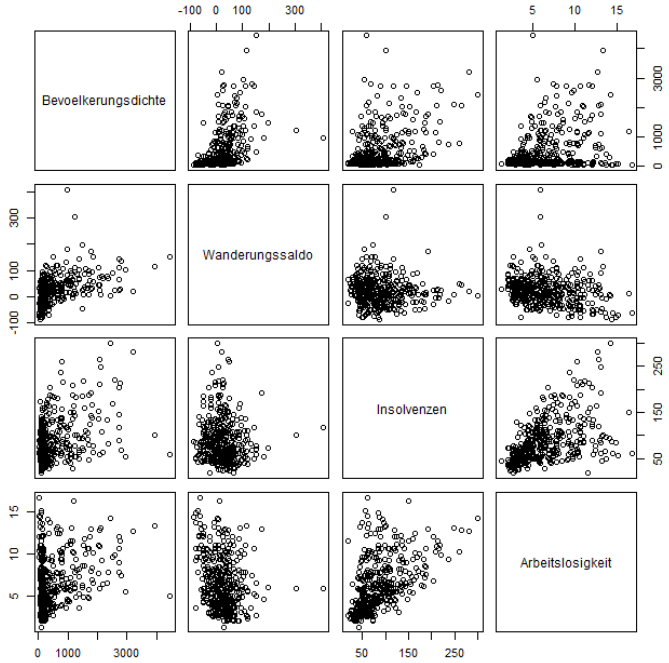
3.2.4 Darstellung gemischt skalierter Fälle

Übersicht

• **Vorgehensweise** • Die im Folgenden präsentierten Darstellungsmöglichkeiten für gemischt-skalierte Fälle (gleichzeitiges Vorliegen metrischer und kategorialer Merkmale) werden in Stocker und Steinke [2017] in Abschnitt 5.3.1 unter „Grafische Analysemöglichkeiten“ erklärt. Einige dort zu findende Abbildungen werden im Nachfolgenden in vereinfachter Form reproduziert. Darüber hinaus wird noch die Erstellung von Streudiagrammen mit kategorisierten Symbolen besprochen, was sich als Spezialfall zur Darstellung gemischt skalierter Fälle im 3-dimensionalen Sinne deuten lässt.

• **Datenbasis** • Im Folgenden werden die Datensätze `Studenten1`, `Studenten2`, `Deutschland` und `Wohnen` als Datenbasis herangezogen. Man beachte hierzu die Hinweise zu den Datensätzen im Vorwort.

Abbildung 3.2.50: Streudiagramm-Matrix ohne Modifikationen



Boxplots: Metrisches vs. kategoriales Merkmal

Stellt man die Verteilung eines metrischen Merkmals in Abhängigkeit von einem kategorialen Merkmal dar, so eignen sich hierfür ***Boxplots***. Hierbei ist dann eine Formelstruktur via „∼“-Operator in der Form

Metrisches Merkmal ∼ Kategoriales Merkmal

vorzugeben. Dies sei im Folgenden anhand des Datensatzes `Studenten2` demonstriert. Dimensionierung und enthaltene Variablen erhält man über

```
> dim(Studenten2)
[1] 214   3
> names(Studenten2)
[1] "Abiturnote" "Alter"      "Master"
```

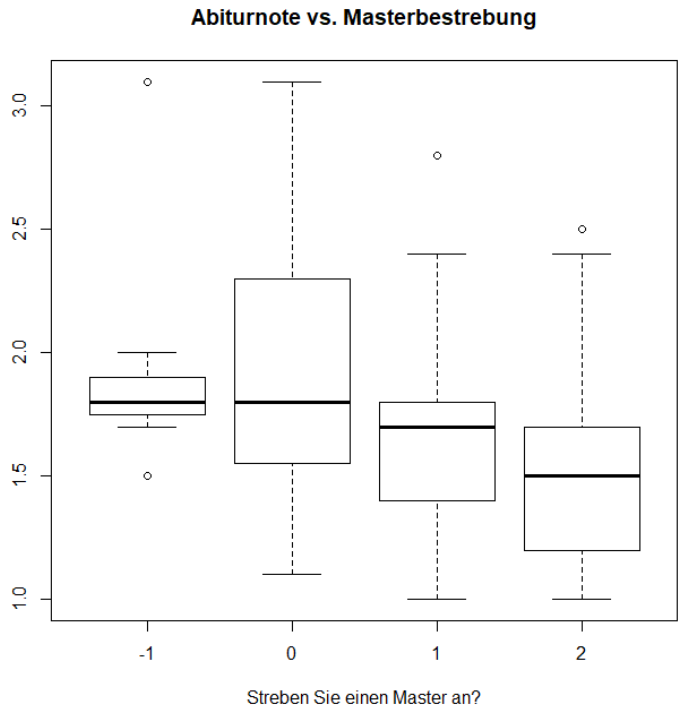
Darauf basierend erhalten wir Abbildung 3.2.51 über

```
> attach(Studenten2)
> boxplot(Abiturnote~Master,xlab="Streben Sie einen Master an?",
+ main="Abiturnote vs. Masterbestrebung")
```

Spinogramme: Kategoriales vs. metrisches Merkmal

Stellt man die Verteilung eines kategorialen Merkmals in Abhängigkeit von einem metrischen Merkmal dar – ein für die meisten Anwender wohl eher weniger geläufiger Ansatz – so eignen

Abbildung 3.2.51: Multipler Boxplot: Metrisches Merkmal vs. 4 Kategorien



sich hierfür ***Spinogramme***. Dies sind im Grunde spezielle segmentierte Säulendiagramme basierend auf klassierten Daten. Dabei werden die Breiten der Säulen in ein proportionales Verhältnis zu den einzelnen Klassenhäufigkeiten gesetzt werden. Hierbei wird die Formelstruktur

$$\text{Kategoriales Merkmal} \sim \text{Metrisches Merkmal}$$

vorgegeben.

Im Unterschied zu den ***Spineplots*** wird beim Spinogramm die kategoriale Variable im Datentyp `factor` übergeben.

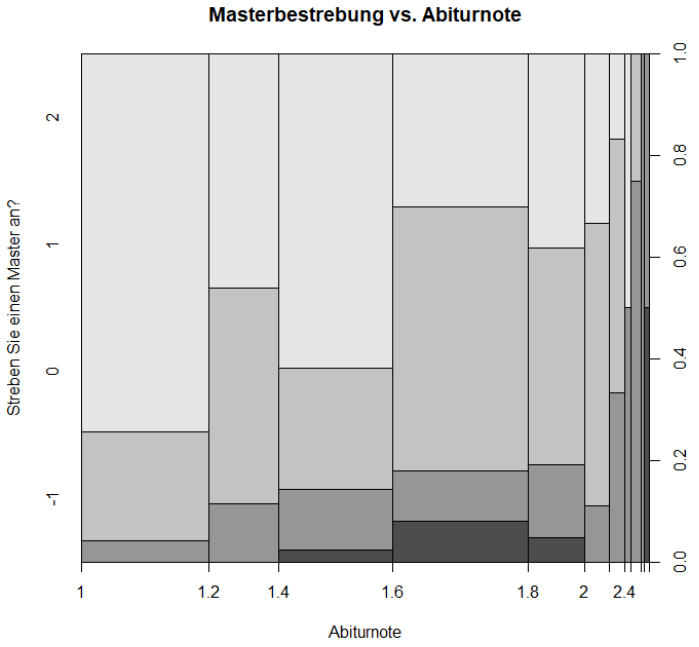
Basierend auf dem Datensatz `Studenten2` erhalten wir Abbildung 3.2.52 über

```
> attach(Studenten2)
> plot(as.factor(Master)~Abiturnote,ylab="Streben Sie einen Master an?",
+   main="Masterbestrebung vs. Abiturnote")
```

Streudiagramme mit kategorisierten Beobachtungspunkten

Anstatt alle 2-dimensionalen Beobachtungen in einem Streudiagramm mit dem gleichen Symbol (etwa durch einen Punkt) oder der gleichen Farbe darzustellen, kann beides auch entsprechend den Ausprägungen eines kategorialen Merkmals variiert werden. Damit erhält man dann im Grunde eine 3-dimensionale grafische Darstellungsform für zwei metrische und ei-

Abbildung 3.2.52: Spinogramm: 4 Kategorien vs. metrisches Merkmal



ne kategoriale Variable. Als Beispiele beachte man die nachfolgenden Abbildungen 3.2.53 und 3.2.54. In ersterer wurde ein Streudiagramm bezüglich Größe und Gewicht erstellt, wobei die Punktsymbole gemäß Geschlecht variieren. In letzterem wurde der Zusammenhang zwischen der Veränderung von Einwohnerzahlen und Preisen dargestellt, wobei für die Beobachtungswerte anstelle von Punkten die Städtenamen abgebildet wurden. Grundsätzlich ließen sich etwa durch Variation von Größe und Farben der einzelnen Symbole durchaus noch höherdimensionale Darstellungsformen gewinnen.

Basierend auf dem Datensatz `Studenten1` erhalten wir Abbildung 3.2.53 über

```
> dim(Studenten1)
[1] 201   5
> names(Studenten1)
[1] "Geschlecht" "Groesse"      "Gewicht"    "Schuhgroesse" "Schlaf"
> attach(Studenten1)
> ifelse(Geschlecht==0,16,1)
  [1] 16 16  1 16  1 16  1 16 16 16 16 16 16 16 16 16  1 16 16 16
 [21]  1  1 16 16  1 16  1 16  1 16  1 16 16 16  1  1 16 16 16 16
 [41] 16 16 16 16 16 16  1  1  1  1 16 16 16 16 16 16 16 16 16 16
 [61]  1  1 16  1 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16  1
 [81] 16 16 16 16  1 16 16 16 16 16 16  1 16 16 16 16 16 16 16 16
[101] 16 16 16 16  1 16  1 16  1 16 16 16 16 16  1 16 16  1  1 16
[121] 16 16  1 16 16 16 16 16  1  1  1  1  1 16 16 16 16 16 16 16
[141] 16 16 16 16 16 16 16 16  1 16 16 16  1 16 16  1  1  1 16 16
[161] 16 16  1  1  1 16 16 16  1 16 16 16 16 16  1  1  1 16  1  1
[181] 16 16 16 16 16 16 16 16  1 16  1 16 16 16  1 16 16  1  1 16
[201]  1
> pchtyp=ifelse(Geschlecht==0,16,1)
```

Abbildung 3.2.53: Streudiagramm mit kategorisierten Beobachtungspunkten – Beispiel 1

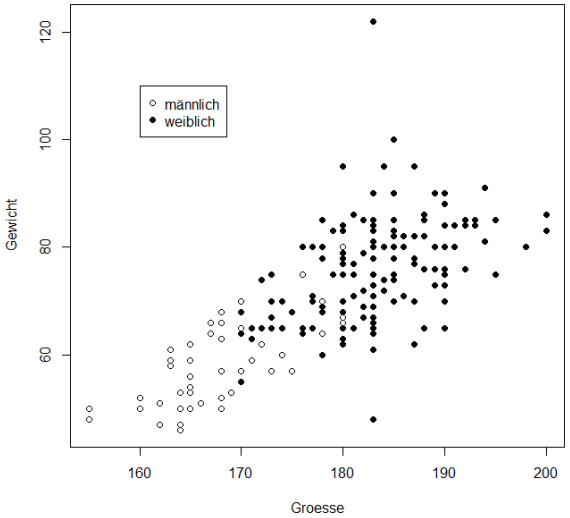
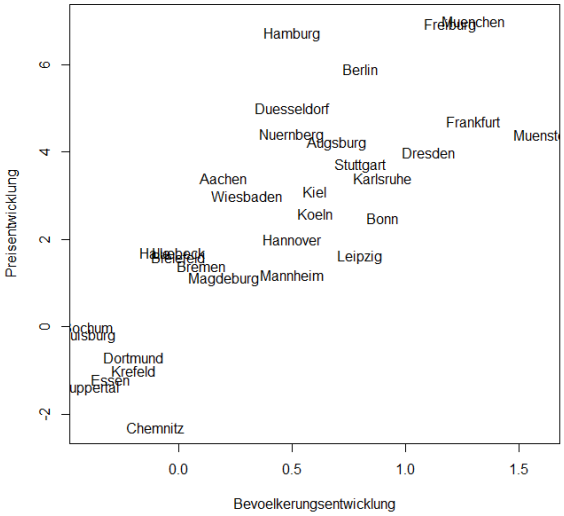


Abbildung 3.2.54: Streudiagramm mit kategorisierten Beobachtungspunkten - Beispiel 2




```
> plot(Groesse,Gewicht,pch=pchtyp)
> legend(160,110,legend=c("männlich","weiblich"),pch=c(1,16))
```

Der Befehl `ifelse()` wird in Abschnitt 2.6 besprochen. Symbol Nr. 16 erzeugt ausgefüllte Punkte, Symbol Nr. 1 hingegen leere Kreise.

Basierend auf dem Datensatz `Wohnen` erhalten wir Abbildung 3.2.54 über

```
> attach(Wohnen)
> plot(Bevoelkerungsentwicklung,Preisentwicklung,type="n")
> text(Bevoelkerungsentwicklung,Preisentwicklung,Stadt)
```

Mit `type="n"` innerhalb des `plot()`-Befehls wird das Einzeichnen von Punkten unterdrückt. Stattdessen werden dann mit dem Low-Level-Befehl `text()` die Namen der Städte in das Schaubild eingefügt. Über Low-Level-Funktionen wie etwa `points()`, `axis()` und `segments()` ließe sich Abbildung 3.2.54 noch dahingehend verfeinern, dass sie Abbildung 5.2.25 aus Stocker und Steinke [2017] weitestgehend entspricht. Das soll hier jedoch nicht weiter ausgeführt werden. Stattdessen sei der Leser auf die Ausführungen in Abschnitt 3.2.1 zu den vielfältigen Gestaltungsmöglichkeiten einer Grafik bzw. eines Streudiagramms verwiesen.

4 Ausgewählte induktive Analysemethoden

4.1 Wahrscheinlichkeitsverteilungen in *R*

In *R* sind statistische Funktionen zu den wichtigsten in der Statistik gebräuchlichen Verteilungen implementiert. I.d.R. kann man auf vier verschiedene Aspekte einer Verteilung zugreifen:

1. die Wahrscheinlichkeitsfunktion bzw. Dichtefunktion,
2. die Verteilungsfunktion,
3. die Quantile und
4. (Pseudo-)Zufallszahlen zu der Verteilung.

Mithilfe der Wahrscheinlichkeits- bzw. Dichtefunktion kann man Verteilungen grafisch veranschaulichen. Die Verteilungsfunktion dient u.a. zur Berechnung von Intervallwahrscheinlichkeiten der entsprechenden Verteilungen. Sie kann auch zur Berechnung von p-Werten von Tests verwendet werden. Quantile sind insbesondere zur Konstruktion von Konfidenzintervallen und Tests nützlich. Unter Verwendung von (Pseudo-)Zufallszahlen kann man statistische Simulationen durchführen und damit z.B. die Gültigkeit von asymptotischen Aussagen prüfen bzw. veranschaulichen.

Die allgemeine Befehlssyntax ist in *R* für alle Verteilungen einheitlich geregelt. Die vier oben erwähnten Aspekte sind in Form von vier Funktionen implementiert. Die Grundstruktur der Funktionen ist

Präfix + Stamm`(x,...)`.

Der Präfix ist einer der Buchstaben `d`, `p`, `q` oder `r`.

1. `d`: Das *„d“* steht für *„density“* (Zufall). Es wird eine Dichte- oder Wahrscheinlichkeitsfunktion an der Stelle `x` berechnet.
2. `p`: Das *„p“* leitet sich von *„probability“* (Wahrscheinlichkeit) ab. Hiermit können die Werte der entsprechenden Verteilungsfunktion bestimmt werden.
3. `q`: Das *„q“* kommt von *„quantile“* (Quantil). Die Funktion ermöglicht die Berechnung von Quantilen zum Niveau `x`.
4. `r`: Das *„r“* leitet sich von *„random“* (Zufall) ab. Mit diesen Funktionen lassen sich `x` (Pseudo-)Zufallszahlen zur vorgegebenen Verteilungen generieren.

Im Folgenden werden die Funktionen an den Beispielen der Normalverteilung und der Binomialverteilung vorgestellt.

4.1.1 Normalverteilung

Bei der Normalverteilung ist der Stamm `norm`. Daraus leiten sich die folgenden vier Funktionen ab:

`dnorm()`, `pnorm()`, `qnorm()` und `rnorm()`.

`dnorm(x)` berechnet den Wert der Dichtefunktion einer Standardnormalverteilung an der Stelle `x` und `pnorm(x)` den Wert der zugehörigen Verteilungsfunktion an der Stelle `x`. Mit `qnorm(x)` kann man das entsprechende Quantil zum Quantilsniveau `x` ermitteln. `rnorm(n)` generiert `n` standardnormalverteilte (Pseudo-)Zufallszahlen. Mit den optionalen Argumenten `mu` bzw. `sd` kann man einen Erwartungswertparameter μ bzw. eine Standardabweichung σ zuweisen und die entsprechenden Funktionen für eine allgemeine Normalverteilung berechnen. Die Funktionen haben damit folgende allgemeine Gestalt:

`znorm(x,mu=0,sd=1)`

Hierbei steht `z` für den Präfix, also für `d`, `p`, `q` bzw. `r`.

```
> x=seq(-5,5,le=200)
> y= dnorm(x)
> plot(x,y,type="l")
```

Mit der Wahl der Option `type="l"` im `plot()`-Befehl werden die einzelnen Datenpunkte durch eine Linie (`"l"`) verbunden. Das Ergebnis finden Sie in Abbildung 4.1.1. Es wird die Dichte einer Standardnormalverteilung dargestellt.

Abbildung 4.1.1: Dichte der Standardnormalverteilung

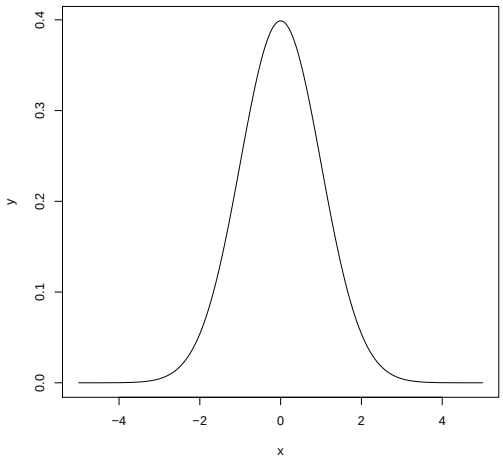
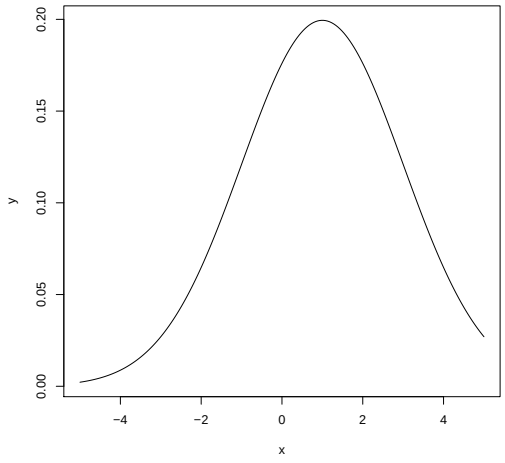


Abbildung 4.1.2: Dichte von $N(1, 4)$



Mittels

```
> dnorm(0,mu=1,sd=2)
[1] 0.1760327
```

kann man die Dichtefunktion einer $N(1,4)$-Verteilung an der Stelle 0 ermitteln, d.h. der Normalverteilung mit Erwartungswert 1 und *Varianz* 4. Das gleiche Ergebnis erhält man mit

```
> dnorm(0,1,2)
[1] 0.1760327
```

Entsprechend liefert

```
> y= dnorm(x,1,2)
> plot(x,y,type="l")
```

Abbildung 4.1.2, S. 121. Die Werte der Verteilungsfunktion einer $N(a, b^2)$ Verteilung an der Stelle x berechnen wir also mit

`pnorm(x,mean=a,sd=b)` oder kurz `pnorm(x,a,b)`.

Seien $Z \sim N(0,1)$ und $X \sim N(1,2^2)$, dann liefern

```
> pnorm(0)
[1] 0.5
> pnorm(0,1,2)
[1] 0.3085375
```

die Werte der jeweiligen Verteilungsfunktionen an der Stelle 0. Da

$$P(2 \leq X \leq 3) = F_X(3) - F_X(2)$$

ist, berechnet sich diese Wahrscheinlichkeit mithilfe von

```
> pnorm(3,1,2)-pnorm(2,1,2)
[1] 0.1498823
```

Hierbei ist zu beachten, dass für jede stetige Zufallsvariable X

$$P(a \leq X \leq b) = P(a < X \leq b) = P(a \leq X < b) = P(a < X < b)$$

gilt. Das 0.6-Quantil der $N(1,4)$-Verteilung erhalten wir mittels

```
> qnorm(0.6,1,2)
[1] 1.506694
```

und das 0.975-Quantil der Standardnormalverteilung aus

```
> qnorm(0.975)
[1] 1.959964
```

Man kann die Funktionen auch mit vektoriellen Argumenten versehen.

```
> qnorm(c(0.25,0.5,0.75))
[1] -0.6744898  0.0000000  0.6744898
```

Das sind das 0.25-, das 0.5- und das 0.75-Quantil der Standardnormalverteilung.

Angenommen, wir möchten 1000 Zufallszahlen aus einer N(1, 4) ziehen. Dies bewerkstelligen wir über

```
> set.seed(12345)
> x=rnorm(1000,1,2)
```

`rnorm()` erzeugt am Computer (Pseudo-)Zufallszahlen gemäß einer Normalverteilung. Die Zahlen sind nicht wirklich zufällig, sondern werden nach gewissen Algorithmen *berechnet*. Sie erscheinen aber wie Realisierungen von Zufallsvariablen. Daher spricht man von *Pseudo*-Zufallszahlen. Mit `set.seed()` wird für den Algorithmus ein Startwert gewählt, so dass auf allen Computern die gleichen Zahlen generiert werden und die resultierenden Ergebnisse vergleichbar werden. Zur Veranschaulichung erstellen wir ein Histogramm.

```
> hist(x,prob=T)
> lines(x,dnorm(x,1,2),col="blue")
```

Das Histogramm finden wir in Abbildung 4.1.3. Wir sehen, dass das Histogramm der am Computer generierten Pseudo-Zufallszahlen einen ähnlichen Verlauf aufweist wie die entsprechende Dichtefunktion. Das nehmen wir an dieser Stelle als Beleg dafür hin, dass die von uns erzeugten Zahlen sich weitgehend wie normalverteilte Beobachtungen verhalten.

Abbildung 4.1.3: Histogramm von $N(1, 4)$-Pseudo-Zufallszahlen

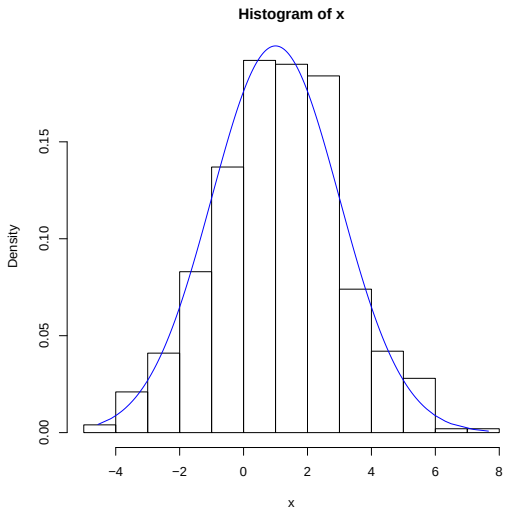
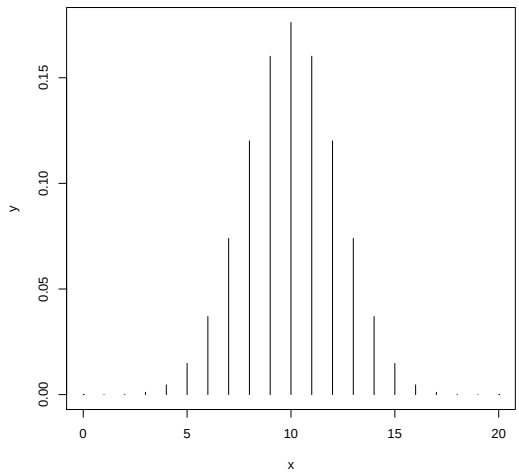


Abbildung 4.1.4: Wahrscheinlichkeitsfunktion von $B(10, 0.5)$



4.1.2 Binomialverteilung

Der Funktionsstamm lautet bei der Binomialverteilung `binom`. Daraus ergeben sich die Funktionen:

`dbinom()`, `pbinom()`, `qbinom()` und `rbinom()`.

Mit `z` als Präfix kann man auch schreiben:

`zbinom(x,size,prob)`.

Sei $X \sim B(n, \pi)$, dann steht `size` für das n und `prob` für den Parameter $\pi \in [0, 1]$.

Speziell sei X binomialverteilt mit $n = 10$ und $\pi = 0.5$. Wir wissen, dass dies gerade dem 10-maligen Werfen einer fairen Münze entspricht. Dabei gebe die Zufallsvariable X gerade die Anzahl von *„Zahl“* (oder *„Wappen“*) in diesen 10 Würfen an.

Wie groß ist die Wahrscheinlichkeit, dass *„genau 8 mal Zahl“* geworfen wird, also $P(X = 8)$? Die Antwort lautet:

```
> dbinom(8,10,0.5)
[1] 0.04394531
```

Die Wahrscheinlichkeitsfunktion dieser Verteilung können wir darstellen über

```
> x=0:10
> y=dbinom(x,10,0.5)
> plot(x,y,type="h")
```

Wir wissen, dass die Trägerpunkte die Zahlen $0, 1, \ldots, 10$ sind. Bei der `plot()`-Funktion stellten wir die Option `type="h"` für *histogram* ein. Die grafische Darstellung befindet sich in Abbildung 4.1.4, S. 123.

Wie groß ist die Wahrscheinlichkeit für $P(2 < X \leq 6)$? Wir berechnen:

```
> pbinom(6,10,0.5)-pbinom(2,10,0.5)
[1] 0.7734375
```

Es ist hier zu beachten, dass im diskreten Fall zwischen „$<$“ und „$\leq$“ unterschieden werden muss. Wie lauten das 0.25-, das 0.5- und das 0.75-Quantil der B(10,0.5)-Verteilung?

```
> qbinom(c(0.25,0.5,0.75),10,0.5)
[1] 4 5 6
```

Wie können wir das 20-malige Werfen einer idealen Münze simulieren?

```
> rbinom(20,1,0.5)
[1] 1 1 1 1 0 1 0 0 0 1 0 0 0 0 0 0 1 1 0 1
```

Dies entspricht gerade der 20-fachen Wiederholung eines Bernoulli-Experiments mit Erfolgswahrscheinlichkeit 0.5, also dem 20-maligen Ziehen aus einer B(1,0.5)-Verteilung. Die Anzahl der Einsen ist dann B(20,0.5)-verteilt.

In völliger Analogie gibt es in *R* für viele weitere Verteilungen entsprechende Befehle, z. B. `dexp()` für die Dichte der Exponentialverteilung oder `dpois()` für die Wahrscheinlichkeitsfunktion der Poisson-Verteilung

4.2 Testen mit *R*

4.2.1 Tests über Erwartungswerte

t-Test

Wir unterstellen, dass die Daten $x_1, \ldots, x_n$ Realisierungen von normalverteilten Zufallsvariablen $X_1, \ldots, X_n \sim N(\mu, \sigma^2)$ sind und wollen

$$H_0 : \mu = \mu_0 \quad \text{vs. } H_1 : \mu \neq \mu_0$$

für einen vorgegebenen Wert μ_0 bei unbekannter Varianz σ^2 mit einem t-Test prüfen.

Dazu wird die *R*-Funktion

```
t.test(x,mu=0,...)
```

verwendet. Das Argument `x` steht für den Beobachtungsvektor. Das optionale Argument `mu` erlaubt die Zuordnung von einem Vorgabewert μ_0.

• Beispiel 4.1 • An einem Flughafen soll die durchschnittliche Abfertigungszeit zur Gepäckannahme geschätzt werden. Hierzu wurden stichprobenhaft 10 Abfertigungszeiten erfasst (in Minuten):

7.8, 10.1, 9.0, 8.0, 11.6, 10.7, 8.1, 8.6, 9.4, 11.9.

Wir unterstellen das Vorliegen einer Normalverteilung und wollen prüfen, ob $H_0 : \mu = 10 = \mu_0$ ist, d.h. ob die im Mittel erwartete Gepäckabfertigungszeit 10 Minuten beträgt.

Der Test kann dann folgendermaßen durchgeführt werden:

```
> x=c(7.8,10.1,9.0,8.0,11.6,10.7,8.1,8.6,9.4,11.9)
> t.test(x,mu=10)
```

Als Ergebnis erhalten wir:

```
        One Sample t-test

data:  x
t = -1.0148, df = 9, p-value = 0.3367
alternative hypothesis: true mean is not equal to 10
95 percent confidence interval:
  8.449989 10.590011
sample estimates:
mean of x
     9.52
```

Wir schauen uns das Ergebnis etwas näher an. `t=-1.0148` ist der Wert der Teststatistik. Da 10 Beobachtungen vorliegen, wird ein t-Test mit `df=9` Freiheitsgraden (*degrees of freedom*) durchgeführt. Der p-Wert des t-Tests wird mit `p-value` bezeichnet und beträgt im Beispiel 0.3367. Die Alternativhypothese ist, dass der wahre Erwartungswert nicht gleich 10 ist. Da

der p-Wert größer als 0.05 ist, wird die Nullhypothese, dass $\mu = 10$ ist, zum Signifikanzniveau 0.05 nicht abgelehnt. Zusätzlich zum t-Test werden die Intervallgrenzen eines 0.95-Konfidenzintervalls berechnet: 8.45 ist die untere Grenze und 10.59 die obere Grenze. Der Schätzwert für den Erwartungswert ist 9.52, das arithmetische Mittel der Beobachtungswerte. Diese Ergebnisse entsprechen den Berechnungen in Stocker und Steinke [2017], Beispiel 10.2.2, S. 480, und Beispiel 11.3.2, S. 549.

Eine Durchführung des t-Tests mit direkten Berechnungen kann auch folgendermaßen aussehen:

```
> mean(x)
[1] 9.52
```

ist das arithmetische Mittel der `x`-Werte und damit der Schätzwert für den Erwartungswert.

```
> tstat= sqrt(10)*(mean(x)-10)/sd(x)
> tstat
[1] -1.014789
```

Der Wert der Teststatistik wurde hier gemäß der Formel

$$t = \sqrt{n}(\bar{x} - \mu_0)/s_X$$

berechnet, vgl. dazu auch Stocker und Steinke [2017], S. 546. Mit

`pt(q,df)` bzw. `qt(p,df)`

berechnen wir den Wert der Verteilungsfunktion einer t-Verteilung mit `df` Freiheitsgraden an der Stelle `q` bzw. das Quantil einer t-Verteilung mit `df` Freiheitsgraden zum Niveau `p`.

H_0 wird zum Niveau $\alpha = 0.05$ abgelehnt, wenn $|t| > t_{n-1,1-\alpha/2} = t_{9,0.975}$ gilt.

```
> qt(0.975,9)
[1] 2.262157
```

So erhalten wir das Quantil. Der Betrag des Wertes der Teststatistik, 1.014789, ist nicht größer als das Quantil, 2.262157. Daher behalten wir die Nullhypothese bei.

Den p-Wert des t-Tests kann man mit der Formel

$$p = 2(1 - F_{n-1}(|t|))$$

berechnen, wobei t der Wert der Teststatistik ist und F_{n-1} hier für die Verteilungsfunktion einer t-Verteilung mit $n-1$ Freiheitsgraden steht.

```
> 2*(1-pt(abs(tstat),9))
[1] 0.3367091
```

Es ergibt sich das gleiche Ergebnis, das wir durch die Anwendung von `t.test()` erhalten haben.

Soll ein einseitiger Test durchgeführt werden, dann kann der optionale Parameter `alternative` (auch kurz `alt`) verwendet werden. Standardmäßig ist er auf `two.sided` für

zweiseitiges Testproblem gesetzt. Für die einseitige Alternative $H_1 : \mu < \mu_0$ ist der Wert auf `"less"` und für $H_1 : \mu > \mu_0$ auf `"greater"` zu setzen.

• **Beispiel 4.2** • Wir verwenden die gleichen Daten, aber untersuchen das Testproblem

$$H_0 : \mu \geq 10 \quad \text{vs. } H_1 : \mu < 10.$$

Dazu verwenden wir die Anweisung:

```
> t.test(x,mu=10,alternative="less")
```

`x` ist dabei erneut der Beobachtungswertvektor, den wir bereits zugeordnet haben. Als Ergebnis ergibt sich:

```
        One Sample t-test

data:  x
t = -1.0148, df = 9, p-value = 0.1684
alternative hypothesis: true mean is less than 10
95 percent confidence interval:
     -Inf 10.38707
sample estimates:
mean of x
     9.52
```

Der Wert der Teststatistik und die Freiheitsgrade sind unverändert. Die Alternative ist, dass der wahre Erwartungswert kleiner als 10 ist. Der p-Wert beträgt jetzt 0.1684 und ist damit immer noch größer als 0.05. Diesen Wert erhält man auch durch

```
> pt(tstat,9)
[1] 0.1683545
```

Vgl. auch Stocker und Steinke [2017], S. 546. Auch für diesen Test wird ein Konfidenzintervall berechnet, aber ein einseitiges, $(-\infty, 10.38707)$.

Weitere Informationen zu `t.test()` finden sich in der R-Hilfe.

4.2.2 Test über Erwartungswertdifferenzen

Gegeben seien Beobachtungsdaten $y_{11}, \ldots, y_{1n_1}$ und $y_{01}, \ldots, y_{0n_0}$ als Realisierungen von Zufallsvariablen $Y_{11}, \ldots, Y_{1n_1} \sim N(\mu_1, \sigma_1^2)$ bzw. $Y_{01}, \ldots, Y_{0n_0} \sim N(\mu_0, \sigma_0^2)$. Es soll ein Testproblem der Form $H_0 : \mu_0 = \mu_1$ vs. $H_1 : \mu_0 \neq \mu_1$ untersucht werden.

Es wird die R-Funktion

```
t.test(y1,y0,mu=0,...).
```

verwendet, wobei `y1` der Vektor der $y_{11}, \ldots, y_{1n_1}$-Daten ist und `y0` die $y_{01}, \ldots, y_{0n_0}$-Werte enthält. Wichtige optionale Parameter sind `paired=FALSE` und `var.equal=FALSE`, wie wir sehen werden.

t-Test bei verbundenen Stichproben

Wir gehen davon aus, dass $n_0 = n_1$ und Y_{0i} und Y_{1i} gemeinsam, z.B. am gleichen Merkmalsträger, erhoben werden. Man könnte dann auch (Y_{0i}, Y_{1i}) schreiben. Y_{0i} und Y_{1i} treten als Paar auf und sind i.d.R. nicht unabhängig.

• **Beispiel 4.3** • Wir interessieren uns für das Wachstum von Efeu. Die Länge von drei Pflanzen wird zu den Zeitpunkten 0 und 1 erfasst.

	Pflanze 1	Pflanze 2	Pflanze 3
Zeitpunkt 0	14	25	36
Zeitpunkt 1	46	55	64

Es soll geprüft werden, ob ein Wachstum zwischen den Zeitpunkten stattgefunden hat, d.h. ob die Erwartungswerte der Pflanzenlängen der Zeitpunkte 0 bzw. 1 identisch sind oder nicht, vgl. auch Beispiel 10.2.4 in Stocker und Steinke [2017], S. 489. Die Daten werden in den Vektoren `y0` und `y1` erfasst.

```
> y0= c(14,25,36)
> y1= c(46,55,64)
```

Den t-Test führen wir durch Aufruf von

```
> t.test(y1,y0,paired=TRUE)
```

durch. Als Ergebnis erhalten wir:

```
        Paired t-test

data:  y1 and y0
t = 25.981, df = 2, p-value = 0.001478
alternative hypothesis: true difference in means is not
    equal to 0
95 percent confidence interval:
 25.03172 34.96828
sample estimates:
mean of the differences
                     30
```

Das Ergebnis ist genauso zu interpretieren wie im vorherigen Beispiel. Getestet wurde in diesem Fall, ob die Erwartungswertdifferenz gleich Null ist, d.h. die y_{0i} und y_{1i}-Daten den gleichen Erwartungswert haben. Der p-Wert (0.001478) ist kleiner als 0.05. Daher ist die Annahme, dass die Erwartungswerte beider Stichproben gleich sind, zum Niveau 0.05 abzulehnen.

Im Wesentlichen das gleiche Ergebnis erhält man mit

```
> t.test(y1-y0)
```

Bei verbundenen Stichproben wird ein Einstichprobentest auf die Differenzen der Beobachtungswerte durchgeführt.

t-Test bei gleichen Varianzen

Jetzt wird angenommen, dass alle Y_{0i} und Y_{1j} unabhängig und normalverteilt sind. Außerdem mögen ihre Varianzen identisch sein, d.h. $\sigma_0^2 = \sigma_1^2$. Zum Prüfen von Hypothesen der Form $H_0 : \mu_0 = \mu_1$ kann man dann erneut einen t-Test mithilfe der Funktion `t.test()` durchführen.

• Beispiel 4.4 • Das Wachstum von Pflanzen soll untersucht werden. Die Pflanzen werden dabei in Gruppe 1 gedüngt und in Gruppe 0 nicht gedüngt. Es soll festgestellt werden, ob die Düngung das Pflanzenwachstum beeinflusst. Die Beobachtungsdaten seien:

Gruppe 0 (ohne Düngung)	14	25	36
Gruppe 1 (mit Düngung)	46	55	64

In jeder Gruppe haben wir drei Pflanzen. Wir unterstellen normalverteilte Daten. Für `y0` und `y1` erhalten wir damit die gleichen Zahlenwerte wie in Beispiel 4.3.

```
> t.test(y1,y0,var.equal=TRUE)
```

Durch das optionale Argument `var.equal=TRUE` wird angegeben, dass die Varianz in beiden Gruppen von Pflanzen als identisch angesehen werden.

```
        Two Sample t-test

data:  y1 and y0
t = 3.656, df = 4, p-value = 0.02166
alternative hypothesis: true difference in means is not
    equal to 0
95 percent confidence interval:
  7.217355 52.782645
sample estimates:
mean of x mean of y
       55        25
```

Der Wert der Teststatistik beträgt 3.656. In den Stichproben sind jeweils $n_1 = 3$ und $n_0 = 3$ Beobachtungen enthalten. Die Anzahl der Freiheitsgrade der t-Verteilung der Teststatistik ist daher $df = n_0 + n_1 - 2 = 4$. Der p-Wert des Tests beträgt 0.02166. Für die Erwartungswertdifferenz ist ein 0.95-Konfidenzintervall mit den Grenzen 7.217 und 52.783 angegeben. Die Mittelwerte der beiden Stichproben betragen 55 (`y1`) und 25 (`y0`).

Eine direkte Berechnung der Teststatistik liefert das gleiche Ergebnis. Zunächst ist dazu der gemeinsame Varianzschätzwert zu bestimmen.

```
> sp2 = (2*var(y1)+2*var(y0))/4
> tstat=(mean(y1)-mean(y0))/sqrt(sp2*(1/3+1/3))
> tstat
[1] 3.656
```

t-Test bei ungleichen Varianzen

Sind die Varianzen nicht gleich, kann die Welch-Version des t-Tests durchgeführt werden. Die Teststatistik berechnet sich dann gemäß

$$t = \frac{\bar{y}_1 - \bar{y}_0}{\sqrt{s_0^2/n_0 + s_1^2/n_1}} \; .$$

Da die Teststatistik nicht mehr t-verteilt ist, wird hier ein approximativer Freiheitsgrad bestimmt, mit deren Hilfe die Testverteilung angenähert wird (Welch [1947]).

• **Beispiel 4.5** • Wir gehen von den gleichen Daten wie in Beispiel 4.4 aus, verzichten aber auf die Annahme $\sigma_0^2 = \sigma_1^2$.

```
> t.test(y1,y0)
```

Hier werden keine optionalen Argumente gesetzt. Das ist die in *R* implementierte Standardversion des t-Tests für Erwartungswertdifferenzen (Zweistichproben-t-Test).

```
        Welch Two Sample t-test

data:  y1 and y0
t = 3.656, df = 3.8491, p-value = 0.02315
alternative hypothesis: true difference in means is not
    equal to 0
95 percent confidence interval:
  6.860673 53.139327
sample estimates:
mean of x mean of y
       55        25
```

Der Wert der Teststatistik ist nur deshalb gleich dem aus Beispiel 4.4, weil die Stichprobenumfänge n_0 und n_1 gleich groß sind. I.d.R. unterscheiden sich die Werte der Teststatistik für die beiden Fälle. Als Freiheitsgrad wurde hier allerdings der Wert `df=3.8491` gewählt. Die Teststatistik ist unter der Nullhypothese nicht mehr t-verteilt, aber mit dem gewählten Wert für die Freiheitsgrade immer noch durch eine t-Verteilung approximierbar, s. dazu Satterthwaite [1941]. Auch der p-Wert und die Grenzen des Konfidenzintervalls unterscheiden sich von den Werten aus Beispiel 4.4 – allerdings nicht deutlich.

4.2.3 χ^2-Tests

χ^2-Anpassungstest

Gegeben seien die Beobachtungen einer diskreten Zufallsvariable X mit k verschiedenen Ausprägungen, die mit $a_1, \ldots, a_k$ bezeichnet werden. Mit Hilfe eines χ^2-Anpassungstests kann geprüft werden, ob

$$H_0 : P(X = a_i) = \pi_i \text{ für } i = 1, \ldots, k$$

für vorgegebene π_i gilt. Ein solcher Test kann mit der Funktion

`chisq.test(x,p,...)`.

durchgeführt werden. Dabei ist `x` ein Vektor, der die Häufigkeit des Auftretens der Ausprägungen $a_1, \ldots, a_k$ in der Stichprobe angibt und `p` ist der Vektor der vorgegebenen Einzelwahrscheinlichkeiten $\pi_1, \ldots, \pi_k$.

• Beispiel 4.6 • Vor einer Wahl wurden stichprobenartig 1369 Personen nach der Partei befragt, die sie wählen wollen. Dieses Umfrageergebnis soll mit dem später erfassten Wahlergebnis verglichen werden.

Partei	1	2	3	4	5	6	7	Summe
n_i	548	370	75	116	123	55	82	1369
$f_i = n_i/n$	0.400	0.270	0.055	0.085	0.09	0.040	0.060	1.0
π_i	0.415	0.257	0.048	0.086	0.084	0.047	0.063	1.0

Die Parteien werden hier mit 1 bis 7 bezeichnet. n_i gibt an, wie viele Personen aus der Umfrage für Partei i gestimmt hätten. π_i gibt an, wie groß der Anteil der Stimmen bei der Wahl war, vgl. dazu auch Beispiel 11.3.3 in Stocker und Steinke [2017], S. 557ff.

```
> x = c(548,370,75,116,123,55,82)
> p = c(0.415,0.257,0.048,0.086,0.084,0.047,0.063)
> chisq.test(x,p=p)
```

Als Ergebnis erhalten wir:

```
        Chi-squared test for given probabilities

data:  x
X-squared = 5.1129, df = 6, p-value = 0.5294
```

Der Wert der Teststatistik beträgt 5.1129. Die diskrete Zufallsvariable X nimmt $k = 7$ verschiedene Ausprägungen an. Die Teststatistik ist unter der Nullhypothese approximativ χ^2-verteilt mit $k - 1 = 6$ Freiheitsgraden (`df`, *degrees of freedom*). Der p-Wert des Tests beträgt 0.5292. Zu einem Niveau 0.05 ist die Nullhypothese, dass die vorgegebenen Zahlen π_i die Einzelwahrscheinlichkeiten zu X sind, nicht abzulehnen.

Eine direkte Berechnung der Teststatistik könnte gemäß der Berechnungsformel

$$\chi^2 = \sum_{i=1}^{k} \frac{(n_i - n\pi_i)^2}{n\pi_i}$$

so aussehen:

```
> n=sum(x); np=n*p
> X2=sum((x-np)^2/np)
> X2
[1] 5.112886
```

Unter der Nullhypothese ist die Teststatistik χ^2-verteilt mit 6 Freiheitsgraden. Den p-Wert erhalten wir entsprechend Stocker und Steinke [2017], S. 546, aus

```
> 1-pchisq(X2,6)
[1] 0.5294184
```

χ^2-Unabhängigkeitstest

Gegeben seien die diskreten Merkmale (X, Y), wobei wir davon ausgehen, dass X die k verschiedenen Werte $x_1, \ldots, x_k$ und Y die m verschiedenen Were $y_1, \ldots, y_l$ annehmen kann. Als Beobachtungsdaten werden die Häufigkeiten n_{ij} der Merkmalskombinationen (x_i, y_j) erfasst. Diese Daten kann man in einer $k \times l$-Matrix zusammenfassen. Mit Hilfe des χ^2-Unabhängigkeitstests soll die Nullhypothese geprüft werden, ob die Merkmale X und Y unabhängig sind.

Dieser Test kann mit der Funktion

```
chisq.test(x,...)
```

durchgeführt werden. In diesem Fall ist `x` die Matrix mit den Einträgen n_{ij}.

• Beispiel 4.7 • Im Rahmen einer Umfrage wurde das Geschlecht (X) und das Rauchverhalten (Y) der befragten Personen festgestellt.

$X \setminus Y$	Raucher	Gelegenheitsraucher	Nichtraucher	Summe
weiblich	4	8	28	40
männlich	12	12	36	60

Es soll geprüft werden, ob Geschlecht und Rauchverhalten unabhängig sind, vgl. dazu auch Stocker und Steinke [2017], Beispiel 11.3.5, S. 567.

```
> x = matrix(c(4,12,8,12,28,36),2,3)
> chisq.test(x)

        Pearson's Chi-squared test

data:  x
X-squared = 1.875, df = 2, p-value = 0.3916
```

Der Wert der Teststatistik ist `X-squared=1.875`. Die Teststatistik ist unter der Nullhypothese χ^2-verteilt mit $(k-1)\cdot(l-1) = (2-1)\cdot(3-1) = 2$ Freiheitsgraden (`df` – degrees of freedom). Der p-Wert (`p-value`) beträgt 0.3916. Er ist größer als 0.05. Zu einem Signifikanzniveau $\alpha = 0.05$ wird die Nullhypothese der Unabhängigkeit der Merkmale X und Y also beibehalten.

Für eine direkte Berechnung ermitteln wir zunächst

$$n_{i\bullet} = \sum_{j=1}^{l} n_{ij}, \quad n_{\bullet j} = \sum_{i=1}^{k} n_{ij}, \quad e_{ij} = \frac{n_{i\bullet} n_{\bullet j}}{n}$$

und damit den Wert der Teststatistik

$$\chi^2 = \sum_{i=1}^{k} \sum_{j=1}^{l} \frac{(n_{ij} - e_{ij})^2}{e_{ij}},$$

vgl. auch Stocker und Steinke [2017], S. 570. Die folgende Berechnung mit R liefert damit den Wert der Teststatistik.

```
> n = sum(x)
> e = rowSums(x) %*% t(colSums(x))/n
> X2 = sum((x-e)^2/e)
> X2
[1] 1.875
```

Mit `rowSums(x)` bzw. `colSums(x)` werden die Vektoren der Zeilen- bzw. Spaltensummen der Matrix `x` ermittelt.

Den p-Wert erhalten wir in Analogie zum χ^2-Anpassungstest.

```
> 1-pchisq(X2,2)
[1] 0.3916056
```

4.2.4 Weitere Tests

Tests für Anteilswerte

Es sei $X \sim B(n, \pi)$, also ist X binomialverteilt mit den Parametern n und $\pi \in (0, 1)$. Es soll das Testproblem $H_0 : \pi = \pi_0$ vs. $H_1 : \pi \neq \pi_0$ geprüft werden.

Approximativer Binomialtest

Zunächst wenden wir einen approximativen Binomial-Test mithilfe der R-Funktion

`prop.test(x,n,p,correct=TRUE,alternative="two.sided")`.

an. `x` wird die Realisierung x zugewiesen, `n` ist der n-Parameter der Binomialverteilung. `p` ist der Nullhypothesenwert π_0. `alternative` gibt an, ob ein zweiseitiger Test (`"two.sided"`), ein oberer einseitiger (`"greater"`) oder ein unterer einseitiger Test (`"less"`) durchgeführt werden soll. Der Parameter `correct` gibt an, ob der Test mit (`TRUE`) oder ohne (`FALSE`) Stetigkeitskorrektur durchgeführt werden soll. Eine Stetigkeitskorrektur ist bekannt aus den Approximationsformeln für die Verteilungsfunktion von Binomialverteilungen durch Normalverteilungen. I.A. ist ihre Anwendung zu empfehlen. Da sie aber in Stocker und Steinke [2017] nicht besprochen wurde, wird sie im folgenden Beispiel nicht angewandt.

• **Beispiel 4.8** • Wir verwenden die Zahlenwerte von Stocker und Steinke [2017], Beispiel 11.1.1. Bei einer Single-Choice-Klausur gab es 30 Fragen mit jeweils zwei Antwortmöglichkeiten, von denen genau eine richtig ist. Ein Student hat 19 Fragen beantwortet. Hätte er

die Antworten nur geraten, wäre die Wahrscheinlichkeit für eine richtige Antwort $\pi_0 = 0.5$. Wir betrachten daher das Testproblem $H_0 : \pi \leq 0.5$ vs. $H_1 : \pi > 0.5$.

```
>  prop.test(19,30,p=0.5,cor=F,alt="greater")
```

Die optionalen Parameter `correct` bzw. `alternative` wurden hier durch `cor` bzw. `alt` abgekürzt. Da ein einseitiger Test durchgeführt wird, ist die Alternative über `alternative="greater"` ($H_1 : \pi > 0.5$) festzulegen. Als Ergebnis erhalten wir:

```
   1-sample proportions test without continuity correction

data:  19 out of 30, null probability 0.5
X-squared = 2.1333, df = 1, p-value = 0.07206
alternative hypothesis: true p is greater than 0.5
95 percent confidence interval:
 0.4832636 1.0000000
sample estimates:
        p
0.6333333
```

Anstelle der üblichen Teststatistik

$$z = \frac{x - n\pi_0}{\sqrt{n\pi_0(1-\pi_0)}} \qquad (*)$$

ist deren quadrierte Version ausgewiesen. Der p-Wert ist 0.07206, also wird H_0 beibehalten. Wir entscheiden bei einem Signifikanzniveau $\alpha = 0.05$ also zugunsten der Nullhypothese, dass der Student nur geraten hat. Es wird ein (einseitiges) 0.95-Konfidenzintervall ausgewiesen. Der Schätzwert für π ist `p=0.6333333` $\approx 19/30$. Eine direkte Berechnung von `p` und der Teststatistik liefert:

```
> x=19; n=30; p0=0.5
> x/n #p
[1] 0.6333333
> z= (x-n*p0)/sqrt(n*p0*(1-p0))
> z^2
[1] 2.133333
```

Das ist der Wert der quadrierten Teststatistik (*).

```
> 1-pnorm(z)
[1] 0.07206352
```

So erhalten wir den p-Wert, vgl. auch Stocker und Steinke [2017], S. 546. Es wird hier $P(Z \geq z) = 1 - \Phi(z)$ mit $Z \sim N(0,1)$ gerechnet.

Exakter Binomialtest

Wenn man einen Test für einen Anteilswert mit *R* durchführt, ist in der Regel ein exakter Binomialtest gegenüber der approximativen Version vorzuziehen. Die Teststatistik ist hier `x` selbst und die Testentscheidung wird mithilfe des p-Wertes durchgeführt. Es wird dazu

```
binom.test(x,n,p=0.5,alternative="two.sided")
```

mit den gleichen Argumenten wie `prop.test` verwendet.

• **Beispiel 4.9** • Auf das Problem von Beispiel 4.8 wird jetzt ein exakter Binomialtest angewendet.

```
> binom.test(19,30,p=0.5,alternative="greater")

    Exact binomial test

data:  19 and 30
number of successes = 19, number of trials = 30,
    p-value = 0.1002
alternative hypothesis: true probability of success is greater
    than 0.5
95 percent confidence interval:
 0.4669137 1.0000000
sample estimates:
probability of success
             0.6333333
```

Der p-Wert ist 0.1002, d.h. die Nullhypothese $H_0 : \pi \leq 0.5$ wird beibehalten. Mit $X \sim B(30, 0.5)$ unter der Nullhypothese erhalten wir den p-Wert auch mittels

$$P(X \geq 19) = 1 - P(X < 19) = 1 - P(X \leq 18) = 1 - F_X(18).$$

```
>   1-pbinom(18,30,0.5)
[1] 0.1002442
```

Das ist der p-Wert von `binom.test()` und auch das Ergebnis aus Stocker und Steinke [2017], Beispiel 11.2.3, S. 544.

Approximativer Binomialtest über eine Anteilswertdifferenz

Es seien $Y_i \sim B(n_i, \pi_i)$ für $i = 0, 1$. Sollen für eine Stichproben zwei Anteilswerte verglichen werden, kann das auch mit der Funktion

```
prop.test(x,n,correct=TRUE)
```

realisiert werden. In diesem Fall ist `x` der Vektor der Ausprägungen von Y_0 und Y_1 und `n` der Vektor der Gruppenstichprobenumfänge n_0 und n_1. Es werden nur $H_0 : \pi_0 = \pi_1$ bzw. die entsprechenden einseitigen Testprobleme geprüft. Tests auf Anteilswertdifferenzen der Form $H_0 : \pi_1 - \pi_0 = \delta_0$ für $\delta_0 \neq 0$, wie sie in Stocker und Steinke [2017], Satz 11.3.6, S. 573, zu finden sind, sind in R nicht vorgesehen.

• **Beispiel 4.10** • Es soll die Wirksamkeit eines Medikaments geprüft werden. Es wurde bei 110 Verabreichungen 85 Mal erfolgreich angewandt. Als Referenz für die Wirksamkeit des Medikaments wurde an 85 Probanden ein Placebo verabreicht. Das führte in 45 Fällen zu einer Besserung. Siehe dazu auch Stocker und Steinke [2017], Beispiel 11.3.7, S. 575.

```
> prop.test(c(45,62),c(85,110),correct=FALSE)
```

Das Ergebnis ist:

```
    2-sample test for equality of proportions without
    continuity correction

data:  c(45, 62) out of c(85, 110)
X-squared = 0.22681, df = 1, p-value = 0.6339
alternative hypothesis: two.sided
95 percent confidence interval:
 -0.1751092  0.1066600
sample estimates:
   prop 1    prop 2
0.5294118 0.5636364
```

Der Wert der Teststatistik ist 0.22681, der p-Wert 0.6339. Ein signifikanter Unterschied zwischen Placebo und Medikament ist daher nicht feststellbar.

Wir prüfen das Ergebnis mit einer manuellen Rechnung.

```
> x1=45; n1=85; x2=62; n2=110
> p1=x1/n1; p2=x2/n2
> p1
[1] 0.5294118
> p2
[1] 0.5636364
> t= (p1-p2)/sqrt(p1*(1-p1)/n1+p2*(1-p2)/n2)
> t^2
[1] 0.2266969
```

Das ist der Wert der Teststatistik `X-squared` aus `prop.test`.

```
> 2*(1-pnorm(abs(t)))
[1] 0.6339839
```

Der p-Wert wird entsprechend Stocker und Steinke [2017], S. 546, als

$$P(|T| \geq |t|) = 2(1 - \Phi(|t|))$$

mit $T \sim N(0, 1)$ berechnet.

Korrelationstest

Gegeben sei der Merkmalsvektor (X, Y). Es soll geprüft werden, ob X und Y unkorreliert sind oder nicht. Wir gehen davon aus, dass Beobachtungspaare

$$(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$$

gegeben sind. Zur Durchführung eines entsprechenden Tests kann die Funktion

```
cor.test(x,y,...)
```

verwendet werden. Dabei ist `x` der Vektor, der die Werte $x_1, x_2, \ldots, x_n$ enthält, und `y` ein Vektor, der die entsprechenden Werte $y_1, y_2, \ldots, y_n$ enthält.

• **Beispiel 4.11** • Gegeben seien die folgenden Beobachtungsvektoren:

$$(2,1),(4,2),(4,3),(5,2),(6,4),(8,5),(9,6),(10,4),(4,5),(7,3),$$

vgl. Stocker und Steinke [2017], Beispiel 11.3.9.

```
> x= c(2,4,4,5,6,8,9,10,4,7)
> y= c(1,2,3,2,4,5,6,4,5,3)
> cor.test(x,y)
```

Wir erhalten das Ergebnis:

```
        Pearson's product-moment correlation

data:  x and y
t = 2.5736, df = 8, p-value = 0.03294
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.07527334 0.91493465
sample estimates:
      cor
0.6730033
```

Es liegen $n = 10$ Beobachtungspaare vor. Wenn r der Schätzwert für den Korrelationskoeffizient ist, s. `cor()`, dann berechnet sich die Teststatistik gemäß

```
> r=cor(x,y)
> sqrt(10-2)*r/sqrt(1-r^2)
[1] 2.573606
```

Die Formel zur Berechnung der Teststatistik findet sich in Stocker und Steinke [2017] auf S. 577. Das Quantil für einen 0.05-Niveau-Test ermitteln wir mittels

```
> qt(0.975,8)
[1] 2.306004
```

Da der Wert der Teststatistik betragsmäßig größer als der Quantilswert ist, wird die Nullhypothese der Unkorreliertheit der Merkmale abgelehnt. Das gleiche Ergebnis würden wir erhalten, wenn wir uns den p-Wert `p-value=0.03295` anschauen, der kleiner als 0.05 ist.

4.3 Einfaches lineares Regressionsmodell

4.3.1 Lineare Regression unter klassischen Annahmen

In einem linearen Regressionsmodell

$$Y_i = \beta_0 + \beta_1 X_i + U_i \tag{4.1}$$

sollen unter klassischen Annahmen, d.h. insbesondere (X_i, Y_i) u.i.v. und

$$U_i|X_i \sim N(0, \sigma^2), \tag{4.2}$$

die Koeffizienten β_i geschätzt und Konfidenzintervalle und Tests für β_i konstruiert werden. Die Funktion zur Anpassung eines linearen Regressionsmodells ist

`lm(y ~ x,...)`

Hierbei ist `y` der Vektor der abhängigen Variablen (y_i) und `x` der Vektor der unabhängigen Variablen (x_i).

Die Funktion `lm( )` gibt eine Liste als Ergebnis zurück. Wichtige Komponenten sind:

- `coefficients` liefert den Vektor der Koeffizientenschätzer $\hat{\beta}_i$.
- `fitted.values` enthält den Vektor der gefitteten Werte $\hat{y}_i = \hat{\beta}_0 + \hat{\beta}_1 x_i$.
- `residuals` enthält den Vektor der KQ-Residuen $\hat{u}_i = y_i - \hat{y}_i$.

• **Beispiel 4.12** • Die nachfolgenden Daten (x_i, y_i) mögen die Bewässerungsmenge x_i und die Wuchshöhe y_i von 5 Pflanzen darstellen:

(1,1), (2,2), (3,1), (4,3), (5,2),

vgl. dazu Stocker und Steinke [2017], Fallbeispiel 1, S.638f. Die Abhängigkeit der Wuchshöhe von der Bewässerungsmenge soll mit Hilfe eines linearen Regressionsmodells modelliert werden.

```
> x = c(1,2,3,4,5)
> y = c(1,2,1,3,2)
> plot(x,y,pch=19,main="Wuchshöhe vs. Bewässerungsmenge")
```

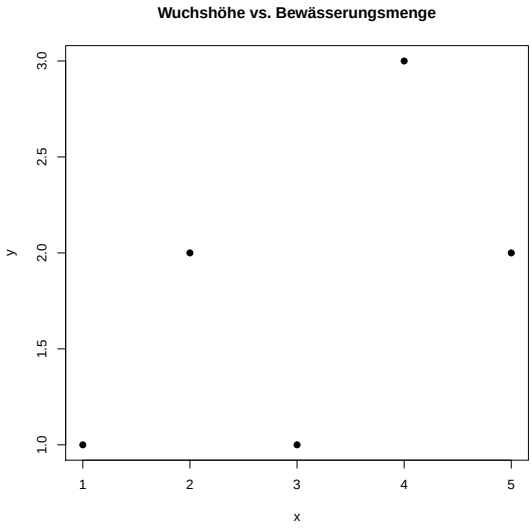
In Abbildung 4.3.1 finden wir eine grafische Darstellung der Daten.

Die Regression wird durchgeführt und die Ergebnisse lassen wir uns ausgeben.

```
> erg=lm(y~x)
> erg$coeficients
(Intercept)           x
        0.9         0.3
> erg$fitted.values
  1   2   3   4   5
1.2 1.5 1.8 2.1 2.4
> erg$residuals
   1    2    3    4    5
-0.2  0.5 -0.8  0.9 -0.4
```

Die gefitteten Werte und die Residuen kann man mit den Koeffizientenschätzern auch leicht direkt berechnen. Man beachte, dass man die Bezeichnungen für die Elemente von Listen auch abkürzen kann, wenn die Eindeutigkeit der Zuordnung gewahrt bleibt. Anstelle von `erg$coefficients` kann man bspw. auch `erg$coef` oder `erg$co` schreiben.

Abbildung 4.3.1: plot(x,y)




```
> yhat = erg$coef[1] + erg$coef[2]*x
> yhat
[1] 1.2 1.5 1.8 2.1 2.4
>   y-yhat
[1] -0.2  0.5 -0.8  0.9 -0.4
```

Wenn man die Funktion `lm( )` anwendet, werden zunächst nur die Schätzwerte der Koeffizienten $\hat{\beta}_0$ und $\hat{\beta}_1$ ausgegeben. Detailliertere Ausgaben zum Ergebnis der Regression erhält man durch Anwendung der Funktion

`summary(erg,...)`.

Hierbei is `erg` das Ergebnis einer `lm()`-Funktion.

• **Fortsetzung von Beispiel 4.12** • Zur zuvor durchgeführten Regression sollen weitere Ergebnisse ausgegeben werden.

```
> summary(erg)
Call:
lm(formula = y ~ x)

Residuals:
   1    2    3    4    5
-0.2  0.5 -0.8  0.9 -0.4

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   0.9000     0.8347   1.078    0.360
x             0.3000     0.2517   1.192    0.319

Residual standard error: 0.7958 on 3 degrees of freedom
Multiple R-squared:  0.3214,    Adjusted R-squared:  0.09524
F-statistic: 1.421 on 1 and 3 DF,  p-value: 0.3189
```

Unter `Residuals` finden wir die Residuen der Regression, vgl. dazu auch die vorherigen Ergebnisse. In der Tabelle `Coefficients` sind nicht nur die Parameterschätzer angegeben, sondern auch deren Standardfehler (`Std. Error`) sowie die Teststatistiken und p-Werte zu den Nullhypothesen $H_0 : \beta_0 = 0$ bzw. $H_0 : \beta_1 = 0$. Beide p-Werte sind größer als 0.05. Daher kann man beide Nullhypothesen zu einem Niveau $\alpha = 0.05$ beibehalten.

Den Standardfehler der Regression (SER) finden wir unter `Residual standard error`. Er beträgt 0.7958 und ist der Schätzwert für σ. Den Wert des Bestimmtheitsmaßes R^2 finden wir unter `Multiple R-squared: 0.3214`. Er ist nicht sehr groß. Das adjustierte Bestimmtheitsmaß `Adjusted R-squared` fällt mit `0.09524` sogar sehr niedrig aus.

Die Ergebnisse entsprechen denen aus Stocker und Steinke [2017], S.639.

Mit der Funktion

```
confint(erg,level=0.95,...).
```

kann man zu den Parameterschätzern auch die zugehörigen Konfidenzintervalle zum Niveau `level` ausgeben lassen. Die Funktion

```
vcov(erg).
```

kann verwendet werden, um die Kovarianzmatrix des Schätzers $\hat{\beta}$ zu bestimmen.

• Fortsetzung von Beispiel 4.12 •

```
> confint(erg)
                 2.5 %   97.5 %
(Intercept) -1.7562785 3.556278
x           -0.5008981 1.100898
```

Die untere bzw. obere Konfidenzintervallgrenze zu β_1, also zur Variablen `x`, sind -0.501 bzw. 1.101. Die Ergebnisse entsprechen – bis auf Abweichungen infolge von Rundungen – denen aus Stocker und Steinke [2017], S.639.

```
> Vbeta=vcov(erg)
> Vbeta
            (Intercept)           x
(Intercept)   0.6966667 -0.19000000
x            -0.1900000  0.06333333
> sqrt(diag(Vbeta))
(Intercept)           x
  0.8346656   0.2516611
```

Die letzte Zeile liefert die Standardfehler von $\hat{\beta}_0$ und $\hat{\beta}_1$.

• Beispiel 4.13: Einfache Regression mit binärem Regressor • Wir betrachten folgende Wuchshöhen von insgesamt 6 Pflanzen ohne bzw. mit Düngung.

Gruppe 0 (ohne Düngung)	14	25	36
Gruppe 1 (mit Düngung)	46	55	64

Unter der Annahme, dass die Wuchshöhen mit gleichen Varianzen aber u.U. unterschiedlichen Erwartungswerten in den beiden Gruppen normalverteilt sind, soll geprüft werden, ob ein signifikanter Unterschied der Erwartungswerte in beiden Gruppen feststellbar ist. Das kann mit einem t-Test über Erwartungswertdifferenzen geschehen (vgl. Beispiel 4.4, S. 129) oder mit einer einfachen linearen Regression. Für letzteren Ansatz wird eine Variable `x` eingeführt, die die Gruppenzugehörigkeit zuordnet.

```
> y=c(14,25,36,46,55,64)
> x=c(rep(0,3),rep(1,3))
> erg=lm(y~x)
> summary(erg)
```

Als Ergebnis erhalten wir:

```
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   25.000      5.802   4.309   0.0126 *
x             30.000      8.206   3.656   0.0217 *
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 10.05 on 4 degrees of freedom
Multiple R-squared:  0.7697,    Adjusted R-squared:  0.7121
F-statistic: 13.37 on 1 and 4 DF,  p-value: 0.02166
```

Über den Koeffizienten von `x` kann der Einfluss der Düngung auf die Wuchshöhe beurteilt werden. Pflanzen, die gedüngt wurden, sind im Mittel 30 Längeneinheiten größer als Pflanzen, die nicht gedüngt wurden. Die Teststatistik für $H_0 : \beta_1 = 0$ vs. $H_1 : \beta_1 \neq 0$ ist gleich 3.656 und der p-Wert 0.0217. Damit ist ein signifikanter Einfluss ($\alpha = 0.05$) des Düngens auf die Wuchshöhe feststellbar. Diese Ergebnisse sind identisch mit denen von Beispiel 4.4 und entsprechen den Ergebnissen aus Stocker und Steinke [2017], Fallbeispiel 4, S. 674.

4.3.2 Lineare Regression unter Heteroskedastizität

Wir betrachten weiterhin das Regressionmodell (4.1) von S.137, allerdings jetzt mit der allgemeineren Bedingung

$$E(U_i|X_i) = 0$$

anstelle von (4.2). Die bedingte Varianz $Var(U_i|X_i)$ muss in diesem Modell nicht mehr konstant (gleich σ^2) sein. Man spricht auch von einem *bedingt heteroskedastischen* Modell. Die Koeffizienten β_0 und β_1 werden immer noch mit den gleichen Formeln geschätzt wie zuvor. Für die Berechnung der Schätzer für die Varianzen von $\hat{\beta}_i$ werden aber andere Formeln verwendet.

• **Beispiel 4.14** • Wir betrachten das Fallbeispiel 2 aus Stocker und Steinke [2017], S. 640. Anhand des California-Testscore-Datensatzes untersuchen wir den Zusammenhang zwischen der erreichten Punktzahl in einem Test (`testscr`) und der Klassengröße (`str`). Wir werden

auf das Beispiel noch einmal in Abschnitt 4.4.2, ab S.148, zurückkommen. Die Daten sind im *R*-Paket *Ecdat* enthalten, das zunächst zu installieren ist.

```
> library(Ecdat)
> data("Caschool")
> attach(Caschool)
```

Das Packet *Ecdat* wird geladen und die Daten bereitgestellt. Zunächst führen wir eine lineare Regression unter klassischen Annahmen durch.

```
> erg=lm(testscr~str)
> summary(erg)
```

Wir erhalten das Ergebnis:

```
Call:
lm(formula = testscr ~ str)

Residuals:
   Min     1Q Median     3Q    Max
-47.73 -14.25   0.48  12.82  48.54

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   698.93       9.47   73.82  < 2e-16 ***
str            -2.28       0.48   -4.75  2.8e-06 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 18.6 on 418 degrees of freedom
Multiple R-squared:  0.0512,    Adjusted R-squared:  0.049
F-statistic: 22.6 on 1 and 418 DF,  p-value: 2.78e-06
```

Der Koeffizient zu `str` beträgt -2.28 und ist signifikant von Null verschieden (p-Wert $\approx 2.8 \cdot 10^{-6} < 0.05$): Umso größer die Klassen sind, umso schlechter sind im Mittel die Testergebnisse. Der R^2-Wert fällt mit 0.0512 sehr niedrig aus. Die Testscore-Daten werden also nicht gut durch die Klassengrößen-Daten beschrieben. Es gibt immer noch viel unerklärte Variabilität in dem Modell. Der Schätzwert für σ ist 18.6 und die Standardfehler zu β_0 und β_1 betragen 9.47 und 0.48.

Wir gehen davon aus, dass die Annahme der Homoskedastizität problematisch sein könnte. Im heteroskedastischen Fall ändern sich die Schätzwerte für β_i nicht, aber deren Standardfehler.

Zur Bestimmung der neuen Schätzer für die Kovarianz verwenden wir die Funktion

`vcovHC(erg, type="const",...)`.

aus dem Paket **sandwich**. `erg` ist die Ergebnisvariable einer linearen Regression, d.h. der Funktion `lm()`. In der Funktion `vcovHC()` sind verschiedene Berechnungsformeln zur Bestimmung der Kovarianzmatrix von $\hat{\beta} = (\hat{\beta}_0, \hat{\beta}_1)^T$ implementiert. Mit

`type="HC"`

wird die Formel nach *White* (White [1980]) ausgewählt, wie sie bspw. in Stocker und Steinke [2017], S. 622, angegeben ist. Als Ergebnis liefert `vcovHC()` die Kovarianzmatrix von $\hat{\beta}$.

• Beispiel 4.14 fortgesetzt •

```
> library(sandwich)
> V=vcovHC(erg,type="HC")
> SE=sqrt(diag(V))
> SE
(Intercept)         str
    10.3397      0.5183
```

`diag()` extrahiert aus der Matrix `V` die Hauptdiagonale. `sqrt(diag(V))` liefert dann die Standardfehler für $\hat{\beta}$. Das sind die Ergebnisse aus Stocker und Steinke [2017], S. 643. Die Standardfehler im heteroskedasischen Modell unterscheiden sich in unserem Fall nicht stark von denen im homoskedastischen Fall.

Konfidenzintervalle berechnen wir mit der Formel

$$\text{Schätzer} \pm \text{Quantil} \cdot \text{Standardfehler}. \tag{4.3}$$

Da die Schätzer $\hat{\beta}_i$ asymptotisch normalverteilt sind, verwenden wir hier die Quantile der Standardnormalverteilung.

```
> q = qnorm(0.975)
> erg$coef - q*SE
(Intercept)         str
    678.668      -3.296
> erg$coef + q*SE
(Intercept)         str
    719.198      -1.264
```

Das sind die oberen und unteren Intervallgrenzen für approximative 0.95-Konfidenzintervalle für β_0 bzw. β_1, vgl. Stocker und Steinke [2017], S. 644. Wollen wir das Testproblem $H_0 : \beta_1 \geq -1$ vs. $H_1 : \beta_1 < -1$ prüfen, dann können wir auch hier den Test direkt durchführen. Die Teststatistik berechnet sich dabei nach der Formel

$$(\text{Schätzer} - \text{Vorgabewert})/\ \text{Standardfehler}. \tag{4.4}$$

Dazu nutzen wir die Ergebnisse der vorherigen Berechnungen.

```
> tstat=(erg$coef[2]-(-1))/SE[2]
> tstat
   str
-2.469
> pnorm(tstat)
     str
0.006766
```

Der Wert der Teststatistik ist -2.469 und der p-Wert ca. 0.0068, vgl. Stocker und Steinke [2017], S. 644.

4.4 Multiples lineares Regressionsmodell

Wir betrachten das multiple lineare Regressionsmodell

$$Y_i = \beta_0 + \beta_1 X_{i1} + \cdots + \beta_p X_{ip} + U_i \tag{4.5}$$

für $i = 1, \ldots, n$. In diesem Modellansatz wird Y_i in Abhängigkeit von p Regressorvariablen $X_{i1}, \ldots, X_{ip}$ beschrieben. Die Vektoren $(X_{i1}, \ldots, X_{ip}, Y_i)$ seien u.i.v. Die einfache lineare Regression erhält man für $p = 1$.

4.4.1 Lineare Regression unter klassischen Annahmen

Unter den klassischen Annahmen unterstellen wir

$$U_i | X_{i1}, \ldots, X_{ip} \sim N(0, \sigma^2), \tag{4.6}$$

d.h. die Fehlerterme U_i sind bedingt normalverteilt.

Bei der Umsetzung mit *R* verwendet man i.W. die gleiche Syntax wie bei der einfachen linearen Regression.

`lm(y ~ x1+x2+..+xp,...)`

Die Regressorvariablen werden hierbei hinter dem Tilde-Symbol als Summe geschrieben.

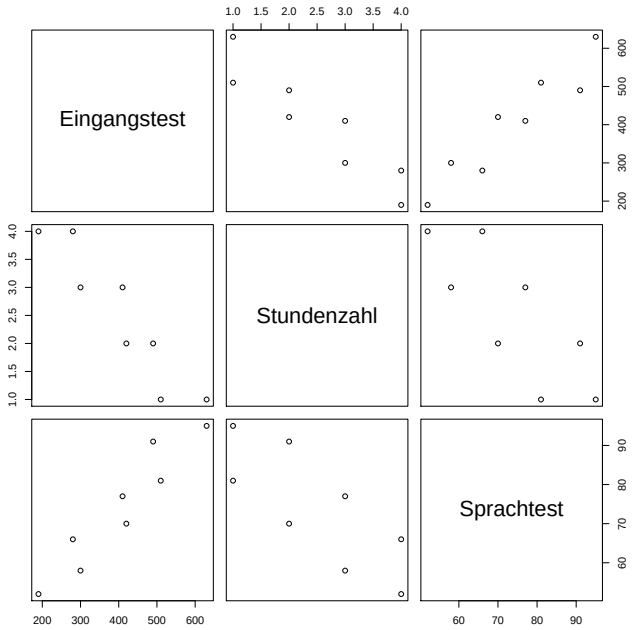
• Beispiel 4.15 • Wir betrachten Beispiel 12.2.1 aus Stocker und Steinke [2017], S. 659. 8 Studenten nehmen Unterricht an einer Sprachschule und buchen eine gewisse Anzahl von Unterrichtsstunden pro Woche (`S`). Zu Beginn des Unterrichts werden die Studenten zur Einstufung einem Eingangstest unterzogen, in dem sie eine gewisse Anzahl von Punkten erreichen (`ET`). Am Ende des Unterrichts wird ein Sprachtest durchgeführt, bei dem die Studenten eine gewisse Punktzahl erreichen (`ST`). Wir interessieren uns primär für den Einfluss der Unterrichtsstunden (`S`) auf den Lernerfolg (`ST`). Dabei sollte man aber die Leistung der Studenten zu Beginn des Sprachunterrichts nicht vernachlässigen (`ET`). Zur Veranschaulichung zeichnen wir zunächst einige Streudiagramme mit Hilfe der *R*-Funktion `pairs()`.

```
> S = c(4,4,3,3,2,2,1,1)
> ET= c(190,280,300,410,420,490,510,630)
> ST= c(52,66,58,77,70,91,81,95)
> daten= cbind(ET,S,ST)
> lab=c("Eingangstest","Stundenzahl","Sprachtest")
> pairs(daten,labels=lab)
```

Das Ergebnis ist in Abbildung 4.4.1 dargestellt. Dabei ist erkennbar, dass Studenten mit besseren Eingangstestergebnis auch i.d.R. ein besseres Sprachtestergebnis erzielen. Andererseits bekommt man anhand des entsprechenden Streudiagramms den Eindruck, dass eine Erhöhung der Anzahl der Sprachunterrichtsstunden das Ergebnis des Sprachtests verschlechtert. Das liegt daran, dass das Vorwissen der Studenten hier nicht berücksichtigt wird: Ein

Student, der bereits gute Sprachkenntnisse besitzt, wird vielleicht nicht so viele Unterrichtsstunden nehmen, aber trotzdem bei dem Sprachtest gut abschneiden, während ein Student mit schwachem Vorwissen viele Unterrichtsstunden braucht, aber trotzdem noch nicht zu den guten Studenten aufschließen kann. Bei einer Regression, bei der man das Eingangstestergebnis nicht berücksichtigt wird, würde man entsprechend einen negativen Zusammenhang zwischen Stundenzahl `S` und Sprachtest `ST` erhalten.

Abbildung 4.4.1: Streudiagramme



Es wird jetzt eine multiple lineare Regression durchgeführt.

```
> erg=lm(ST~S+ET)
> summary(erg)
```

Alternativ kann man auch alle Regressorvariablen in einer Matrix zusammenfassen und die Regression bzgl. der Matrix durchführen. Diese Vorgehensweise kann insbesondere beim Vorliegen von sehr vielen Regressoren vorteilhaft sein.

```
> X=cbind(S,ET)
> erg1=lm(ST~X)
> summary(erg1)
```

Wir erhalten das folgende Ergebnis:

```
Call:
lm(formula = ST ~ X)

Residuals:
     1        2        3        4        5        6        7        8
 1.0167   0.4542  -2.9024  -1.7010  -2.4396   7.2340   1.8774  -3.5392

Coefficients:
```

```
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -11.27766   20.42406  -0.552  0.60460
XS            7.87949    3.52261   2.237  0.07551 .
XET           0.16181    0.02956   5.473  0.00277 **
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 4.169 on 5 degrees of freedom
Multiple R-squared:  0.9459,    Adjusted R-squared:  0.9243
F-statistic: 43.75 on 2 and 5 DF,  p-value: 0.0006793
```

Die Variablen `S` bzw. `ET` werden als Bestandteil von `X` bei der Ausgabe als `XS` bzw. `XET` bezeichnet.

Das Ergebnis entspricht jetzt eher unseren Erwartungen: Umso mehr Stunden Unterricht genommen wurden, umso besser ist das Ergebnis im Sprachtest ($\hat{\beta}_S \approx 7.88 > 0$). Andererseits erzielen Studenten mit einem besseren Ergebnis im Eingangstests bei gleicher Anzahl von Unterrichtsstunden auch ein besseres Ergebnis im Sprachtest ($\hat{\beta}_{ET} \approx 0.16 > 0$).

Eine weitere Möglichkeit, obige Regression durchzuführen, ist folgende:

```
> daten=data.frame(ET,S,ST)
> erg2=lm(ST~.,data=daten)
> summary(erg2)
```

Die y und alle x-Variablen werden in einem Datensatz (*daten*) zusammengefasst. In `lm( )` wird die y-Variable wie zuvor angegeben. Rechts von $\sim$ wird aber ein Punkt gesetzt. Damit werden alle Variablen des Datensatzes außer der zuvor spezifizierten y-Variable als Regressorvariablen gesetzt.

• Beispiel 4.16: ANOVA-Modelle • Wir betrachten folgende y-Beobachtungen, die in drei Gruppen erhoben wurden.

Gruppe 0	1.0	1.3	0.7		
Gruppe 1	2.7	2.1	1.9	2.3	
Gruppe 2	1.3	1.2	1.8	1.4	1.6

Es werden normalverteilte Beobachtungen unterstellt. Bei gleichen Varianzen soll geprüft werden, ob die Gruppenerwartungswerte identisch sind oder nicht. Das entspricht einer Verallgemeinerung des Tests über Erwartungswertdifferenzen bei gleichen Varianzen aus Abschnitt 4.2.2.

```
> y=c(1.0,1.3,0.7,1.7,2.1,1.9,2.3,1.3,1.2,1.8,1.4,1.6)
> x= c(rep(0,3),rep(1,4),rep(2,5))
> plot(x,y)
```

Die Daten werden eingelesen und mit einem Streudiagramm grafisch veranschaulicht. In Abbildung 4.4.2, S. 147, kann man sehen, dass die y-Werte in den einzelnen Gruppen erkennbar im Mittel unterschiedlich verteilt sind. `x` ist eine Variable, die angibt, welcher Gruppe die einzelnen y-Beobachtungen zuzuordnen sind. Die Bezeichnung der Gruppen mit 0 bis 2 ist

dabei willkürlich. Sie hätten auch mit 1, 2 und 3 oder mit 3, 7 und 37 bezeichnet werden können. Eine solche Variable, die eine Zuordnung vornimmt, bezeichnet man auch als **Faktor**. Das muss für *R* kenntlich gemacht werden, z.B. mit der Funktion `as.factor()`.

```
> xf= as.factor(x)
> xf
```

Wir erhalten:

```
 [1] 0 0 0 1 1 1 1 2 2 2 2 2
Levels: 0 1 2
```

Die Werte (von `x`) sind unverändert. Sie werden aber jetzt als Stufe (level) des Faktors `xf` interpretiert. Für die Faktorvariable `xf` stellt *R* auch ein verändertes Ergebnis der `plot()`-Funktion bereit.

```
> plot(xf,y)
```

Die Darstellung befindet sich in Abbildung 4.4.3, S.147. Anstelle eines Streudiagramms werden für jede Gruppe Boxplots gezeichnet. Mit deren Hilfe kann man noch besser die Lage und Streuung der Daten in den verschiedenen Gruppen vergleichen.

Abbildung 4.4.2: `plot(x,y)`

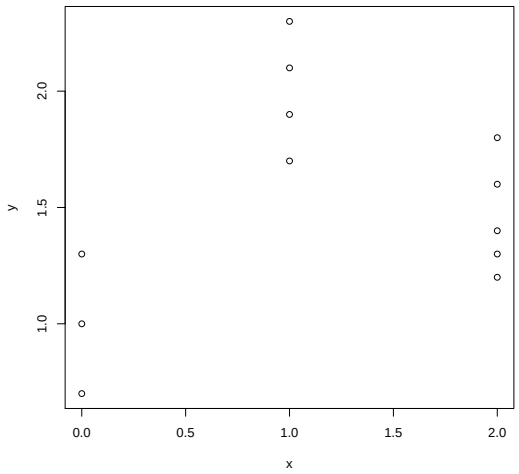
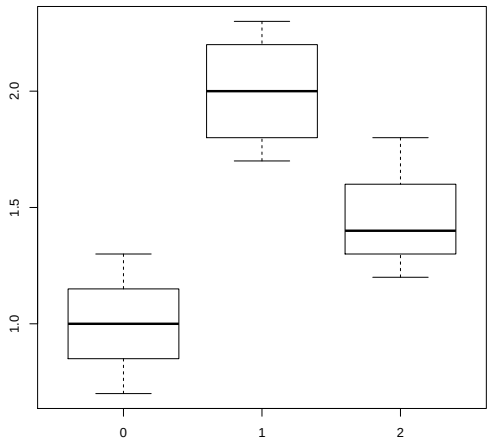


Abbildung 4.4.3: `plot(xf,y)`



Die lineare Regression wird jetzt bzgl. der Faktorvariable durchgeführt.

```
> summary(lm(y~xf))
```

Wir erhalten:

```
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   1.0000     0.1506   6.642 9.46e-05 ***
xf1           1.0000     0.1992   5.021 0.000718 ***
xf2           0.4600     0.1904   2.415 0.038898 *
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

```
Residual standard error: 0.2608 on 9 degrees of freedom
Multiple R-squared:  0.741,     Adjusted R-squared:  0.6834
F-statistic: 12.87 on 2 and 9 DF,  p-value: 0.002292
```

Bei den Koeffizienten finden wir insgesamt 3 Einträge, für jede Stufe der Faktorvariable einen. Besonders wichtig ist hier das Ergebnis in der letzten Zeile. Mit einem F-Test kann geprüft werden, ob die Erwartungswerte aller Gruppen identisch sind (Nullhypothese). Der Wert der Teststatistik wird hier als `F-statistic` bezeichnet. Er nimmt den Wert 12.87 an. Der p-Wert des Tests beträgt `p-value:` 0.002292. Das heißt, zum Signifikanzniveau $\alpha = 0.05$ werden nicht alle Erwartungswerte der Gruppen als identisch angesehen; es gibt statistisch signifikante Unterschiede. Dieses Ergebnis findet sich auch als 2. Teil des Fallbeispiels 4 in Stocker und Steinke [2017], S. 676.

Das obige Ergebnis lässt sich reproduzieren, wenn man binäre Hilfsvariablen, so genannte Dummy-Variablen einführt, vgl. dazu Tabelle 12.2.3, Stocker und Steinke [2017], S. 677.

```
> x1=x2=rep(0,length(y))
> x1[x==1]=1
> x2[x==2]=1
```

Die Variable `x1` ist genau für Beobachtungen von Gruppe 1 gleich 1 und `x2` ist genau für Beobachtungen von Gruppe 2 gleich 1.

```
> summary(lm(y~x1+x2))
```

Es wird hier eine multiple lineare Regression bzgl. der binären Variablen `x1` und `x2` durchgeführt. Wir erhalten bis auf die Notation der Regressorvariablen (`x1` anstelle von `xf1` und `x2` anstelle von `xf2`) das gleiche Ergebnis.

4.4.2 Lineare Regression unter Heteroskedastizität

Auf die Verteilungsannahme (4.6) von S.144 wird jetzt verzichtet. Insbesondere ist die bedingte Varianz $Var(Y_i|X_{i1}, \ldots, X_{ip})$ nicht mehr zwingend konstant.

• Beispiel 4.17 • Wir setzen Beispiel 4.14 von S. 141 fort. Neben der Klassengröße `str` nehmen wir aber auch die Variable `mealpct` („Begünstigten"-Anteil) in das Modell zur Erklärung des Testergebnisses `testscr` auf. `mealpct` gibt den Anteil der Schüler an, die ein vergünstigtes Mittagessen beziehen. Diese Schüler kommen zumeist aus finanzschwachen Familien und haben oft verstärkt Probleme in der Schule. Wir führen zunächst erneut eine Regression unter klassischen Annahmen durch.

```
> erg=lm(testscr~str+mealpct)
> summary(erg)

Call:
lm(formula = testscr ~ str + mealpct)

Residuals:
   Min     1Q Median     3Q    Max
```

```
-33.66  -5.52  -0.17   5.64  34.49

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 702.9113     4.7002  149.55  < 2e-16 ***
str          -1.1172     0.2404   -4.65  4.5e-06 ***
mealpct      -0.5998     0.0168  -35.78  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 9.22 on 417 degrees of freedom
Multiple R-squared:  0.767,     Adjusted R-squared:  0.766
F-statistic:  686 on 2 and 417 DF,  p-value: <2e-16
```

Beide Koeffizienten zu `str` und `mealpct` sind negativ und betragsmäßig so groß, dass sie im Modell mit klassischen Annahmen signifikant sind. Ein hoher Begünstigten-Anteil reduziert also auch den Test-Score. R^2 fällt hierbei mit 0.767 deutlich höher aus als im Modell ohne die Variable `mealpct`, vgl. Beispiel 4.14. Die Klassengröße reduziert immer noch die Testergebnisse, aber nicht mehr so stark: Der Koeffizient zu `str` beträgt jetzt -1.1172 im Gegensatz zu -2.28 in Beispiel 4.14.

Im Modell mit bedingt heteroskedastischen Fehlertermen müssen die Standardfehler und die Werte der Teststatistiken und p-Werte zu den Testproblemen $H_0 : \beta_i = 0$ neu berechnet werden.

```
> V=vcovHC(erg,type="HC")
> SE = sqrt(diag(V))
```

`SE` enthält die Werte für die Standardfehler im Modell mit bedingt heteroskedastischen Fehlern.

```
> tstat=erg$coef/SE
> pvalue=2*(1-pnorm(abs(erg$coef/SE)))
```

Das sind die Werte für die Teststatistiken und die p-Werte. Die Berechnungen beruhen auf asymptotischen Ergebnissen. Es wird eine asymptotische Normalverteilung für die Verteilung der Schätzer unterstellt. Schließlich werden die Grenzen für asymptotische 0.95-Konfidenzintervalle ermittelt, vgl. (4.3) von S. 143.

```
> lower=erg$coef - qnorm(0.975)*SE
> upper=erg$coef + qnorm(0.975)*SE
```

Jetzt bringen wir die Ergebnisse in eine Form, wie sie der Ausgabe bei einer linearen Regression üblicherweise entspricht.

```
> M=cbind(erg$coef,SE,tstat,pvalue,lower,upper)
> rownames(M)[1]="const"
> colnames(M)[1]="estimate"
> M
```

Alle Zwischenergebnisse werden in einer Matrix mit `cbind` zusammengefasst. Mit den `rownames`- und `colnames`-Kommandos werden der ersten Zeile und der ersten Spalte geeignete Bezeichner zugewiesen. Wir erhalten das Ergebnis:

```
         estimate       SE    tstat    pvalue    lower    upper
const    702.9113 5.49523 127.913 0.000e+00 692.1409 713.6818
str       -1.1172 0.27072  -4.127 3.679e-05  -1.6478  -0.5866
mealpct   -0.5998 0.01714 -34.984 0.000e+00  -0.6334  -0.5661
```

Die ersten beiden Ergebnisspalten finden wir auch in Stocker und Steinke [2017], Tabelle 12.2.2, S. 670, Modell 3. Das Ergebnis im heteroskedastischen Modell unterscheidet sich nicht sehr stark von dem im homoskedastischen Modell. Auch die Interpretation des Ergebnisses bleibt damit dieselbe.

5 Ergänzende und weiterführende Themen

5.1 Fortgeschrittene Grafiken

5.1.1 Farbpalette

Möchte man auf Farben zugreifen bzw. Farben zuordnen, dann kann man das über die Standardfarben machen, die über Zahlen (0,1,2,...) kodiert sind.

```
> x=c(1,2,2,3,3,3,4,4,5)
> barplot(table(x),col=1:5)
```

Wir erhalten Abbildung 5.1.1. Die Option `col` erlaubt die Zuweisung von Farben zu den Balken des Balkendiagramms. Hier werden die Farben mit den Nummern 1 bis 5 verwendet.

Abbildung 5.1.1: Standardfarben

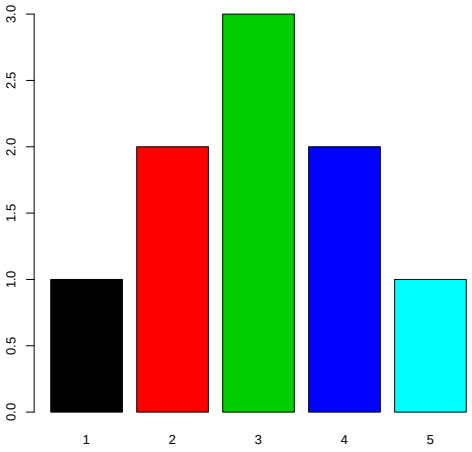
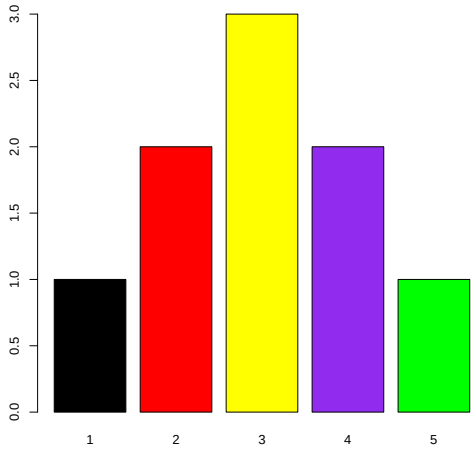


Abbildung 5.1.2: Vorgegebene Farben



Alternativ kann man auf Namensbezeichner für die Farben zugreifen.

```
> barplot(table(x),col=c("black","red","yellow",
+                        "purple2","green"))
```

Wir erhalten Abbildung 5.1.2. Die Bezeichner sind hierbei selbsterklärend: `"black"` steht für „schwarz“, `"red"` für „rot“ usw. Mit der Anweisung

```
> colors()
```

werden alle verfügbaren vordefinierten Farbbezeichner aufgelistet. Z.B. erhalten wir mittels

```
> colors()[550:560]
```

die Bezeichner zu den Indizes 550 bis 560:

```
 [1] "purple3" "purple4" "red" "red1" "red2" "red3" "red4"
 [8] "rosybrown" "rosybrown1" "rosybrown2" "rosybrown3"
```

Mithilfe der `palette( )`-Anweisung kann man sich die aktuellen Standardfarben anzeigen lassen und sie bei Bedarf auch verändern.

```
> palette()
```

liefert

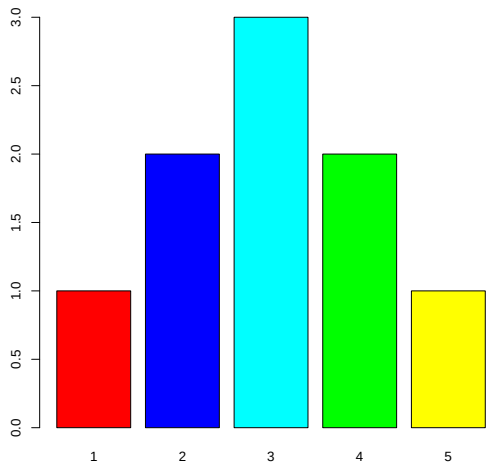
```
[1] "black" "red" "green3" "blue" "cyan" "magenta" "yellow"  "gray"
```

8 Farben sind vordefiniert. Insbesondere sind das die ersten 5 Farben aus Abbildung 5.1.1. Mit `palette` kann man die Standardfarbzuordnung auch überschreiben.

```
> palette(c("red","blue","cyan","green","yellow"))
> barplot(table(x),col=1:5)
```

Wir erhalten Abbildung 5.1.3.

Abbildung 5.1.3: Veränderte Farbpalette



Die Anweisung

```
> palette("default")
```

stellt die Standardbelegung der Farben wieder her.

5.1.2 Verwendung mathematischer Ausdrücke in Grafiken

Zur Beschriftung von Grafiken ist normaler Text u.U. nicht ausreichend. Die Verwendung von speziellen mathematischen Symbolen, griechischen Buchstaben oder Indizierungen wäre vorteilhaft. Mithilfe der `expression( )`-Anweisung kann das bewerkstelligt werden.

```
> x=0:20
> y=dpois(x,5)
```

```
> plot(x,y,type="h",lwd=2,ylab=expression(f[X](x)),
+       main=expression("Wahrscheinlichkeitsfunktion zu Po("
+                       *lambda*"), "*lambda==5))
> abline(h=0)
```

Wir erhalten Abbildung 5.1.4. Die `expression()`-Funktion generiert Text mit Sondersymbolik und kann in Zusammenhang mit Überschriften (`main`), Achsenbeschriftungen (`xlab`, `ylab`) oder Texteinträgen (`text()`) verwendet werden. Normaler Text wird weiterhin in Anführungsstrichen (`" "`) geschrieben. Griechische Buchstaben können über ihre Bezeichner

alpha, beta, gamma, delta, lambda, Gamma, Lambda, ...

verwendet werden. Um normalen Text mit speziellen Symbolen zu verknüpfen, wird `*` verwendet. Vergleichs- oder auch Zuweisungsoperationen werden mittels

==, <=, >= , !=

geschrieben. Für Indizes bzw. Potenzen verwendet man die Syntax

x[2] bzw. `x^2`.

Auf diese Weise kann man x_2 bzw. x^2 ausdrücken. Um $\hat{x}$, $\tilde{y}$ bzw. $\bar{z}$ darzustellen, kann man

hat(x), tilde(y) bzw. bar(z)

schreiben.

Abbildung 5.1.4: Mathematische Symbole

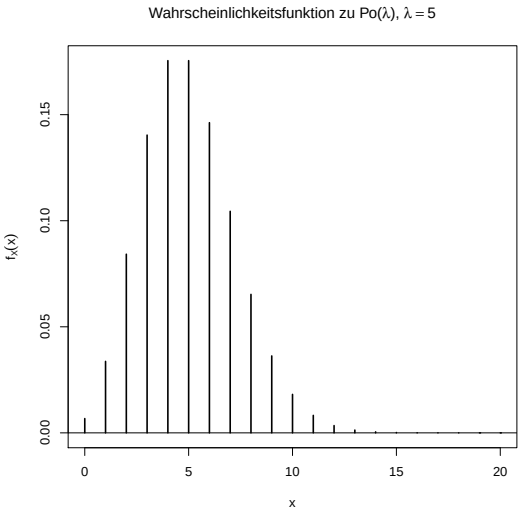
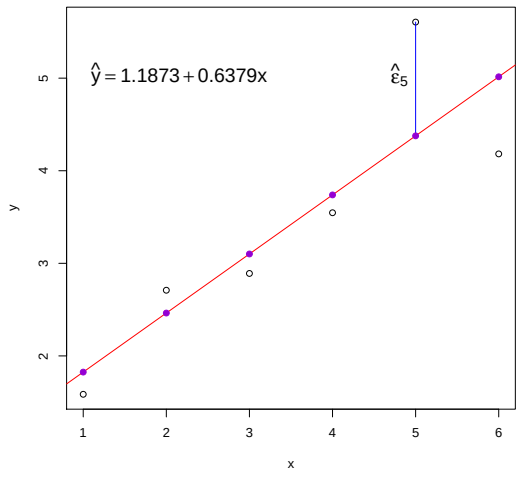


Abbildung 5.1.5: Lineare Regression



Einen Überblick über die zur Verfügung stehende mathematische Notation erhält man durch Aufruf von

```
> help(plotmath)
```

Betrachten wir ein weiteres Beispiel.

```
> set.seed(12345); n=6; x=1:n
> y=x+rnorm(n)
> er=lm(y~x)
> print(er$co)
```

Es wird eine lineare Regession durchgeführt. Die KQ-Koeffizienten sind

```
(Intercept)           x
  1.1872563   0.6379328
```

Die Daten und die Regressionsgerade sollen nun veranschaulicht werden.

```
> plot(x,y)
> abline(er$co[1],er$co[2],col="red")
> yhat=1.1873+0.6379*x
> text(1,5,expression(hat(y)==1.1873+0.6379*x),cex=1.5,pos=4)
> lines(c(5,5),c(y[5],yhat[5]),col="blue")
> points(x,yhat,pch=19,col="darkviolet")
> text(5,(y[5]+yhat[5])/2,expression(hat(epsilon)[5]),cex=1.5,
+    pos=2)
```

Das Ergebnis finden Sie in Abbildung 5.1.5. Die `text()`-Anweisung wird hier verwendet, um die Grafik um Beschriftungen zu ergänzen. Neben der Verwendung von griechischen Zeichen und Indizierung wurde hier auch das Dach-Symbol ($\hat{y}$) verwendet.

Schauen wir uns ein letztes Beispiel mit einer etwas komplexeren Formel an.

```
> ex=expression(varphi(x)*"="*frac(1,sqrt(2*pi))~exp*
+                   bgroup("(",-frac(1,2)*x^2,")"))
> curve(dnorm(x),-3,3,ylab="y",main="N(0,1)-Dichte")
> text(0,0.1,ex,cex=1.3)
```

Das Ergebnis finden Sie in Abbildung 5.1.7. Mit `frac(a,b)` wird ein Bruch und mit `sqrt(a)` eine Wurzel dargestellt. Mittels `bgroup(...)` kann man Klammern verwenden, deren Größe sich an den Ausdruck zwischen den Klammern anpasst.

5.1.3 Speichern von Grafiken

Grafiken, die man in *R* erstellt hat, möchte man häufig für die weitere Verwendung, z.B. im Rahmen einer Publikation, speichern. Das Speichern der Grafiken kann man manuell vornehmen. Arbeitet man mit *RGui* kann man die Grafik auswählen und sie dann über *Datei/Speichern als* in einem Grafikformat speichern. Zur Verfügung stehen hier z.B. *pdf*, *jpeg* und *png* als Dateiformate. In *RStudio* kann das analog über *Export/Save as Image ...* bewerkstelligt werden. Möchte man mehrere Abbildungen im Rahmen einer *R*-Sitzung erstellen, dann kann es sinnvoll sein, das Abspeichern der Grafik im *R*-Quelltext zu realisieren. Dazu verwenden wir die Anweisung `savePlot()`.

Abbildung 5.1.6: Speichern

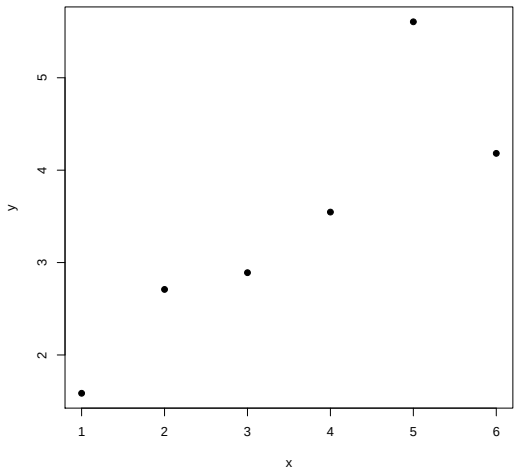
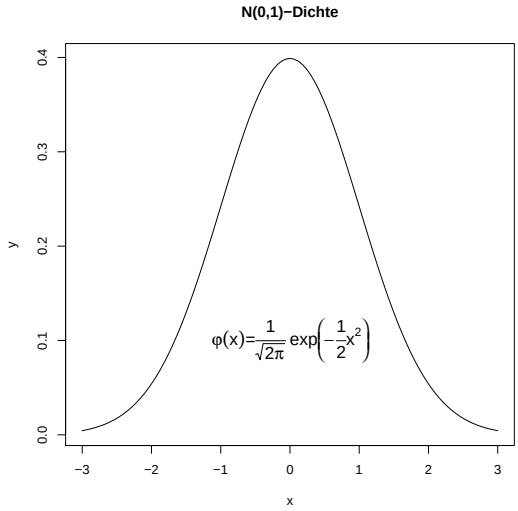


Abbildung 5.1.7: Mathematische Symbole in *R*




```
> setwd("C:/bilder")
> savePlot("grafik1","pdf")
```

Mit `setwd( )` wird zunächst das Arbeitsverzeichnis (*working directory*) festgelegt. Anschließend wird die aktuelle Grafik in diesem Verzeichnis unter dem Namen `grafik1.pdf` gespeichert. `savePlot` unterstützt verschiedene Grafikformate, z.B. pdf, jpeg, png etc. Mehr Informationen findet man in der Hilfe. Möchte man die `savePlot()`-Anweisung in *RStudio* anwenden, muss man die Grafikausgabe u.U. zunächst über

```
> windows()
```

auf ein externes Fenster umleiten.

Eine weitere Vorgehensweise ist die Verwendung der Anweisungen `pdf( )`, `png( )` bzw. `postscript( )`, mit deren Hilfe Grafiken im *pdf*-, *png*- bzw. postscript-Format gespeichert werden. Betrachten wir ein einfaches Beispiel.

```
> set.seed(12345); n=6; x=1:n
> y=x+rnorm(n)
```

Zunächst wird ein Datensatz erstellt.

```
> pdf("grafik2.pdf")
> plot(x,y,pch=19)
> dev.off()
```

Das Streudiagramm der (x_i, y_i)-Punkte wird im Arbeitsverzeichnis unter dem Namen *grafik2.pdf* gespeichert. Die Grafikanweisungen, die in der pdf-Datei gespeichert werden, stehen hierbei zwischen den Anweisungen `pdf(...)` und `dev.off( )`. Grafikanweisungen, die vor `pdf()` oder hinter `dev.off()` stehen, werden in der *pdf*-Datei nicht berücksichtigt.

Mit weiteren Optionen kann man das Bild an seine Vorstellungen anpassen.

```
> pdf("Grafik2a.pdf",width=6/2.54,height=4/2.54)
> plot(x,y,pch=19)
> dev.off()
```

Mit den Optionen `width` und `height` kann man bspw. die Breite und die Höhe des Bildes vorgeben. Die Größenangaben sind hier in Zoll (*inch*) zu machen. Das Teilen durch 2.54 bewirkt eine Umrechnung in *cm*. Die Grafik, die erzeugt wird, soll also im Beispiel 6 cm breit und 4 cm hoch sein. `png()` und `postscript()` kann man auf ähnliche Weise wie `pdf()` verwenden.

5.2 Einführung in die Programmierung

Was versteht man eigentlich unter Programmieren? *R* ist eine Programmiersprache bzw. eine Programmierumgebung für statistische Analysen. War all das, was wir in den vorherigen Kapiteln behandelt haben, keine Programmierung? Immerhin haben wir ja uns der Programmiersprache *R* bedient und sämtliche Befehle in die *R*-Konsole, also das Programmierfenster von *R*, eingegeben. Es wäre sicherlich nicht ganz verkehrt, dies unter den Begriff Programmierung zu stellen. Allerdings waren die Funktionen (Befehle), die wir bisher verwendet haben, wie z.B. `seq()`, `matrix()`, `dnorm()` oder `hist()`, ausnahmslos Funktionen, die in *R* vorimplementiert waren oder aus Zusatzpaketen von *R* stammten. Die Befehle waren also stets in einer nicht von uns bestimmten Form anzuwenden. Den Hilfedateien zu diesen Befehlen konnten wir entnehmen, welche Argumente zulässig sind und welches Objekt als Ergebnis ausgegeben wird. In diesem Abschnitt werden wir uns damit befassen, wie wir selbst Funktionen kreieren können. Zunächst befassen wir uns aber mit Schleifen. Schleifen spielen z.B. im Zusammenhang statistischer Simulationen eine wichtige Rolle und sind unverzichtbarer Bestandteil vieler Programme. Als weiterführende Literatur sei z.B. auf Ligges [2008] verwiesen.

5.2.1 Verwendung von Schleifen

Eine `for`-Schleife ist eine Prozedur, bei der eine Variable eine bestimmte Routine wiederholend durchläuft und dabei ihren Wert gemäß den Werten einer vorgegebenen Datenmenge (meist numerisch) verändert.

Die allgemeine Syntax einer for-Schleife ist

```
for(i in indexvector)
{
   Anweisungen ...
}
```

Die Index-Variable `i` nimmt dabei nacheinander die Werte des Vektors `indexvector` an. Für jeden dieser Werte von `i` werden die Anweisungen im Anweisungsblock { ... } durchgeführt.

Die Schleife

```
for(i in c(2,5,7))
{
   print(i)
}
```

liefert

```
[1] 2
[1] 5
[1] 7
```

Die Indexvariable `i` nimmt zuerst den Wert 2 an. Dann werden alle Anweisungen im Anweisungsblock durchgeführt. In diesem Beispiel ist es nur die `print(i)`-Anweisung. Dann nimmt `i` den zweiten Wert des Vektors `c(2,5,7)` an, also 5, und `print(i)` wird angewendet. Schließlich nimmt `i` den Wert 7 an und `print(i)` wird wieder angewendet.

Ist nur eine Anweisung durchzuführen, kann man die geschweiften Klammern auch weglassen und verkürzend

```
for(i in c(2,5,7)) print(i)
```

schreiben. Das Ergebnis ist dasselbe.

Mithilfe von Schleifen kann man sich z.B. die verschiedenen Farben der Farbpalette darstellen lassen.

```
plot(-1,-1,xlim=c(0,5),ylim=c(0,5),xlab="",ylab="",axes=F)
for(i in 1:5) for(j in 1:5)
    text(i-0.5,j-0.5,5*(j-1)+i,col=5*(j-1)+i,cex=2)
```

Als Ergebnis erhalten wir Abbildung 5.2.1. Beachten Sie, dass auch größere Zahlen zur Bezeichnung von Farben verwendet werden können, sich die Farben aber ab dem Zahlenwert 9 periodisch wiederholen.

Als weiteres Beispiel veranschaulichen wir das Gesetz der Großen Zahlen.

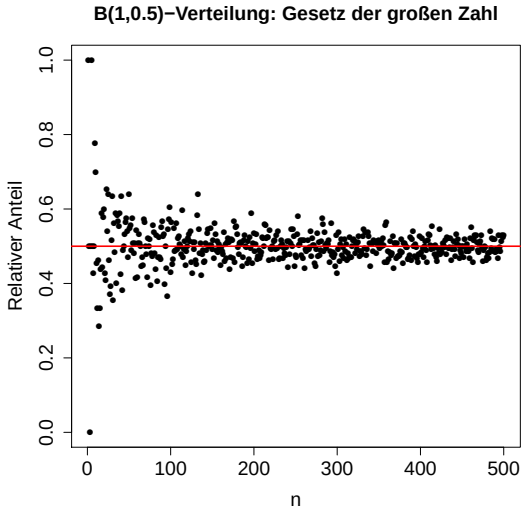
```
set.seed(123456)
N=500
RelAnteil=rep(0,N)

for (n in 1:N)
{
  Stichprobe=rbinom(n,1,0.5)
  RelAnteil[n]=mean(Stichprobe)
}
```

Für die Stichprobenumfänge von $n = 1$ bis $n = N = 500$ werden jeweils n Bernoulli-verteilte Pseudozufallszahlen zum Parameter $\pi = 0.5$ erzeugt. Der Mittelwert dieser so erzeugten Zahlen ist ein Schätzer für den Anteilswert 0.5. In diesem Beispiel stehen zwei Anweisungen im Anweisungsblock der `for`-Schleife. Wir veranschaulichen das Ergebnis der Simulation mit einer Abbildung.

Abbildung 5.2.1: Standardfarben

Abbildung 5.2.2: Simulation mit Schleifen




```
mtext="B(1,0.5)-Verteilung: Gesetz der großen Zahl"
plot(1:N,RelAnteil,main=mtext,xlab="n",ylab="Relativer Anteil",
     pch=16,cex.axis=1.5,cex.lab=1.5,cex.main=1.5)
abline(h=0.5,lwd=2,col=2)
```

Das Ergebnis der Simulation finden Sie in Abbildung 5.2.2.

5.2.2 Definition von Funktionen und bedingte Programmausführung

Definition

Bisher haben wir die in *R*-implementieren Anweisungen und Funktionen verwendet. Es ist häufig nützlich, eigene Funktion zu definieren, um mit ihrer Hilfe häufig wiederkehrende Aufgaben zu erfüllen. Die allgemeine Funktionssyntax hat folgende Form:

```
fName = function(x){
  Anweisungen ...
  return(y)
}
```

Das Schlüsselwort ist `function`. `fName` steht für den Namen der Funktion. Eine Funktion kann ein, mehrere oder auch kein Argument haben. Beim Funktionsaufruf werden eine oder mehrere Anweisungen durchgeführt und ggf. ein Objekt als Funktionswert zurückgegeben (`return()`).

Angenommen, wir möchten die Funktion $f(x) = (x^2 - 1) \cdot e^{-x}$ berechnen. Dann können wir sie folgendermaßen definieren:

```
fu = function(x){
```

```
   y=(x^2-1)*exp(-x)
   return(y)
}
```

Der Name der Funktion ist im Beispiel `fu`. Sie hat als einziges Argument `x`. Der Funktionswert `y` wird berechnet und zurückgegeben. Anstelle von `return(y)` hätte man hier auch einfach `y` schreiben können. Oder man hätte die zweite Zeile der Funktionsdefinition ganz weglassen können, da der Wert, der in der Funktion als letztes berechnet wurde, als Funktionswert interpretiert wird. Gleichwertig zu obiger Definition ist also

```
fu = function(x){
     (x^2-1)*exp(-x)
}
```

`fu` ist der Name, der von uns definierten Funktion. Er kann verwendet werden wie wir es mit Standardfunktionsnamen tun würden.

```
> fu(2)
```

liefert den Funktionswert der Funktion an der Stelle 2.

```
[1] 0.4060058
```

Da unsere Funktion nur in einer Zeile definiert wird, können wir auch die verkürzende Schreibweise ohne geschweifte Klammern verwenden.

```
fu = function(x) (x^2-1)*exp(-x)
```

Eine Funktion kann auch mehr als ein Argument haben.

```
addiere = function(x,y)  x+y
```

Typen von Variablen müssen in *R* – anders als in vielen anderen Programmiersprachen – den Argumenten nicht zu gewiesen. Erst zur Laufzeit zeigt sich, ob die Funktion richtig verwendet wurde.

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

Die Funktion `addiere` lässt sich auf Zahlen, aber auch auf Vektoren und Matrizen anwenden.

```
> addiere("a","b")
Error in x + y : non-numeric argument to binary operator
```

Zeichenketten können nicht mit „+“ addiert werden.

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

Wie wir aus Abschnitt 2.2 wissen, kann man auch Zahlen zu Vektoren addieren.

Bedingte Programmausführung

Angenommen, wir möchten Berechnungen mit folgender Dichtefunktion durchführen:

$$f(x) = \begin{cases} 0.5 & , \text{ wenn } 0 < x < 1, \\ 0.75 - 0.25x & , \text{ wenn } 1 \leq x < 3, \\ 0 & , \text{sonst.} \end{cases}$$

Diese Funktion kann man mithilfe von

`if`(*Bedingung*){ *Anweisungen* } `else` { *Anweisungen* }

definieren. Nach der Formulierung einer Bedingung folgen die Anweisungen, die auszuführen sind, falls die Bedingung erfüllt ist. Diese werden als Funktionsblock in geschweifte Klammern gesetzt. Wenn die Bedingung nicht erfüllt ist, kann ein optionaler `else`-Befehlsblock folgen. Die Definition der obigen Funktion kann dann folgendermaßen aussehen:

```
fu = function(x){
  if(x>0&x<1){
    y=0.5
  }
  else {
    if(x>=1&x<3) {
      y=0.75-0.25*x
    }
    else {
      y=0
    }
  }
  y
}
fu(2.5)
```

Das Ergebnis is 0.125. Da nach einer Bedingung nur eine Anweisung folgt, kann man die geschweiften Klammern auch weglassen und die Funktion kompakter und übersichtlicher definieren.

```
fu = function(x){
  y=0
  if(x>0&x<1) y=0.5
  else if(x>=1&x<3) y=0.75-0.25*x;
  y
}
fu(2.5)
```

Zu beachten ist, dass die beiden folgenden Befehle zu Fehlermeldungen führen.

```
> fu(c(2.5,2.8))
> curve(fu(x),-1,4)
```

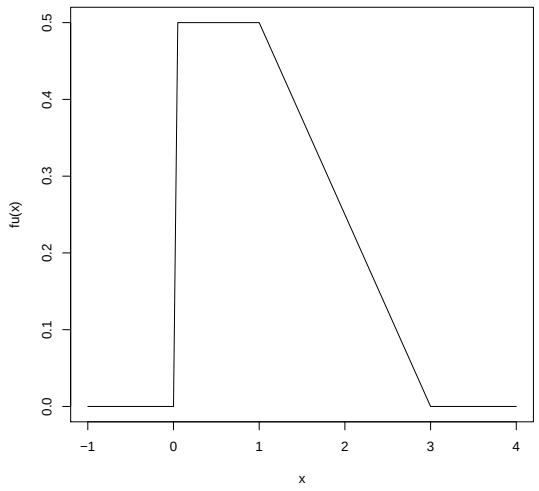
Die `if`-Anweisung kann keine Vektoren verarbeiten. Unsere Funktion akzeptiert nur Zahlen, aber keine Vektoren als Argument. Deshalb funktioniert die `curve()`-Funktion, siehe auch

S.87, im Zusammenhang mit `fu` nicht. Abhilfe schafft die Verwendung von `ifelse( )`, siehe auch S.55, da die Bedingung von `ifelse()` auch vektoriell sein kann.

```
fu = function(x){
  y1=ifelse(x>0&x<1,0.5,0)
  y2=ifelse(x>=1&x<3,0.75-0.25*x,0)
  y1+y2
}
curve(fu(x),-1,4)
```

Wir erhalten Abbildung 5.2.3.

Abbildung 5.2.3: Bedingte Funktionsdefinition



Noch kürzer kann man f mit Hilfe der Indikator-Funktion `I( )` definieren. `I`(*Bedingung*) liefert den Wert 1, wenn die Bedingung erfüllt ist, sonst 0. Vektorielle Ausdrücke werden als Bedingung akzeptiert. Die Funktionsdefinition lautet jetzt:

```
fu = function(x) 0.5*I(x>0&x<1)+(0.75-0.25*x)*I(x>=1&x<3)
```

`fu` akzeptiert Vektoren als Argumente und die `curve()`-Funktion kann zur grafischen Darstellung verwendet werden.

Optionale Parameter

Selbstdefinierte Funktionen können auch optionale Argumente besitzen.

```
Indi = function(x,a=0,b=1) ifelse((a<x)&(x<=b),1,0)
```

Wir definieren eine eigene Indikatorfunktion, die nur dann 1 wird, wenn x größer als a und kleiner oder gleich b ist, also $I_{(a,b]}(x)$. Die Standardwerte für `a` bzw. `b` sind dabei 0 bzw. 1. Die Zuweisung erfolgt wie im Beispiel in der Form `Parameter=Vorgabewert`.

```
> Indi(0.5)
[1] 1
> Indi(-0.5)
[1] 0
> Indi(1.5,0,2)
[1] 1
```

`Indi` kann mit 1, 2 oder 3 Argumenten verwendet werden. Bei der Verwendung mit nur einem Argument werden bspw. a und b auf ihre Standardwerte gesetzt. Gemäß der Definition sind im letzten Beispiel $x = 1.5$, $a = 0$ und $b = 2$. Anders könnte man auch schreiben:

```
> Indi(b=2,a=0,x=1.5)
[1] 1
```

Verwendet man die Bezeicher der Argumente, dann sind diese beliebig vertauschbar. Das Argument einer Funktion kann ein beliebiges Objekt sein, auch erneut eine Funktion.

```
verwende = function(x,fu=sin) fu(x)
```

`fu` ist hier ein Bezeichner für eine Funktion. Der Standardwert ist die Sinusfunktion `sin`. Schauen wir uns ein paar einfache Anwendungen an.

```
> verwende(1)
[1] 0.841471
> verwende(1,cos)
[1] 0.5403023
```

Es werden sin(1) und cos(1) berechnet. Im ersten Beispiel wurde dabei auf den Standardwert `fu=sin` zurückgegriffen.

Als Funktionen können auch selbst definierte Funktionen verwendet werden.

```
> verwende(1.5,Indi)
[1] 0
> verwende(1.5,fu)
[1] 0.375
> verwende(1.5,verwende)
[1] 0.997495
```

Dabei wurden für `Indi` die Standardwerte $a = 0$ und $b = 1$ verwendet und für `verwende` der Standardwert `fu=sin`.

6 Zusammenfassung

6.1 Einige Grundlagen

Funktion	Anwendung
`boxplot()`	Boxplot
`exp()`, `log()`	Expoential-, Logarithmusfunktion
`help()`	Aufruf der *R*-Hilfe
`help.search()`	Aufruf der *R*-Hilfe
`install.packages()`	Installation eines Pakets
`library()`	Laden eines Pakets
`ls()`	Ausgabe der aktuell vorhandenen *R*-Objekte
`mean()`	arithmetisches Mittel
`rm()`	Löschen von *R*-Objekten
`sin()`, `cos()`	Sinus-, Cosinusfunktion
`source()`	Ausführen einer *R*-Skriptdatei
`sum()`	Summe

6.2 Wichtige Datenstrukturen und deren Arithmetik

Funktion	Anwendung
`c()`	Erstellen eines Vektors
`cbind()`	Verknüpfung eines Spaltenvektors mit einer Matrix
`data.frame()`	Erstellen eines *Data Frame*
`dim()`	Dimension einer Matrix
`length()`	Länge eines Vektors oder einer Liste
`load()`	Importieren von *R*-Objekten
`hist()`	Histogramm
`list()`	Erstellen eines `list`-Objektes
`matrix()`	Erstellen einer Matrix
`rbind()`	Verknüpfung eines Zeilenvektors mit einer Matrix
`read.csv()`	Einlesen von *Data Frames* aus *CSV*-Dateien
`read.table()`	Einlesen von *Data Frames* aus Textdateien
`read.xlsx()`	Einlesen von *Excel*-Dateien, Paket `xlsx`
`save()`	Exportieren von *R*-Objekten
`subset()`	Auswählen eines Teildatensatz aus *Data Frame*
`summary()`	Zusammenfassung eines *R*-Objekts
`write.table()`	Schreiben von *Data Frames* in Textdateien
`write.xlsx()`	Schreiben von *Excel*-Dateien, Paket `xlsx`

6.3 Deskriptive Analysemethoden

Funktion	Anwendung
`abline()`	Zeichnen einer Gerade
`assoc()`	Assoziationsplot, Paket `vcd`
`axis()`	Zeichnen einer Achse
`barplot()`	Säulen- und Balkendiagramme
`boxplot()`	Boxplot
`chisq.test()`	Bestimmung des χ^2-Koeffizienten
`cor()`	Korrelationskoeffizient
`cov()`	(korrigierte) Stichprobenkovarianz
`curve()`	Zeichnen von Funktionen
`hist()`	Histogramm
`lm()`	lineare Regression
`mad()`	Median Absolute Deviation
`mean()`	arithmetisches Mittel
`median()`	Median
`mosaic()`	Mosaikplot, Paket `vcd`
`pairs()`	Streudiagramm-Matrizen
`par()`	globale Grafikeinstellungen
`pie()`	Kreisdiagramm
`plot()`	Streudiagramme
`points()`	Punkte einer Grafik hinzufügen
`prop.table()`	Erstellen einer Tabelle relativer Häufigkeiten
`quantile()`	Quantilberechnung
`rank()`	Ermittlung der Ränge
`rq()`	Regression nach der LAD-Methode, Paket `quantreg`
`sd()`	(korrigierte) Stichprobenstandardabweichung
`segments()`	eine Strecke einer Grafik hinzufügen
`sort()`	Sortieren eines Vektors
`stem()`	Stammblattdiagramme
`table()`	Erstellen einer Häufigkeitstabelle
`text()`	Text einer Grafik hinzufügen
`var()`	(korrigierte) Stichprobenvarianz

6.4 Induktive Analysemethoden

Funktion	Anwendung
`as.factor(x)`	Generierung einer Faktorvariablen
`binom.test()`	exakter Binomialtest auf Anteilswerte
`cor.test()`	Korrelationstest
`dbinom(x,)`, `pbinom(x,)`	Wahrscheinlichkeits- und Verteilungsfunktion einer Binomialverteilung
`dnorm(x,)`, `pnorm(x,)`	Dichte und Verteilungsfunktion einer Normalverteilung
`chisq.test()`	χ^2-Anpassungs-/Unabhängigkeitstest
`pairs()`	paarweise Darstellung von Streudiagrammen
`prop.test()`	approximativer Test auf Anteilswerte oder Anteilswertdifferenzen
`pt()`, `qt()`	Verteilungsfunktion und Quantile der t-Verteilung
`qnorm(x,)`, `rnorm(x,)`	Quantile und Zufallszahlen zu einer Normalverteilung
`set.seed()`	Startwert für Pseudozufallszahlen
`t.test()`	t-Test für Erwartungswert und Erwartungswertdifferenzen
`lm(y~x)`	lineare Regression
`summary(erg)`	Zusammenfassung der linearen Regression
`confint(erg,)`	Konfidenzintervalle
`vcov(erg)`	geschätzte Kovarianzmatrix der Koeffizienten
`vcovHC(erg,)`	geschätzte Kovarianzmatrix der Koeffizienten im bedingt hetereoskedastischen Modell

6.5 Ergänzende und weiterführende Themen

Funktion	Anwendung
`colors()`	Gibt die vordefinierten Farbbezeichner aus
`expression()`	Verwendung mathematischer Ausdrücke in Grafiken
`for(){ }`	for-Schleifen
`function(){ }`	Definition von Funktionen
`palette()`	Farbpalette der Standardfarben
`pdf(),png()`	Speichern von Grafiken

Literatur

[1] Albert, J. (2009): *Bayesian computation with R.* Dordrecht u.a.: Springer.

[2] Cowles, M.K. (2013): *Applied Bayesian Statistics With R and OpenBUGS Examples.* New York: Springer.

[3] Dalgaard, P. (2002): *Introductory Statistics with R.* New York: Springer.

[4] Dolić, D. (2004): *Statistik mit R.* München: Oldenbourg.

[5] Efron, B., Tibshirani, R.G. (1993): *An Introduction to the Bootstrap.* New York: Chapman and Hall.

[6] Everitt, B., Hothorn, T. (2011): *An Introduction to Applied Multivariate Analysis with R.* New York: Springer.

[7] Farnsworth, G.V. (2006): *Econometrics in R.* Internetpublikation.

[8] Fox, J. (2008): *Applied Regression and generalized linear models.* Los Angeles u.a.: Sage.

[9] Grolemund, G. (2014): *Hands-On Programming with R: Write Your Own Functions and Simulations.* Sabastopol: O'Reilly Media.

[10] Ligges, U. (2008): *Programmieren mit R.* 3. Auflage. Berlin, Heidelberg: Springer.

[11] Luhmann, M. (2015): *R für Einsteiger: Einführung in die Statistiksoftware für die Sozialwissenschaften.* 4. Auflage. Weinheim, Basel: Beltz.

[12] Satterthwaite, F.E. (1941): *Synthesis of variance.* Psychometrika 6, S.309–316.

[13] Shao, Y., Tu, D. (1995): *The Jackknife and Bootstrap.* New York: Springer.

[14] Stocker, T.C., Steinke, I. (2017): *Statistik. Grundlagen und Methodik.* Berlin, Boston: DeGruyter Oldenbourg.

[15] Stocker, T.C., Steinke, I. (2017): *Statistik. Übungsbuch.* Berlin, Boston: DeGruyter Oldenbourg.

[16] Toomey, D. (2014): *R for Data Science.* Birmingham: Packt Publishing.

[17] Welch, B.L. (1947): *The Generalization of Student's Problem When Several Different Population Variances Are Involved.* Biometrika 34, S.28–35.

[18] White, H. (1980): *A Heteroskedasticity-Consistent Covariance Matrix Estimator and a Direct Test for Heteroskedasticity.* Econometrica 45, S.817-838.

[19] Wickham, H. (2015): *R Packages.* Sebastopol: O'Reilly Media.

[20] Wickham, H., Grolemund, G. (2016): *R for Data Science.* O'Reilly Media.

[21] Witting, H., Müller-Funk, U. (1995): *Mathematische Statistik II.* Stuttgart: Teubner.

[22] Wollschläger, D. (2016): *R Kompakt: Der schnelle Einstieg in die Datenanalyse.* 2. Auflage. Berlin, Heidelberg: Springer.

Index