Statistik mit R

Release 1.0

Jens Pönisch

10.02.2022

Inhalt

Voraussetzungen

Grundkenntnisse der Programmierung (z. B. aus WI-Praktikum 1).

Grundkenntnisse der linearen Algebra (Vektoren, Matrizen) und der Statistik sind wünschenswert.

KAPITEL 1

Literatur und Online-Quellen

1.1 Monographien

Eine gute Erläuterung statistischer Techniken und ihrer Umsetzung in R bietet [hatz14]. Dieses Buch bildet auch die Hauptquelle dieses Scripts. Auch [braun07] ist gute Einführung zur Nutzung von *R*, bietet aber nur wenig zu statistischen Auswertungen.

Recht preiswert ist [rrzn11], das im *URZ* erhältlich ist.

Wer über umfangreiche Statistikkenntnisse verfügt, findet in [adler10] zahlreiche Umsetzungsmöglichkeiten in *R*.

Ein wichtiges Standardwerk zu statistischen Methoden stellt [sachs15] (bzw. eine neuere Auflage) dar, dessen Beispiele in *R* implementiert sind.

Speziell mit verbesserten Diagrammen setzt sich [wic09] auseinander.

1.2 Online-Quellen

Mit der Installation von [miniconda] ist eine eigenständige Installation von *R* ([r-home] und [rstudio]) nicht erforderlich, diese sind im *Anaconda*-Paketsystem enthalten.

1.3 Statistische Daten

Als Quellen für statistische Daten Deutschlands dienen u. a. [StatJahrbuch] und [dtl_in_zahlen].

Der Blog [unstatistik] analysiert fehlerhafte oder falsche Statistiken, die in der Presse veröffentlicht werden und ist hilfreich, um typische Fehlinterpretationen zu erkennen. Leider bietet er meist keine konkreten Daten zum Nachvollziehen der Rechnungen.

Einen allgemeinen, nichtmathematischen Einstieg in die Statistik bietet C. Hesses Buch [hesse14].

KAPITEL 2

Installation

- *R und RStudio*
- *Jupyter Notebook*
- *Installation und Test (Python-Grundpaket)*
- *Arbeit mit Jupyter Notebook (R)*

2.1 R und RStudio

Wenn wir *R* nur über die Kommandozeile und mit einer klassischen *IDE* verwenden wollen, kann eine eigenständige Installation auf dem System sinnvoll sein.

R wird über die *URL* [r-home] bereitgestellt. Für die meisten Linux-Distributionen existieren schon fertige Pakete (Debian/Ubuntu: `r-base`). Die *IDE* Rstudio kann von [rstudio] bezogen werden. Sinnvoll für Webanwendungen ist sicher die Konvertierung von und nach *JSON*, für die in Debian/Ubuntu das Paket `r-cran-json` zu installieren ist.

Ein Installationstest erfolgt unter Linux durch Aufruf von

```
R
```

bzw.

```
rstudio &
```

Unter Windows werden die Programme wie gewohnt aufgerufen.

Zum Beenden benutzen wir den Funktionsruf

```
q()
```

und wählen anschließend aus, ob wir die Arbeitsumgebung speichern wollen. Soll die Rückfrage unterdrückt werden, verwenden wir

```
q("no")
```

```
# bzw.
q("yes")
```

2.2 Jupyter Notebook

Für das Praktikum verwenden wir die webbasierte Umgebung *Jupyter Notebook*. *Jupyter Notebook* ist eine webbasierte Entwicklungsumgebung, die häufig für die Datenanalyse eingesetzt wird. Sie unterstützt u. a. die drei für diesen Einsatzzweck wichtigen Programmiersprachen *Julia*, *Python* und *R*, aus denen der Name *Jupyter* zusammengesetzt ist.

Programme werden in Form von sogenannten *Notebooks* erstellt, die Programmcode, die Ergebnisse der Berechnung und Dokumentation vereinigen. Diese können problemlos in Programme der jeweiligen Sprache und wichtiger noch als Textdokumente exportiert werden, die als Protokoll der Datenanalyse dienen können.

2.3 Installation und Test (Python-Grundpaket)

Jupyter Notebook basiert auf Python und benötigt eine Vielzahl von Paketen in der passenden Version. Um die Installation zu vereinfachen, bietet sich die speziell für Datenanalysen zusammengestellte Distribution *Anaconda 3* an, die für mehrere Betriebssysteme eine passende virtuelle Umgebung bereitstellt.

Wir können entweder die komplette *Anaconda*-Umgebung installieren, sinnvoller ist es jedoch, mit dem Basispaket *Miniconda* zu starten und die benötigten Pakete nachzuinstallieren, da hier mit weniger Problemen aufgrund verschiedener Paketversionen zu rechnen ist.

Wir laden den Installer von der *Anaconda*-Webseite [miniconda] für das passende Betriebssystem.

Anschließend kann der Installer (Linux) gestartet werden:

```
sh {Installer}.sh
```

Unter Windows wird einfach der Installer ausgeführt.

Wir müssen die Lizenzbestimmung bestätigen und das Installationsverzeichnis wählen. Der Vorgabewert *Homeverzeichnis*`/miniconda` kann beibehalten werden, günstiger für unser Praktikum ist jedoch das Installationsverzeichnis `~/wip3/miniconda3` (Das Symbol `~` steht für das Homeverzeichnis). Am Ende der Installation werden wir gefragt, ob wir die Umgebung sofort initialisieren wollen, wir wählen zunächst `n` und beenden den Installer.

Unter Windows kann der Installationspfad beibehalten werden. Im Startmenü entsteht ein Eintrag *Anaconda 3* und unter diesem ein Eintrag *Anaconda 3 Prompt*. Wir starten diesen und erhalten ein Konsolenfenster. Hier kontrollieren wir das Arbeitsverzeichnis. Wir können dieses ändern, wenn wir den Menüeintrag mit der rechten Maustaste anklicken und *Mehr→Speicherort öffnen* auswählen. Ein Rechtsklick auf die entsprechende Datei öffnet die Eigenschaften, in denen wir das Arbeitsverzeichnis setzen.

Nun kann die *Anaconda*-Umgebung getestet werden. Wir aktivieren dazu die virtuelle Umgebung:

```
. ~/wip3/miniconda3/bin/activate
# oder
source ~/wip3/miniconda3/bin/activate
```

(Der Befehl beginnt mit einem Punkt, gefolgt von einem Leerzeichen). Der Prompt ändert sich. Unter Windows ist diese Umgebung mit dem Start der *Anaconde*-Konsole automatisch aktiviert.

Wir aktualisieren zunächst die installierte *Anaconda*-Umgebung:

```
conda update --all
```

und installieren anschließend die *Jupyter*-Pakete:

```
conda install jupyter
```

```
cd wip3
jupyter notebook
```

Es öffnet sich ein Browserfenster mit der Übersicht aller Dateien des Verzeichnisses. Über den Schalter `New` erstellen wir ein neues Notebook und wählen in der Auswahlliste als Sprache `Python 3` aus.

Es erscheint ein neues Notebook. Wir wechseln durch Anklicken der Eingabezelle oder Drücken von `Enter` in den Eingabemodus und geben nun einen Python-Befehl zum Testen ein:

```
print('Jupyter')
```

Nach dem Anklicken des `Run`-Schalters oder Drücken von `StrgEnter` wird der Code der Zelle ausgeführt und das Ergebnis angezeigt.

Nach diesem kurzen Test verlassen wir das Notebook über das *File*-Menü. In der Konsole beenden wir den Server durch die Eingabe von `StrgC` und bestätigen den Abbruch mit `y`.

Nach Beeden der Arbeit in der virtuellen Umgebung kann diese wieder beendet werden:

Nun muss die Unterstützung für *R* installiert werden. Da leider in den Standardpaketen derzeit die Fontunterstützung in Graphiken nicht korrekt funktioniert, wählen wir die Version aus `conda-forge`. Wir suchen zunächst die aktuell verfügbare Version:

```
conda search -c conda-forge r-base
```

Diese wollen wir nun installieren:

```
conda install r-base=4.1.1
conda install r-essentials
```

Ohne Angabe der Version wird eine ältere *R*-Version genutzt.

Nun sollte ein Kernel für *R* exisistieren, wir können das mit folgendem Befehl prüfen:

```
jupyter kernelspec list
```

In der Liste muss ein Kernel mit dem Namen `ir` auftauchen.

Nun können wir erneut den Notebook-Server starten und sollten nun in der Auswahl unter *New* den Eintrag *R* finden. Wir erzeugen nun ein solches Notebook und führen in der ersten Zelle den Befehl

```
R.version
```

aus. Nun testen wir noch die Graphik. Wir erzeugen mit `B` oder über den Schalter der Werkzeugleiste eine neue Zelle und geben den Befehl

```
curve(sin(x), main='Umlaute äöüß')
```

ein. Wir sollten sowohl den Graphen als auch korrekt beschriftete Achsen sehen.

Wenn der Titel und die Achsenbeschriftungen kryptische Zeichen enthalten, müssen wir das Notebook speichern, den Server beenden und die Pakete `r-cairo` und eventuell `fonts-anaconda` nachinstallieren.

```
conda install r-cairo fonts-anaconda
```

Wir testen erneut die Diagrammanzeige, die nun korrekt sein sollte. Diese Anzeige sollten wir noch für den Diagrammexport testen. Wir schreiben in eine Zelle

```
png('graphtest.png')
curve(sin(x), main='Umlaute äöüß')
dev.off()
```

und führen die Zelle aus. Wenn wir in das Fenster der Dateiübersicht wechseln, sollte eine Datei `graphtest.png` entstanden sein, die wir nun kontrollieren.

Wenn wir unsere Arbeit komplett beenden wollen, können wir nach Beenden des Jupyter-Servers auch die *Miniconda*-Umgebung wieder deaktivieren:

```
conda deactivate
```

2.4 Arbeit mit Jupyter Notebook (R)

Nach dem Start eines neuen Notebooks sehen wir zunächst eine Zelle, in der wir einen Befehl eingeben können. Wir prüfen zunächst die R-Version durch Eingabe von

```
R.version
```

Mit der Eingabe von `StrgEnter` wird die Zelle ausgeführt.

Weitere Zellen werden über den Schalter `+` oder die Taste `B` hinter der aktuellen Zelle eingefügt, mit `A` kann man eine Zelle vor der aktuellen Zelle einfügen.

Eine neue Zelle ist zunächst für die Code-Eingabe vorgesehen. Mit `M` schalten wir sie in eine Dokumentationszelle um, in der wir *Markdown*-Code schreiben können. Auch dieser wird mit `StrgEnter` oder dem Schalter `Run` übersetzt. Ein Ändern einer Zelle in den Programm-(Code-)Modus erfolgt mit `Y`.

Wollen wir eine Zelle ausführen und anschließend zur nächsten Zelle wechseln, verwenden wir `UmschaltEnter`.

Sollte sich die Ausführung festfahren, können wir sie durch Neustart des Kernels über den Schalter abbrechen. Ebenso können wir ein geladenes Notebook komplett neu berechnen, indem wir den Schalter `>>` anklicken.

Neue Notebooks erhalten automatisch den Name `Untitled`. Um den Überblick zu behalten, sollten die Notebooks deshalb über das File-Menü einen sinnvollen Namen erhalten.

Unser Notebook wird etwa alle 2 Minuten automatisch gespeichert. Ein manuelles Speichern geschieht mit `StrgS` oder einfach `S`. Insbesondere vorm Verlassen des Notebooks sollte nochmals gespeichert werden, um auch die letzten Änderungen tatsächlich zu sichern.

Im File-Menü finden wir ebenfalls die Export-Funktionen unter *Download as*. Interessant sind zum einen der Export als R-Script, um den Code direkt mit `Rscript` oder in *RStudio* ausführen zu können, und zum anderen in einem Dokumentenformat wie HTML, Markdown oder PDF für die Dokumentation. Für die Erzeugung von PDF ist eine vorhandene LaTeX-Installation (incl. `xelatex`) erforderlich.

KAPITEL 3

Einführung und Grundlagen

3.1 Was ist R?

R ist eine Programmiersprache für statistische Anwendungen, die als freie Alternative zur Sprache *S* seit 1993 entwickelt wird.

R ist im Gegensatz zu *SPSS* kommandozeilenorientiert. Der Nutzer gibt Befehle ein, die von der Anwendung ausgeführt werden und deren Ergebnis wiederum auf der Konsole angezeigt wird. Die Arbeit ist dadurch am Anfang vielleicht etwas ungewohnt, allerdings erleichtet dieses Vorgehen eine automatische Datenverarbeitung ungemein.

Für die interaktive Arbeit bietet sich die Nutzung einer *IDE* wie *RStudio* oder *Jupyter Notebook* bzw. *Jupyter Lab* an. Wir werden von dieser zweiten Möglichkeit Gebrauch machen.

3.2 Hilfe

R enthält eine eingebaute Hilfe zu allen verfügbaren Objekten. Diese kann wie folgt aufgerufen werden:

```
?funktionsName
```

help(funktionsName)

Ist der Objektname unbekannt, kann über ähnliche Begriffe nach dem passenden Objekt gesucht werden:

help.search("begriff")

Als Ergänzung zur Funktionsbeschreibung ist für viele Objekte ein Beispiel hinterlegt, das die Nutzung demonstriert.

example(funktionsName)

Beispiel: Hilfe zum Arbeitsverzeichnis.

```
> ?getwd

getwd                   package:base                   R Documentation

Get or Set Working Directory

Description:

     'getwd' returns an absolute filepath representing the current
     working directory of the R process; 'setwd(dir)' is used to set
     the working directory to 'dir'.

Usage:

     getwd()
     setwd(dir)

Arguments:

     dir: A character string: tilde expansion will be done.

Details:

     See files for how file paths with marked encodings are
     interpreted.

Value:

     'getwd' returns a character string or 'NULL' if the working
     directory is not available.  On Windows the path returned will use
     '/' as the path separator and be encoded in UTF-8.  The path will
     not have a trailing '/' unless it is the root directory (of a
     drive or share on Windows).

     'setwd' returns the current directory before the change, invisibly
     and with the same conventions as 'getwd'.  It will give an error
     if it does not succeed (including if it is not implemented).

Note:

     Note that the return value is said to be *an* absolute filepath:
     there can be more than one representation of the path to a
     directory and on some OSes the value returned can differ after
     changing directories and changing back to the same directory (for
     example if symbolic links have been traversed).

See Also:
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
     'list.files' for the _contents_ of a directory.

     'normalizePath' for a 'canonical' path name.

Examples:

     (WD <- getwd())
     if (!is.null(WD)) setwd(WD)


> help.search('working directory')

Help files with alias or concept or title matching 'working directory'
using fuzzy matching:


base::getwd             Get or Set Working Directory
devtools::wd            Set working directory.
ps::ps_cwd              Process current working directory as an
                        absolute path.
usethis::proj_sitrep    Report working directory and usethis/RStudio
                        project
withr::with_dir         Working directory
xfun::in_dir            Evaluate an expression under a specified
                        working directory


Type '?PKG::FOO' to inspect entries 'PKG::FOO', or 'TYPE?PKG::FOO' for
entries like 'PKG::FOO-TYPE'.




> example('getwd')


getwd> (WD <- getwd())
[1] "/data/home/jens/lehre/R"

getwd> if (!is.null(WD)) setwd(WD)
```

3.3 Einfache Rechnungen

Zahlen werden in *R* als Dezimalzahlen betrachtet, einen speziellen *Integer*-Typ gibt es für arithmetische Rechnungen nicht. Für diese Zahlen stehen die üblichen Grundrechenarten `+ - * /` zur Verfügung, Potenzieren ist mit dem Operator `^` möglich. Die üblichen mathematischen Funktionen stehen ebenfalls zur Verfügung, eine Übersicht kann mit

help(funktionsgruppe)

angezeigt werden. Funktionsgruppen:

abs Betrag und Wurzel.

log Logarithmus- und Exponentialfunktionen.

`sin` Winkelfunktionen.

`Special` Spezielle, meist statistische Funktionen (Fakultät, Beta- und Gamma-Funktion, …)

Beispiele:

```
3/4
[1] 0.75
2*1.5
[1] 3
2^-0.5
[1] 0.7071068
sqrt(2) # Wurzelfunktion ist verfügbar
[1] 1.414214
```

Für die ganzzahlige Division und den Divisionsrest werden die Operatoren `%/%` und `%%` verwendet. Im ersten Fall werden die Operanden auf ganze Zahlen gerundet, der Divisionsrest kann auch für Dezimalzahlen ermittelt werden:

```
3 %/% 2
[1] 1
5 %% 3
[1] 2
4.7 %% 1.5
[1] 0.2
```

Eine Klammersetzung erfolgt wie üblich.

Notebook

Download: `basics_1.ipynb`.

3.4 Variablen und Namen

R speichert alle Daten als Objekte, wobei Objekte über Namen (Identifier) angesprochen werden. Die Namensbildung ist dabei sehr flexibel, man sollte sich jedoch auf (*ASCII*-)Buchstaben, Ziffern, den Unterstrich und den Punkt beschränken. Groß- und Kleinschreibung wird unterschieden. Der Punkt kann dabei zur sinnvollen Gruppierung verwandter Objekte verwendet werden (z. B. `person.mean` und `person.max`).

Ergebnisse von Rechnungen und eingelesene Daten können solchen Objekten zugewiesen werden. Die Zuweisung erfolgt mit den Operatoren `<-` bzw. `->`. Eine Zuweisung mit `=` ist möglich, sollte aber vermieden werden.

```
# Zinsertrag nach 10 bzw. 30 Jahren bei 1000 EUR Einlage und 3% Zinssatz
b <- 1000
p <- 0.03
b*(1+p)^10 -> zinsen.10 # äquivalent zu: zinsen.10 <- berechnung
b*(1+p)^30 -> zinsen.30
zinsen.10
[1] 1343.916
zinsen.30
[1] 2427.262
```

3.5 Objektübersicht

Mit den Funktionen `ls()` bzw. `objects()` wird eine Übersicht aller bisher erzeugten und geladenen Objekte angezeigt. *RStudio* zeigt diese Übersicht automatisch in einem eigenen Fenster an.

Sollen Objekte gelöscht werden, geschieht dies mit der Funktion `rm()`:

rm(objekt1*[, ...]*)

Vor dem Ausführen von Scripten und der Bearbeitung eines neuen Problems ist es sinnvoll, alle bisherigen Objekte zu löschen:

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

3.6 Scripte und Kommentare, Arbeitsverzeichnis

Für komplexere Probleme ist es sinnvoll, Befehlsfolgen als *Script* abzuspeichern und diese automatisch auszuführen. Scriptdateien können mit jedem beliebigen Editor erstellt werden und erhalten meist die Endung `.R`. RStudio beinhaltet einen solchen Editor, der mit der Funktion *File* → *New* bzw. *File* → *Open* geöffnet werden kann. Mit dem Schalter `Run` kann die aktuelle Zeile oder der markierte Bereich direkt ausgeführt werden. Alternativ kann der Schalter `Source on Save` gesetzt werden, dann wird das Script bei jedem Speichern automatisch ausgeführt.

Es ist sinnvoll, Scripte zu kommentieren. Dies geschieht mit dem Zeichen `#`:

Kommentar

Das Laden eines Scripts im Kommandofenster geschieht mit der Funktion

source("scriptDatei")

Die Datei wird im Arbeitsverzeichnis gesucht, das mit dem Kommando `getwd()` ermittelt werden kann. Das Setzen eines neuen Arbeitsverzeichnisses geschieht mit

setwd("neuesVerzeichnis")

Das neue Verzeichnis kann absolut oder relativ zum bisherigen Arbeitsverzeichnis angegeben werden.

3.7 Ausgaben

Normalerweise erzeugt bei der *interactiven Nutzung* jeder *R*-Ausdruck, der einen Wert liefert, auch eine Ausgabe, allerdings erscheint diese im internen Format. Beispiel:

```
"Hallo\n"
[1] "Hallo\n"
1:3
[1] 1 2 3
```

Führen wir Scripte aus, wird nur das Ergebnis der letzten Anweisung ausgegeben. Die Ausgabe von Zwischenergebnissen können wir mit der Funktion `print()` veranlassen:

print(ausdruck)

Es kann nur hier nur ein Ausdruck angegeben werden!

Wollen wir die Ausgabe schöner gestalten, können wir die Funktion `cat()` benutzen, die auch Escape-Sequenzen in Strings auflöst:

cat(wert1*[, ...]*)

Beispiele:

```
cat("Hallo, \"Egon\"!\n")
Hallo, "Egon"!
cat("Sequenz: ", 1:3, "\n")
Sequenz:  1 2 3
```

Diese Ausgaben können auch in eine Datei geschrieben werden:

```
f <- file("dateiname)
cat(..., file=f)
close(f)
```

3.8 Weitere einfache Datentypen und spezielle Werte

Neben Zahlen unterstützt *R* auch Strings und boolsche Werte. Strings werden in einfache oder doppelte Hochkommas eingeschlossen.

Die logischen Werte werden mittels `FALSE` und `TRUE` bzw. kurz als `F` und `T` spezifiziert.

Um fehlende Daten (z. B. in einer Stichprobe) zu kennzeichnen, wird der spezielle Wert `NA` (not available) verwendet, der sowohl zu Zahlen als auch Strings kompatibel ist.

3.9 Eingaben

Um in *R*-Scripte Daten interaktiv einzulesen, verwenden wir die Funktion `readline()`:

```
stringVar <- readline("Text")
```

Oft benötigen wir den Wert jedoch als Zahl, die Umwandlung erfolgt mit der Funktion `as.numeric()`:

```
zahlVar <- as.numeric(string)
```

Für das Einlesen mehrerer Zahlen bietet es sich an, eine Funktion zu schreiben:

```
readNum <- function(text) {
  x <- readline(text)
  val <- as.numeric(x)
  return(val)
}

# Verwendung:
a <- readNum("Zahl eingeben: ")
```

Für die Umwandlung von Strings in ganze Zahlen existiert eine vergleichbare Funktion `as.integer()`. Ganze Zahlen werden zwar nicht für Rechnungen, aber zum Indicieren von Elementen in Datenstrukturen benötigt (Feldindex, Faktor).

KAPITEL 4

Statistische Grundlagen

4.1 Aufgaben

Definitionen aus [sachs15]:

> Statistik ist die Kunst, Daten zu gewinnen, darzustellen, zu analysieren und zu interpretieren, um zu neuem Wissen zu gelangen.
>
> Statistik ist die Lehre von der Variabilität / Streuung in den Beobachtungen.

Wie die beiden Definitionen andeuten, hat die Statistik zwei wesentliche Aufgaben.

Die *Beschreibende Statistik* versucht eine große Anzahl von Beobachtungen so zu Tabellen, Graphiken und Kennzahlen zu verdichten, dass es einfach möglich ist, grundlegende Aussagen über die Daten zu machen: Entwicklung und Verteilung von Einkommen über die Zeit, Geschlechter, Berufsgruppen, Industriestruktur in verschiedenen Regionen, Wahlergebnisse nach Regionen, …

Die *Beurteilende Statistik* versucht dagegen, anhand der Beobachtung verschiedener Daten Zusammenhänge zwischen diesen zu erkennen: Ist ein Medikament oder ein Schädlingsbekämpfungsmittel wirklich wirksam, sind Frauen beim Einkommen systematisch benachteiligt, …

Wir werden uns hier im wesentlichen mit den Verfahren der beschreibenden Statistik auseinandersetzen und nur sehr einfache Verfahren der beurteilenden Statistik behandeln.

4.2 Skalen (Datentypen)

Beobachtete Daten und Messwerte werden je nach ihren Eigenschaften verschiedenen *Skalenniveaus* (= Datentypen) zugeordnet. Diese Skalen sind wichtig für die korrekte Interpretation und Verarbeitung der Daten.

Metrische Skala Die Daten sind reelle Zahlen, zwischen ihnen lassen sich sinnvoll Differenzen (Abstände) definieren. Hier wird noch einmal zwischen *Intervallskalen* und *Verhältnisskalen* unterschieden. Bei ersteren ist die Null willkürlich festgelegt (z. B. Temperatur in °C, Himmelsrichtung in Grad, Wochentagnummer), bei letzteren ist die Null natürlich gegeben, es lassen sich Verhältnisse zwischen den Daten (doppelt so groß …) angeben (z. B. Gewicht, Alter).

Ordinalskala Daten, die eine definierte Reihenfolge haben (schlecht, mittel, gut, Bewertung mit 1…5 Sterne).

Nominalskala Zugehörigkeit zu einer bestimmten Gruppe ohne sinnvolle Rangordnung bzw. Reihenfolge (rot, grün, blau; Medikament.1, Medikament.2).

Je nach Skala der Daten sind unterschiedliche Operationen zwischen den Daten möglich. Nominalskalen erlauben lediglich den Test auf Gleichheit bzw. Ungleichheit zweier Beobachtungen. Ordinalskalen definieren zusätzlich eine Ordnungsrelation (größer als, besser als) zwischen Beobachtungen, die Abstände zwischen den Werten sind jedoch nicht unbedingt gleich (z. B. bei Bewertung mit 1 bis 5 Sternen). Bei metrischen Skalen kann nicht nur die Reihenfolge der Beobachtung, sondern auch die Differenz der Werte analysiert werden. Verhältnisskalen erlauben zusätzlich auch die Betrachtung von Quotienten.

4.3 Kenngrößen der beschreibenden Statistik

Neben dem Darstellen der Daten in geeigneten Diagrammen werden folgende Möglichkeiten der Datenverdichtung verwendet:

Histogramm (Kontigenztafel, Kreuztabelle) Geeignet für alle Skalen, wichtiges Werkzeug für Nominal- und Ordinalskalen. Es wird gezählt und dargestellt, wie häufig jeder Wert beobachtet wurde.

Für sehr viele oder unendlich viele (bei metrischen Skalen) mögliche Werte fasst man die beobachteten Werte zu *Klassen* zusammen und zählt die Werte je Klasse. Diese zweite Form wird in *R* als Histogramm bezeichnet, Kontigenztafeln werden mit der `table()`-Funktion erzeugt.

Mittelwert Nur für metrische Skalen. Mittel aller Beobachtungen (Summe aller beobachteten Werte, geteilt durch deren Anzahl). Wert allein ist nicht allzu aussagekräftig und kann zu völligen Fehlinterpretationen führen (Durchschnittseinkommen).

Wird auch bei Ordinalskalen häufig berechnet, besitzt dann aber noch weniger Aussagekraft.

Median Metrische Skalen, Alternative zum Mittelwert. Die Werte werden sortiert, als Median ist der mittlere Wert, so dass die Hälfte aller Werte kleiner, die andere Hälfte der Werte größer als der Median ist. Bei sehr ungleich verteilten Daten häufig aussagekräftiger.

Quartile und Quantile Metrische Skalen. Für Quartile wird der Wertebereich so in 4 disjunkte Intervalle zerlegt, so dass in jedes 25% der Beobachtungen fallen. Die beiden mittleren Intervalle enthalten also die Hälfte aller beobachteten Werte. Für Quantile werden statt 25% andere Werte (z. B. 5%) für die beiden Randintervalle verwendet.

Varianz und Streuung Metrische Skalen. Unter der Annahme, dass die beobachteten Werte einer bestimmten Wahrscheinlichkeitsverteilung unterliegen, versucht man, die Abweichung vom Mittelwert genauer durch einen Zahlenwert festzulegen. Aus der Theorie ergibt sich mit der *Varianz* ein quadratischer Wert. Die Wurzel aus diesem Wert ist die *Streuung* σ, die meist leichter zu interpretieren ist. Sind die Beobachtungen *normalverteilt*, so liegen etwa 60% der Beobachtungen im Bereich $\pm\sigma$ um den Mittelwert und sogar 99% der Daten im Bereich $\pm 3\sigma$ um diesen Wert.

Beispiel zur Aussagekraft von Mittelwert und Median: In einem Handwerksbetrieb verdient der Chef monatlich 6000 €, seine 3 Angestellten 1100, 1200 und 2000 €.

Lösung mit R:

```
# Erzeugen eines Datenvektors
gehalt <- c(6000, 1100, 1200, 2000)

# Ausgabe des Vektors
cat("Einzelne Einkommen:    ", gehalt, "\n")
# Mittelwertberechnung und -ausgabe
gehalt.mittel <- mean(gehalt)
cat("Mittleres Einkommen:   ", gehalt.mittel, "\n")
# Medianberechnung und -ausgabe
gehalt.median <- median(gehalt)
cat("Median des Einkommens: ", gehalt.median, "\n")
```

Ergebnis:

```
Einzelne Einkommen:     6000 1100 1200 2000
Mittleres Einkommen:    2575
Median des Einkommens:  1600
```

Die Berechnung weiterer Kenngrößen wird später behandelt.

4.4 Beurteilende Statistik

Um anhand der Beobachtungen Gesetzmäßigkeiten abzuleiten, ist es erforderlich, *Modelle* aufzustellen.

Eine erste wichtige Aufgabe ist das Prüfen von Fragestellungen. Dazu werden *Hypothesen* aufgestellt, wie z. B. ob die beobachteten Daten einen bestimmten Mittelwert haben, einer bestimmten Wahrscheinlichkeitsverteilung unterliegen oder dass zwischen verschiedenen beobachteten Größen ein Zusammenhang besteht. Ein statistischer Test gibt uns die dann die Antwort, ob die Hypothese zutrifft oder abgelehnt werden muss. Diese Aussage ist dabei im allgemeinen eine Wahrscheinlichkeitsaussage («trifft mit 95%er Wahrscheinlichkeit nicht zu»).

Statistische Modelle gehen einen Schritt weiter und versuchen, den Zusammenhang zwischen beobachteten unabhängigen und abhängigen Größen mathematisch zu beschreiben. Im einfachsten Fall suchen wir lineare Zusammenhänge zwischen diesen Größen. Eine erster Indikator ist der *Korrelationskoeffizient*, eine Zahl zwischen -1 und 1. Ist dieser Wert nahe 0, besteht zwischen den Größen kein erkennbarer Zusammenhang, ist er nahe -1 oder 1, dann deutet das auf einen starken Zusammenhang hin.

4.5 Stichproben

Grundlagen für das Erstellen und Überprüfen von vermuteten Zusammenhängen sind häufig *Stichproben*, da das Erheben aller verfügbaren Daten entweder zu aufwendig oder völlig unmöglich ist.

Dazu wird aus der Gesamtmenge der Daten eine Teilmenge ausgewählt, für die die Untersuchungen (z. B. eine Befragung) durchgeführt wird. Aus dem Ergebnis für die Teilmenge wird auf die Gesamtmenge geschlossen.

Um tatsächlich aussagekräftige Ergebnisse für die Gesamtmenge zu erhalten, muss die Stichprobe *repräsentativ* und ausreichend groß sein, sie muss eine gleiche Verteilung der Eigenschaften wie die Gesamtmenge haben. Meist versucht man dies durch eine *zufällige Auswahl* der Strichprobenelemente zu erreichen, was jedoch nicht immer möglich ist.

KAPITEL 5

Vektoren

In der Statistik arbeiten wir meist nicht mit Einzeldaten, sondern mit größeren Datenmengen. Die einfachste Struktur zu Verwaltung derartiger Daten ist der *Vektor*, der einem eindimensionalen *Feld* oder *Array* sonstiger Programmiersprachen entspricht.

5.1 Erzeugung und Zugriff

Vektoren sind geordnete Listen von Werten gleichen Typs. Sie werden mit der Funktion `c()` erzeugt:

```
vektorObjekt <- c(wert1, wert2, ...)
```

Im mathematischen Sinne wird ein Vektor als Spaltenvektor interpretiert.

Auf einzelne Elemente des Vektors kann wie auf Feldelemente mit dem Indexoperator `[]` zugegriffen werden:

```
element <- vektor[index]
```

Hinweis: Die Indizierung in *R* beginnt mit 1 und *nicht* mit 0 wie in C/C++.

Benötigt man ein Intervall natürlicher Zahlen, kann ein entsprechender Vektor einfach wie folgt erzeugt werden:

intervall <- startWert:endWert

Beispiel: Vektor der Zahlen von 1 bis 10.

```
interval.1.10 <- 1:10
```

Vertiefung

Komplexere Folgen lassen sich mit den Funktionen `seq()` und `seq.int()` erzeugen.

Mit Hilfe dieser Intervalle lassen sich Teilvektoren bilden:

teilVektor <- vektor[startIndex:endIndex]

Beispiel: Teilvektor vom 3. bis zum 5. Element.

```
v <- c(1, 1, 2, 3, 5, 8, 13)
vp <- v[3:5]
vp
[1] 2 3 5
```

Vertiefung

Statt eines Intervalls kann auch ein beliebiger Indexvektor angegeben werden, um ein Teilvektor zu erzeugen.

Wiederholte Werte können mit `rep()` erzeugt werden. Beispiel: Vektor aus 10 Nullen und 5 Einsen.

```
null.eins <- c(rep(0, 10), rep(1, 5))
null.eins
 [1] 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1
```

Jupyter-Notebook

Download: `vector_sel.ipynb`.

5.2 Vektoroperationen

Die Länge eines Vektors kann mit der Funktion `length()` ermittelt werden:

laenge <- **length(**vektor**)**

Verknüpft man einen Zahlenvektor und eine skalare Größe mit einer mathematischen Operation, wird diese Operation für jedes Element ausgeführt, es entsteht ein Vektor mit den berechneten Werten:

```
v <- c(1, 2, 3, 4, 5)
v+3
[1] 4 5 6 7 8
3+v
[1] 4 5 6 7 8
v*2.5
[1]  2.5  5.0  7.5 10.0 12.5
v/2
[1] 0.5 1.0 1.5 2.0 2.5
2/v
[1] 2.0000000 1.0000000 0.6666667 0.5000000 0.4000000
```

Ebenso können Funktionen auf alle Vektorelemente angewendet werden:

```
sqrt(v)
[1] 1.000000 1.414214 1.732051 2.000000 2.236068
```

Das Skalarprodukt zweier Vektoren können wir mit Hilfe der Matrixmultipikation `%*%` ermitteln. Im Sinne der linearen Algebra muss dazu der linke Vektor mittels `t()` in einen Zeilenvektor transponiert werden. *R* «denkt» aber mit und erlaubt die Operation auch zwischen zwei Spaltenvektoren:

```
w <- c(2, 1, 4, 3, 0)
t(v) %*% w
     [,1]
[1,]   28
v %*% w
     [,1]
[1,]   28
```

Das Ergebnis wird als 1×1-Matrix ausgegeben, was uns zunächst nicht weiter stören soll.

Mit der Funktion `sum()` können alle Elemente eines Vektors aufsummiert werden.

```
sum(v)
[1] 15
sum(w)
[1] 10
```

Damit können wir einfach einen Ausdruck konstruieren, der die euklidische Norm eines Vektors bestimmt ($< v, v >^{\frac{1}{2}}$):

```
sqrt(sum(v^2))
```

Benötigen wir diese Formel häufiger, können wir mit diesem Ausdruck einfach eine Funktion schreiben. Wir ergänzen dazu gleich eine zweite Funktion, die einen Vektor normiert (ihn also auf die Norm 1 skaliert):

```
norm_vec <- function(v) { sqrt(sum(v^2)) }
normalize_vec <- function(v) { v/norm_vec(v) }
```

Wir testen diese Funktionen:

```
norm_vec(v)
[1] 7.416198
normalize_vec(v)
[1] 0.1348400 0.2696799 0.4045199 0.5393599 0.6741999
nv <- normalize_vec(v)
sum(nv^2)
[1] 1
```

5.3 Einlesen und Speichern von Vektoren

Vektordaten können auch von der Konsole oder aus einer Datei eingelesen werden:

vektorVariable **= scan(***["*dateiname*"]***)**

Die Werte sind dabei durch Zeilenschaltungen oder Leerzeichen zu trennen. Fehlende Werte können durch `NA` eingefügt werden (kann umdefiniert werden). Enthält die Datei Kommentarzeilen, die z. B. mit dem Zeichen `#` beginnen, kann die Scananweisung um den Parameter

comment.char = "zeichen"

ergänzt werden. Das Dezimaltrennzeichen (Vorgabe: `.`) lässt sich mit dem Parameter `dec` umdefinieren.

Beispiel: Einlesen von «deutschen» Dezimalzahlen mit Kommentarzeilen.

```
values <- scan("zahlen.dat", comment.char="#", dec=",")
```

Datendatei: `zahlen.dat`.

Ganz analog lässt sich ein Vektor in einer Textdatei ablegen:

write(vektor**, file=**"fileName"**,** *[***ncolumns=**anzahl*]***)**

Ohne den Parameter `ncolumns` werden bei numerischen Daten 5 Werte in eine Zeile geschrieben.

5.4 Listenoperationen

Neben Operationen auf einzelne Elemente benötigen wir häufig Operationen über den gesamten Vektor. Wir verwenden dazu als Beispieldaten den mitgelieferten Vektor `rivers` von Flusslängen in den USA sowie den Vektor `aup-a1.dat` mit den Punktzahlen einer AuP-Prüfungsaufgabe, die NA-Werte enthält.

```
aup.a1 <- scan("aup-a1.dat")
```

Enthält der Vektor keine Zahlen, sondern z. B. Strings, dann kann mit dem Parameter `what` der Datentyp angegeben werden. Beispiel: Lesen der beobachteten Ampelfarbe zu zufälligen Zeitpunkten `ampel.dat`.

ampel <- scan("ampel.dat", what=character()) ampel

Ein Vektor kann sortiert werden:

sortierterVektor <- **sort(**vektor*[***, decreasing=T***]***)**

Der zweite Parameter erzeugt eine absteigende Sortierung.

Sortieren wir die Punktzahlen, stellen wir fest, dass die `NA`-Werte verschwunden sind. Durch einen weiteren Boolean-Parameter `na.last` können diese entweder vorn oder hinten einsortiert werden.

Soll die Reihenfolge der Elemente des Vektors umgedreht werden, verwenden wir die Funktion `rev()`:

reverseVektor <- **rev(**vektor**)**

Haben wir sehr lange Vektoren und wollen uns für Tests nur auf einige Werte beschränken, können wir die Funktionen `head()` bzw. `tail()` verwenden:

teilvektor <- **head(**vektor*[***,** länge*]***)**
teilvektor <- **tail(**vektor*[***,** länge*]***)**

Ohne Angabe der Länge werden die ersten bzw. letzten 6 Werte ausgewählt.

Viele statistische Vektoroperationen blenden `NA`-Werte automatisch aus. Wir können diese aber auch explizit aus dem Vektor entfernen, damit sie z. B. auch nicht mehr gezählt werden:

```
neuerVektor <- vektor[!is.na(vektor)]
```

Alternativ können wir alle `NA`-Werte auch durch 0 (oder einen anderen Wert) ersetzen:

```
vektor[is.na(vektor)] <- 0
```

Der Vektor behält dabei seine Länge, statistische Auswertungen liefern damit unterschiedliche Ergebnisse.

Generell kann als Vektorindex auch eine Bedingung angegeben werden. Es werden in den Ergebnisvektor nur die Werte übernommen, die diese Bedingung erfüllen.

Beispiel: Ergebnis «aufhübschen» indem wir alle Punktzahlen kleiner 2 weglassen:

```
aup.a1.gut <- aup.a1[aup.a1 > 1 & !is.na(aup.a1)]
```

5.5 Filter

Um aus Vektoren Teilvektoren zu bilden, können in den eckigen Klammern auch Ausdrücke angegeben werden (wie z. B. oben `is.na()`).

neuerVektor <- vektor**[**bedingung(vektor)**]**

Es entsteht ein neuer Vektor, der nur die Werte enthält, die die angegebene Bedingung erfüllen. Alternativ können mit der Funktion `which()` auch die Indices der Elemente ermittelt werden, die die Bedingung erfüllen:

indexVektor <- **which(**bedingung(vektor)**)**

Beispiel:

```
# Werte von 11 bis 20
v <- 11:20
# Alle Werte größer 15
v[v>15]
[1] 16 17 18 19 20
# Indices der Werte größer 15
idx <- which(v>15)
idx
[1]  6  7  8  9 10
# und die zugehörigen Werte
v[idx]
[1] 16 17 18 19 20
```

Für den Index des größten und kleinsten Elements eines Vektors gibt es die speziellen Funktionen `which.max()` bzw. `which.min()`.

Beispiel:

```
v <- c(7, 3, 8, 11, 17, 11, 9, 2)
idx <- which.min(v) # → 8
# alternativ mittels which
idx <- which(v == min(v)) # → 8
```

5.6 Faktoren für nominale und ordinale Merkmale

Zur Verarbeitung nominaler (diskrete Eigenschaften ohne vorgegebene Ordnung, z. B. Geschlecht, Farbe) und ordinale Merkmale (diskrete Eigenschaften mit natürlicher Ordnung, z. B. (gut, mittel, schlecht)) bietet sich der spezielle Datentyp `factor` an. Er ordnet einem String-Vektor intern Zahlen zu, was eine schnellere Verarbeitung erlaubt:

nominalVektor <- **factor(**stringVektor**)**

Ist eine Ordnung erforderlich, definiert man diese in einem zweiten Vektor und übergibt diesen als zweiten Parameter:

ordnung <- **c(**level1**,** level2**,** ...**)**
ordinalVektor <- **factor(**stringVektor**,** *[***level =** *]*ordnung**)**

Beispiele:

```
# nominal
gender <- c("m", "w", "w", "m", "m", "-", "m")
gender <- factor(gender)
# ordinal
score.order <- c("schlecht", "mittel", "gut")
score.values <- c("schlecht", "schlecht", "schlecht", "mittel", "mittel")
evaluation <- factor(score.values, score.order)
```

5.7 Verändern von Vektoren (Listenoperationen)

Will man Vektoren als *ADT* Liste, Vektor oder Stack einsetzen, muss man am Anfang oder Ende Werte einfügen und löschen können.

Einen leeren Vektor kann man mit der Funktion `vector()` erzeugen:

v <- **vector()**

Die Standardvariante zum Einfügen von Werten am Anfang oder am Ende ist die Funktion `append()`:

 # Vorn einfügen
v <- **c(**wert**,** v**)**
 # hinten anhängen
v <- **c(**v**,** wert**)**

Zum Löschen betimmter Werte eines Vektors kann man vor den Index ein Minuszeichen setzen. Damit kann der erste Wert mit

v <- v**[-1]**

und der letzte mit

v <- v**[-length(**v**)]**

gelöscht werden.

5.8 Einfache statistische Vektoroperationen

Häufig enthalten Vektoren diskrete Beobachtungen einer Zufallsvariable (Stichprobe), die statistisch ausgewertet werden sollen. Wir beschränken uns zunächst auf Zahlenvektoren.

Einen Überblick über die wichtigsten Kenngrößen liefert die Funktion `summary()`

```
summary(rivers)
    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   135.0   310.0   425.0   591.2   680.0  3710.0
```

Sofort klar sein sollten Minimum und Maximum. *Mean* gibt die mittlere Flusslänge (arithmetisches Mittel) an, der *Median* ist die Flusslänge, für die die Hälfte der Flüsse kürzer und die andere Hälfte länger als dieser Wert sind. Die Quantile geben das Interval der häufigsten Flusslängen an: 25% der Flüsse sind kürzer als der Wert des ersten Quantils, 25% der Flüsse sind länger als der Wert des dritten Quantils, 50% der Flusslängen liegen demnach im Intervall zwischen ersten und dritten Quantil.

Eine große Differenz zwischen Median und Mittel weist auf eine recht ungleiche Verteilung der Größen hin. Beispiel: In einer Firma mit einem Chef und 10 Angestellten gönnt sich der Chef ein Gehalt von 500 k€, seinen Angestellten zahlt er dagegen 30 k€ Jahresgehalt.

```
gehaelter <- c(rep(30, 10), 500)
gehaelter
[1]  30  30  30  30  30  30  30  30  30  30 500
summary(gehaelter)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  30.00   30.00   30.00   72.73   30.00  500.00
```

Das gemittelte Einkommen ist also 73 k€, aber davon haben 10 von 11 Personen nichts. Der Median gibt hier eine realistischere Aussage.

Diese und weitere statische Grundgrößen können natürlich auch einzeln abgefragt werden:

`min(vektor` Kleinster Wert

`max(vektor` Größter Wert

`mean(vektor` Arithmetisches Mittel

`median(vektor` Median

`quantile(vektor` Quantile (auch 50%- (= Median) und 100%-Quantil)

`var(vektor` Varianz der Werte

`sd(vektor` Standardabweichung der Werte

`sum(vektor` Summe aller Werte

`cumsum(vektor` Kumulierte Summen des Vektors (ergibt einen neuen Vektor)

`diff(vektor` Differenz zwischen benachbarten Werten (Umkehroperation zu `cumsum()`). Die Länge des Ergebnisses ist eins kleiner als der Originalvektor.

Die meisten Funktionen kennen einen Boolean-Parameter `na.rm`, mit dem NA-Werte entfernt werden können, andernfalls scheitert die Berechnung meistens.

Die Varianz einer Stichprobe gibt an, wie stark die Werte vom Mittelwert abweichen. Da dies eine quadratische Größe ist, bietet es sich häufig an, daraus die Wurzel zu ziehen(*Standardabweichung*), was die Funktion `sd()` realisiert.

Die Funktion `cumsum` geht den Vektor elementweise durch und bestimmt jeweils die Summe vom ersten bis zu diesem Element, die an den Ergebnisvektor angehängt wird. Beispiel: Partialsummen der arithmetischen Folge.

```
arith <- 1:10
x <- cumsum(arith)
x
[1]  1  3  6 10 15 21 28 36 45 55
# Und wieder die Originaldaten
y <- diff(x)
y
[1]  2  3  4  5  6  7  8  9 10
# Ersten Wert aus x ergänzen
y <- c(x[1], y)
y
[1]  1  2  3  4  5  6  7  8  9 10
```

5.9 Häufigkeitsverteilungen

Entält unser Vektor Daten einer Ordinal- oder Nominalskala, interessiert uns häufig, wie oft jeder Wert auftritt. Eine einfache tabellarische Darstellung erhalten wir mittels

```
table(vektor)
#relativ
table(vektor)/length(vektor)
#alternativ:
prop.table(table(vektor))
```

Für ordinale Daten können wir eine *ASCII*-Graphik erzeugen:

```
stem(vektor)
```

`NA`-Werte werden dabei ausgeblendet. Für unser Beispiel `aup.a1` erhalten wir eine aussagekräftige Anzeige, für `rivers` ist das Ergebnis (wie auch von `table()`) nicht verwendbar.

Da `stem()` einen Zahlenvektor erwartet, müssen wir für Nominalwerte, die als String vorliegen, zunächst einen Faktor bilden und diesen dann numerisch auswerten:

```
as.integer(faktor)
```

Beispiel: Ampelwerte als Tabelle und Stem-Diagramm:

```
table(ampel)
ampel
     green        red red-yellow     yellow
         3          5          1          1

fampel <- factor(ampel)
stem(as.integer(fampel))

  The decimal point is at the |

  1 | 000
  2 | 00000
  3 | 0
  4 | 0
```

Eigentlich sind *Stem-and-Leaf*-Diagramme zum Gruppieren von Daten nach Klassen (hier Vorkommastellen, Stem) und listet für jede Klasse die Werte (hier Nachkommastellen, Leaf, bei uns immer 0) auf.

Zur Verdeutlichung erzeugen wir 50 normalverteilte Zufallszahlen und zeigen das Stem-Diagramm an. Mit dem Parameter `scale` können wir die Klassenzahl beeinflussen.

```
# 80 normalverteilte Zufallszahlen mit Mittel 5 und Varianz 2
x <- rnorm(80, 5, 2)
"Stem-Diagramm:"
stem(x)
"Höher aufgelöst:"
stem(x,scale=2)
```

Mögliches Ergebnis (von konkreten Zufallszahlen abhängig (von konkreten Zufallszahlen abhängig):

```
[1] "Stem-Diagramm:"

  The decimal point is at the |

  -2 | 2
  -0 | 41
   0 | 29068
   2 | 01255557812333456666679
   4 | 011222233334566890013345688 8
   6 | 01236790000012479
   8 | 2745

[1] "Höher aufgelöst:"

  The decimal point is at the |

  -2 | 2
  -1 |
  -0 | 41
   0 | 29
   1 | 068
   2 | 012555578
   3 | 12333456666679
   4 | 01122223333456689
   5 | 00133456888
   6 | 0123679
   7 | 0000012479
   8 | 27
   9 | 45
```

5.10 Aufgaben

Aufgabe 5.1 Berechnen Sie Mittelwert und Varianz eines numerischen Vektors ohne Verwendung der statistischen Funktionen. Vergleichen Sie Ihre berechneten Werte mit den Ergebnissen der Statistikfunktion.

Aufgabe 5.2 *Sammeln von Daten, Hausaufgabe (6 Punkte)*

Erfassen Sie von Amazon oder einem anderen Online-Anbieter Ihrer Wahl für mindestens 3 Produkte (z. B. Notebook bestimmter Ausstattung, Kopfhörer, Schreibtischlampe) jeweils mindestens 20 Angebote mit (mindestens) folgenden Informationen:

- Individuelle Artikelbezeichnung,

- mindestens zwei Ausstattungsmerkmale (z. B. Speichergröße, Leistung, Gewicht, Farbe), mindestens eines davon metrisch,
- Anbieter,
- Preis,
- Versandkosten,
- Anzahl abgegebene Bewertungen,
- Mittlere Bewertung,

Ersetzen Sie alle Nicht-ASCII-Zeichen (Umlaute und ß) durch ASCII-Umschreibungen (ä → ae, ß → ss etc.).

Speichern Sie diese Daten als eigene *CSV*-Datei für jedes Produkt. Die Datei soll eine Kopfzeile mit kurzen Spaltennamen enthalten. Auch die Kopfzeilen dürfen nur ASCII-Zeichen enthalten.

Erzeugen Sie weiterhin für jedes Produkt eine eigene Datei, die lediglich die Anzahl der Bewertungen zeilenweise enthält.

Erzeugen Sie von Ihrem Lieblingsprodukt (also nur einem!) zwei Dateien, die die Preise bzw. die Versandkosten zeilenweise enthalten.

Schreiben Sie ein Jupyter-Notebook, das

- diese Bewertungszahlen aller Produkte sowie Preise und Versandkosten des ausgewählten Produkts einzeln in Vektoren einliest,
- den Gesamtpreis jedes Artikels Ihres gewählten Produkts berechnet,
- die statistischen Kenngrößen Median, Mittelwert und Standardabweichung für den Gesamtpreis berechnet und ausgibt,
- die Zahlen der abgegebenen Bewertungen für alle Produkte zu einem Vektor zusammenfasst,
- für die Anzahl der Bewertungen jedes Produkts sowie für alle abgegebenen Bewertungen ein Stem-diagramm erzeugt.

Beantworten Sie folgende Fragen in Form von Markdown-Zellen nach der jeweiligen Berechnung bzw. am Ende des Notebooks:

- Weichen Mittelwert und Median stark voneinander ab? Was können wir daraus ableiten?
- Welche konkrete Aussage liefert uns die Standardabweichung?
- Welche Informationen können wir aus den einzelnen und dem gesamten Stem-Diagramm ableiten?

Hinweise zur Abgabe: Packen Sie die *CSV*-Datei, die Spaltendateien und Ihr Jupyter-Notebook zu einem *ZIP*-File zusammen, das Sie über das Bildungsportal abgeben. Die *CSV*-Dateien sind auch bei allen folgenden Hausaufgaben wieder mit abzugeben.

Aufgabe 5.3 Erzeugen Sie für eine der obigen Artikelkategorien eine Datei, die die mittleren Bewertungen für jeden Artikel zeilenweise enthält, lesen Sie diese als Vektor ein und erzeugen Sie eine Verteilungstabelle der Bewertungen.

Aufgabe 5.4 Wählen Sie einen deutschen und einen englischen Text aus und zerlegen Sie diesen in seine Buchstaben, die Sie durch Leerzeichen bzw. Zeilenschaltung getrennt in jeweils eine Datei schreiben. Jeder der beiden Texte sollte etwa 1000 Zeichen umfassen. Verwenden Sie dazu ein Programm einer beliebigen Programmiersprache (auch in *R* möglich, aber noch nicht behandelt). Großbuchstaben sollen dabei in Kleinbuchstaben, Umlaute und ß ignoriert werden, es sind also genau 26 verschiedene Zeichen zu erfassen. Erzeugen Sie daraus pro Sprache jeweils eine Textdatei mit den Buchstaben. Beispiel:

```
d a s i s t e i n t e x t
u n d n o c h e i n e z e i l e
```

Lesen Sie diese Vektoren in R ein. Bestimmen Sie jeweils die relative Häufigkeit des Buchstaben `e` (Verhältnis der Anzahl des Buchstaben zur Gesamtzahl der Buchstaben). Erzeugen Sie eine Verteilungstabelle und ein Stem-Diagramm für die Häufigkeit der vorkommenden Buchstaben. Hinweis: Mit dem Parameter `scale=2` werden alle Zeilen ausgegeben.

Wandeln Sie den Zeichen- in einen Zahlenvektor um (a = 1, b = 2, …) und berechnen Sie Mittelwert, Median und Standardabweichung (und nicht mehr). Vergleichen Sie die Werte mit der 10-mal wiederholten Folge der Zahlen von 1 bis 26 (die damit offenbar gleichverteilt sind). Hinweis: Einen Vektor aller Kleinbuchstaben enthält das Objekt `letters`.

Aufgabe 5.5 Vergleichen Sie die statischen Eigenschaften der Prüfungsergebnisse der Aufgabe `aup-a1.dat`, wenn Sie zum einen die NA-Werte entfernen und zum zweiten durch 0 ersetzen.

Aufgabe 5.6 Erzeugen Sie aus dem bereinigten Vektor der Prüfungsergebnisse Vektoren der 20 besten und 20 schlechtesten Punktzahlen und werten Sie diese statistisch aus.

KAPITEL 6

Data Frames

6.1 Grundlagen

Data Frames (`data.frame`) sind vermutlich die wichtigsten Objekte für statistische Auswertungen. Sie bestehen üblicherweise aus mehreren gleichlangen Datenspalten, wobei jede Spalte einen Namen hat. Die Zeilen entsprechen den einzelnen Datensätzen und sind meist einfach durchnumeriert.

Vertiefung

Data Frames sind eine Spezialform des Datentyps `list`, der es erlaubt, unterschiedliche Objekte zu einem neuen Objekt zusammenzufassen (vergleichbar mit dem *ADT Dictionary* oder den Datentypen `struct` oder `class` in C/C++).

Aus statistischer Sicht ist es wichtig, dass jede Zeile des Data Frames die Daten eines Beobachtungsobjektes darstellt. Folgendes Konstrukt ist demnach kein Data Frame (Wirkungsweise verschiedener Maßnahmen auf jeweils gleichgroße Guppen), da unterschiedliche Objekte (Personen) einander zugeordnet werden:

```
Gruppe.1 Gruppe.2 Gruppe.3
      10        8       -3
      -4        2       15
       5        7       -1
```

Richtig wäre folgende Modellierung:

```
  Wirkung Gruppe
1      10      1
2       8      2
3      -3      3
4      -4      1
5       2      2
6      15      3
7       5      1
8       7      2
9      -1      3
```

6.2 Konstruktion und Zugriff

Wir konstruieren einen Data Frame aus einen oder mehreren gleichlangen Vektoren:

dataFrame <- **data.frame**(vector1*[, ...]*)

Stringvektoren werden in *R*-Version 3 automatisch in Faktoren umgewandelt, in Version 4 dagegen nicht. Dieses Verhalten wird durch den Parameter `stringAsFactors` gesteuert. Ist er `TRUE` erfolgt diese Umwandlung, ist er `FALSE`, unterbleibt sie.

Beispiel: Jahreszeiten und mittlere Temperatur.

```
temp <- c(0.6, 8.3, 14.1, 7.9)
seasons <- c('Winter', 'Spring', 'Summer', 'Autumn')
tempdata <- data.frame(seasons, temp)
tempdata

   seasons temp
1   Winter  0.6
2   Spring  8.3
3   Summer 14.1
4   Autumn  7.9
```

Die Spaltenüberschriften sind die Namen der Vektoren. Das wird problematisch, wenn in eine Spalte der Tabelle nicht als Objekt vorliegt, sondern erst im Aufruf von `data.frame()` erzeugt wird:

```
fib <- data.frame(c(1, 1, 2, 3, 5, 8, 13))
fib

  c.1..1..2..3..5..8..13.
1                       1
2                       1
3                       2
4                       3
5                       5
6                       8
7                      13
```

Es ist deshalb oft sinnvoll, einzelne oder alle Spalten umzubenennen:

```
# Einzelspalte
names(dataframe)[index] <- "name"
# Alle Spalten
names(dataframe) <- c(stringvektor)
```

Beispiel:

```
names(fib)[1] <- "Fib"
# alternativ: names(fib) <- c("Fib")
fib
  Fib
1   1
2   1
3   2
4   3
5   5
6   8
7  13
```

Auf einen einzelnen Wert innerhalb eines Data Frames kann ähnlich wie auf ein zweidimensionales C-Array über ein Indexpaar `[zeile, spalte]` zugegriffen werden. Im Gegensatz zu C/C++ verwenden wir nur ein Klammenpaar und trennen die Indices durch Komma:

```
frame[zeile, spalte]
```

Wie schon bei Vektoren können Zeilen- und Spaltenindex auch Indexlisten sein, um einen Teilframe zu erzeugen.

Die einzelnen Spalten sind weiter als Vektoren nutzbar. Der Zugriff erfolgt entweder über die Spaltennummer, die in *doppelte* eckige Klammen eingeschlossen wird, die Angabe des Spaltenindex, wobei der Zeilenindex frei bleibt, oder über den Spaltennamen, der mit einem `$`-Zeichen an den Framenamen angehängt wird:

```
dataframe[[index]]
dataframe[,index]
dataframe$spaltenname
```

Die zweite Variante kann auch für die Auswahl einer Zeile genutzt werden, indem der Spaltenindex frei bleibt:

```
dataframe[index,]
```

Vertiefung

Soll einem bereits vorhandenen Data Frame eine neue Spalte hinzugefügt werden, geschieht dies einfach durch Vergabe eines neuen Spaltennamens und der Zuordnung des Spaltenvektors:

```
frame$neueSpalte <- vektor
# oder
frame['neueSpalte'] <- vektor
```

Statt des Vektors kann auch ein Einzelwert verwendet werden, der dann in jeder Zeile in der neuen Spalte eingetragen wird.

Neben Data Frames unterstützt *R* auch *Matrizen* und mehrdimensionale Felder. Eine Matrix ist ebenfalls zweidimensional, aber alle enthaltenen Daten haben den gleichen Typ, es gibt keine Zeilen- und Spaltennamen. Da wir diesen Datentyp zunächst nicht benötigen, wollen wir an dieser Stelle nicht darauf eingehen, werden diesen aber später z. B. beim Transponieren eines Data Frames benötigen (*Transponieren von Datenframes*).

6.3 Bearbeiten von Data Frames

Sollen Data Frames in *R* interaktiv bearbeitet werden, kann dies mit den Funktionen `edit()` bzw. `fix()` geschehen:

neuerFrame <- **`edit(`**datenFrame**`)`**
`fix(`datenFrame**`)`**

Es öffnet sich ein graphischer Tabelleneditor, in dem die Werte bearbeitet und ergänzt werden können. Nach dem Schließen stehen im Zielobjekt die neuen Daten bereit. Dabei lässt `edit()` den ursprünglichen Frame unverändert und liefert einen neuen Data Frame, `fix()` kopiert diesen automatisch auf den alten Data Frame zurück.

Hinweis: Eine Möglichkeit des Abbrechens ohne Veränderung der Werte scheint es nicht zu geben. Es ist deshalb sinnvoller, mit `edit()` zunächst einen neuen Data Frame zu erzeugen und das ursprüngliche Objekt erst nach Prüfung der Daten zu überschreiben.

6.4 Einlesen und Speichern von Data Frames

Sollen Data Frames aus einer Datei gelesen werden, sollte die Datei textbasiert sein und in jeder Zeile einen Datensatz enthalten, dessen Daten z. B. durch Leerzeichen getrennt sind. Eine Kopfzeile ist zulässig (gleicher Aufbau), Kommentarzeilen mit einleitendem `#` sind ebenfalls möglich.

Beispiel: Unterrichtsstunden pro Tag.

```
# Unterrichtsstunden pro Tag
Tag Vorlesung Uebung
Mo          4      2
Di          8      0
Mi          2      6
Do          4      4
Fr          2      0
```

Das Einlesen geschieht mit der Funktion

frame <- **`read.table(`**"fileName"***`[, header=T][, dec=`***"dezimaltrenner"***`]`***)

Der Dezimaltrenner ist laut Vorgabe ein Punkt. Wenn dieser bei unseren Daten ein Komma ist, müssen wir den Parameter `dec` entsprechend anpassen.

Wir hatten schon bei Vektoren gesehen, dass für Auswertungen nach Nominal- und Ordinalskalen sinnvoll *Faktoren* verwendet werden sollten. Beim Einlesen von Textspalten unterscheiden sich die *R*-Versionen 3 und 4: in der älteren Version erfolgt eine automatische Konvertierung in Faktoren, in Version 4 unterbleibt sie. Auch hier kann das gewünschte Verhalten gezielt gesteuert werden, der Parameter heißt `as.is`. Mit dem Wert `FALSE` erfolgt die Umwandlung, mit `TRUE` unterbleibt sie.

Warnung: Vergessen Sie nicht das Ausblenden der Kopfzeile, falls diese vorhanden ist. Andernfalls gehen alle Daten (auch metrische) in Nominaldaten über!

Diese Funktion unterstützt eine Vielzahl weiterer Parameter, die festlegen, welche Zeichen als Trennzeichen zwischen den Daten möglich sind, welche Zeichenketten für fehlende Werte (`NA`) stehen und mit welchen Zeichen Strings eingeschlossen werden. Speziell für typische *CSV*-Dateien existieren angepasste Funktionen `read.csv()` (Trennung mit Komma, Dezimalpunkt) und `read.csv2()` (Trennung mit Semikolon, Dezimalkomma) mit geeigneten Voreinstellungen.

Soll ein Data Frame im gleichen Format gespeichert werden, benutzen wir die Funktion `write.table()`, müssen aber die Zeilennamen ausblenden:

write.table(datenframe, "fileName", **row.names=F**)

6.5 Analysen

Nach dem Einlesen oder Erzeugen eines Data Frames sollte man dessen Struktur zunächst analysieren. Dies geschieht in der einfachsten Form durch seine Ausgabe:

frameName

Beispiel: Kückengewichte bei verschiedenen Dieten.

```
ChickWeight
```

Ausgabe:

```
    weight Time Chick Diet
1       42    0     1    1
2       51    2     1    1
3       59    4     1    1
4       64    6     1    1
5       76    8     1    1
6       93   10     1    1
7      106   12     1    1
8      125   14     1    1
9      149   16     1    1
10     171   18     1    1
11     199   20     1    1
...
```

Da dieser Frame sehr lang ist, kann man sich mit den Funktionen `head()` bzw. `tail()` auf die ersten bzw. letzen Einträge beschränken.

head(dataFrame *[* , anzahl*]*)
tail(dataFrame *[* , anzahl*]*)

```
head(ChickWeight)
tail(ChickWeight, 3)
```

Ausgaben:

```
  weight Time Chick Diet
1     42    0     1    1
2     51    2     1    1
3     59    4     1    1
4     64    6     1    1
5     76    8     1    1
6     93   10     1    1

    weight Time Chick Diet
576    234   18    50    4
577    264   20    50    4
578    264   21    50    4
```

Falls uns unbekannt ist, ob unsere Daten überhaupt den richtigen Datentyp haben, können wir dies mit den Funktionen `class()` und `typeof()` feststellen.

```
class(faithful)
[1] "data.frame"
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
class(ChickWeight)
[1] "nfnGroupedData" "nfGroupedData"  "groupedData"    "data.frame

typeof(faithful)
[1] "list"
typeof(ChickWeight)
[1] "list"
```

Wir sehen, dass das Objekt `ChickWeight` eigentlich kein Data Frame ist (aber einen enthält). Wir können mit diesen Daten wie mit einem Data Frame arbeiten.

Die Zahl der Zeilen und Spalten eines Data Frames liefert uns die Funktion `dim()`:

```
dim(ChickWeight)
[1] 578   4
dim(faithful)
[1] 272   2
```

Eine genauere Analyse erlaubt die Funktion `str()`, die uns alle enthaltenen Objekte und Attribute auflistet:

```
str(faithful)
'data.frame':   272 obs. of  2 variables:
 $ eruptions: num  3.6 1.8 3.33 2.28 4.53 ...
 $ waiting  : num  79 54 74 62 85 55 88 85 51 85 ...

lehre <- read.table("lehre.dat", header=T)
str(lehre)
'data.frame':      5 obs. of  3 variables:
 $ Tag      : Factor w/ 5 levels "Di","Do","Fr",..: 5 1 4 2 3
 $ Vorlesung: int  4 8 2 4 2
 $ Uebung   : int  2 0 6 4 0
```

Ebenso können wir die Funktion `summary()` für die wichtigsten statischen Kenngrößen verwenden, die dann einzeln für jede Spalte angewendet wird:

```
summary(lehre)
Tag      Vorlesung     Uebung
 Di:1   Min.   :2   Min.   :0.0
 Do:1   1st Qu.:2   1st Qu.:0.0
 Fr:1   Median :4   Median :2.0
 Mi:1   Mean   :4   Mean   :2.4
 Mo:1   3rd Qu.:4   3rd Qu.:4.0
        Max.   :8   Max.   :6.0
```

Die Spaltennamen können mit der Funktion `names()` abgefragt und auch geändert werden:

```
# Anzeige
names(frame)
# Alle Spaltennamen neu setzen
names(frame) <- c("spalte1"[, ...])
# Einen Spaltennamen ändern
names(frame)[index] <- "spaltenName"
```

6.6 Sortieren von Data Frames

Auch Frames können zeilenweise sortiert werden. Dies geschieht einfach durch einen speziellen Ausdruck für den Zeilenindex, wobei der Spaltenindex frei bleibt:

sortierterFrame <- frame**[order(**spalte1*[, ...]***) ,]**

Es können mehrere Spalten angegeben werden. Ein Minuszeichen vor der Spalte bewirkt eine absteigende Sortierung (= aufsteigendes Sortieren der negierten Werte).

Beispiel: Größte Eruptionen des *Faithful*-Geysirs.

```
rf <- faithful[order(-faithful$eruptions,faithful$waiting),]
head(rf, 10)
```

Ergebnis:

```
    eruptions waiting
149     5.100      96
76      5.067      76
151     5.033      77
168     5.000      88
138     4.933      86
243     4.933      86
86      4.933      88
100     4.900      82
113     4.900      89
56      4.883      83
```

Die erste Spalte zeigt die ursprünglichen Zeilennummern an, die intern als String verwaltet werden. Damit unterscheiden sich nach der Sortierung die Zugriffe

```
rf["1"]
rf[1]
```

Im ersten Fall wird die Zeile 1 des ursprünglichen, unsortierten Frames, die nun an einer neuen Stelle steht, im zweiten Fall die Zeile 1 des sortierten Frames angezeigt.

6.7 Frames vereinigen

Oft ist es erforderlich, mehrere Data Frames mit unterschiedlichen Beobachtungen zu vereinen (weitere Beobachtungsdaten für bestimmte Objekte, verschiedene Beobachtungsserien).

Haben wir einen weiteren Data Frame der gleichen Struktur (also z. B. eine weitere Stichprobe), können beide Frames mit `rbind()` aneinander gehängt werden.

```
gesamtFrame <- rbind(frame1, frame2, )
```

Beispiel: Laufzeiten von Sortierverfahren mit verschiedenen Optimierungen: `sort_o0.res` und `sort_o2.res`. Die Einträge sollen in einem Data Frame vereinigt werden. Um die Daten unterscheiden zu können, fügen wir eine Spalte `opt` hinzu, die für den ersten Frame den Wert 0 und den zweiten den Wert 2 erhält.

```
sort.o0 <- read.table("sort_o0.res")
sort.o2 <- read.table("sort_o2.res")
# Spaltennamen
names(sort.o0) <- c("method", "size", "time")
names(sort.o2) <- c("method", "size", "time")
# Spalte "opt" hinzufügen
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
sort.o0$opt <- 0
sort.o2$opt <- 2
# Vereinigen
sort.all <- rbind(sort.o0, sort.o2)
```

Komplexer ist eine Vereinigung verschiedener Beobachtungen der gleichen Objekte, z. B. bei einer erneuten Befragung der gleichen Personen. Wir haben zwei Data Frames mit meist einer übereinstimmenden Variable (Personennummer), aber unterschiedlichen Spalten, die alle der richtigen Person zuzuordnen sind. Dabei gibt es jedoch häufig Objekte, die nur in einem der Data Frames auftauchen (Person nimmt an zweiter Umfrage nicht teil).

Am einfachsten gehen wir dabei analog zu einem *Natural Join* in *SQL* vor:

1. In beiden Frames bekommen die Join-Spalte(n) den gleichen Namen.
2. Alle Nicht-Join-Spalten erhalten in den beiden Data Frames unterschiedliche Namen.

Die Vereinigung erfolgt mit der Funktion `merge()`.

res <- **merge**(f1, f2*[*, **all=T***]*)

Hinweis: Mit `merge()` können nur genau zwei Frames vereinigt werden. Wollen wir mehrere Frames vereinigen, wenden wir die Funktion mehrfach an:

```
fAll <- merge(f1, f2)
fAll <- merge(fAll, f3)
...
```

Ohne den Parameter `all=T` werden nur die Zeilen übernommen, in denen die Werte aller Joinspalten übereinstimmen. Mit diesem Parameter werden alle Zeilen übernommen, die in einer der beiden Tabellen auftauchen. Fehlende Werte werden mit `NA` aufgefüllt.

Beispiel: Anzahl der Service-Anrufe eines (unterbeschäftigten) Admins in zwei verschiedenen Wochen:

```
# Reihenfolge der Faktoren
tage <- c("MO", "DI", "MI", "DO", "FR")
# Woche 1
# Tage mit Anrufen, Reihenfolge erhalten
t1 <- factor(c("MO", "DI", "MI"), tage)
# Anzahl und Dauer
w1 <- data.frame(t1, c(2, 1, 3), c(10, 15, 5))
# Namen vergeben, Tag und Anrufe sollen Join-Spalten sein
names(w1) <- c("Tag", "Anrufe", "Dauer.1")
# Woche 2
t2 <- factor(c("DI", "MI", "FR"), tage)
# Anzahl und Dauer
w2 <- data.frame(t2, c(1, 2, 4), c(25, 20, 30))
names(w2) <- c("Tag", "Anrufe", "Dauer.2")
```

Erzeugte Data Frames:

```
[1] "w1"
  Tag Anrufe Dauer.1
1  MO      2      10
2  DI      1      15
3  MI      3       5
[1] "w2"
  Tag Anrufe Dauer.2
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
1  DI      1      25
2  MI      2      20
3  FR      4      30
```

Eine einfache Vereinigung wählt genau die Zeilen aus, in der alle Spalten gleichen Namens gleich sind:

```
merge(w1, w2)
  Tag Anrufe Dauer.1 Dauer.2
1  DI      1      15      25
```

Mit dem Parameter `all=T` erhalten wir alle Zeilen. Tagesnamen können mehrfach erscheinen, wenn sich die Anrufszahl (ebenfalls Join-Spalte) unterscheidet:

```
merge(w1, w2, all=T)
  Tag Anrufe Dauer.1 Dauer.2
1  MO      2      10      NA
2  DI      1      15      25
3  MI      2      NA      20
4  MI      3       5      NA
5  FR      4      NA      30
```

Um das vermutlich gewünschte Ergebnis – Zeilen sind die Wochentage, Spalten sind die Beobachtungen verschiedener Wochen – erzeugen, müssen die Anrufsspalten der beiden Frames unterschiedlich benannt werden:

```
names(w1)[2] <- "Anrufe.1"
names(w2)[2] <- "Anrufe.2"

# Nur die Tage, die in beiden Wochen auftreten
merge(w1, w2)
  Tag Anrufe.1 Dauer.1 Anrufe.2 Dauer.2
1  DI        1      15        1      25
2  MI        3       5        2      20

# Alle Tage (bis auf FR, der in beiden Frames fehlt)
merge(w1, w2,all=T)
  Tag Anrufe.1 Dauer.1 Anrufe.2 Dauer.2
1  MO        2      10       NA      NA
2  DI        1      15        1      25
3  MI        3       5        2      20
4  FR       NA      NA        4      30
```

Hinweis: Falls uns die `NA`-Werte stören, können wir sie leicht durch 0 oder einen anderen Wert ersetzen:

```
frame[is.na(frame)] <- 0
```

6.8 Frequenztabellen (Histogramm)

Wollen wir aus Häufigkeiten einen Data Frame bauen, können wir einfach die Tabellendarstellung (`table()`) mit der Funktion `as.data.frame` in einen Data Frame umwandeln. Anschließend ist es meist sinnvoll, die Spalten umzubenennen.

Beispiel: Zensurenvektor.

```
zensuren <- c(1, 2, 2, 3, 3, 2, 1, 5, 4, 1, 2, 3)
frame.zensuren <- as.data.frame(table(zensuren))
frame.zensuren
  v Freq
  1 1    3
  2 2    4
  3 3    3
  4 4    1
  5 5    1
names(frame.zensuren) <- c("Zensur", "Anzahl")
```

6.9 Gruppieren von Daten

Enthalten eine oder mehrere Spalten eines Data Frames einen Faktor, ist oftmals eine gruppierte Auswertung nach diesem Faktor gewünscht. Wir betrachten dazu die anonymisierten Ergebnisse einer AuP-Klausur, die nach Studiengängen Wirtschaftsinformatik, Biomedizintechnik und Informatik (= Rest) gruppiert sind: `erg-aup2015.dat`. Achtung: Die Daten sind nur ein Zwischenstand, das Endergebnis war deutlich besser.

Nach dem Einlesen ersetzen wir die NA-Werte durch Nullen.

```
aup <- read.table(".../erg-aup2015.dat", header=T)
aup[!is.na(aup)] <- 0
```

Wollen wir nur eine Spalte gruppieren, können wir die Funktion `tapply()` verwenden:

gruppierterVektor <- **tapply(**Datenspalte**,** gruppierungsSpalten**, FUN=**funktionsname**)**

Beispiel: Durchschnittspunktzahl, Anzahl Teilnehmer, Anzahl Bestanden pro Studiengang.

```
rm(list=ls())
#setwd("~/afs/lehre/R")
aup <- read.table("samples/testdata/erg-aup2015.dat", header=T)
aup[is.na(aup)] <- 0
cat("Durchschnittspunktzahl pro Studiengang\n")
print(tapply(aup$Summe, aup$Studg, FUN=mean))
cat("Anzahl Teilnehmer pro Studiengang\n")
print(tapply(aup$Summe, aup$Studg, FUN=length))
cat("Bestanden pro Studiengang\n")
print(tapply(aup$Bestanden, aup$Studg, FUN=sum))
```

Ergebnis:

```
Durchschnittspunktzahl pro Studiengang
      BT       IF       WI
22.90909 28.33333 28.44828
Anzahl Teilnehmer pro Studiengang
 BT  IF  WI
 33 162  29
Bestanden pro Studiengang
BT IF WI
 3 39  6
```

Statt der vordefinierten Funktion können wir auch eine selbstdefinierte Funktion übergeben, die auf einem Vektor arbeitet.

Wollen wir Gruppierungen über mehrere Spalten vornehmen, können wir die Funktion `aggregate()` verwenden. Die Nutzung ist recht ähnlich.

gruppierterFrame <- **aggregate**(datenframe, **list**(Gruppierungsspalten), **FUN=**funktionsname)

Beispiel: Mittelwerte für alle Aufgaben je Studiengang.

```
rm(list=ls())
#setwd("~/afs/lehre/R")
aup <- read.table("samples/testdata/erg-aup2015.dat", header=T)
aup[is.na(aup)] <- 0
cat("Durchschnittspunktzahl pro Aufgabe und Studiengang\n")
aup.studg <- aggregate(aup[,2:8], list(aup$Studg), FUN=mean)
print(aup.studg)
```

Ergebnis:

```
Durchschnittspunktzahl pro Aufgabe und Studiengang
  Group.1       HA       A1       A2       A3        A4       A5       A6
1      BT 2.272727 5.121212 5.424242 3.030303 0.5454545 1.272727 3.454545
2      IF 5.080247 5.061728 5.037037 5.580247 1.0061728 1.641975 3.771605
3      WI 4.448276 5.965517 5.241379 2.827586 2.8965517 2.206897 3.344828
```

6.10 Aufgaben

Aufgabe 6.1 Vereinigen Sie die beiden Data Frames aus `sort_o0.res` und `sort_o2.res` so, dass die Laufzeiten beider Optimierungen als verschiedene Spalten eines Frames erscheinen (die Zeilenzahl also der der Einzelframes entspricht).

Aufgabe 6.2 Diese Aufgabe dient als Vorbereitung weiterer Hausaufgaben mit statistischen Auswertungen und sollte deshalb zeitnah bearbeitet werden.

Erfassen Sie ca. 30 Ihrer Lieblingsbücher mit Autor und Titel (Kurzform möglich, nur zur Identifizierung), Kategorie, Seitenzahl und Preis in einer Form, die Sie mit *R* als Data Frame einlesen können.

Erfassen Sie ebenso z. B. aus https://de.wikipedia.org/wiki/Liste_von_Fl%C3%BCssen_in_Deutschland die wichtigsten deutschen Flüsse (ca. 20) mit Name, Gesamtlänge, Abfluss, Gesamteinzugsgebiet.

Aufgabe 6.3 *Data Frames, statistische Grundgrößen, Hausaufgabe (8 Punkte)*

- Lesen Sie Ihre *CSV*-Dateien mit den Artikeldaten in je einen Data Frame ein.
- Geben Sie für jeden Data Frame die Quartile für jede metrische Variable und die Häufigkeit jedes Wertes für jede nominale Variable aus.
- Ergänzen Sie für jeden Data Frame je eine Spalte für den Gesamtpreis (Preis+Versand) sowie den relativen Gesamtpreis, bezogen auf den höchsten Gesamteinzelpreis (der den relativen Preis von 1 hat).
- Erzeugen Sie aus den getrennten Data Frames der verschiedenen Produkte einen gemeinsamen Data Frame, der die allen gemeinsamen Variablen/Spalten Produktname, Artikelbezeichnung, Preis, Versandkosten, Gesamtpreis, mittlere Bewertung, Anzahl Bewertungen enthält.
- Speichern Sie diesen Gesamtdatenframe unter dem Namen `preisvergleich.csv`. Der Spaltentrenner soll ein Semikolon, der Dezimaltrenner ein Komma sein, Spaltennamen sollen vorhanden, Zeilennamen dagegen weggelassen werden.
- Ermitteln Sie *ausgehend von Ihrem Gesamtdatenframe* den Mittelwert und die Standardabweichung für den Gesamtpreis, gruppiert nach jedem Produkt.

Geben Sie neben dem Jupyter-Notebook Ihre *CSV*-Dateien für die einzelnen Produkte als ZIP-Datei im OPAL ab.

Aufgabe 6.4 Erzeugen Sie aus Ihren Vektoren für die deutsche, englische und gleichverteilte Buchstabenzahl einen Data Frame, der für jeden Buchstaben die absolute und die relative Häufigkeit für beide Sprachen und für die Gleichverteilung enthält. Beispiel:

```
   buchst  de.abs de.rel en.abs en.rel gl.abs gl.rel
1       a      91   0.05     65   0.03     20   0.04
...
```

Erzeugen Sie daraus einen neuen Frame, der nur die Buchstaben und die drei relativen Häufigkeiten enthält.

Werten Sie diesen Data Frame mit `summary()` aus. Vergleichen Sie Mittelwerte und Median für die Sprachen und die gleichverteilten Buchstaben und interpretieren Sie gleiche Werte bzw. Unterschiede.

Geben Sie diesen Frame absteigend sortiert nach deutscher und nach englischer Buchstabenhäufigkeit aus.

Geben Sie bitte Ihre Originaltexte und das Programm zum Erzeugen der Buchstabenvektoren erneut mit ab.

KAPITEL 7

Mitgelieferte Testdaten

7.1 Überblick

R liefert eine Anzahl von Datensätzen mit, die innerhalb der R-Hilfe und -Beispiele benutzt werden. Diese können wir ebenso für eigene Tests verwenden. Diese Daten sind im Paket `datasets` enthalten.

Eine Übersicht der vorhandenen Daten kann mit dem Kommando

```
library(help="datasets")
```

angezeigt werden. Detaillierte Informationen zu einem bestimmten Datensatz erhält man mit

`help("objektname")`

Die Daten sind oft umfangreich, deshalb ist es sinnvoll, diese für einen ersten Eindruck mit `head()` anzuzeigen.

7.2 Einige geeignete Beispieldaten

`rivers` Flusslängen der USA. Einfacher Zahlenvektor zum Untersuchen von statistischen Daten mit metrischer Verhältnisskala.

faithful Wartezeit und Eruptionshöhe des Geysirs *Old Faithful* im *Yellowstone National Park* in einem bestimmten Zeitraum. Datenframe mit gut korrelierten Daten.

ToothGrow Zahnwachstum bei Guinea-Schweinen in Abhängigkeit von Vitamin C und der Art der Verabreichung. Datenframe mit Nominal- und metrischen Skalen.

cars Geschwindigkeit und Bremsweg von Autos um 1920. Datenframe mit gut korrelierten Daten.

quakes Lage und Stärke von Erdbeben auf den Fiji-Inseln seit 1964. Datenframe mit Geodaten.

AirPassengers Flugpassagiere einer Fluglinie von 1949 bis 1960. Zeitreihe.

Nile Stärke des Nilhochwassers von 1871 bis 1970. Zeitreihe.

Zeitreihen wurden bisher nicht besprochen. Einfache statistische Auswertungen sind wie bei Vektoren möglich. Ebenso können die Daten graphisch dargestellt werden (s. u.).

KAPITEL 8

Plots

8.1 Testdaten

Als Testdaten für Plots wollen wir verschiedene Datenframes verwenden. Ein Datensatz mit einem starken funktionalem Zusammenhang ist der mitgelieferte Datensatz `faithful`, der die Pausen und Dauer der Eruptionen des Yellowstone-Geysirs *Faithful* in einem bestimmten Zeitraum darstellt.

Mit einem zweiten Datensatz wollen wir untersuchen, wie stark der Zusammenhang zwischen Seitenzahl und Preis bei Fachbüchern ist. Ein entsprechender Datensatz, der gern erweitert werden kann, ist `buchpreise.dat`

Als dritten Datensatz verwenden wir die anonymisierten Teilergebnisse einer AuP-Prüfung. Die Aufgabe A2 wurde zum Zeitpunkt der Datenübernahme nur bei einer geringen Zahl der Teilnehmer, die Aufgabe A6 fast noch gar nicht bewertet. NA steht dabei für keine Bearbeitung oder Nichtteilnahme an der Prüfung im Gegensatz zu 0 für eine fehlende oder völlig falsche Lösung des Teilnehmers: `res.dat`. Hier

wollen wir die Punktverteilungen bei den einzelnen Aufgaben betrachten. Ebenso können wir untersuchen, ob es Zusammenhänge zwischen den Erfolgen bei verschiedenen Aufgaben gibt.

Als weiterer Datensatz sollen die Ergebnisse der österreichischen Bundespräsidentenwahl im Vergleich zur Wahrnehmung in den sozialen Medien dienen (S.P.O.N. – Die Mensch-Maschine: Das Versagen in der digitalen Fußgängerzone, der Datensatz befindet sich hier: `oesiwahl.dat`.

Weitere bereitgestellte Datensätze umfassen eine Liste der beliebtesten IT-Pioniere der Informatik-Studienanfänger 2016 sowie die Laufzeiten verschiedener Sortierverfahren in Abhängigkeit der Datenmenge.

Das Einlesen geschieht wie bereits besprochen mit

```
books <- read.table("buchpreise.dat", header=T)
aup <- read.table("res.dat", header=T)
```

Warnung: Nicht alle Datensätze haben einen Kopf, bitte vorher prüfen.

Der Datenframe `faithful` steht als Datenframe mit dem *R*-System direkt zur Verfügung.

Aufgabe 8.1 Untersuchen Sie die Struktur aller drei Datenframes.

8.2 Streudiagramme (Scatterplots, XY-Plots)

XY-Plots dienen zur Veranschaulichung funktionaler bzw. statistischer Zusammenhänge zwischen zwei metrischen Größen. Wir benötigen zwei Vektoren gleicher Länge, wobei der erste die unabhängige, der zweite die abhängige Größe beschreibt.

Obwohl der `plot()`-Befehl die Angabe von zwei Vektoren erlaubt, verwenden wir hier zur Verdeutlichung eine *Formel*, die in *R* für die Beschreibung aller Arten solcher Zusammenhänge benutzt wird:

abhängigeGröße ~ ausgangsGröße

Beispiel: Plotten einer Funktion. Wir erzeugen einfach zwei Vektoren mit Argumenten und Funktionswerten (eine andere Möglichkeit betrachten wir später).

```
x <- 1:100
y <- sin(x)
plot(y ~ x)
```

Ergebnis:

Hier ist aus der Graphik nicht zu erkennen, dass ein funktionaler Zusammenhang besteht. Deshalb besteht die Möglichkeit, durch Angabe des Parameters `type` die Punkte zu verbinden:

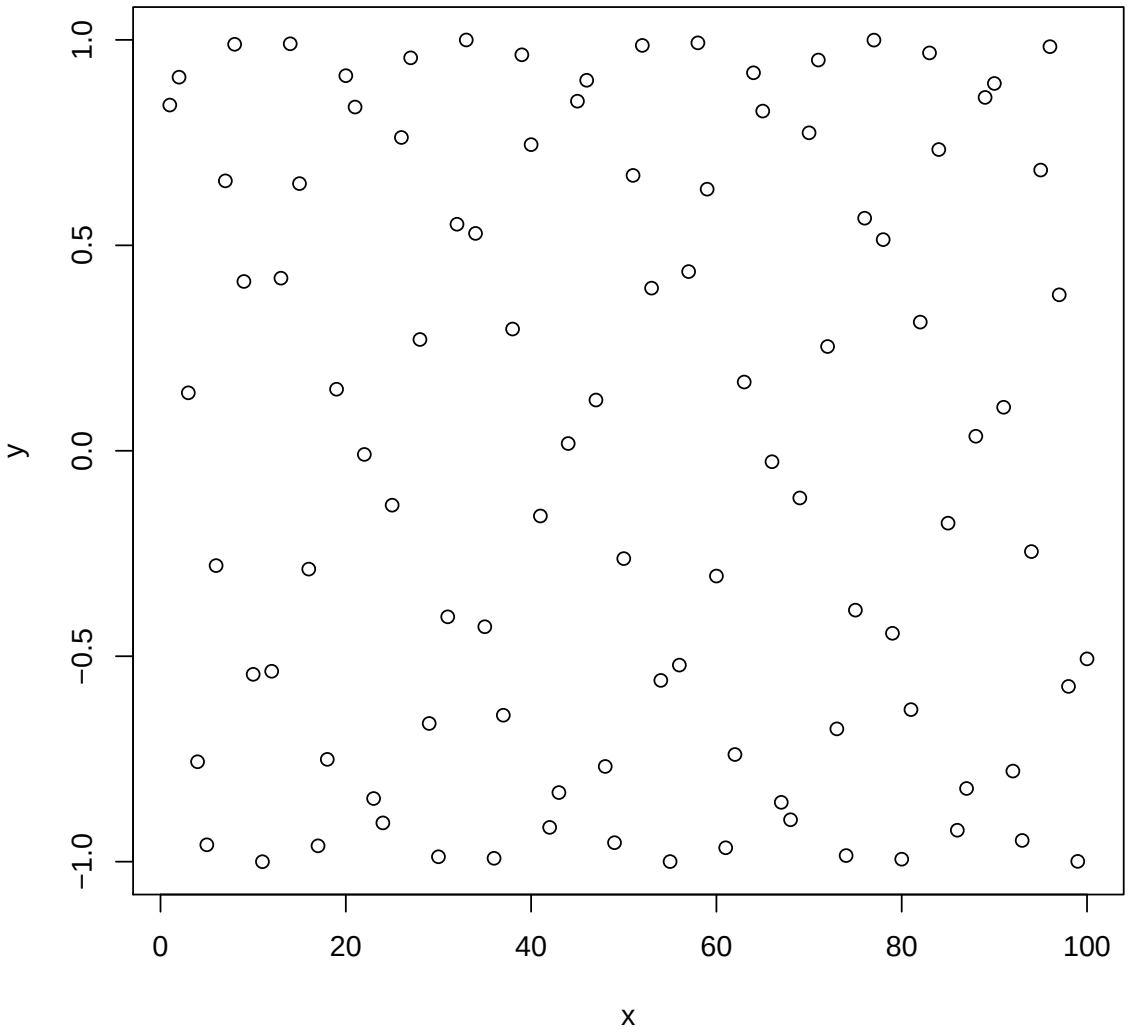
```
plot(y ~ x, type="l")
```

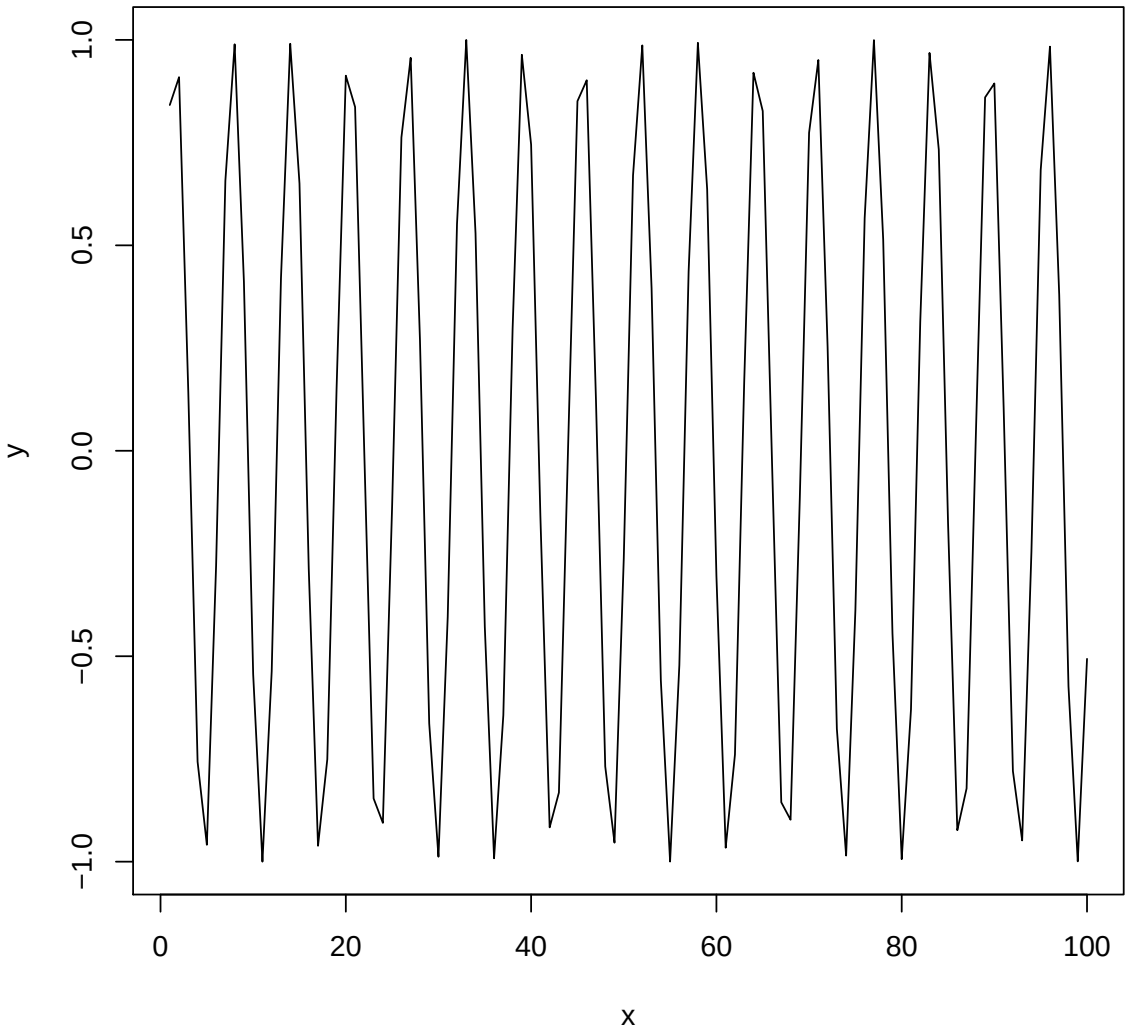
Sollen sowohl Linien als auch Punkte sichtbar sein, gibt man für den Parameter `type` die Werte `"b"` (both) oder `"o"` (overplotted) an. Die erste Variante lässt bei den Datenpunkten Lücken in den Linien, die zweite zeichnet einen durchgehenden Linienzug.

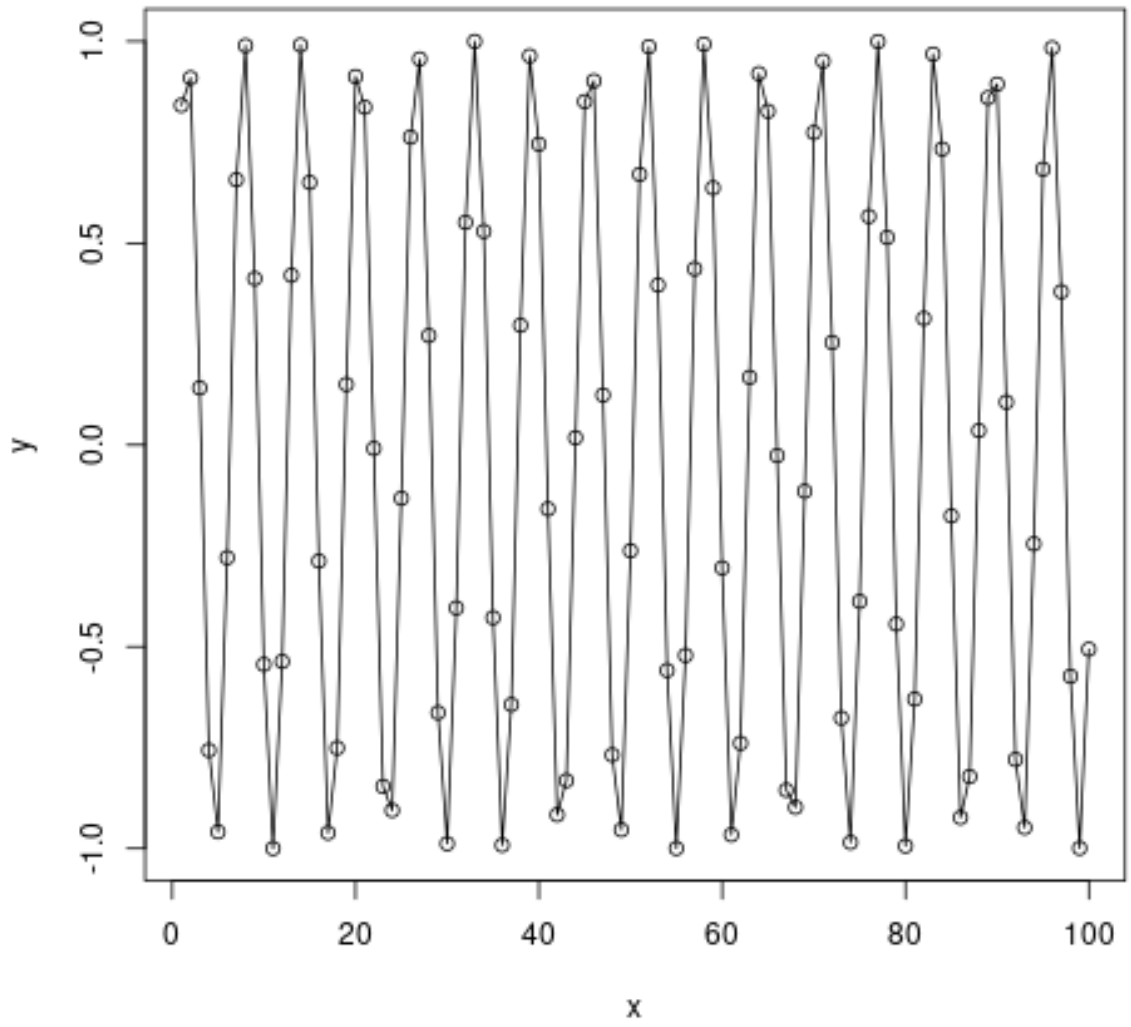
Beispiel mit `type="o"`

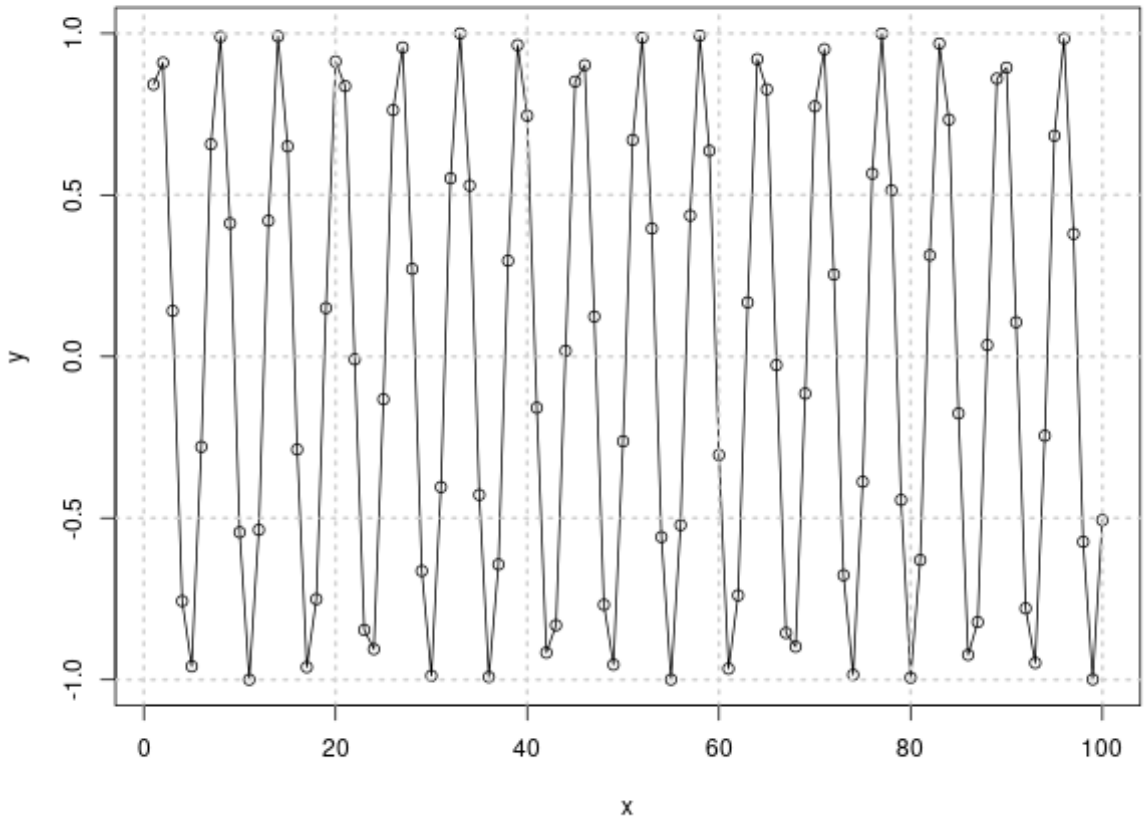
Hinweis: Voraussetzung für die Verbindung mit Linien ist das Ordnen der Daten nach der x-Koordinate, ansonsten entsteht nie eine aussagekräftige Kurve.

Oft ist es hilfreich, das Diagramm mit einem Gitter zu versehen, um die Werte besser ablesen zu können. Dies geschieht mit der Funktion `grid()`, die einfach als Folgebefehl nach dem Plot aufgerufen wird.


y
1.0
0.5
0.0
−0.5
−1.0
0
20
40
60
80
100
x


y
1.0
0.5
0.0
−0.5
−1.0
0
20
40
60
80
100
x


y
1.0
0.5
0.0
-0.5
-1.0
0
20
40
60
80
100
x

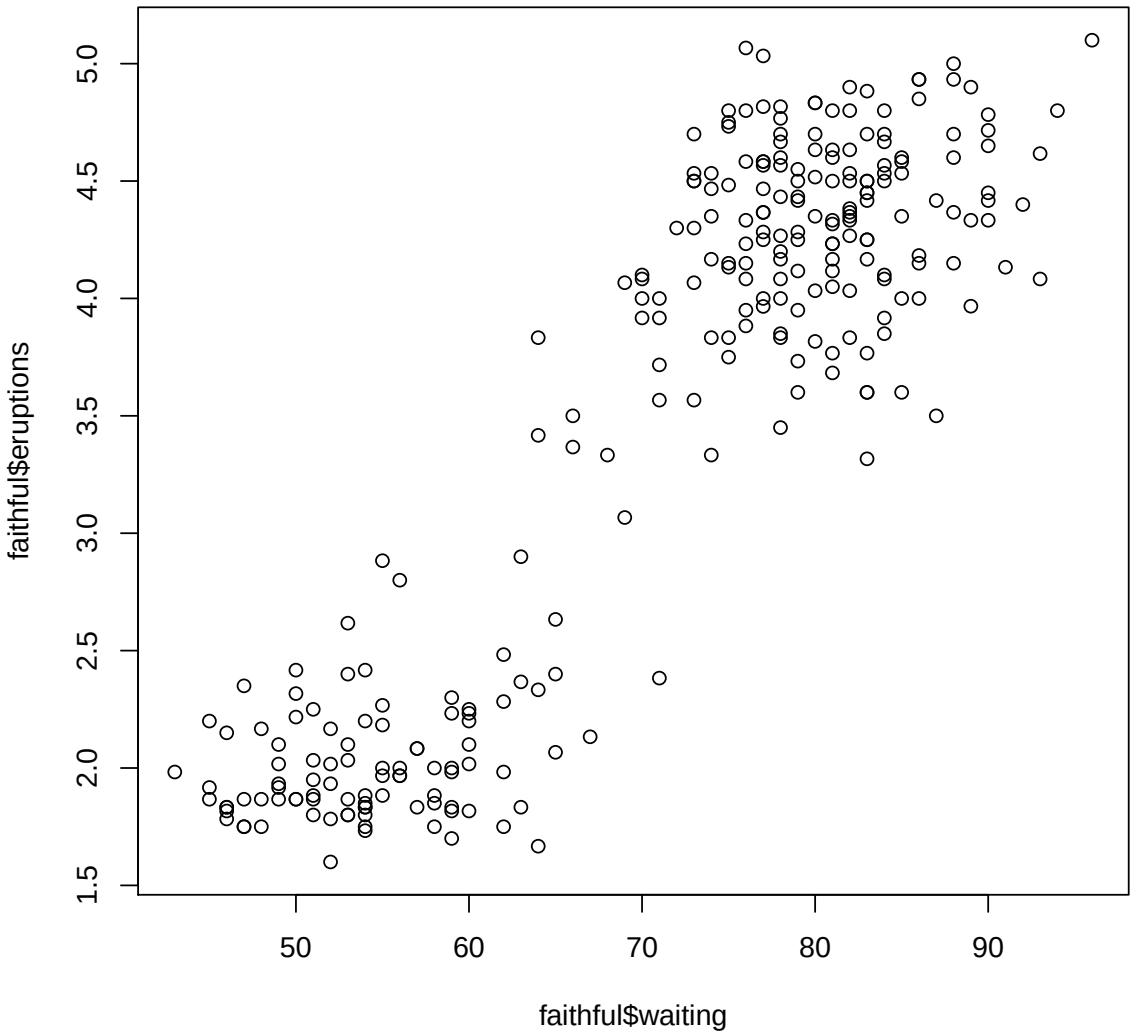

y
1.0
0.5
0.0
-0.5
-1.0
0
20
40
60
80
100
x

Oft wollen wir statistische Daten plotten, um eventuelle Zusammenhänge zu erkennen. Die Daten liegen üblicherweise als Datenframe vor, wir wählen die gewünschten Spalten aus.

Beispiel: Zusammenhang zwischen Wartezeit und Eruptionsdauer des Geysirs Faithful.

```
plot(faithful$eruptions ~ faithful$waiting)
# alternativ ohne Formel:
plot(faithful$waiting, faithful$eruptions)
```

Ergebnis:



Das Diagramm lässt einen linearen Zusammenhang zwischen den beiden Größen vermuten. Wir können versuchen, die Datenpunkte durch Linien zu verbinden, nachdem wir diese nach der Wartezeit sortiert haben.

```
faithful.order <- faithful[order(faithful$waiting),]
plot(faithful.order$eruptions ~ faithful.order$waiting, type="b")
```

Das Ergebnis ist allerdings nicht besonders aussagekräftig:

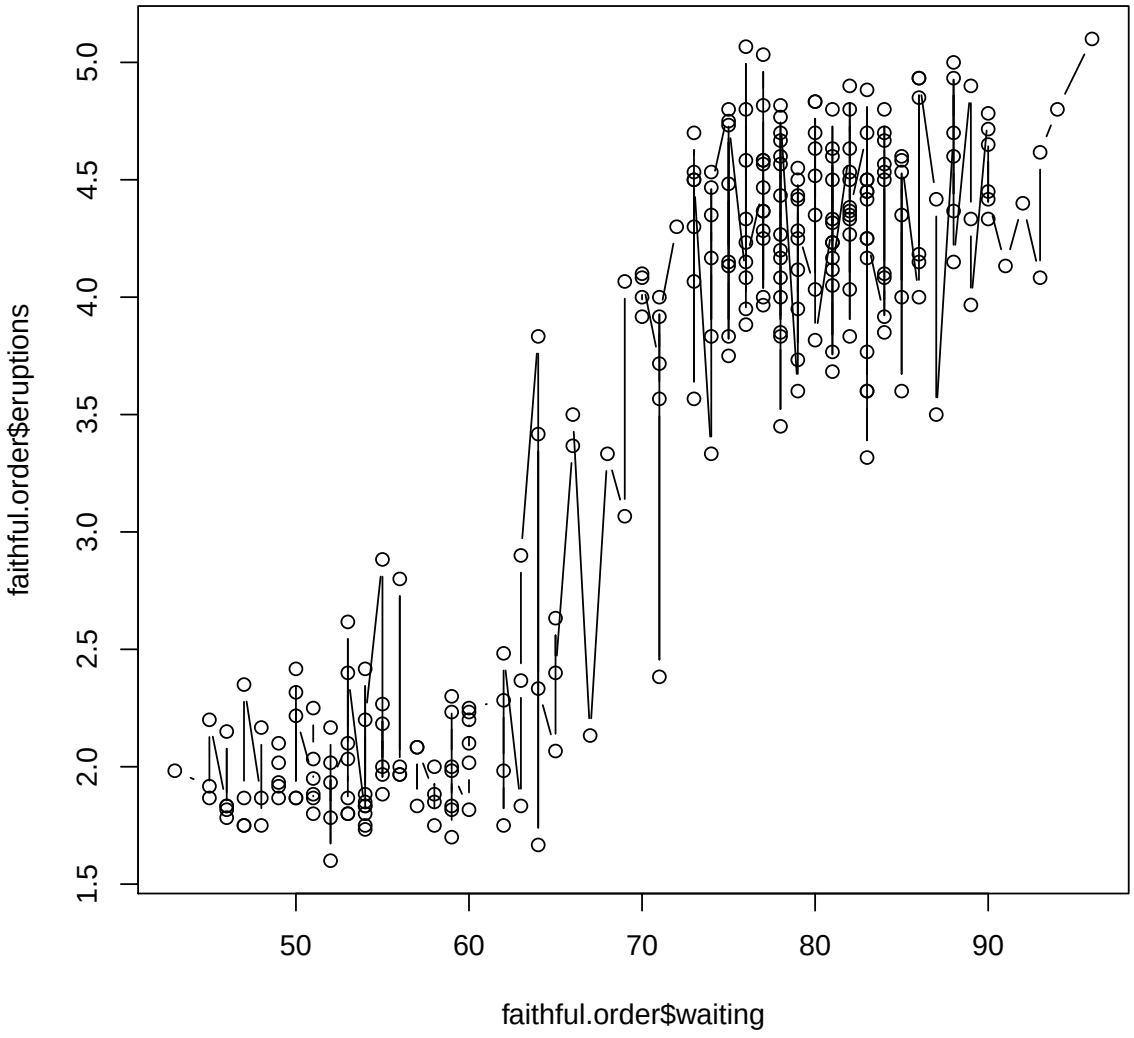


Abb. 8.1: Sortierung nach Wartezeit und mit Linien verbunden.

8.3 Diagramme in Jupyter Notebook

Falls uns die Größe der Diagramme in Jupyter Notebook zu klein ist, können wir diese mit den Optionen `repr.plot.width`, `repr.plot.height` und `repr.plot.res` beeinflussen. Die ersten beiden Angaben sind in Zoll, die letzte gibt die Auflösung (Bildpunkte je Zoll) an.

Beispiel:

```
options(repr.plot.width=9, repr.plot.height=7, repr.plot.res=200)
```

8.4 Beschriftungen

Um das Diagramme sinnvoll interpretieren zu können, benötigen wir Beschriftungen. Diese werden den Plot-Funktionen als weitere Parameter in der Form

parameter=wert*[, ...]*

durch Komma getrennt in beliebiger Reihenfolge übergeben.

Die Hauptbeschriftungen erzeugen wir mit folgenden Parametern:

`main` Haupttitel

`sub` Untertitel

`xlab` Hauptbeschriftung der x-Achse (Ordinate)

`ylab` Hauptbeschriftung der y-Achse (Abszisse)

Weitere Parameter steuern je nach Diagrammtyp die angezeigten Intervalle, Verhältnis zwischen x- und y-Scalierung, Symbol- und Linientypen und Farben.

8.5 Weitere Diagrammparameter

Enthält das Diagramm Datenpunkte, kann deren Darstellung mit dem Parameter `pch` beeinflusst werden. Der Wert ist entweder eine Zahl oder ein Vektor von Zahlen für jeden Punkt mit Werten zwischen 0 und 25 oder eine symbolische Stringangabe wie `"*"` oder `"+"`. Mit folgendem Trick können alle möglichen Symbole ausgegeben werden:

```
plot(0:25, pch=0:25)
# Beschriftung der Punkte (s. u.)
text(1:26, 0:25, 0:25, pos=1)
```

Der Parameter `col` legt die Farbe des Symbols oder der Linie fest. Er wird als Zahl (bzw. Zahlenvektor) zwischen 1 und 8 oder als String mit einem vordefinierten Farbnamen oder einem hexadezimalen *RGB*-Wert mit vorgestellter Raute (z. B. `"#003e2f"` für TU-Farbe) angegeben.

Werden Linien verwendet, steuert der Parameter `lty` den Linientyp. Die Angabe erfolgt durch eine Zahl zwischen 0 und 6 (0=unsichtbar, 1=durchgezogen) oder einen symbolischen String (`"blank"`, `"solid"`, `"dashed"`, `"dotted"`).

Wollen wir den dargestellent Datenbereich beeinflussen, geben wir die Parameter `xlim` bzw. `ylim` mit, die jeweils einen Vektor mit dem Minimum und Maximum erwarten:

```
plot(faithful$eruptions ~ faithful$waiting,
     xlim=c(0, 120), ylim=c(0, 7), pch="+",
     ...)
```

Dabei werden zum angegebenen Intervall 4% zugegeben. Wollen wir diesen Effekt abschalten, geben wir die Parameter `xaxs="i"` bzw. `yaxs="i"` mit.

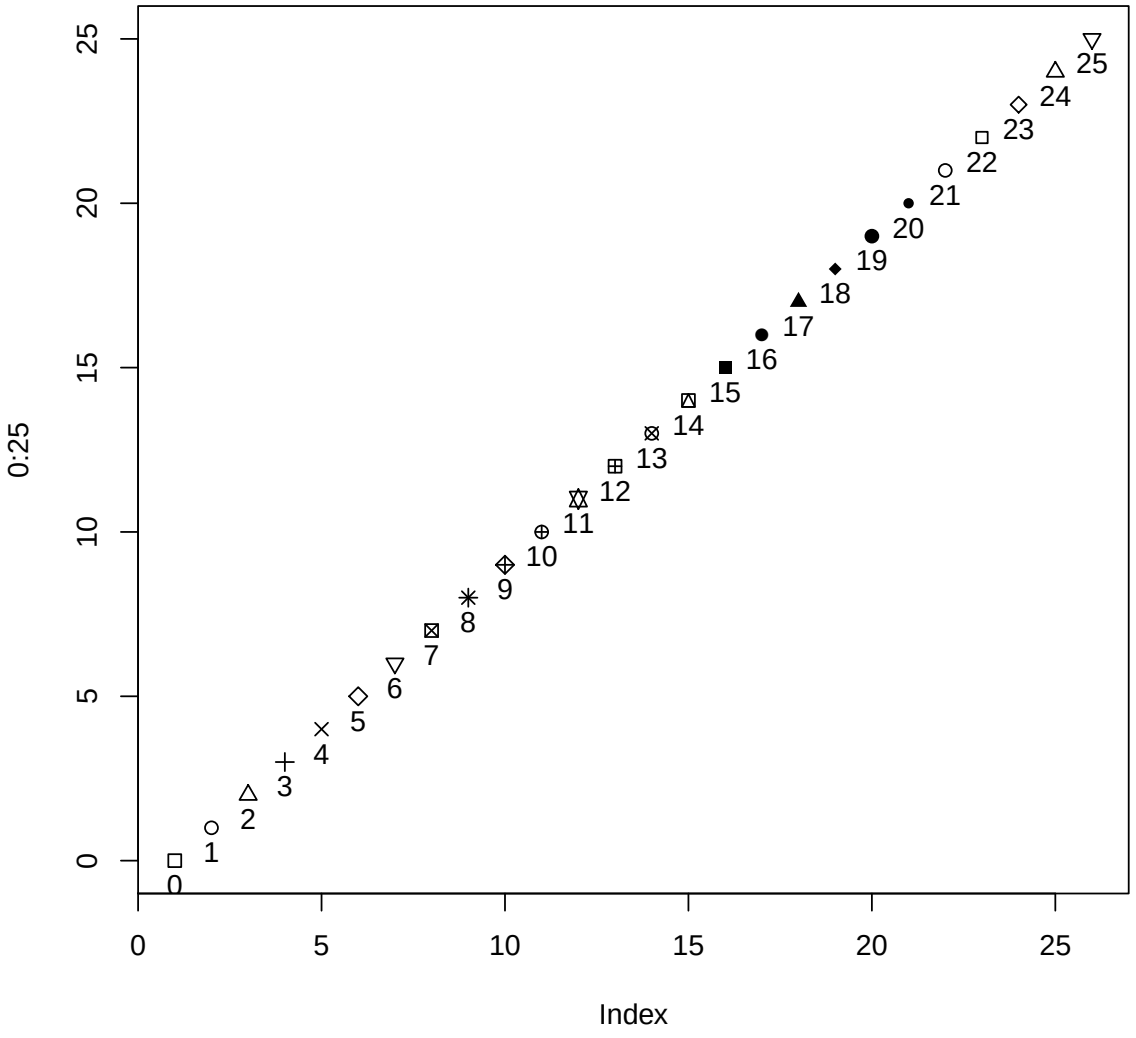


Abb. 8.2: Symbole für `pch`

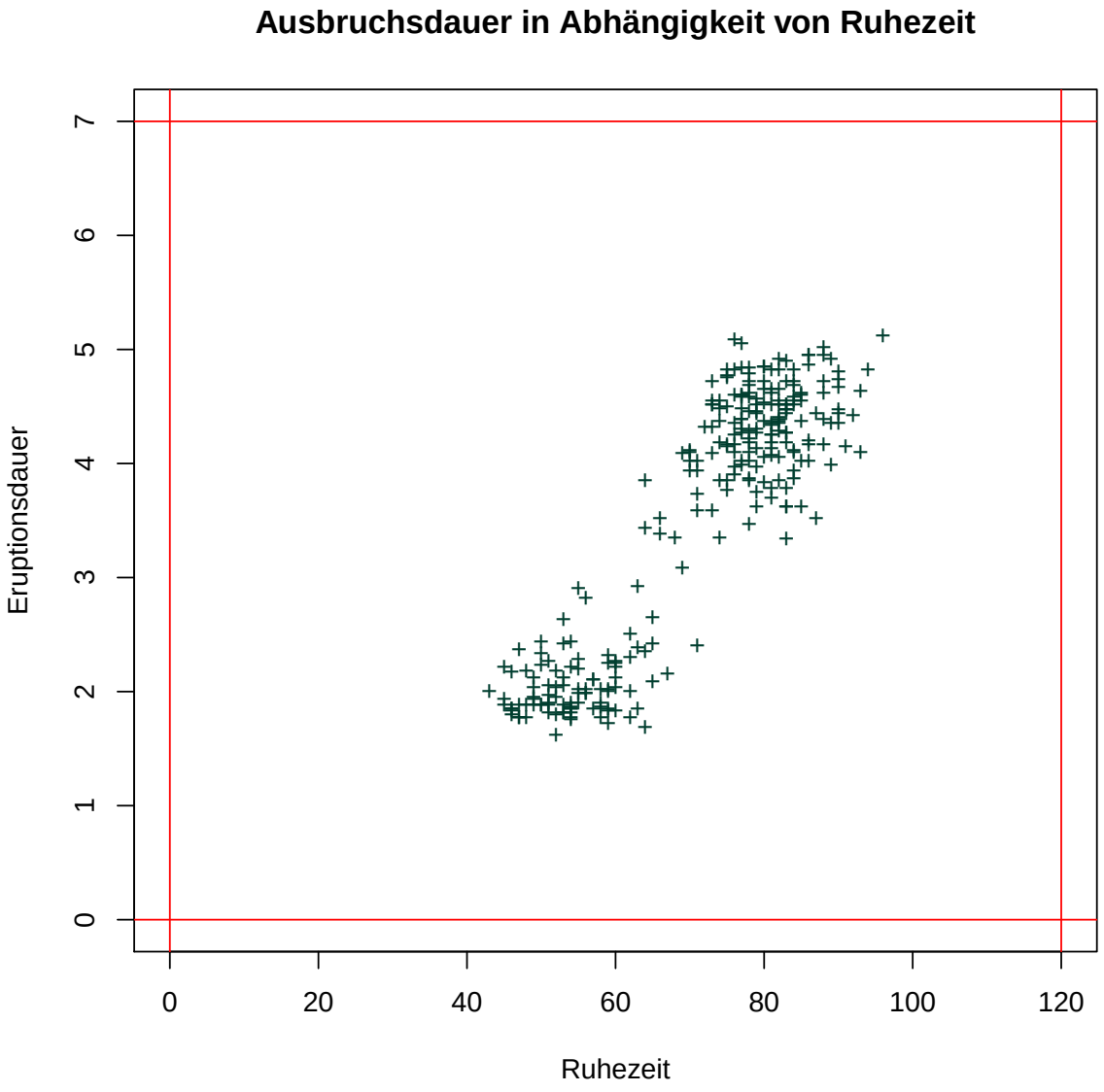
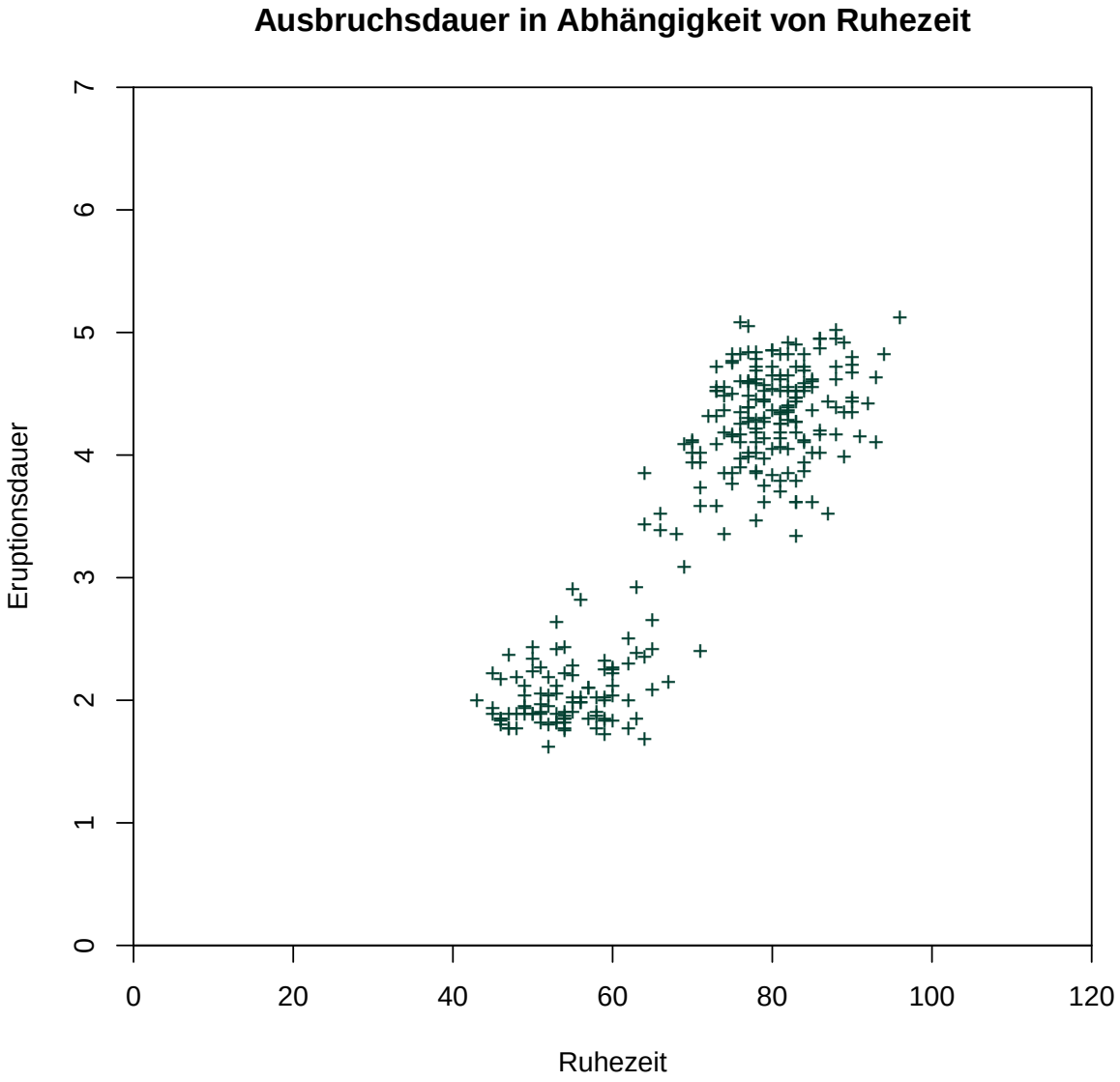


Abb. 8.3: Bereichsangabe, rote Linien zeigen die `xlim`- und `ylim`-Intervalle an.

```
plot(faithful$eruptions ~ faithful$waiting,
     xlim=c(0, 120), ylim=c(0, 7), pch="+",
     xaxs="i", yaxs="i",
     ...)
```



Überdecken die Daten einen Bereich von mehreren Zehnerpotenzen, kann es sinnvoll sein, eine logarithmische Achsenteilung zu benutzen. Dies erreicht man mit dem Parameter `log`, der die Werte `"x"`, `"y"` oder `"xy"` hat.

Beispiel: Plotten der Laufzeiten verschiedener Sortierverfahren in Abhängigkeit der Datenmenge. Die Erzeugung der Legende wird später erläutert. Datensatz: `sort.dat`.

```
data <- read.table('sort.dat', header=F)
names(data) <- c('Method', 'Size', 'Time')
# Faktor als Zahl
methNr <- as.integer(data$Method)
# lineare Skalen
plot(data$Time ~ data$Size, col=methNr, pch=methNr,
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
     main='Sortierzeiten', xlab='Feldgröße', ylab='Zeit (s)')
legend('topleft', legend=c('Bubble', 'Heap', 'Insertion', 'Selection'),
       col=1:4, pch=1:4)
# logarithmische Skalen
plot(data$Time ~ data$Size, col=methNr, pch=methNr, log='xy',
     main='Sortierzeiten', xlab='Feldgröße', ylab='Zeit (s)')
legend('topleft', legend=c('Bubble', 'Heap', 'Insertion', 'Selection'),
       col=1:4, pch=1:4)
```

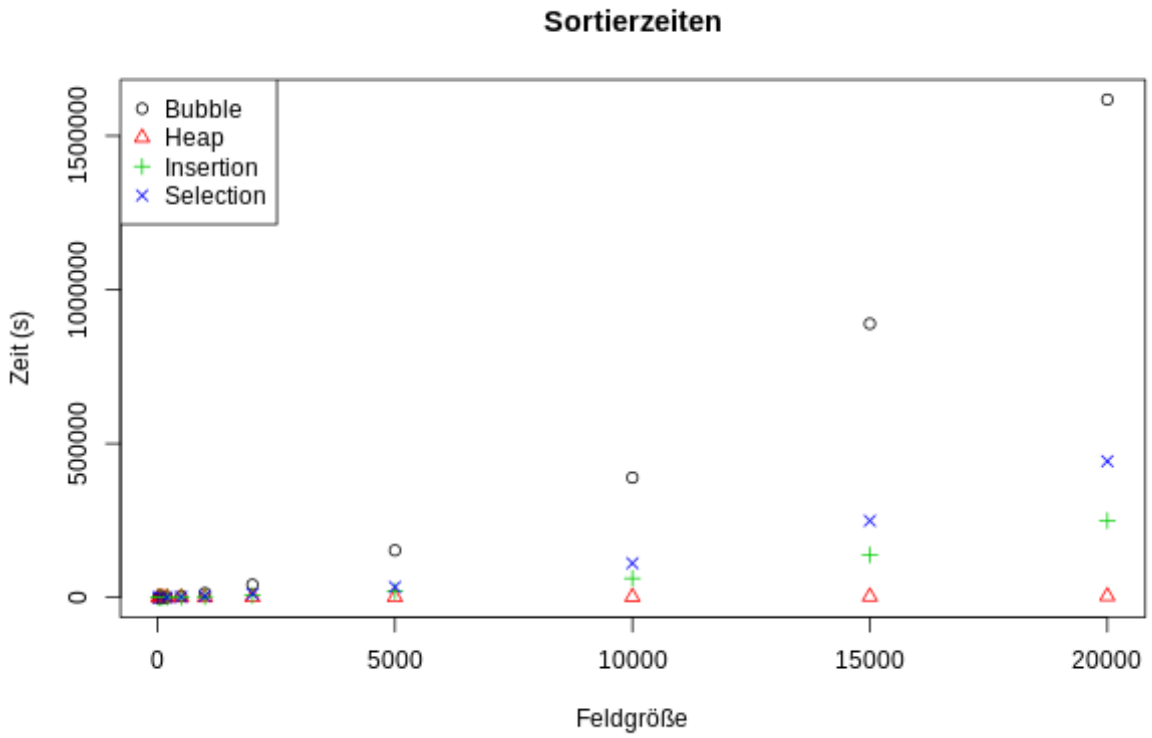


Abb. 8.4: Lineare Skalen

8.6 Funktionsgraphen

Neben statistischen Diagrammen können auch Funktionsgraphen dargestellt werden, die eng verwandt mit den XY-Plots sind:

curve(f(x), **xlim=**(xmin,xmax)*[, ...]*)

Beispiel: Glockenkurve (Achtung: Das sind nicht die korrekten Parameter für eine Verteilungsfunktion).

```
curve(exp(-x^2), xlim=c(-2,2), ylim=c(-0.5,1.5),
      main="Funktionsgraph",sub="Glockenkurve", ylab="y")
```

Ergebnis:

Vertiefung

Für viele statistische Verteilungsfunktionen existieren fertige Funktionsdefinitionen. So kann die Glockenkurve auch mit der Dichtefunktion der Normalverteilung gezeichnet werden:

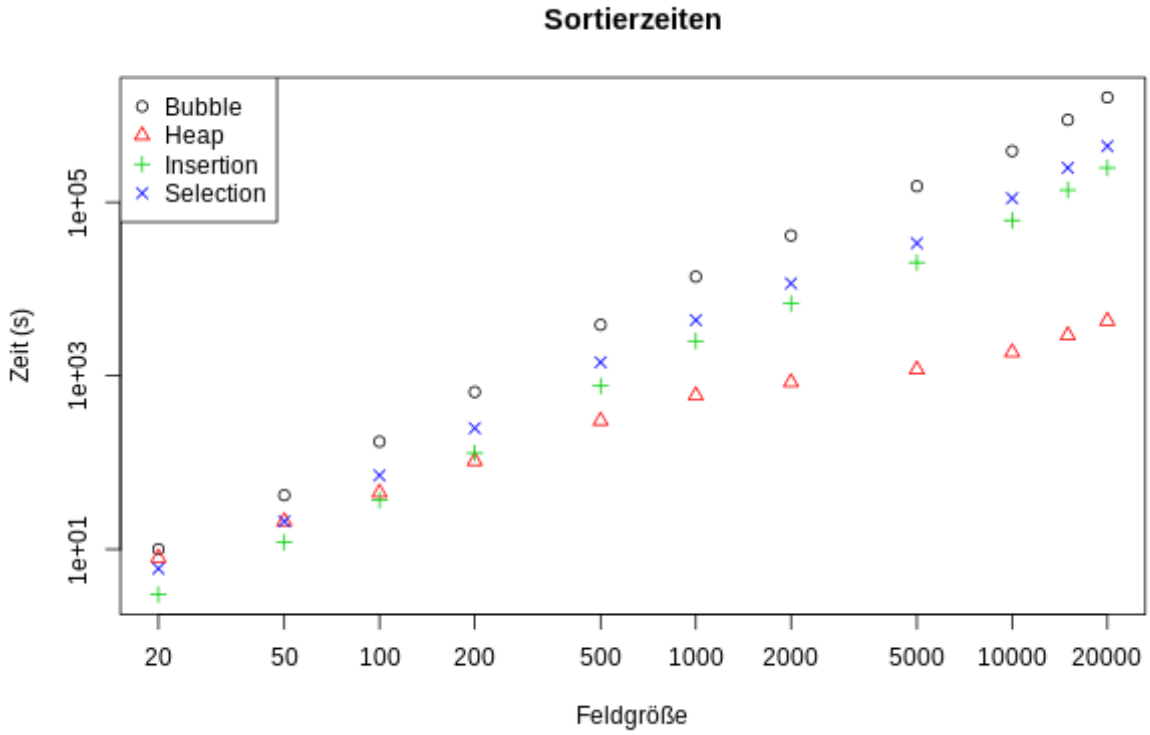


Abb. 8.5: Logarithmische Skalen

`dnorm(`x, mittelwert, varianz`)`

8.7 Balkendiagramme

Für die graphische Darstellung von Daten mit einer Nominal- und einer Verhältnisskala (Person-Punktzahl, Artikel-Preis, …) bieten sich oft auch Balkendiagramme an. Die x-Achse stellt die Nominalskala, die y-Achse die Verhältnisskala dar.

Zum Zeichnen wird zunächst nur der Vektor mit den Verhältnisdaten benötigt:

`barplot(`datenVektor`)`

Beispiel: Diskussionbeiträge Oesterreich-Wahl.

```
wahl <- read.table("oesiwahl.dat", header=T)
names(wahl) <- c("Name", "Talking", "Result")
barplot(wahl$Talking)
```

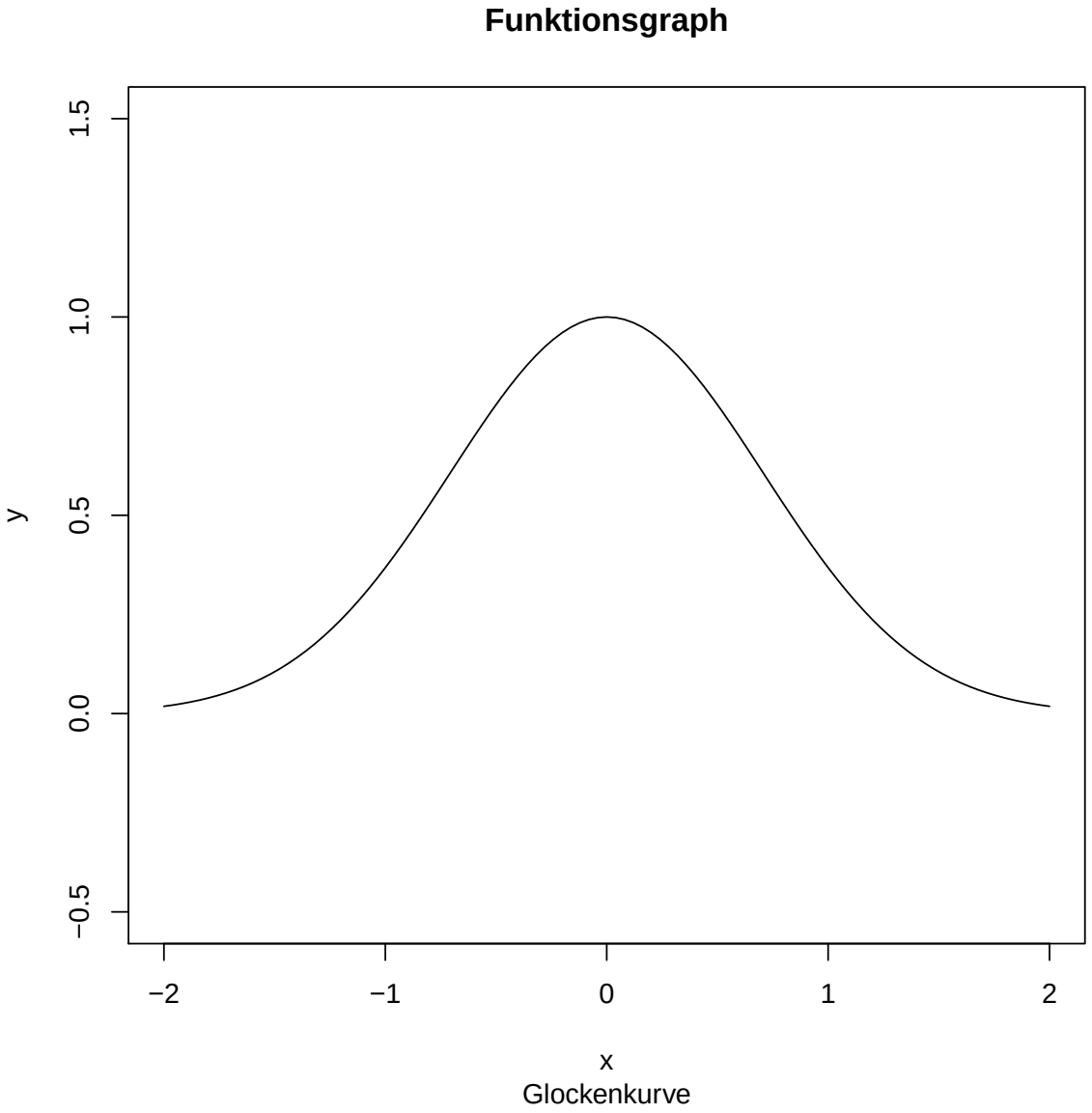
Ergebnis:

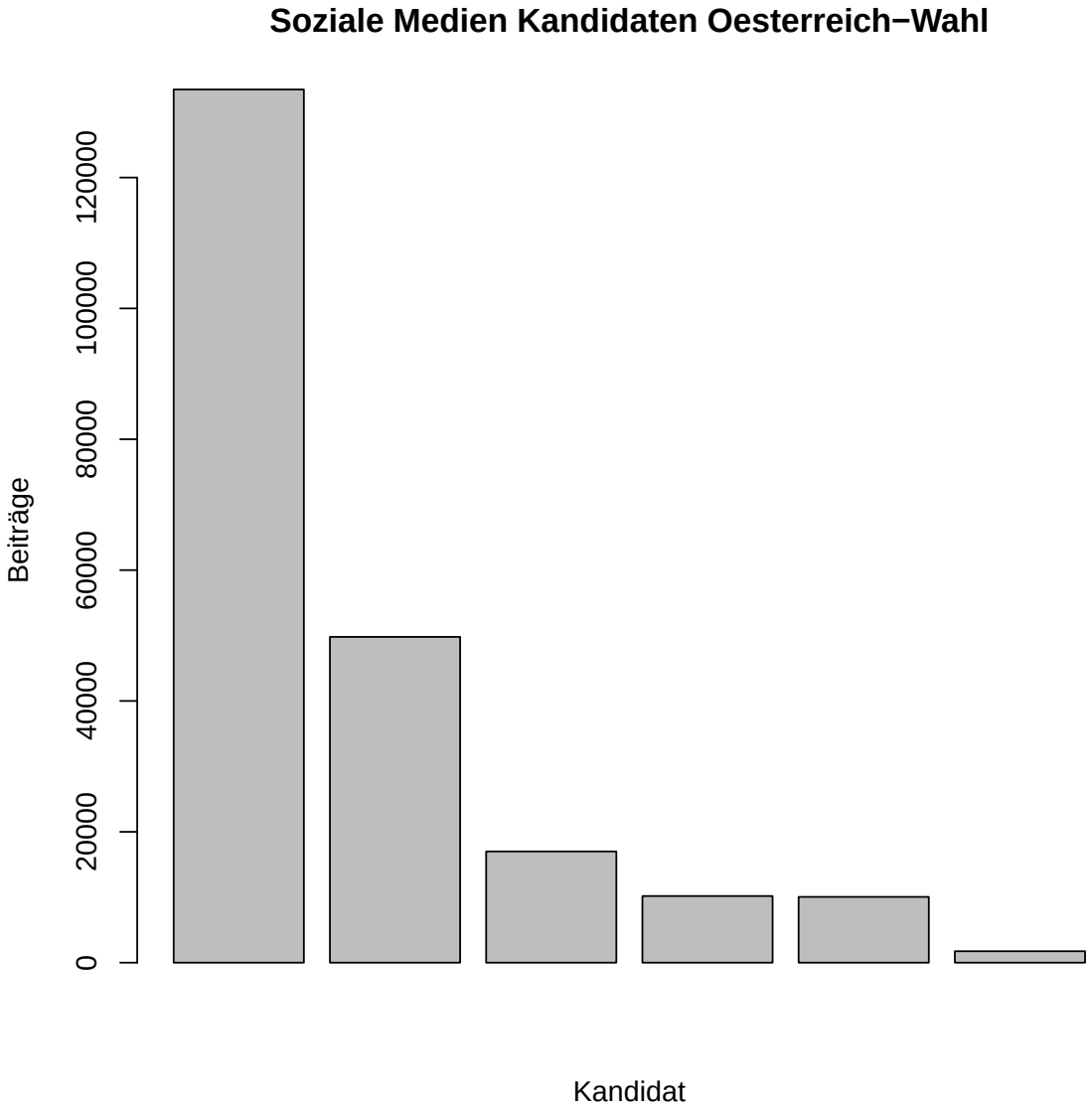
Oft ist es sinnvoll, die Werte vorher zu sortieren.

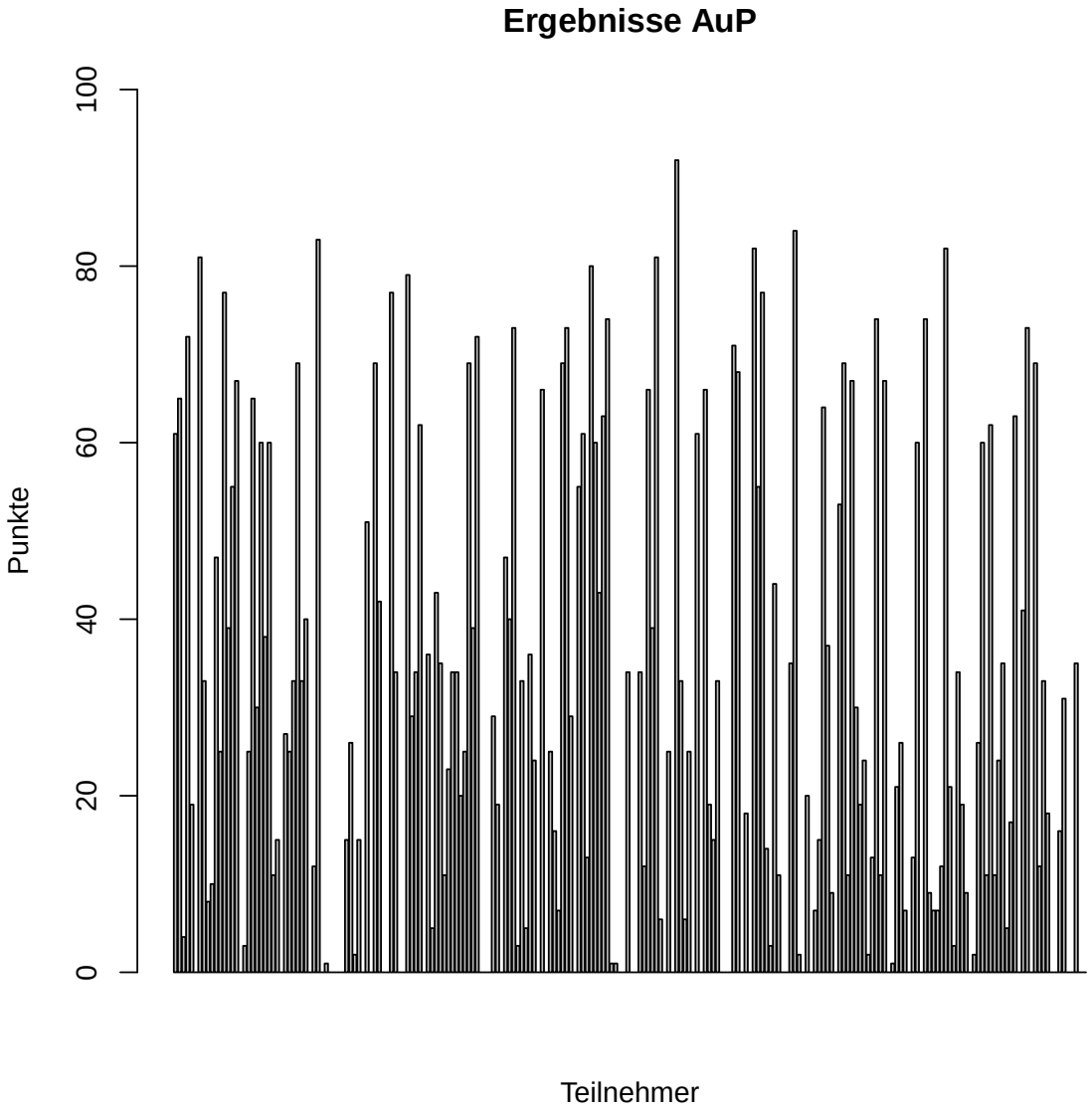
Beispiel: Gesamtpunktzahl AuP-Prüfung ungeordnet.

```
barplot(aup$Summe, ...)
```

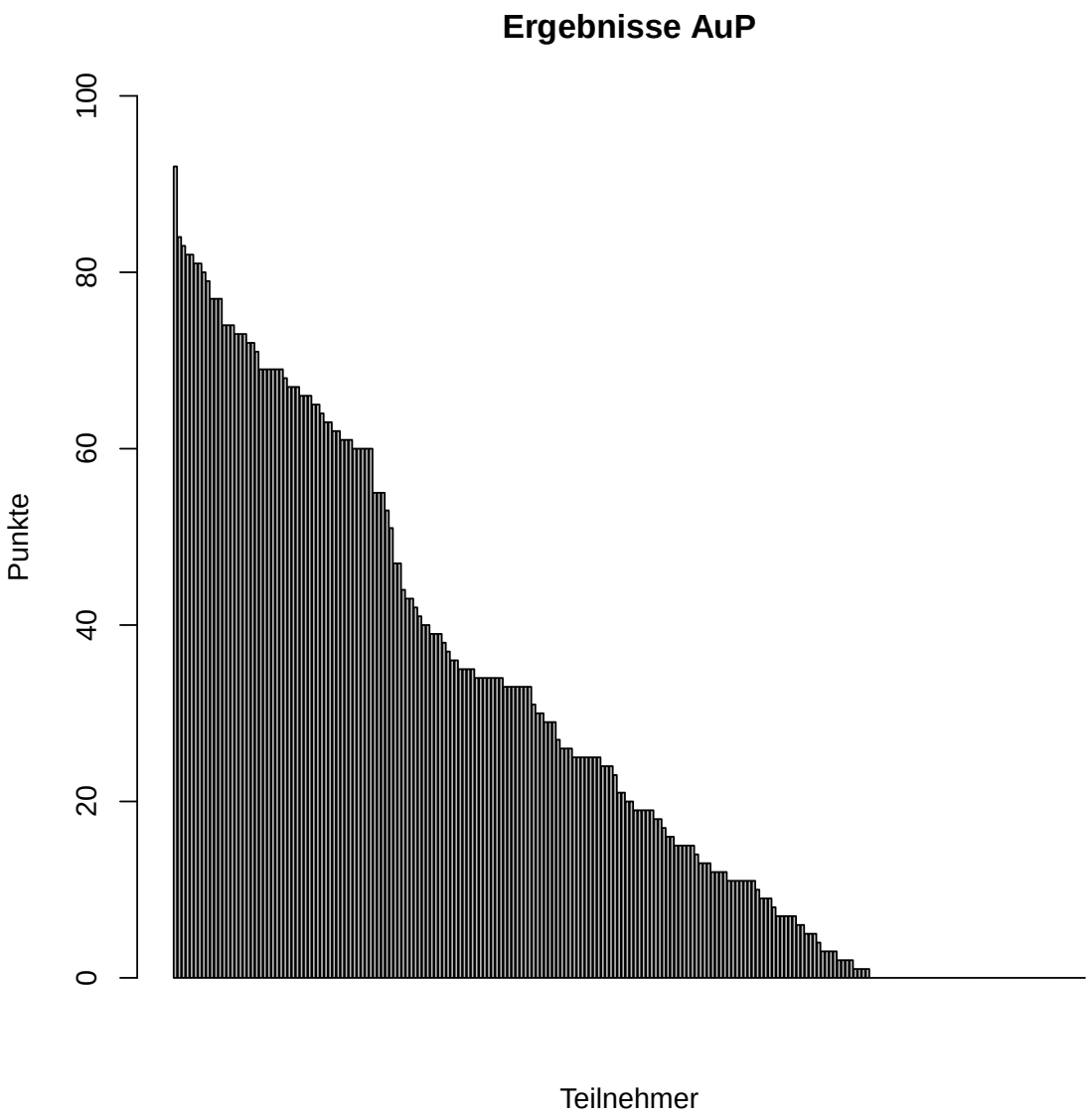
Nach dem Sortieren der Werte wird die Graphik aussagekräftiger:


Funktionsgraph
y
1.5
1.0
0.5
0.0
−0.5
−2
−1
0
1
2
x
Glockenkurve


Soziale Medien Kandidaten Oesterreich−Wahl
Beiträge
0
20000
40000
60000
80000
100000
120000
Kandidat


Ergebnisse AuP
Punkte
0
20
40
60
80
100
Teilnehmer

```
aup.sort <- aup[sort(-aup$Summe),]
barplot(aup.sort$Summe, ...)
```



Ist die Zahl der Balken überschaubar, ist es oft sinnvoll, ihnen Namen zuzuordnen. Das geschieht mit dem Parameter `names.arg`.

barplot(datenVektor**, names.arg=**beschriftungsVektor**)**

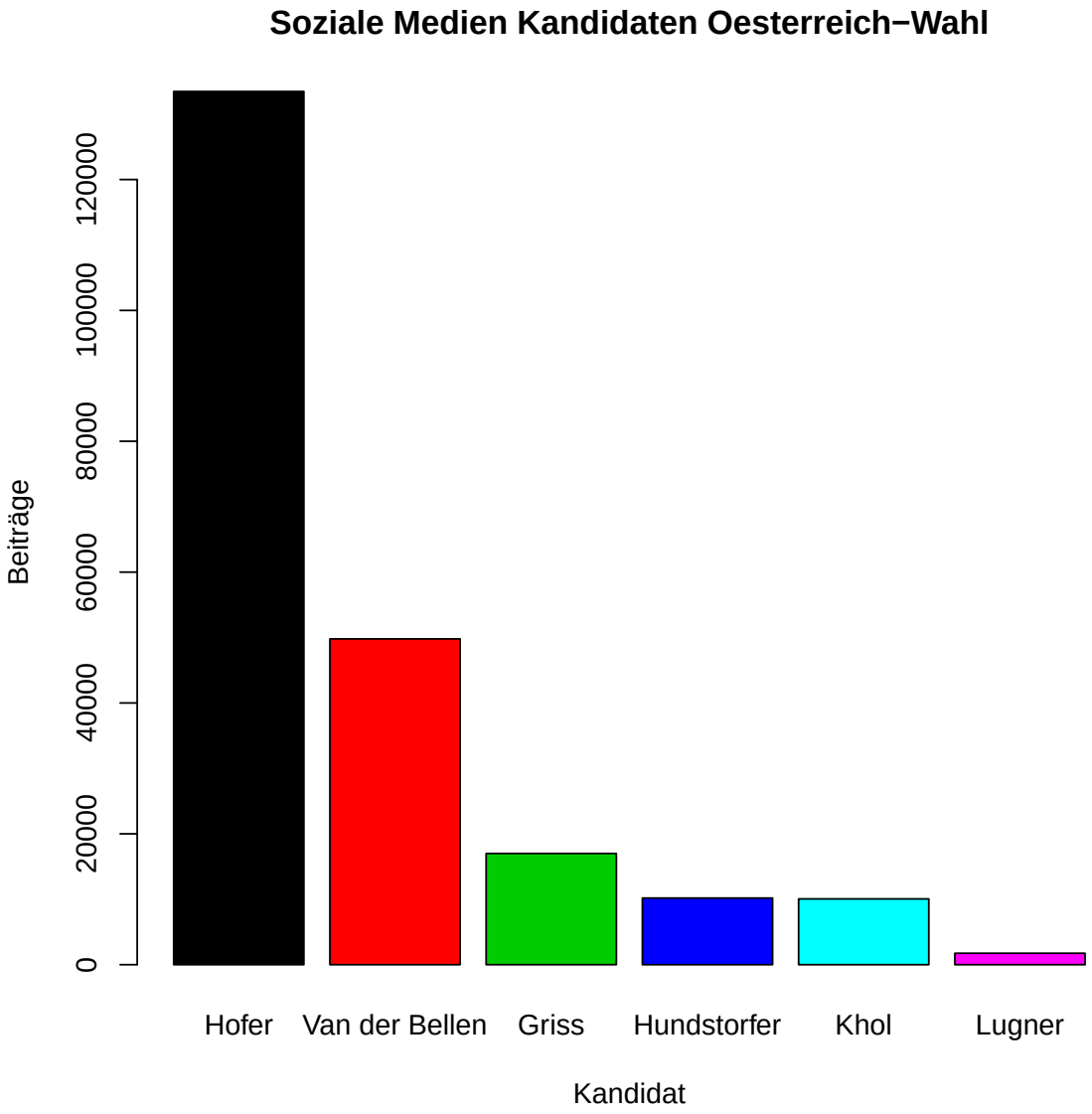
Beispiel: Kandidatennamen der Oesterreichwahl.

```
barplot(wahl$Talking, names.arg=wahl$Name, col=1:length(wahl$Name), ...)
```

Ergebnis:

Wollen wir aus einem Datenframe mehre Spalten als Balkendiagramm nebeneinander darstellen, müssen wir den gewünschen Bereich zunächst in eine Matrix umwandeln. Das ist nur mit Spalten des gleichen Typs möglich!

matrix <- **as.matrix(**frame**[**zeilenwahl**,** spaltenwahl**]**


Soziale Medien Kandidaten Oesterreich−Wahl
Beiträge
0
20000
40000
60000
80000
100000
120000
Hofer
Van der Bellen
Griss
Hundstorfer
Khol
Lugner
Kandidat

Diese Matrix kann nun als Balkendiagramm dargestellt werden, wobei jede Spalte eine Gruppe bildet. Die Werte einer Gruppe werden normalerweise gestapelt. Wollen wir für jeden Wert einen eigenen Balken haben, setzen wir den zusätzlichen Parameter

`beside=T`

Um die Daten vergleichbar zu machen, sollten sie das gleiche Intervall umfassen, z. B. alle relative (prozentuale) Werte darstellen. Wir können dafür neue Spalten im Datenframe anlegen.

```
frame$data.relativ <- frame$data / sum(frame$data)
```

Beispiel: Barplot der Oesterreichwahl, Anteil Beiträge und Wahlergebnis.

```
wahl$talkrel <- wahl$Talking/sum(wahl$Talking)
wahl$resrel <- wahl$Result/sum(wahl$Result)
barplot(as.matrix(wahl[,4:5]), names.arg = c("Beiträge", "Ergebnis"),
        col = 1:l,
        main="Soziale Medien Kandidaten Oesterreich-Wahl",
        xlab="Kandidat", ylab="Beiträge")
```

Die Beschriftung der x-Achse sind nun die geplotteten Gruppen, die Einzelwerte sind zunächst nicht beschriftet.

Die Beschriftung der einzelnen Balkenabschnitte bzw. Balken geschieht am einfachtsten mit dem zusätzlichen Parameter `legend=stringVektor`. Die Balken sollten dazu passend eingefärbt sein.

Beispiel: Barplot der Oesterreichwahl mit Einzelbalken und Legende.

```
...
barplot(as.matrix(wahl[,4:5]), names.arg = c("Beiträge", "Ergebnis"),
        beside=T, legend=wahl$Name,
        col = 1:l,
        main="Soziale Medien Kandidaten Oesterreich-Wahl",
        xlab="Kandidat", ylab="Beiträge")
```

Ergebnis:

8.8 Tortendiagramme

Für die Darstellung von Verhältniszahlen weniger Klassen sind auch Tortendiagramme (*Pie Charts*) geeignet.

pie(datenvektor*[,*labelvektor *...]***)**

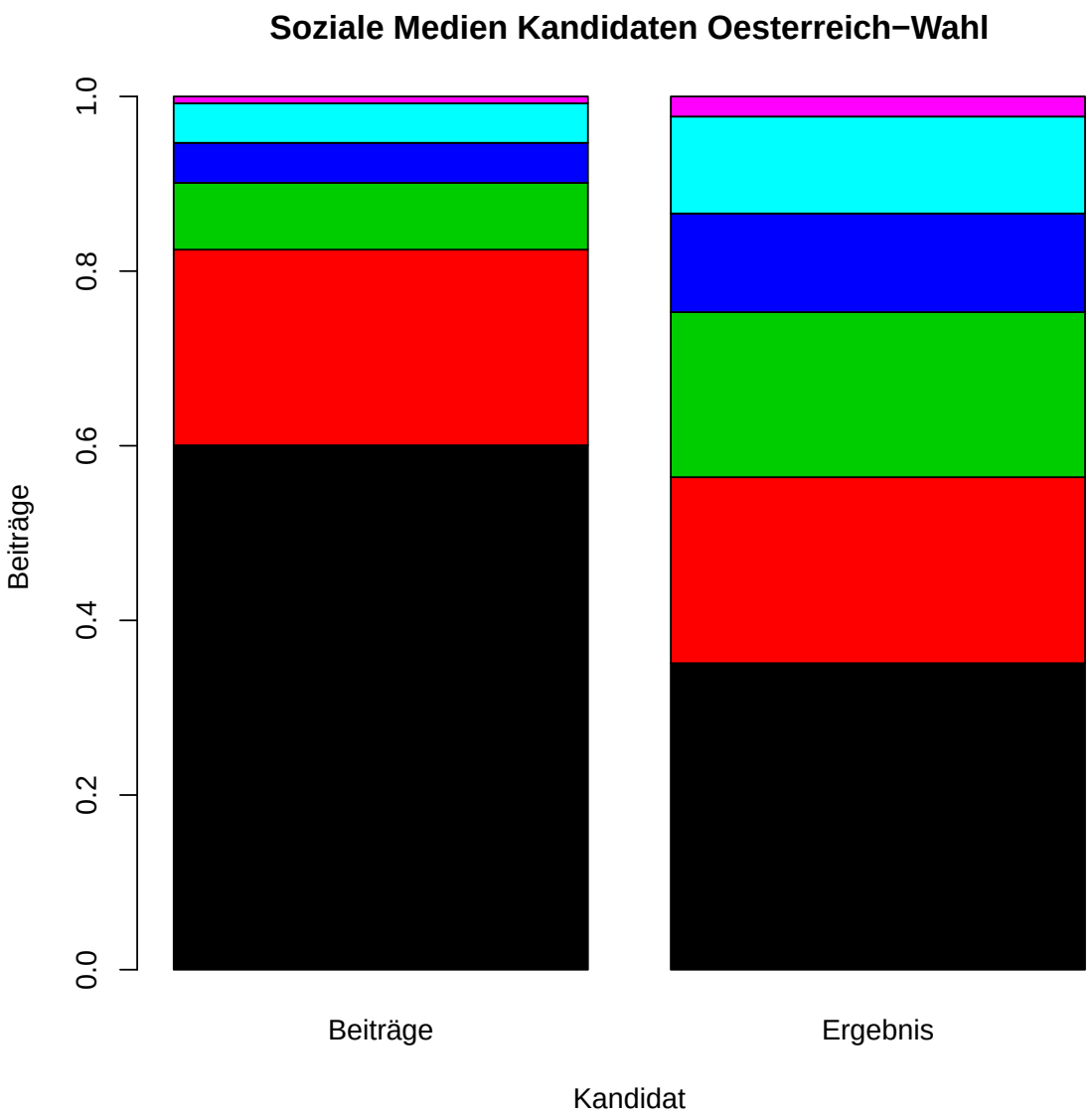
Beispiel: Wahlergebnis Bundespräsident Österreich 2016, 1. Runde.

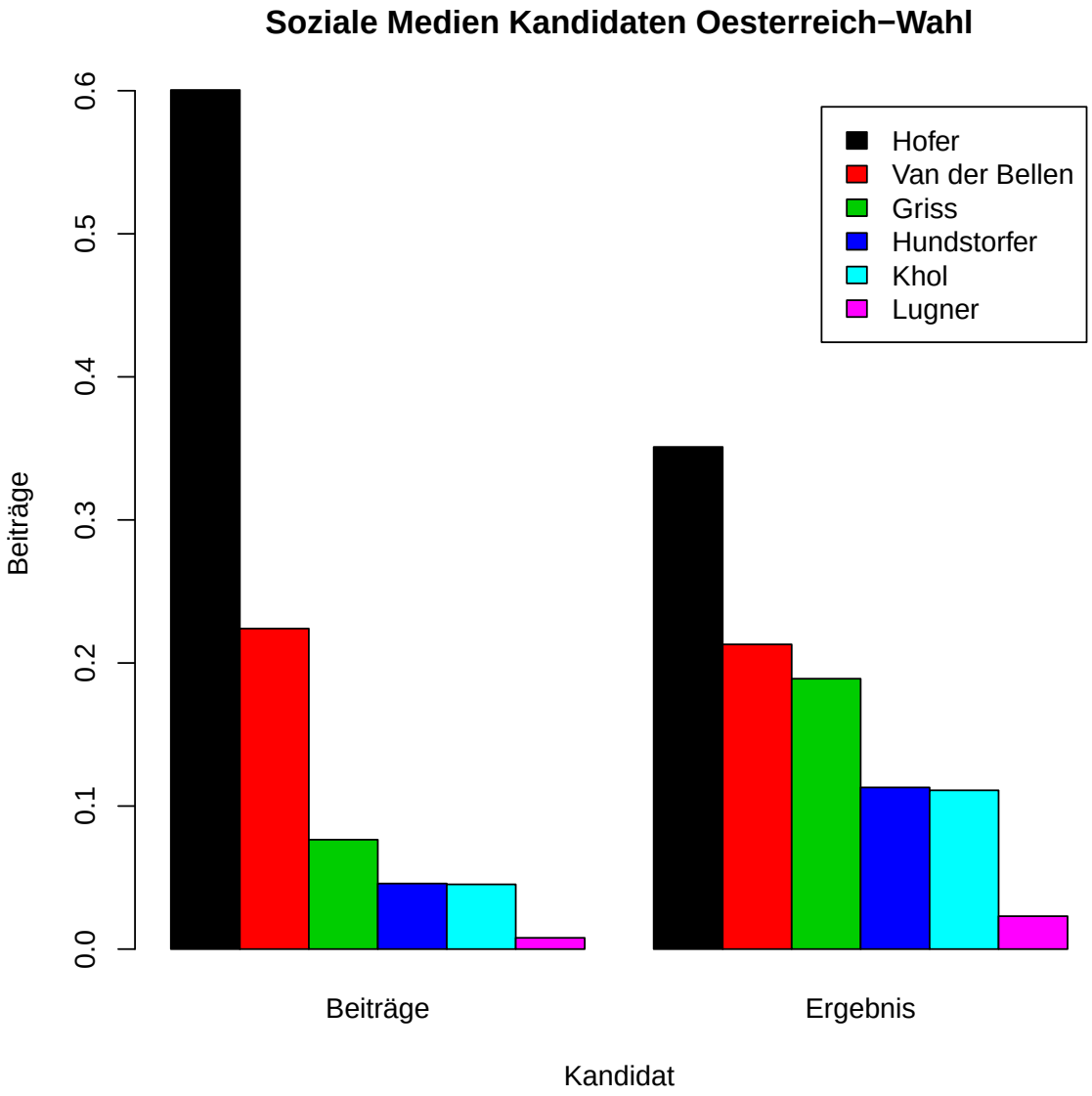
```
wahl <- read.table("samples/testdata/oesiwahl.dat", header=T)
names(wahl) <- c("Name", "Talk", "Result")
pie(wahl$Result, wahl$Name, main="Wahl Bundespräsident 2016")
```

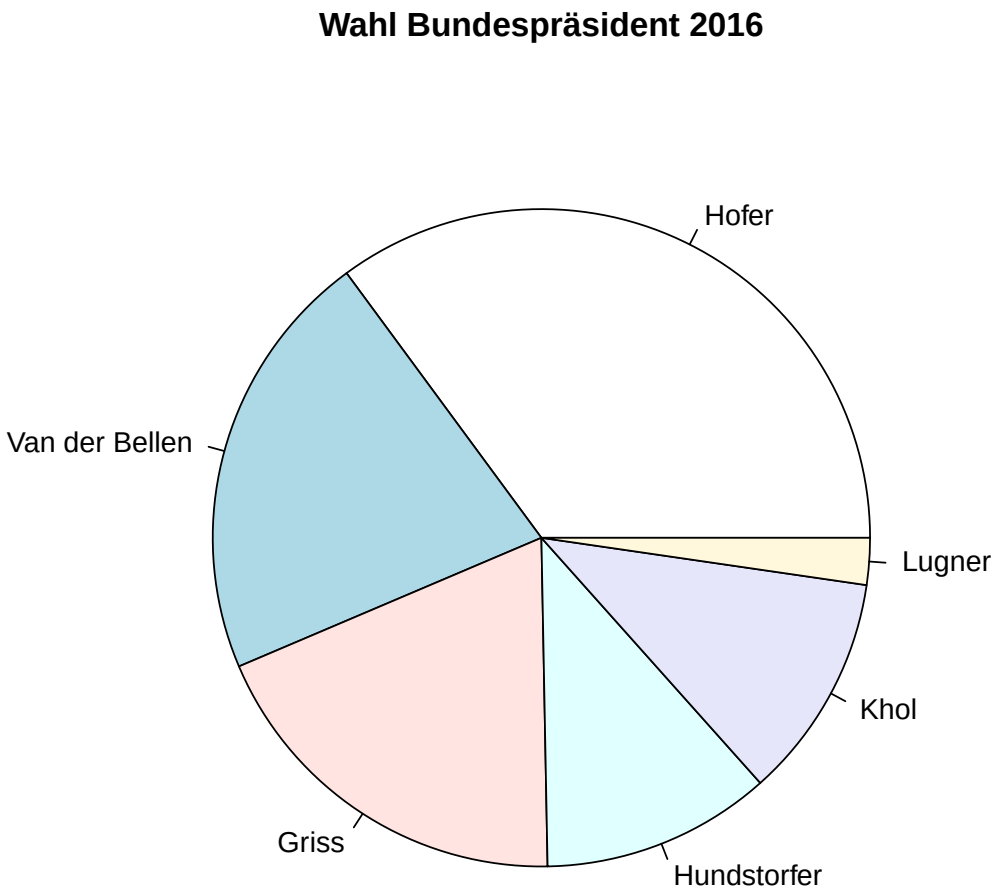
Ergebnis:

Wollen wir neben dem Namen auch das Ergebnis anzeigen, können wir mit der Funktion `paste()` Strings oder Stringvektoren zusammensetzen:

```
neuerStringVektor <- paste(stringVektor, vektorOderWert )
```


Soziale Medien Kandidaten Oesterreich−Wahl
Beiträge
0.0
0.2
0.4
0.6
0.8
1.0
Beiträge
Ergebnis
Kandidat

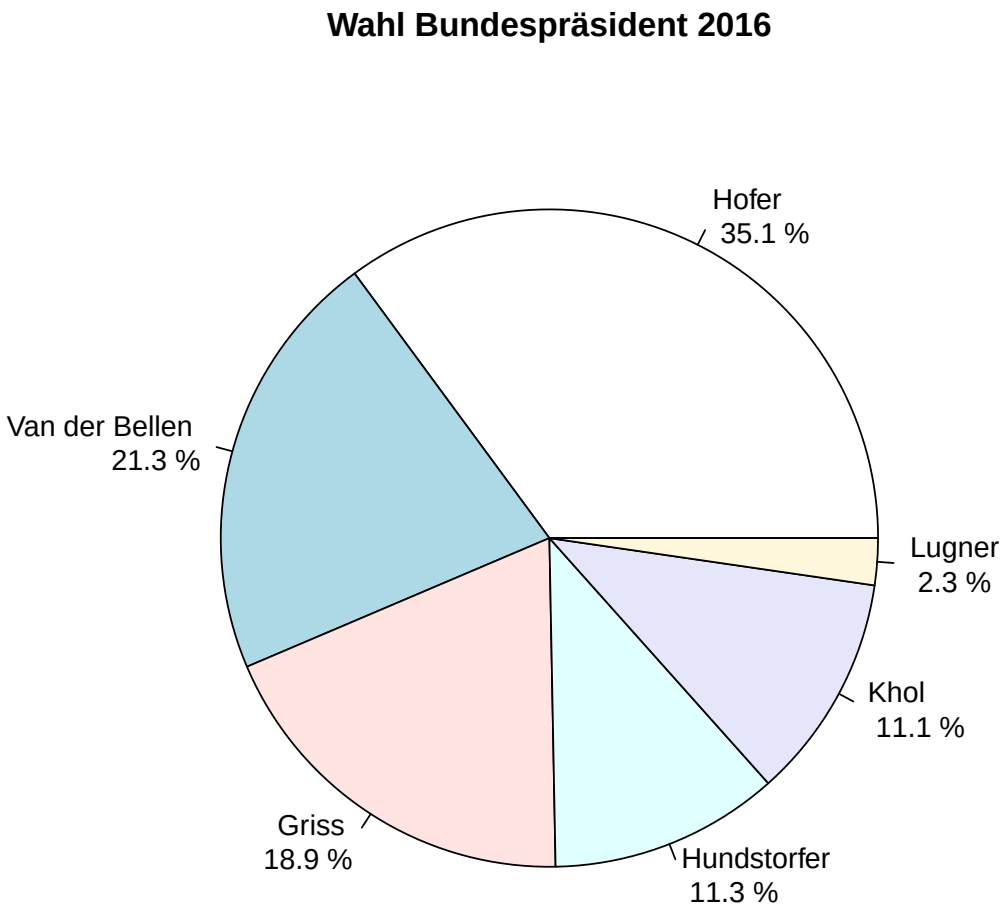

Soziale Medien Kandidaten Oesterreich−Wahl
Beiträge
0.0
0.1
0.2
0.3
0.4
0.5
0.6
Hofer
Van der Bellen
Griss
Hundstorfer
Khol
Lugner
Beiträge
Ergebnis
Kandidat


Wahl Bundespräsident 2016
Hofer
Van der Bellen
Lugner
Khol
Griss
Hundstorfer

und diesen Vektor als Label verwenden. Für unser Tortendiagramm verwenden wir:

```
lab <- paste(wahl$Name, "\n", wahl$Result, "%")
pie(wahl$Result, lab, ...)
```

Ergebnis:



8.9 Histogramme

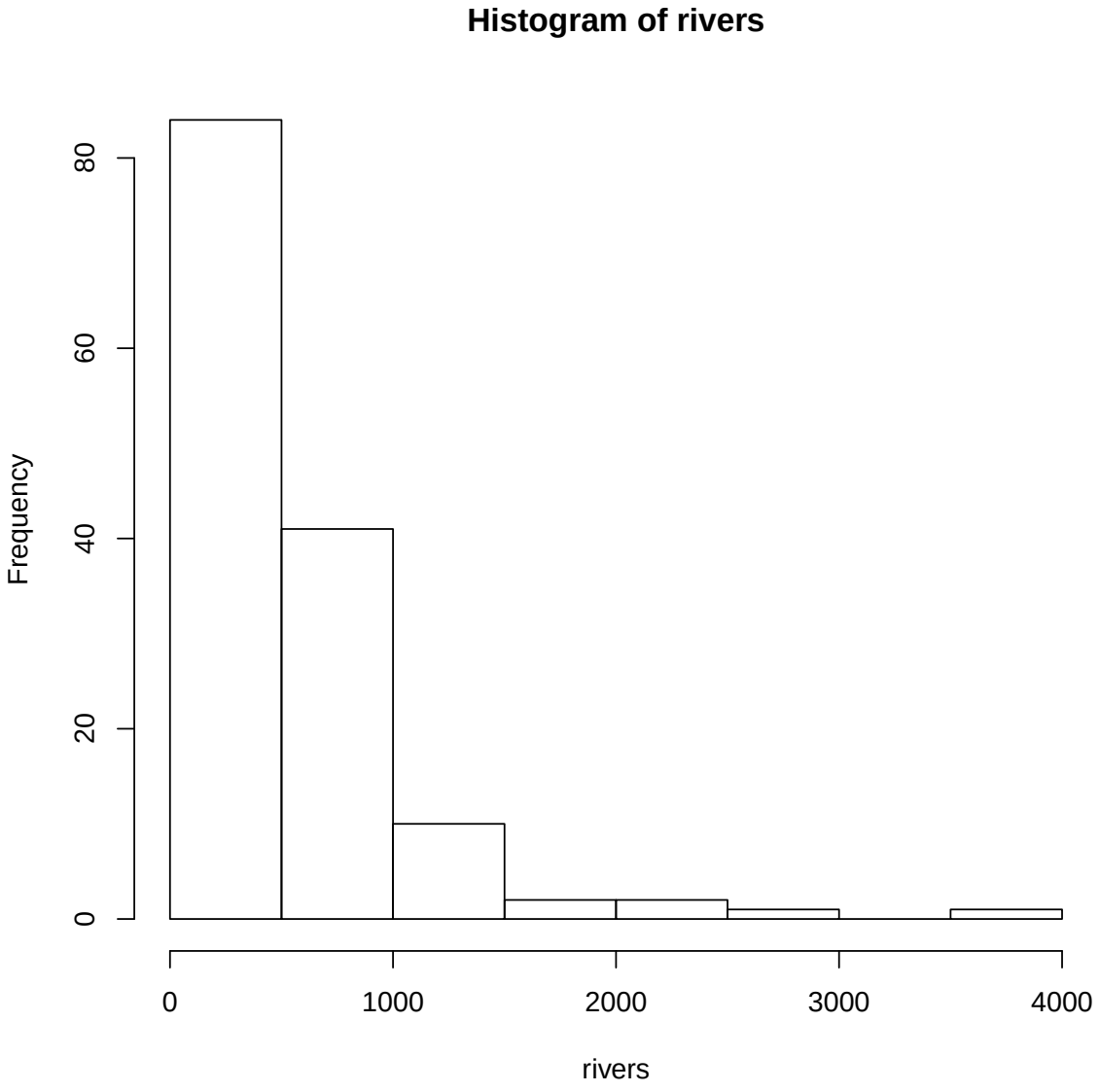
Wollen wir die Verteilung einer metrischen Beobachtung visuell veranschaulichen, bieten sich *Histogramme* an. Der Wertebereich wird dabei entweder automatisch oder manuell in disjunkte Klassen (Teilintervalle) aufgeteilt. Je Klasse wird die Anzahl bzw. Häufigkeit der Werte als Balken dargestellt.

hist(metrischerVektor*[*, **breaks=**klassenZahl *...]***)**

Die Klassenzahl wird entweder automatisch bestimmt oder mit dem Parameter `breaks` selbst festgelegt, der entweder eine einzelne Zahl oder einen Vektor der Klassengrenzen (inkl. linker und rechter Rand) erwartet.

Beispiel: Flusslängen USA.

```
hist(rivers)
```



Feinere Klassenunterteilung, Beschriftung.

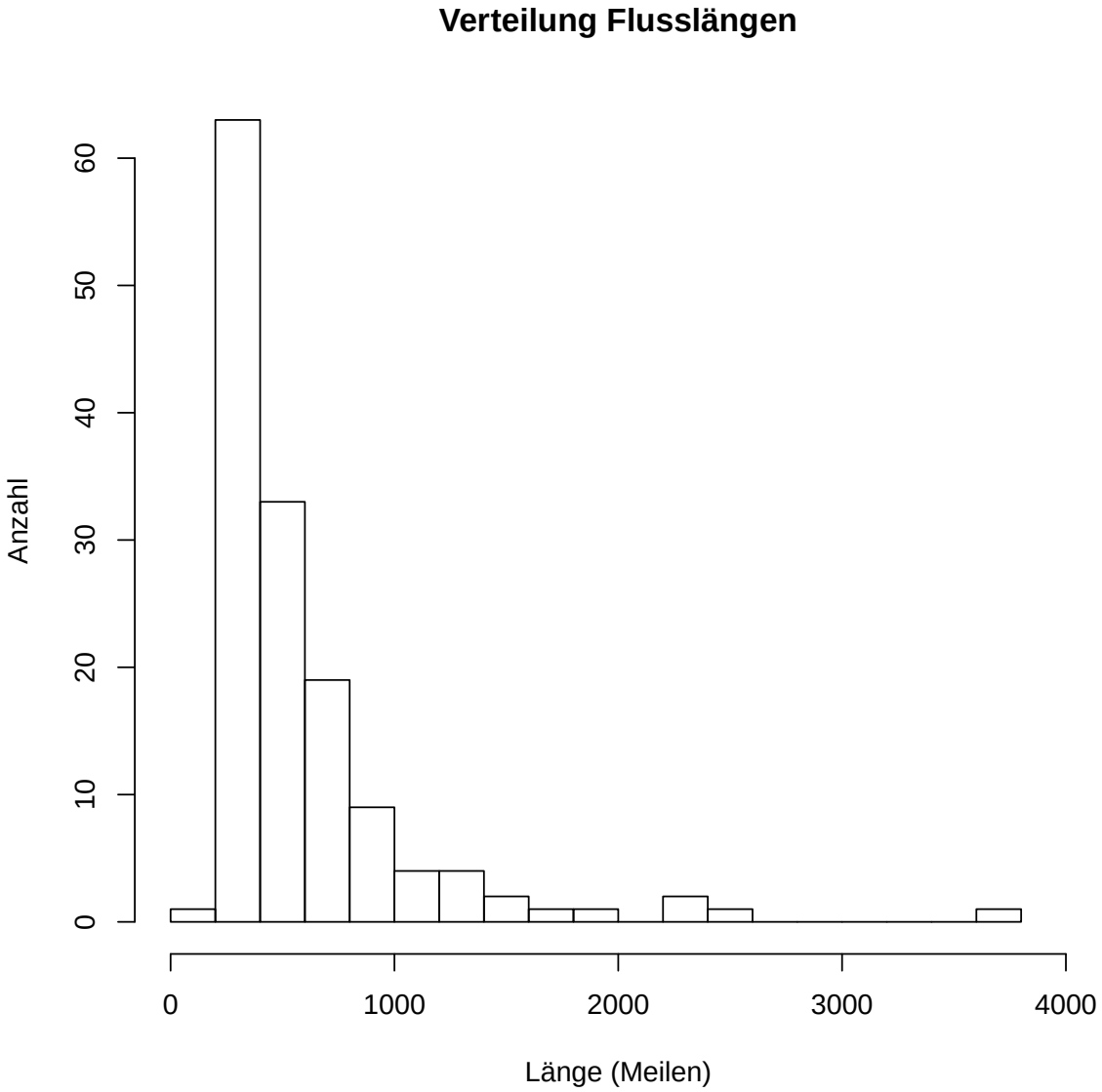
```
hist(rivers, breaks=20, xlim=c(0, 4000), ...)
```

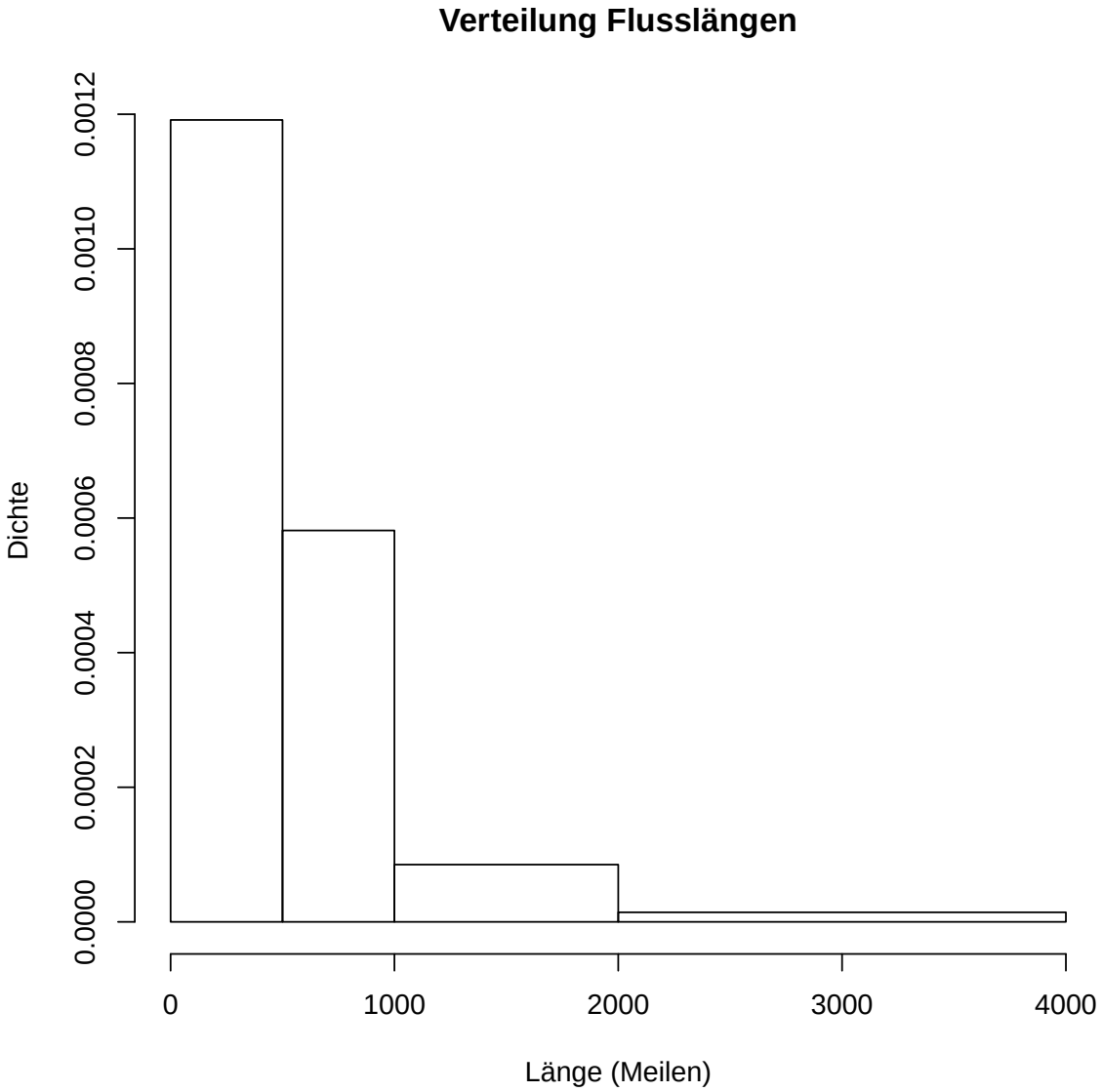
Werden die Klassen unterschiedlich groß gemacht, erhalten wir eine Warnung (wir müssen dazu den Parameter `freq=T` setzen):

```
Die FLÄCHEN im Plot sind falsch -- besser 'freq=FALSE' nutzen
```

Wir befolgen deshalb diesen Hinweis und setzen den Parameter `freq=F`. Statt der absoluten Zahl wird nun der Wert der Dichtefunktion angezeigt.

```
hist(rivers, breaks=c(0, 500, 1000, 2000, 4000), freq=F, ...)
```

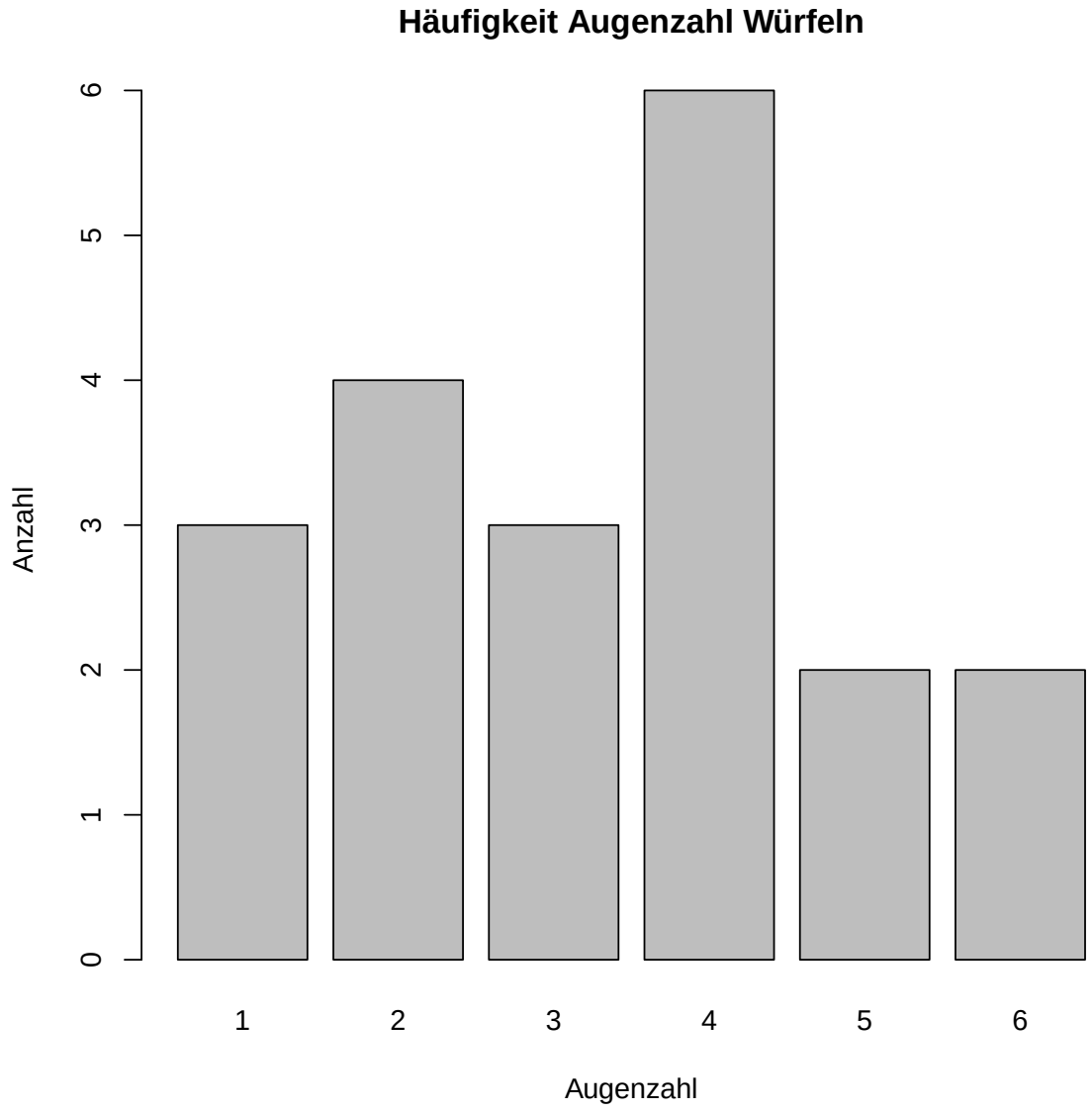

Verteilung Flusslängen
Anzahl
0
10
20
30
40
50
60
0
1000
2000
3000
4000
Länge (Meilen)


Verteilung Flusslängen
Dichte
0.0000
0.0002
0.0004
0.0006
0.0008
0.0010
0.0012
0
1000
2000
3000
4000
Länge (Meilen)

Warnung: Die Funktion `hist()` ist nicht geeignet, um Histogramme von Daten mit Nominalskalen anzuzeigen (Anzahl Augenzahlen beim Würfeln, Wochentag für Beginn Krankheit, …). Hier verwenden wir die Funktion `table()` zum Gruppieren und `barplot()` zum Darstellen der Daten.

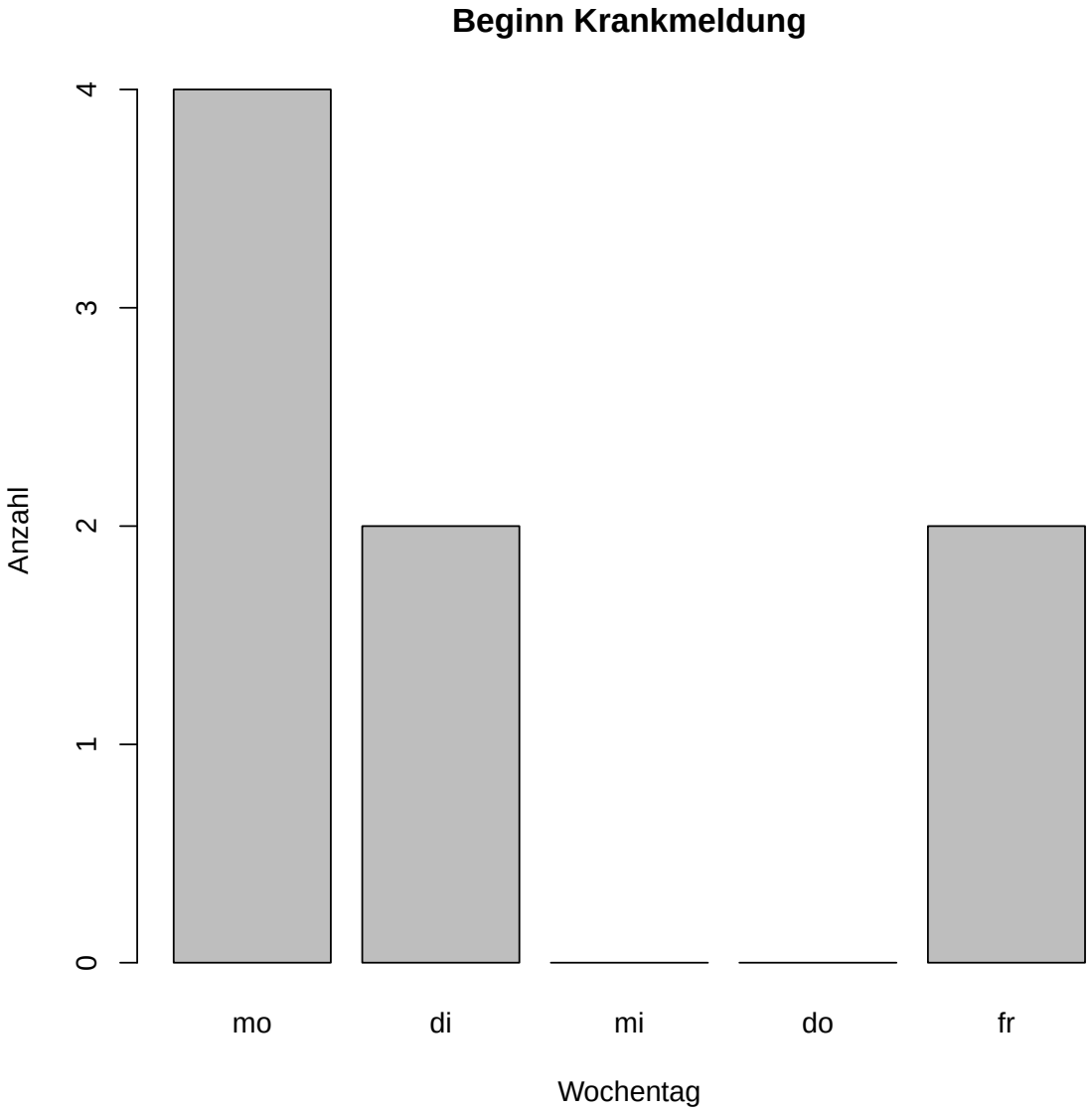
Beispiel: Verteilung Augenzahl beim 20-maligen Würfeln.

```
dice <- sample(1:6, 20, replace=T)
barplot(table(dice), ...)
```



Beispiel: Wochentag Krankheitsbeginn.

```
days <- c("mo", "fr", "di", "mo", "di", "mo", "fr", "mo")
fdays <- factor(days, c("mo", "di", "mi", "do", "fr"))
barplot(table(fdays), main="Krankheitsbeginn")
```


Beginn Krankmeldung
Anzahl
0
1
2
3
4
mo
di
mi
do
fr
Wochentag

8.10 Boxplots

Boxplots dienen zum Auswerten metrischer Daten. Wir sehen eine Box, die 50% der Daten vom 1. bis zum 3. Quantil einschließt. Diese Box wird durch den Median geteilt. Nach unten und oben zeigen sogenannte *Whiskers*, die die Mehrzahl der Daten einschließen, und Ausreißer außerhalb dieses Bereichs.

Ein Boxplot für einen metrischen Vektor wird wie folgt erzeugt:

> **boxplot(**datenvektor*[...]***)**

Der Mittelwert wird nicht automatisch angezeigt, er kann jedoch leicht ergänzt werden (s. a. nächstes Kapitel):

> **points(median(**datenvektor*[,* **pch=**symbolNummer*]***)**

Beispiel: Flusslängen der 30 längsten Flüsse der USA.

```
riv30 <- head(rivers[order(-rivers)], 30)
boxplot(riv30,main="Flusslängen", ylab="Länge (km)")
points(mean(riv30), pch=16)
legend("topright", legend=c("Ausreißer", "Mittelwert"), pch=c(1, 16))
```

Ergebnis:

Sinnvoll werden Boxplots häufig dann, wenn wir verschiedene Gruppen miteinander vergleichen. Einfach, aber nur selten sinnvoll, können wir verschiedene Spalten eines Datenframes miteinander in Beziehung setzen. Dann sollten aber alle Spalten etwa den gleichen Wertebereich umfassen (*identische Verteilung* besitzen), um eine interpretierbare Aussage ableiten zu können.

Beispiel: Punktzahlen der Prüfungsteilnehmer bei verschiedenen Aufgaben. Die erreichbare Punktzahl war allerdings nicht identisch! Um die Mittelwerte jeder Aufgabe zu berechnen, benutzen wir eine Schleife, um Tipparbeit zu sparen (s. u.).

```
aup <- read.table("samples/testdata/erg-aup2015.dat", header=T)
# NAs bereinigen
aup[is.na(aup)] <- 0
boxplot(aup[,3:9], xlab="Aufgabe", ylab="Punkte",
  main="Ergebnisse nach Aufgabe")
# Mittelwertvektor der gewünschten Länge anlegen, Werte werden überschrieben
aup.means <- rep(0,9)
for (i in 3:9) { aup.means[i] <- mean(aup[,i]) }
points(aup.means[3:9], pch=16)
legend("topright", legend=c("Ausreißer", "Mittelwert"), pch=c(1, 16))
```

Interessanter ist häufiger die Gruppierung nach einem Faktor. Wir geben dann statt eines Datenframes oder Vektors eine Formel an:

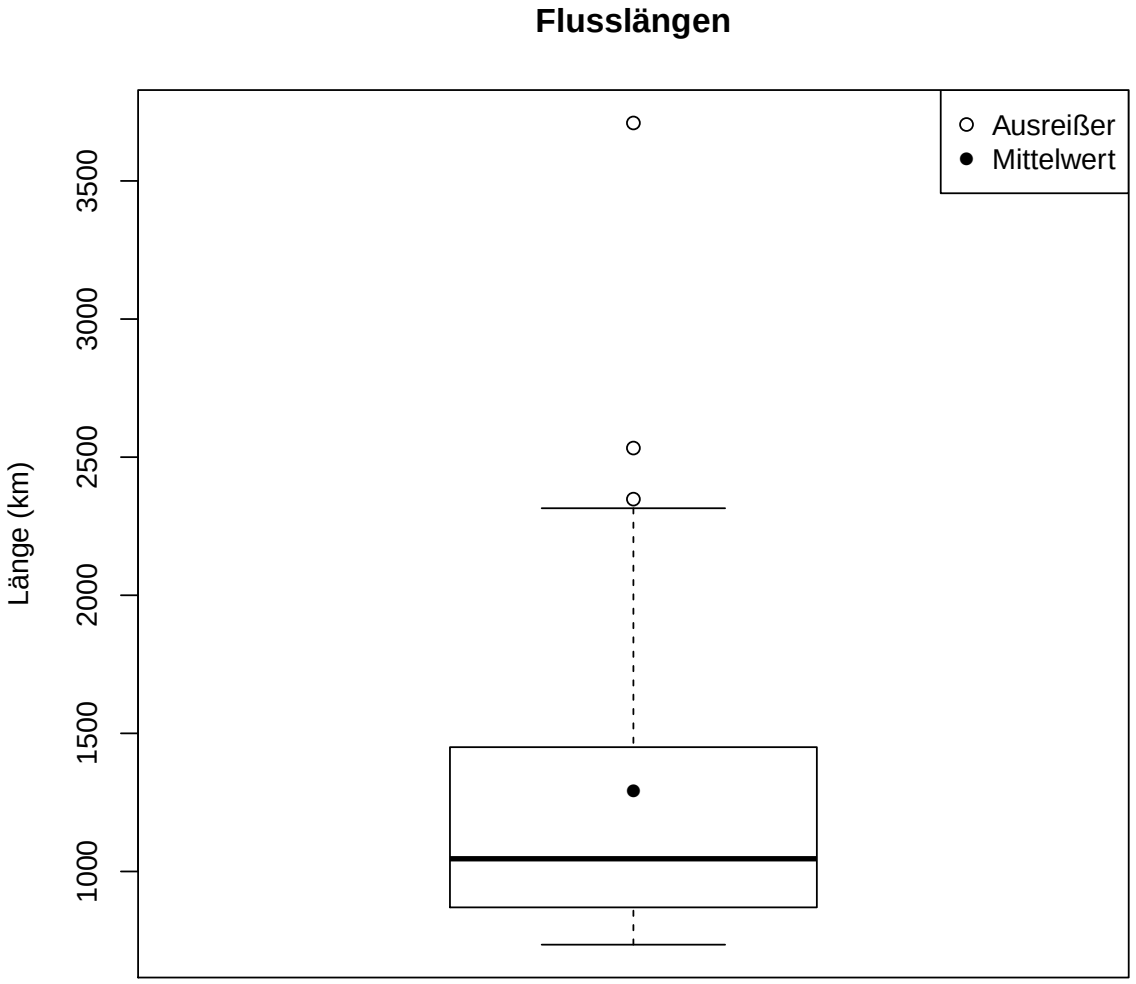
> metrischerVektor **~** faktorVektor

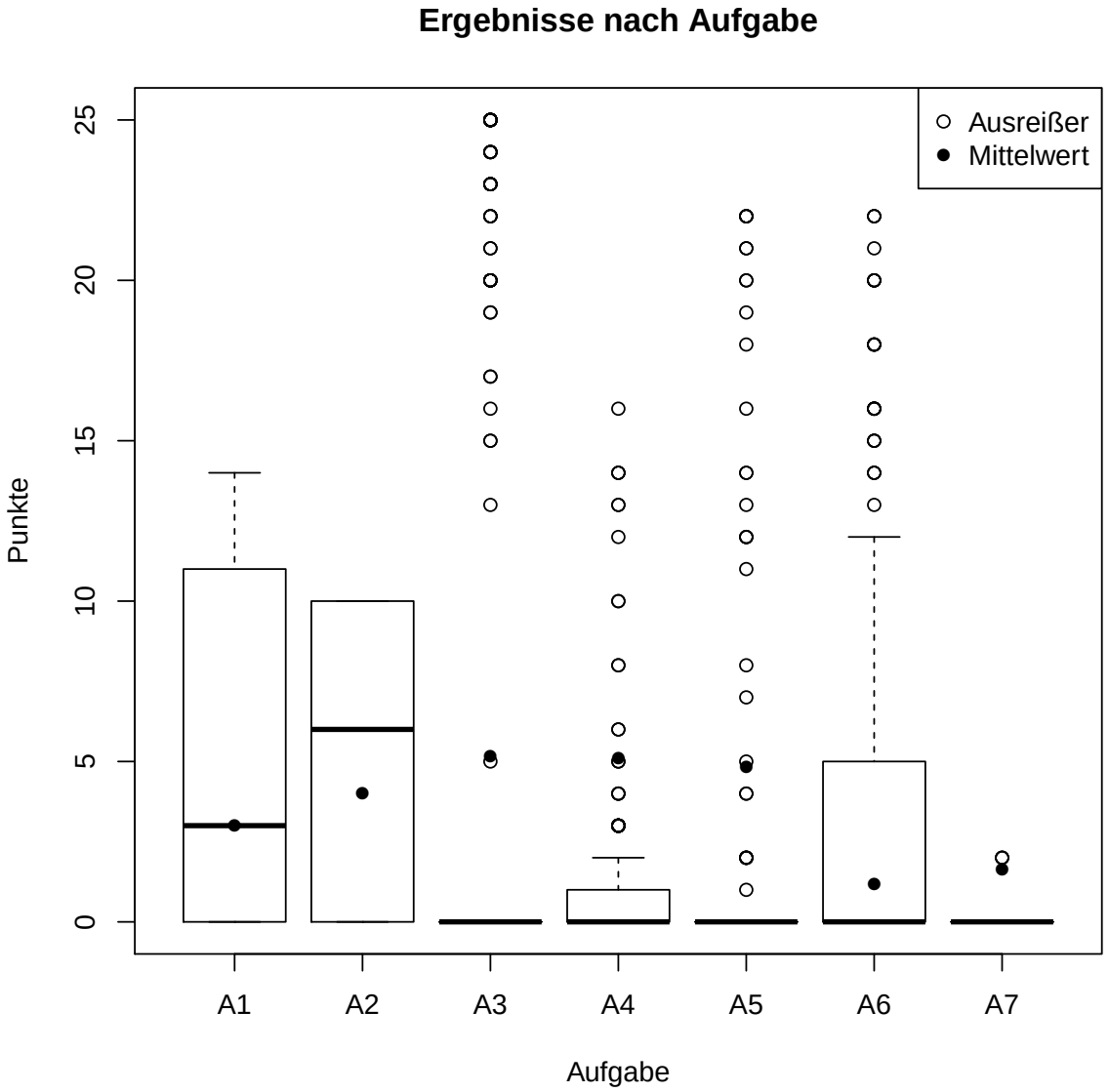
Um hier ebenfalls die Mittelwerte ergänzen zu können, müssen wir diese je Gruppe berechnen, was mit der Funktion `aggregate()` möglich ist:

> aggregatFrame <- **aggregate(**metrischerVektor **~** faktorVektor**, FUN=**funktionsName**)**

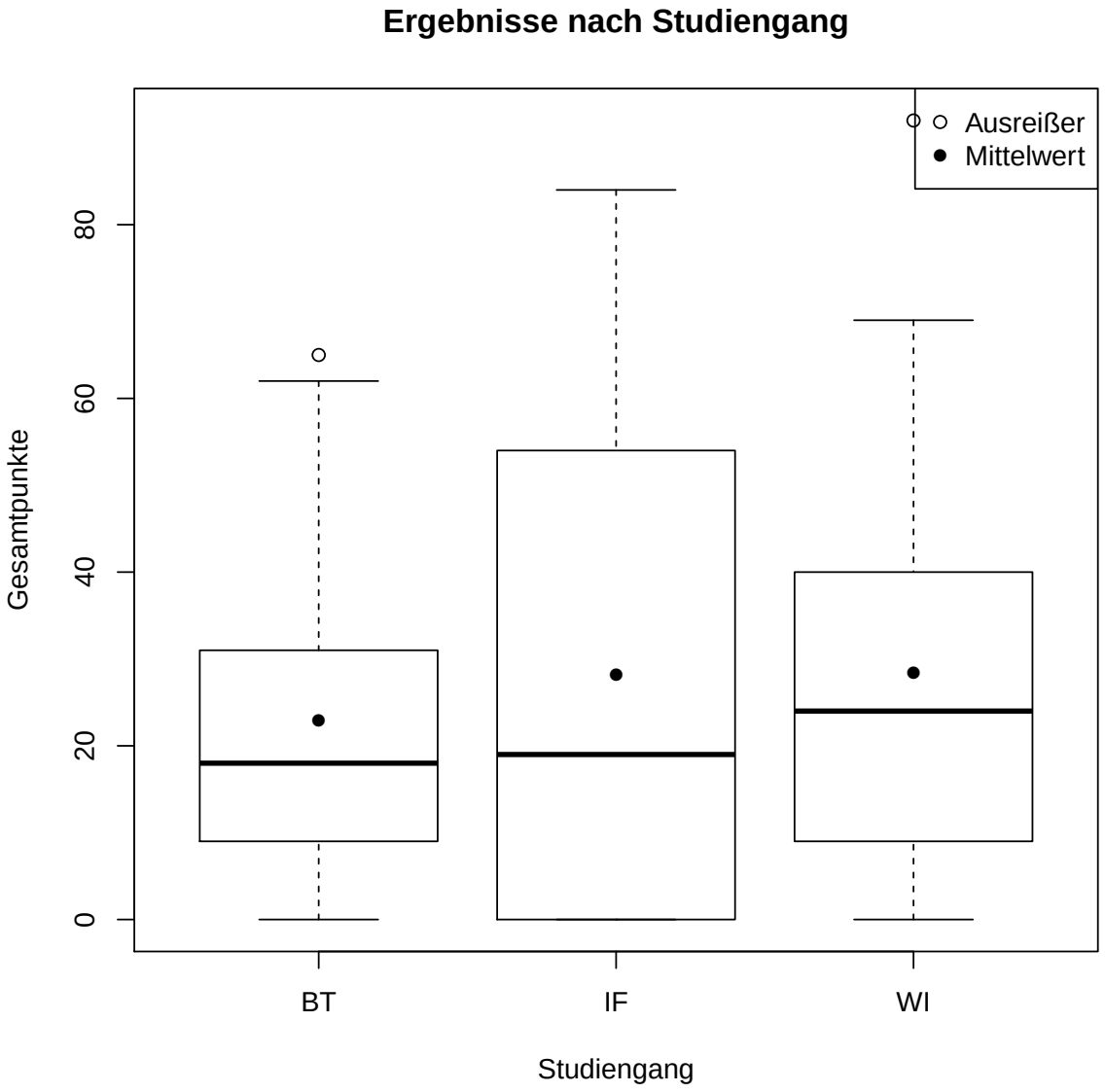
Hier verwenden wir `mean` als Funktion.

Beispiel: Punktzahlen nach Studiengang (Nominalskala, beim Einlesen in Faktor umgewandelt):


Flusslängen
Ausreißer
Mittelwert
Länge (km)
1000
1500
2000
2500
3000
3500


Ergebnisse nach Aufgabe
Ausreißer
Mittelwert
Punkte
0
5
10
15
20
25
A1
A2
A3
A4
A5
A6
A7
Aufgabe

```
aup <- read.table("samples/testdata/erg-aup2015.dat", header=T)
# NAs bereinigen
aup[is.na(aup)] <- 0
# Gruppierungsformel
aup.groupSum <- aup$Summe ~ aup$Studg
boxplot(aup.groupSum, xlab="Studiengang", ylab="Gesamtpunkte",
  main="Ergebnisse nach Studiengang")
# Mittelwerte je Gruppe
aup.means <- aggregate(aup.groupSum, FUN=mean)
points(aup.means, pch=16)
legend("topright", legend=c("Ausreißer", "Mittelwert"), pch=c(1, 16))
```



8.11 Dichteplots

Eine alternative Darstellung der Wertestreuung oder -verteilung einer metrischen Größe ist ein Dichteplot. Hier steht keine direkte Plot-Funktion zur Verfügung, sondern wir berechnen zunächst die geschätzte Dichtefunktion mit `density()`. Deren Ergebnis können wir einfach mit `plot()` darstellen:

```
dens <- density(x)
plot(dens)
```

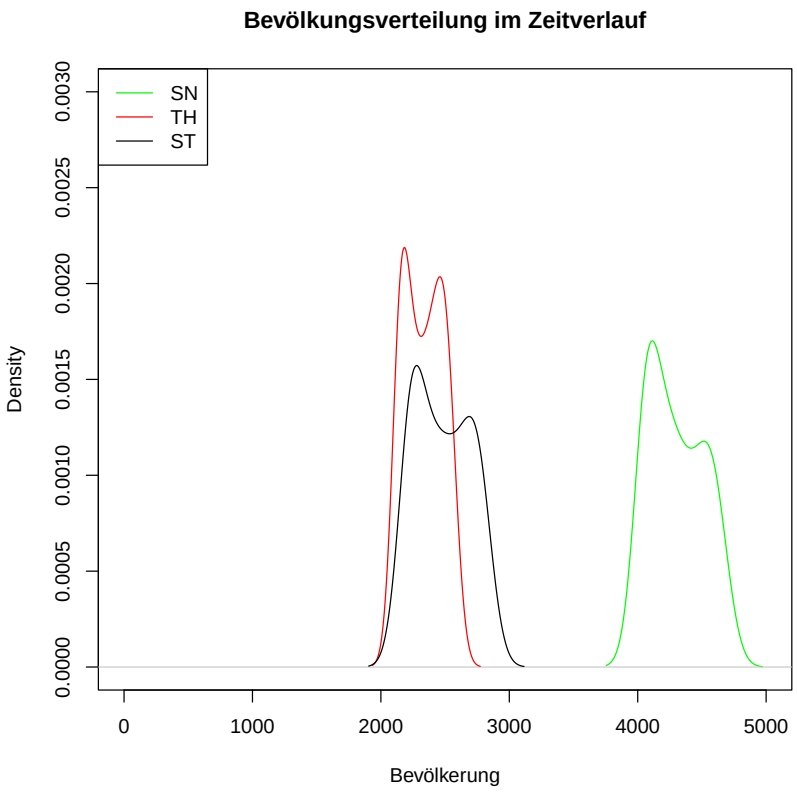
Wollen wir in diesem Diegramm eine weitere Dichtefunktion darstellen, verwenden wir den Graphikbefehl `lines()`.

```
dens2 <- density(x2)
lines(dens2)
```

Beispiel: Bevölkerungsverteilung verschiedener Bundesländer im Zeitverlauf. Datendatei: `bevoelkerung.csv`. Wir transponieren diese Datei zunächst, damit die Zeilen die Jahre und die Spalten die Bundesländer darstellen.

```
data <- read.table('bevoelkerung.csv', header=T, sep=';', dec=',')
data2 <- as.data.frame(t(data[,2:ncol(data)]))
names(data2) <- data[,1]

plot(density(data2$Sachsen), xlim=c(0, 5000), ylim=c(0,3e-3), col='green',
   main='Bevölkungsverteilung im Zeitverlauf', xlab='Bevölkerung')
lines(density(data2$Thueringen), col='red')
lines(density(data2$SachsenAnhalt))
```

Ergebnis:



Wir sehen hier deutlich detaillierter die Schwankungen der Bevölkerungszahl als im Boxplot, der uns nur Median und Quantile darstellt. Bei der Interpretation müssen wir beachten, dass die x-Achse *keinen* zeitlichen Verlauf, sondern lediglich die Bevölkerungszahl abbildet, wir erhalten hier also keine Informationen über eine zeitliche Entwicklung.

Download Notebook: `density-inhab.ipynb`.

8.12 Aufgaben

Aufgabe 8.2 Untersuchen Sie graphisch, ob ein Zusammenhang zwischen der Punktzahl der Hausaufgaben und der Gesamtpunktzahl der Klausur in AuP erkennbar ist.

Aufgabe 8.3 Erzeugen Sie ein Streudiagramm, dass die Buchstabenhäufigkeiten der englischen Sprache in Bezug auf die deutsche Sprache darstellt. Benutzen Sie dazu die bereits analysierten Testtexte und die relativen Buchstabenhäufigkeiten.

Erzeugen Sie weiterhin ein Histogramm für die Häufigkeit der deutschen Buchstaben. Diese soll absteigend sortiert dargestellt werden.

Hinweis: Die im folgenden Kapitel beschriebenen Diagrammerweiterungen wie z. B. das Beschriften der Datenpunkte dürfen verwendet werden.

KAPITEL 9

Ergänzungen in Plots

9.1 Erzeugen von Graphikdateien

Die *IDE RStudio* erlaubt es uns, aus dem Graphikfenster die Graphikdateien zu exportieren. Benötigen wir die Graphiken für Veröffentlichungen, bietet es sich an, auch diesen Export in R-Scripte einzubauen, um diesen reproduzierbar zu gestalten und bei Änderungen auch die Graphiken automatisch zu aktualisieren. Wir öffnen dazu ein passendes Graphik-Device durch Angabe eines Dateinamens und weiterer Parameter:

```
pdf("filename")
png("filename"[, width=breite][, height=höhe])
svg("filename")
```

Alle Zeichnungen erfolgen nun in diese Datei. Sind wir fertig, schließen wir diese.

```
dev.off()
```

Alle folgenden Zeichnungen erscheinen nun wieder in unserem Graphikfenster der *IDE*.

9.2 Manuelle Diagrammerstellung

Wollen wir einen Plot komplett mittels Low-Level-Funktionen zeichnen, funktioniert dies z. B. mit folgendem Konstrukt:

plot(NA,xlim=c(xmin,xmax**), ylim=c(**ymin,ymax**)***[*, ann=F, xaxt='n', yaxt='n'*]***)

Die automatischen Achsenteilungen lassen sich mit den Parametern `xaxt='n'` bzw. `yaxt='n'` abschalten, was immer dann nützlich ist, wenn die automatisch erzeugte Achseneinteilung nicht sinnvoll erscheint. Die Achsen können dann manuell mit der Funktion `axis()` erzeugt werden. Meist müssen dann auch die automatischen Achsen- und Diagrammbeschriftungen weggelassen werden, da die Abstände zwischen Achsenteilung und -beschriftung zu klein sind. Dies geschieht mit dem Parameter `ann=F`.

Eine Achse wird manuell mit der Funktion `axis()` erzeugt.

axis(richtung**, at=**tickPositionen***[*, labels=**beschriftungsVektor*][*las=**orientierung*]***)

Der erste Parameter gibt die Seite an (1 = unten, 2 = links, 3 = oben, 4 = rechts). Der Parameter `at` erwartet einen Vektor der Tick-Positionen, bezogen auf den Wertebereich von x bzw. y. Mit `labels` kann eine alternative Beschriftung gesetzt werden. Dieser Vektor muss die gleiche Länge wie der `at`-Parameter haben. Der Parameter `las` steuert die Ausrichtung der Beschriftung: 0 bedeutet parallel, 1 senkrecht zur Achse. Dieser Parameter kann auch in der `plot()`-Funktion mit den vorgegebenen Achsteilungen und den Werten 0 bis 3 verwendet werden. Ein weiterer Parameter `cex.axis` steuert die Schriftgröße, die relativ angegeben wird. 1 bedeutet normale, 0.5 halbe und 2 doppelte Schriftgröße.

Die Gesamtbezeichnung der Achsen kann manuell mit `mtext()` gesetzt werden.

mtext('label**',** orientierung***[*, line=**abstand*]***)

Die Orientierung hat den gleichen Wert wie bei den Achsen. Mit `line` kann der Abstand zwischen Achsenteilung und der Beschriftung verändert werden, um ein Überdecken zu verhindern.

Der Gesamttitel wird mit der Funktion `title()` erzeugt.

title('Titel**'***[*sub='**untertitel**'***]*** ***[*,line=**abstand*]***)

Das Vergrößern des `line`-Parameters führt oft dazu, dass der Text außerhalb der Bildgrenzen erscheint. Falls dies passiert, müssen wir vor der Plotfunktion und *nach* dem eventuellen Öffnen der Graphikdatei für die Ausgabe die Funktion `par()` aufrufen, mit der der unter anderem der Rand eingestellt werden kann.

par(mar=c(unten, links, oben, rechts**)**

Der Rand wird in der gleichen Einheit wie die Textabstände angegeben. Nach dem Schließen der Ausgabe mit `dev.off()` werden wieder die Vorgabewerte benutzt.

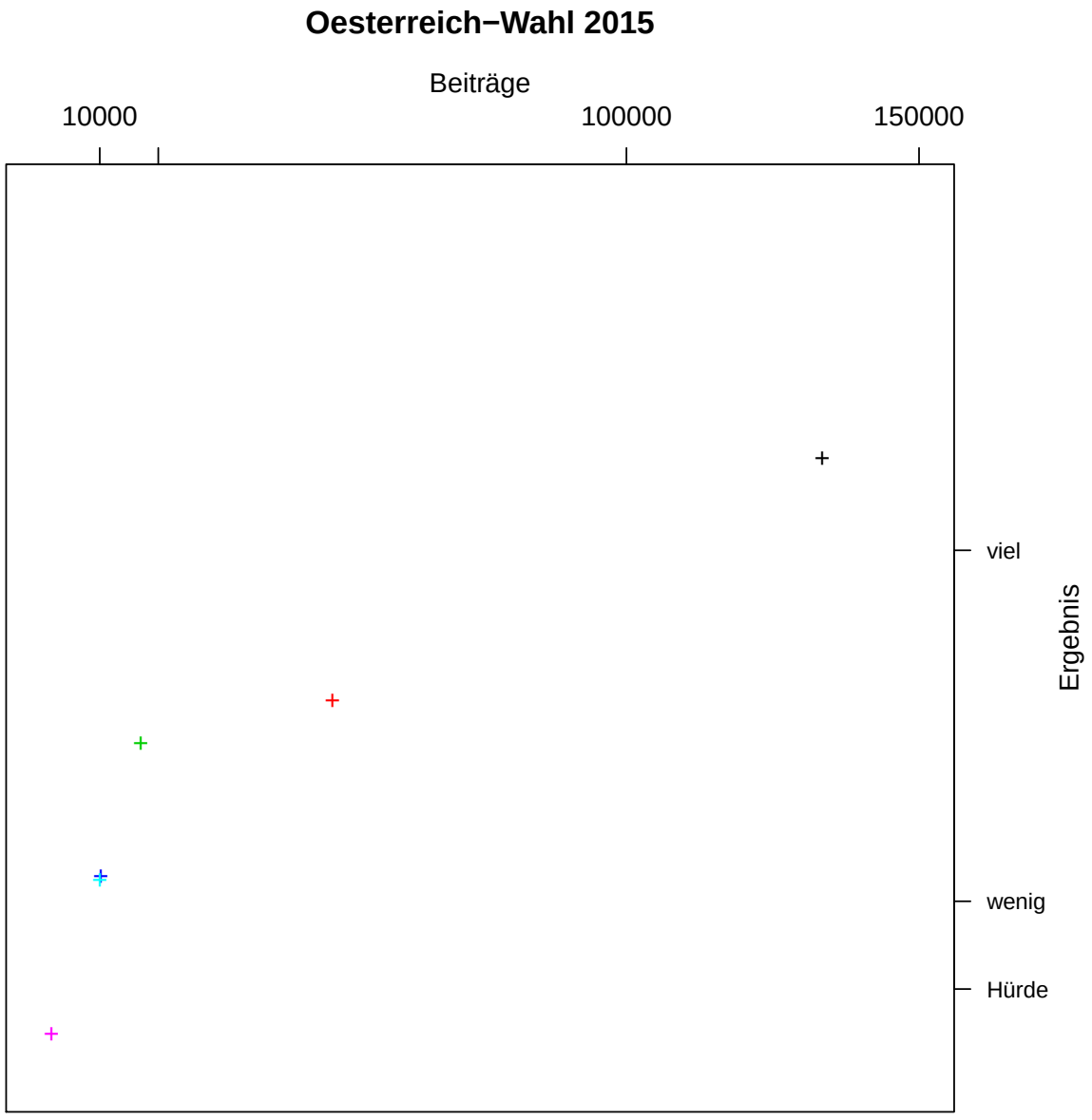
Beispiel: Wir zeichnen einen leeren XY-Plot (erster Parameter `NULL`, geben aber `xlim` und `ylim` an) und die Ergebnisse der Oesterreich-Wahl anschließend manuell mit der `points()`-Funktion. Auch die Achsen werden manuell oben und rechts mit eigener Achsteilung und für die y-Achse mit individueller Beschriftung gesetzt. Da der Abstand zwischen Achsteilung, -beschriftung und Diagrammtitel zu klein ist, werden auch die Beschriftungen manuell erzeugt.

```
oesi <- read.table("oesiwahl.dat", header=T)
names(oesi) <- c("Name", "Talk", "Result")
# Ränder vergrößeren
par(mar=c(1,1,5,5))
# keine Achsteilung und -beschriftung
plot(NA, xlim=c(0, 150000), ylim=c(0, 50), ann=F,
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
     xaxt='n', yaxt='n')
# Diagrammfläche
points(oesi$Talk, oesi$Result, pch="+", col=1:length(oesi$Name))
# Teilung x-Achse und Darstellung oben
xticks <- c(10000, 20000, 100000, 150000)
axis(3, at=xticks)
# x-Label
mtext('Beiträge', 3, line=2)
# Teilung y-Achse, eigene Labels, rechts
yticks <- c(5, 10, 30)
axis(4, at=yticks, labels=c('Hürde', 'wenig', 'viel'), las=2,
     cex.axis=0.8)
# y-Label
mtext('Ergebnis', 4, line=3)
# Diagrammtitel, ohne Subtitel
title('Oesterreich-Wahl 2015', line=4)
```



9.3 Punkte und Punktbeschriftungen

XY-Plots und ähnliche Diagramme können nun weiter ergänzt werden. So können Datenpunkte manuell hinzugefügt werden:

points(xVektor, yVektor*[, ...]*)

Weitere Parameter sind z. B. Farbe (`col`) und dargestelltes Symbol (`pch`).

Wollen wir Datenpunkte beschriften, ordnen wir Text an bestimmten Koordinaten an. Es werden drei Vektoren benötigt: x-Koordinaten, y-Koordinaten, Texte.

text(xVektor, yVektor, textVektor*[,* **pos=**posVektor*][,* **cex=**faktor...*]*)

Der `pos`-Vektor oder -Einzelwert gibt an, in welcher Richtung vom Datenpunkt die Beschriftung erfolgen soll (1: unterhalb, 2: links, 3: oberhalb, 4: rechts). Ohne Angabe (bzw. `NULL`) wird der Text zentriert zum Punkt ausgegeben. Mit `cex` kann die Schriftgröße skaliert werden. Es kann auch ein Farbvektor (`col`) übergeben werden, um z. B. die Beschriftung in der gleichen Farbe der Datenpunkte vorzunehmen.

Alternativ kann die Beschriftung auch mit einer Legende erfolgen.

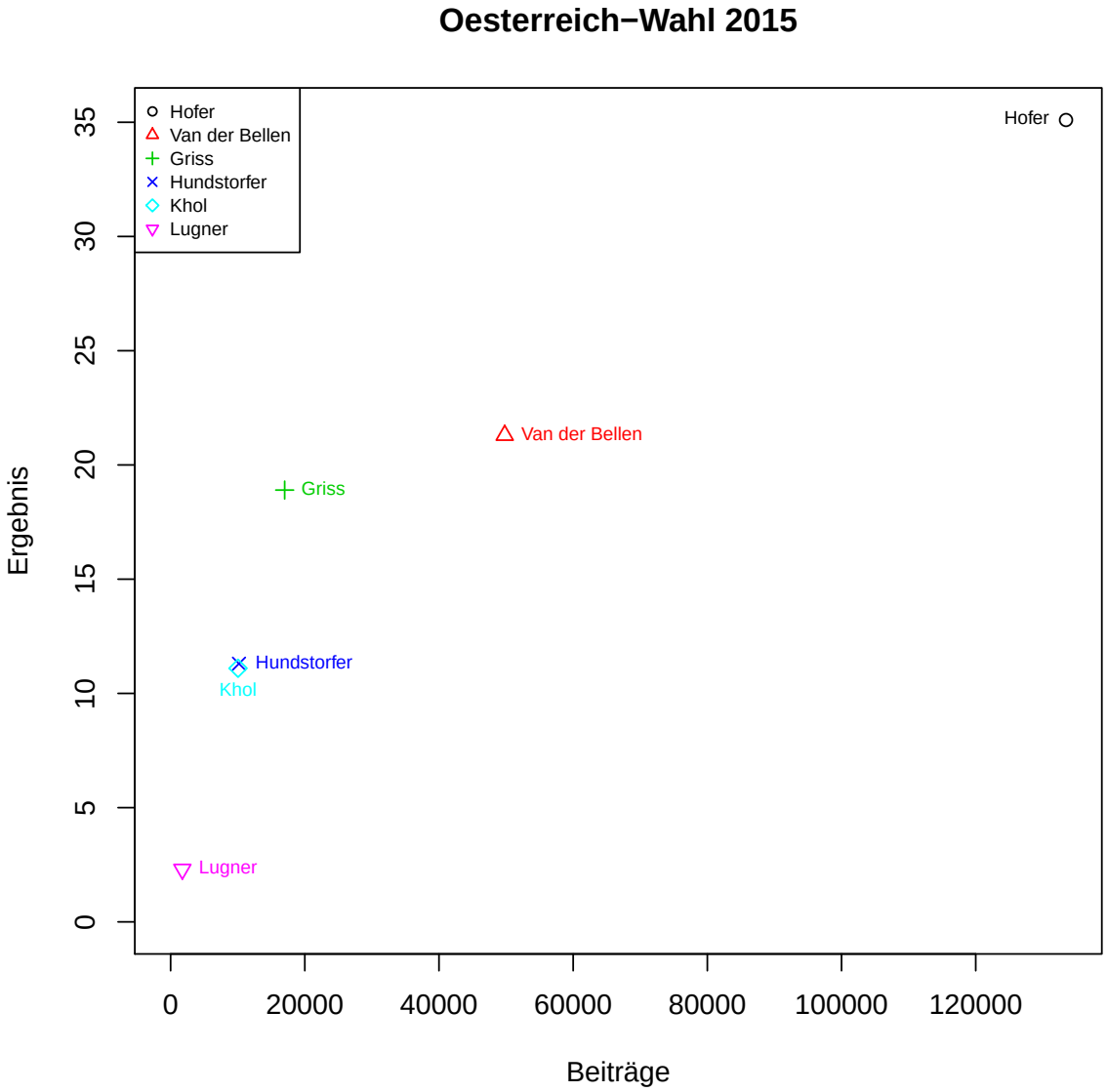
legend(position, **legend=**stringVektor*[, ...]*)

Wir wollen mit der Legende wahlweise Datenpunkte und Linien beschriften. Die Legendensymbole für Punkte geben wir als Vektor oder Einzelwert mit dem Paramater `pch` an. Ein Wert `NA` zeigt kein Symbol an. Linien werden durch den Parameter `lty` ausgewählt, wobei der Wert 0 keine Linie anzeigt. Treten in einer Legende sowohl Symbole für Datenpunkte als auch Linien auf, benötigen wir sowohl `pch` als auch `lty` als Vektor, wobei an jeder Position in einem der Vektoren eine Auswahl für die Anzeige erfolgt, im anderen Vektor dagegen die Ausgabe unterdrückt wird (Beispiel s. u.).

Beispiel: Farbige Anzeige der Datenpunkte und parallele Aufnahme in Legende. In realen Auswertungen sollte die Doppelbeschriftung allerdings vermieden werden.

```
setwd("~/lehre/R")
oesi <- read.table("samples/testdata/oesiwahl.dat", header=T)
names(oesi) <- c("Name", "Talk", "Result")
# Anzahl Datensätze
anz <- length(oesi$Name)
# Scatterplot, jeder Datensatz eigenes Symbol und Farbe
plot(oesi$Result ~ oesi$Talk, xlim=c(0, max(oesi$Talk)), ylim=c(0, max(oesi$Result)),
     pch=1:anz, col=1:length(oesi$Name),
     main="Oesterreich-Wahl 2015", xlab="Beiträge", ylab="Ergebnis")
# Text normalerweise rechts
pos <- rep(4, anz)
# Werte für Zeilen 4 und 5 fast identisch, deshalb 1x anders beschriften
pos[5] <- 1
# erster Wert rechts oben, deshalb links beschriften
pos[1] <- 2
# Für Text gleiche Farbe wie für Datenpunkte, Schrift verkleinert
text(oesi$Talk, oesi$Result, oesi$Name, col=1:length(oesi$Name),
     pos=pos, cex=0.7)
# Legende muss Symbole und Farben des Plots wiederholen, Schrift ebenfalls etwas␣
↪kleiner
legend("topleft", legend=oesi$Name, cex=0.7, col=1:length(oesi$Name), pch=1:anz)
```

Ergebnis:


Oesterreich−Wahl 2015
Hofer
Van der Bellen
Griss
Hundstorfer
Khol
Lugner
Ergebnis
0
5
10
15
20
25
30
35
0
20000
40000
60000
80000
100000
120000
Beiträge

9.4 Einfache Linien

Waagerechte, senkrechte und speziell geneigte Linien können einfach mit der Funktion `abline()` in Diagrammen ergänzt werden. Die Grundformen sind

abline(a=intercept**, b=**anstieg**)**
abline(h=yWert**)**
abline(v=xWert**)**

wobei *Intercept* den y-Wert bei x=0 beschreibt, die Linie also die Geradengleichung $y{=}a{+}b \cdot x$ erfüllt. Diese beiden Werte können auch als Vektor an den Parameter `coef` übergeben werden, wie sie bei der Berechnung linearer Modelle entstehen (s. u.)

Für horizontale Linien verwenden wir statt `a` und `b` den Parameter `h=y-Wert`, für vertikale den Parameter `v=x-Wert`.

Die Farbe wird wie üblich mit dem Parameter `col` gewählt, mit dem Parameter `lty` kann der Linientyp gewählt werden.

Beispiel: Anzeige Mittelwerte und Mediane für die Oesterreichwahl 2015 sowohl für die Zahl der Beiträge als auch für das Wahlergebnis.

```
oesi <- read.table("samples/testdata/oesiwahl.dat", header=T)
names(oesi) <- c("Name", "Talk", "Result")
mean.talk <- mean(oesi$Talk)
median.talk <- median(oesi$Talk)
mean.result <- mean(oesi$Result)
median.result <- median(oesi$Result)

plot(oesi$Result ~ oesi$Talk, pch="+",
     main="Wahl Bundespräsident Oesterreich 2015", xlab="Beiträge", ylab="Ergebnis (%)
↪")
pos <- rep(3, length(oesi$Name))
pos[1] <- 2
pos[4] <- 4
text(oesi$Talk, oesi$Result, oesi$Name, cex=0.7, pos=pos)
abline(v=mean.talk, col="red")
abline(v=median.talk, col="blue")
abline(h=mean.result, col="red")
abline(h=median.result, col="blue")
legend("topleft", legend=c("Kandidat", "Mittelwert", "Median"),
       pch=c("+", NA, NA), lty=c(0, 1, 1), col=c("black", "red", "blue"))
```

Ergebnis:

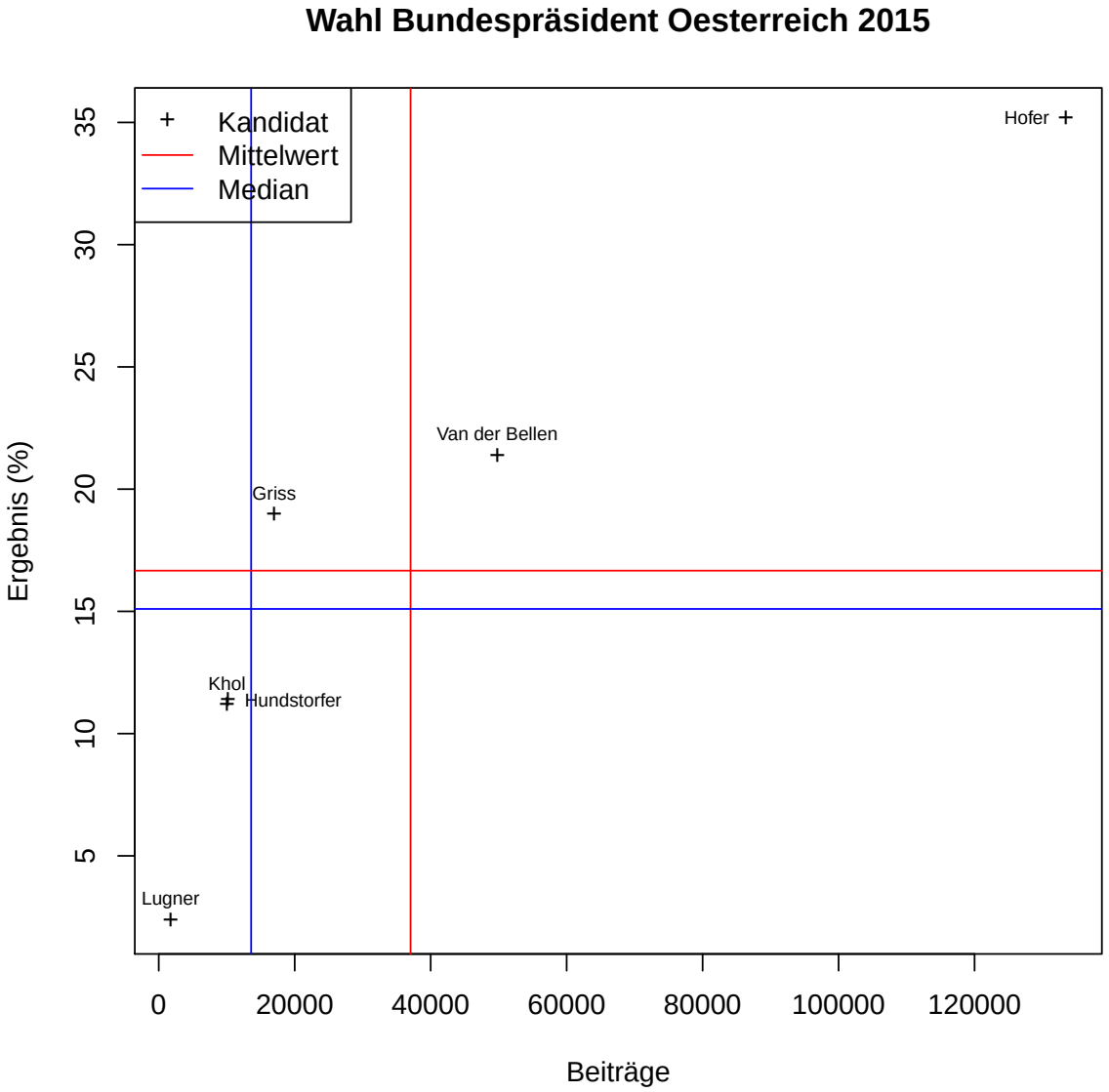
Beispiele für geneigte Linien sehen wir bei der Behandlung linearer Modelle.

9.5 Linienzüge und Polygone

In einem XY-Plot stehen uns mit Linienzügen und Polygonen weitere Zeichenfunktionen für die Diagrammgestaltung zur Verfügung.

line(xVektor**,** yVektor*[, ...]***)**
polygon(xVektor**,** yVektor*[, ...]***)**

Beide Funktionen erwarten mindestens zwei Parameter, die Vektoren der x- und der y-Koordinaten. Während die erste Funktion alle angegebenen Koordinaten durch einen Linienzug verbindet, füllt `polygon()`


Wahl Bundespräsident Oesterreich 2015
Kandidat
Mittelwert
Median
Hofer
Van der Bellen
Griss
Khol
Hundstorfer
Lugner
Ergebnis (%)
5
10
15
20
25
30
35
0
20000
40000
60000
80000
100000
120000
Beiträge

die durch die Koordinaten eingeschlossene Fläche, wobei automatisch der erste mit dem letzten Punkt verbunden wird.

Beispiel: Funktionsgraph von *sin(x)* mit Low-Level-Funktion, Fläche unter der Kurve von 2 bis 7.

```
plot(NA, xlim=c(0,10), ylim=c(-2,2),xlab="x", ylab="y",main="Funktionsgraph und␣
↪Integral")
x <- c(0:100)*0.1
y <- sin(x)
lines(x, y)
idx <- which(x>=2 & x<=7)
xx <- x[idx]
yy <- y[idx]
xx <- append(xx, c(7, 2))
yy <- append(yy, c(0, 0))
polygon(xx, yy, col="gray")
abline(h=0,col="blue")
abline(v=2,col=2)
abline(v=7,col=2)
legend("topright", legend=c("sin(x)", "y=0", "Grenzen", "Integral"),
       lty=c(1,1,1,NA),pch=c(NA,NA,NA,15),col=c(1,"blue", 2, "gray"))
```

Ergebnis:

9.6 Aufgaben

Aufgabe 9.1 Stellen Sie die Kurve einer Normalverteilung mit Mittelwert 5 und Varianz $\sigma^2 = 2$ im Intervall (0, 10) dar (Funktion `dnorm()`, Dichte der Normalverteilung). Zeichnen Sie die Werte σ und $2 \cdot \sigma$ als senkrechte Linien und die beiden symmetrischen Bereiche der Kurve, die zusammen weniger als 5% der Fläche beanspruchen, als graue Fläche dar (2,5%-Quantile, Wert kann mit Funktion `qnorm()` ermittelt werden).

Mögliches Ergebnis:

Aufgabe 9.2 *Hausaufgabe (10 Punkte)*

Erzeugen verschiedener Plots aus der zusammengefassten Datei aller Angebote. Die Einzeldateien dürfen hier nicht verwendet werden!

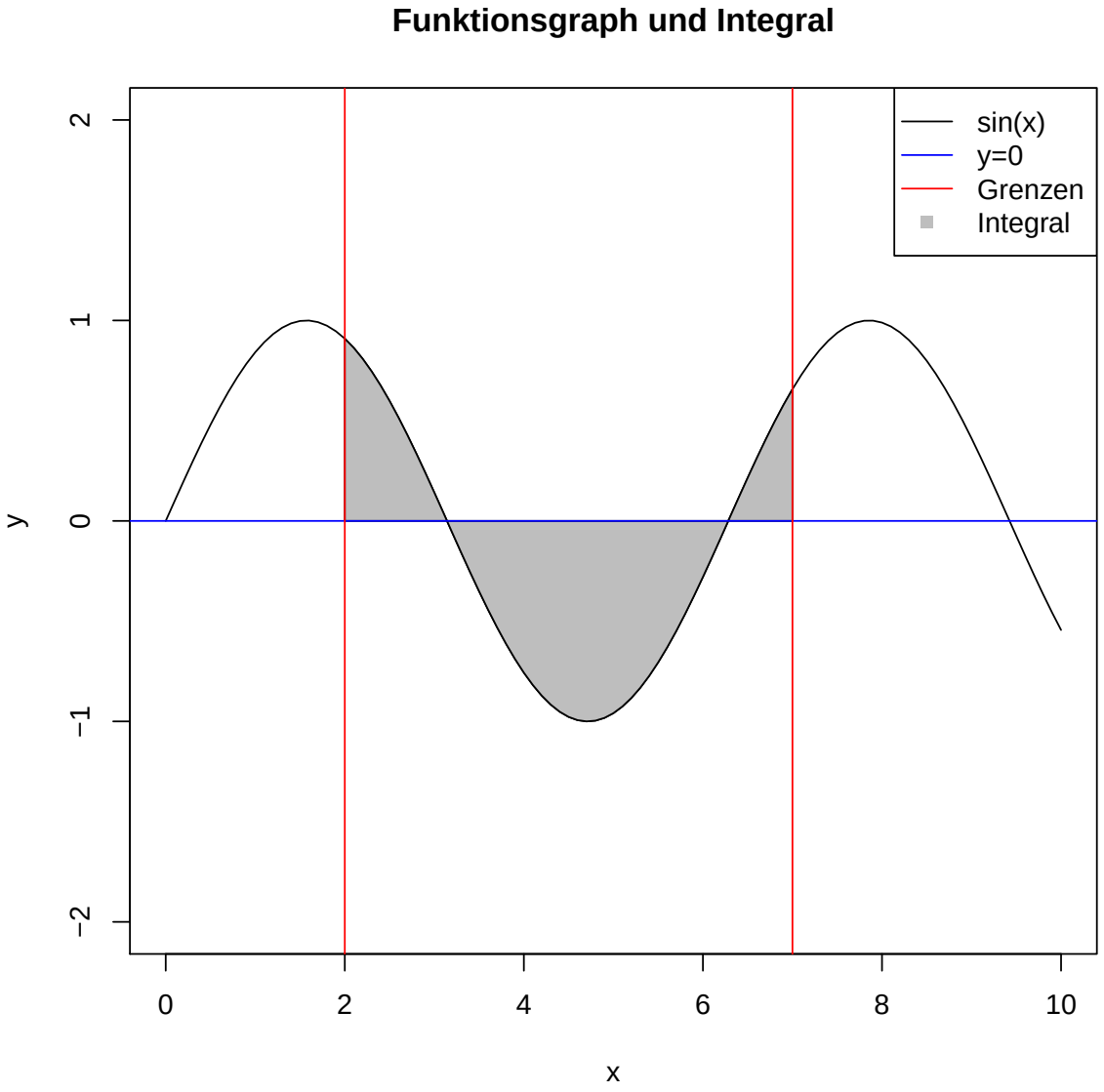
- Berechnen Sie für jedes Produkt den mittleren Preis und ergänzen Sie für jedes Angebot einen normierten Preis bezüglich des mittleren Artikelpreises (ein Angebot mit genau dem mittleren Preis hat also den normierten Preis 1, also *Normpreis = Artikelpreis/Mittelpreis_Produktgruppe*).

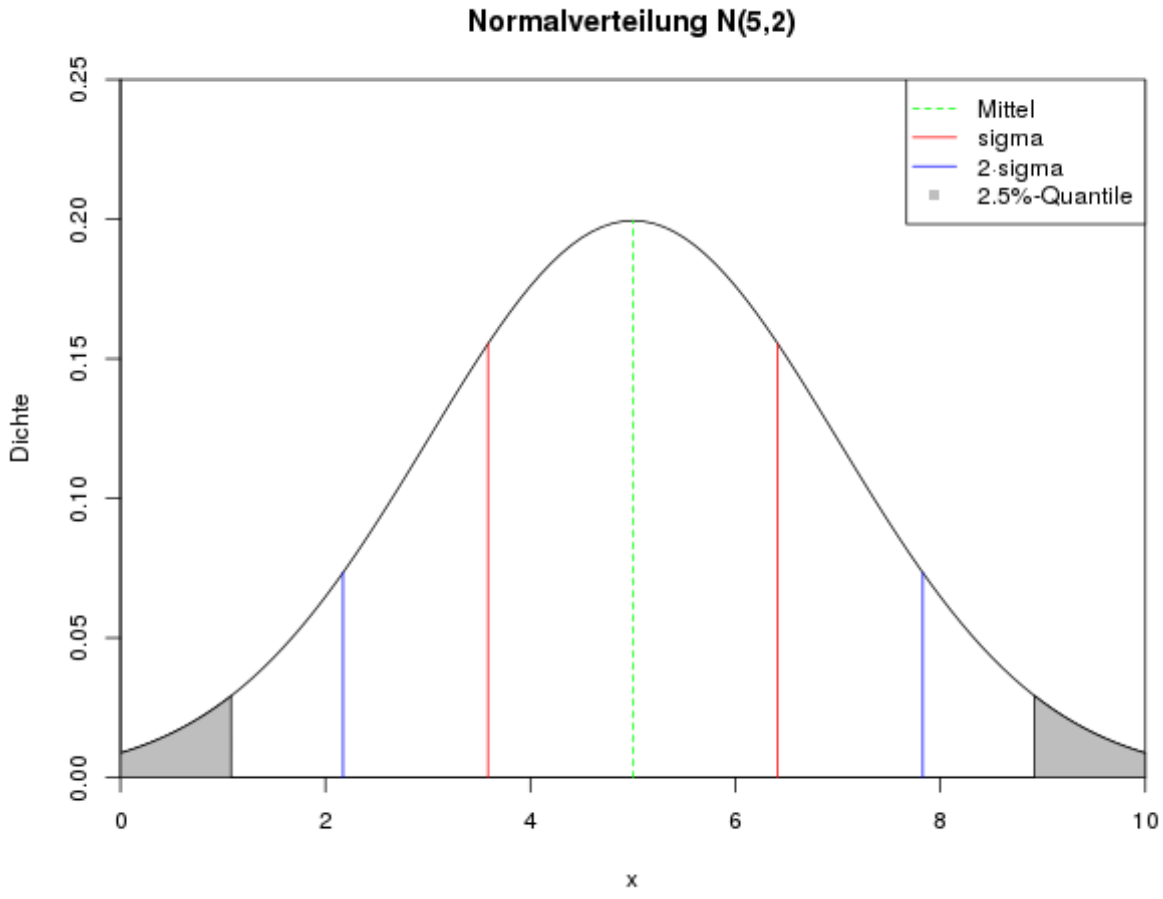
  Hinweis: Sie können hier sinnvoll die Funktion `ave()` (s. Dokumentation) nutzen, das ist aber nicht zwingend erforderlich.

- Erzeugen Sie einen Scatterplot, der für jedes Angebot die Anzahl der Bewertungen bzgl. des normierten Produktpreises darstellt. Die Produkte sollen dabei durch Farbe und Symbol voneinander unterschieden werden (Legende nicht vergessen).

- Erzeugen Sie ein Balkendiagramm, das die Anzahl der Bewertungen je Angebot absteigend nach Häufigkeit zeigt. Die Balken sind nach Produkt unterschiedlich zu färben, eine Legende für die Farbzuordnung ist erforderlich.

- Erzeugen Sie ein Histogramm für die Verteilung der Anzahl an Bewertungen aller Angebote, zusammengefasst für alle Produkte.


Funktionsgraph und Integral
sin(x)
y=0
Grenzen
Integral
y
x
2
1
0
−1
−2
0
2
4
6
8
10


Normalverteilung N(5,2)
Mittel
sigma
2·sigma
2.5%-Quantile
Dichte
0.00
0.05
0.10
0.15
0.20
0.25
0
2
4
6
8
10
x

- Erzeugen Sie zwei Diagramme, die zum einen die absolute Preisverteilung der Angebote je Produkt, zum anderen die normierte Preisverteilung je Produkt darstellt. Welches Diagramm ist aussagekräftiger?
- Stellen Sie das Verhältnis der Gesamtbewertungszahlen aller Angebote, gruppiert für jedes Produkt graphisch dar. Die prozentualen Anteile sollen auch textuell ablesbar sein.

Alle Diagramme sind aussagekräftig zu beschriften und die Ergebnisse zu interpretieren.

Geben Sie neben Ihrem Jupyter-Notebook auch Ihre Datendatei mit den zusammengefassten Angeboten für alle Produkte ab.

KAPITEL 10

Matrizen und lineare Algebra

10.1 Grundlagen, Erstellung

Matrizen sind zweidimenisionale Felder, deren Elemente über einen Zeilen- und einen Spaltenindex adressiert werden. Im Gegensatz zu Datenframes, die zum Darstellen und Bearbeiten von Beobachtungen mit Daten verschiedener Skalen (Typen) geeignet sind, dürfen in Matrizen nur Werte des gleichen Datentyps (meist Zahlen) stehen.

Matrizen werden aus einem oder mehreren Vektoren erzeugt. Liegt nur ein Vektor vor, wird dieser in gleiche Abschnitte zerlegt, die die Spalten der Matrix bilden. Die Anzahl der Zeilen bzw. Spalten wird beim Erzeugen durch den Parameter `nrow` bzw. `ncol` festgelegt. Diese Zahlen müssen Teiler der Elementzahl des Vektors sein, andernfalls erhalten wir eine Warnung, zum Auffüllen der letzten Elemente wird der Vektor erneut von vorn gelesen.

```
values <- 1:6

A <- matrix(values, nrow=2)
cat('A (rnow=2):\n')
print(A)

B <- matrix(values, ncol=2)
cat('B (ncol=2):\n')
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
print(B)

X <- matrix(values, ncol=4)
cat('X (ncol=4):\n')
print(X)
```

Ergebnis:

```
A (rnow=2):
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
B (ncol=2):
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
Warnmeldung:
In matrix(values, ncol = 4) :
  Datenlänge [6] ist kein Teiler oder Vielfaches der Anzahl der Spalten [4]
X (ncol=4):
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    1
[2,]    2    4    6    2
```

Zum Zusammenfügen mehrerer gleichlanger Vektoren zu einer Matrix können wir die Befehle `rbind()` und `cbind()` verwenden. Der erste Befehl betrachtet die Einzelvektoren als Zeilenvektoren und fügt diese untereinander als Zeilen in die Matrix ein, der zweite verwendet die Vektoren als Spalten der Matrix. Beispiel: Ordne zu einem Größenvektor eine laufende Nummer zu (beides sind Zahlen).

```
# Vektoren erstellen
sizes <- c(182, 175, 168, 174, 186)
idx <- 1:length(sizes)
# zeilenweise verbinden
g1 <- rbind(idx, sizes)
# spaltenweise verbinden
g2 <- cbind(idx, sizes)

# Typ- und Strukturprüfung
cat('Datentyp von g1:\n')
class(g1)
cat('\nAufbau von g1:\n')
str(g1)

# Ausgabe
cat('\ng1:\n')
print(g1)

cat('\ng2:\n')
print(g2)
```

Ausgabe:

```
Datentyp von g1:
[1] "matrix"

Aufbau von g1:
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
 num [1:2, 1:5] 1 182 2 175 3 168 4 174 5 186
 - attr(*, "dimnames")=List of 2
  ..$ : chr [1:2] "idx" "sizes"
  ..$ : NULL

g1:
      [,1] [,2] [,3] [,4] [,5]
idx      1    2    3    4    5
sizes  182  175  168  174  186

g2:
     idx sizes
[1,]   1   182
[2,]   2   175
[3,]   3   168
[4,]   4   174
[5,]   5   186
```

Wie wir hier sehen, können zusätzlich zum Index den Zeilen und Spalten auch Namen zugewiesen werden. Dies geschieht mit den Funktionen `rownames()` bzw. `colnames()`.

Der Elementzugriff erfolgt wie bei Datenframes durch die Angabe von Zeilen- und Spaltenindex. Sind Namen zugeordnet, so können auch diese verwendet werden. Wird ein Index weggelassen, so wird eine ganze Zeile bzw. Spalte ausgwählt. Durch die Angabe von Indexbereichen können auch Teilmatrizen ausgewählt werden.

Häufig ist es erforderlich, Matrizen zu transponieren (Zeilen und Spalten zu vertauschen), z. B. wenn die Eingabedaten zeilenweise organisiert sind, die Matrix aber normalerweise spaltenweise aufgebaut wird. Dies erledigt die Funktion `t()`:

```
A <- matrix(1:6, nrow=2)
cat('A:\n')
print(A)

# Transponieren
A.T <- t(A)

cat('\nt(A):\n')
print(A.T)
```

Ausgabe:

```
A:
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

t(A):
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
```

10.2 Matrizen und Datenframes

Ein Datenframe kann in eine Matrix umgewandelt werden, sofern alle Spalten numerische Werte enthalten.

matrix <- **as.matrix(**dataFrame**)**

Wir betrachten die am Beispiel der Bevölkerungsentwicklung in Deutschland `bevoelkerung.csv` (Quelle: deutschlandinzahlen.de). Diesen lesen wir wie folgt ein:

```
data <- read.csv2('bevoelkerung.csv', header=T, comment='#', dec='.')
```

Die erste Spalte enthält das Bundesland, die folgenden Spalten die Zahlen pro Jahr. Wir müssen also die erste Spalte abtrennen, dann können wir ihn in eine Matrix umwandeln.

```
xdata <- data[,-1]
# oder: xdata <- data[, 2:ncol(data)]
# oder: xdata <- subset(data, select=-c(Bundesland))
mdata <- as.matrix(xdata)
```

Umgedreht können wir eine Matrix mit der Funktion `data.frame()` wieder in einen Datenframe umwandeln:

dataFrame <- **data.frame(**matrix**)**

Diesem fehlen jedoch meist sinnvolle Kopfzeilen, die wir mit der Funktion `names()` ergänzen können.

Beispiel: Rückwandlung der Bevölkerungszahlen in einen Datenframe. Die Spaltenüberschriften setzen wir neu.

```
ndata <- data.frame(mat)
years <- 1991:2019
names(ndata) <- paste('Jahr', years, sep='.')
```

10.3 Transponieren von Datenframes

Hin und wieder ist es erforderlich, in einem Datenframe die Zeilen und Spalten zu vertauschen. Damit dies möglich ist, müssen alle Werte den gleichen Datentyp haben, da die Originalzeilen zu Spalten und damit zu einer Variable werden.

Wir gehen wie folgt vor:

- Speichern und Entfernen der ersten Spalte,
- Transponieren des restlichen Datenframes,
- Verwenden der ersten Spalte als neue Spaltennamen.

Für die Analyse der zeitlichen Entwicklung je Bundesland ist die Anordnung in Spalten pro Bundesland aber praktischer.

Wir schneiden deshalb die erste Spalte ab und transponieren die restlichen Spalten. Eine vorherige Umwandlung in eine Matrix erfolgt implizit und muss deshalb nicht vorher vorgenommen werden. Das Ergebnis ist jedoch eine Matrix und kein Datenframe.

```
laender <- data[,1] # oder laender <- data$Bundesland
xdata <- data[,-1]
# oder: xdata <- data[, 2:ncol(data)]
# oder: xdata <- subset(data, select=-c(Bundesland))
mat.trans <- t(xdata)
```

Die Matrix wird wieder in einen Datenframe umgewandelt. Die Spaltennamen erhalten wir aus der ersten Spalte des ursprünglichen Datenframes.

```
tdata <- data.frame(t(xdata))
names(tdata) <- laender
```

10.4 Matrix- und Vektormultiplikationen

Während der Operator * eine elementweise Multiplikation durchführt, verwenden wir für «echte» Matrix- und Vektormultiplikationen den Operator **%*%**. Je nach Art der Operanden werden dabei unterschiedliche Berechnungen ausgeführt.

Zwei Vektoren Berechnung des Skalarproduktes. Ergebnis wird als 1×1-Matrix dargestellt. Mit der Funktion `drop()` erhält man daraus den Einzelwert.

```
v1 <- c(1, 2, 3)
v2 <- c(-1, 4, -2)

cat('v1:\n')
print(v1)
cat('\nv2:\n')
print(v2)

v1.v2 <- v1 * v2
cat('\nv1 * v2:\n')
print(v1.v2)

scalprod <- v1 %*% v2
cat('\n v1 %*% v2 (Skalarprodukt):\n')
print(scalprod)

# zur Einzelzahl:
norm <- drop(scalprod) # alternativ: c(scalprod)
cat('\nAls Einzelwert:\n')
print(norm)
```

Ergebnis:

```
v1:
[1] 1 2 3

v2:
[1] -1  4 -2

v1 * v2:
[1] -1  8 -6

 v1 %*% v2 (Skalarprodukt):
     [,1]
[1,]    1

Als Einzelwert:
[1] 1
```

Matrix und Vektor Der Vektor wird als Spaltenvektor interpretriert und von rechts an die Matrix multipliziert. Die Spaltenzahl der Matrix muss mit der Länge des Vektors übereinstimmen. Das Ergebnist ist ein Spaltenvektor.

```
A <- matrix(c(1, 2, 3, 4), nrow=2)
v <- c(5, -1)

cat('A:\n')
print(A)
cat('\nv:\n')
print(v)

A.v <- A * v

cat('\nA * v (elementweise):\n')
print(A.v)

Av <- A %*% v
cat('\nA %*% v (inneres Produkt):\n')
print(Av)

# In Vektor umwandeln:
vec.Av <- c(Av)
cat('\nA %*% v als Vektor:\n')
print(vec.Av)
```

Ergebnis:

```
A:
     [,1] [,2]
[1,]    1    3
[2,]    2    4

v:
[1]  5 -1

A * v (elementweise):
     [,1] [,2]
[1,]    5   15
[2,]   -2   -4

A %*% v (inneres Produkt):
     [,1]
[1,]    2
[2,]    6

A %*% v als Vektor:
[1] 2 6
```

Vektor und Matrix Der Vektor wird als Zeilenvektor interpretiert und von links an die Matrix multipliziert. Die Zeilenzahl der Matrix muss mit der Länge des Vektors übereinstimmen. Das Ergebnis ist ein Zeilenvektor.

```
A <- matrix(c(1, 2, 3, 4), nrow=2)
v <- c(5, -1)

cat('A:\n')
print(A)
cat('\nv:\n')
print(v)
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
v.A <- v * A

cat('\nv * A (elementweise):\n')
print(v.A)

vA <- v %*% A
cat('\nv %*% A (inneres Produkt):\n')
print(vA)

# In Vektor umwandeln:
vec.vA <- c(vA)
cat('\nv %*% A als Vektor:\n')
print(vec.vA)
```

Ergebnis:

```
A:
     [,1] [,2]
[1,]    1    3
[2,]    2    4

v:
[1]  5 -1

v * A (elementweise):
     [,1] [,2]
[1,]    5   15
[2,]   -2   -4

v %*% A (inneres Produkt):
     [,1] [,2]
[1,]    3   11

v %*% A als Vektor:
[1]  3 11
```

Matrix und Matrix Matrixmultiplikation. Die Spaltenzahl der ersten Matrix muss mit der Zeilenzahl der zweiten Matrix übereinstimmen. Das Ergebnis ist eine neue Matrix. Die Matrixmultiplikation ist nicht kommutativ (aber assoziativ).

```
A <- matrix(c(1, 2, 3, 4), nrow=2)
B <- matrix(c(-1, 1, -2, 2), nrow=2)

cat('A:\n')
print(A)
cat('\nB:\n')
print(B)

A.B <- A * B

cat('\nA * B:\n')
print(A.B)

B.A <- B * A

cat('\nB * A == A * B:\n')
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
print(B.A)

AB <- A %*% B

cat('\nA %*% B:\n')
print(AB)

BA <- B %*% A

cat('\nB %*% A   A %*% B:\n')
print(BA)
```

Elementweise und Matrixmultiplikation (inneres Produkt):

```
A:
     [,1] [,2]
[1,]    1    3
[2,]    2    4

B:
     [,1] [,2]
[1,]   -1   -2
[2,]    1    2

A * B:
     [,1] [,2]
[1,]   -1   -6
[2,]    2    8

B * A == A * B:
     [,1] [,2]
[1,]   -1   -6
[2,]    2    8

A %*% B:
     [,1] [,2]
[1,]    2    4
[2,]    2    4

B %*% A   A %*% B:
     [,1] [,2]
[1,]   -5  -11
[2,]    5   11
```

Bei der Multiplikation von Matrix und Vektor bzw. Vektor und Matrix entsteht ein Spalten- bzw. Zeilenvektor, der jedoch als Matrix dargestellt wird. Eine Umwandlung in einen eindimensionalen Vektor geschieht mit der Funktion `c()`. Mit dieser Funktion kann auch eine Matrix wieder in einen Vektor konvertiert werden, wobei die Elemente spaltenweise durchlaufen werden. Diese Umwandlung wurde in obigen Beispielen bereits eingebaut.

Im Gegensatz zur Multiplikation erfolgt die Addition zweier Matrizen elementweise, wir können also einfach das Additionszeichen verwenden:

```
A + B
```

Die Addition von Matrizen ist kommutativ.

10.5 Lösen von linearen Gleichungssystemen

Mit *R* lassen sich lineare Gleichungssysteme einfach lösen. Die Funktion `solve()` erwartet als Parameter eine quadratische Matrix und einen Vektor gleicher Länge oder eine zweite Matrix gleicher Zeilenzahl (die man sich als eine Liste verschiedener rechter Seiten vorstellen kann, für die das System gelöst wird). Fehlt der zweite Parameter, ist das Ergebnis die inverse Matrix.

```
A <- matrix(c(1, 2, 3, 4), nrow=2)
b <- c(5, 8)

cat('Koeffizienten (A):\n')
print(A)
cat('\nRechte Seite (b):\n')
print(b)

# Lösen des Gleichungssystems:

x <- solve(A, b)
cat('\nLösung:\n')
print(x)

# Probe:
b.test <- A %*% x

cat('\nProbe (Differenz zwischen Ax und b):\n')
print(b.test - b)

# Inverse von A:
A.inv <- solve(A)
cat('\nInverse von A:\n')
print(A.inv)

# Probe:
I <- A.inv %*% A
cat('\nProbe: A.inv · A = I:\n')
print(I)
```

Ergebnis:

```
Koeffizienten (A):
     [,1] [,2]
[1,]    1    3
[2,]    2    4

Rechte Seite (b):
[1] 5 8

Lösung:
[1] 2 1

Probe (Differenz zwischen Ax und b):
     [,1]
[1,]    0
[2,]    0

Inverse von A:
     [,1] [,2]
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
[1,]   -2  1.5
[2,]    1 -0.5

Probe: A.inv · A = I:
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

Ist das Gleichungssystem nicht lösbar, weil die Koeffizientenmatrix *singulär* ist (Matrixzeilen sind linear abhängig, Determinante ist 0), dann erhalten wir eine Fehlermeldung:

```
N <- matrix(c(1, 2, 3, 4, 5, 6, 5, 7, 9), nrow=3)
cat('Singuläre Matrix N:\n')
print(N)

N.inv <- solve(N)
```

Ergebnis:

```
Singuläre Matrix N:
     [,1] [,2] [,3]
[1,]    1    4    5
[2,]    2    5    7
[3,]    3    6    9
Fehler in solve.default(N) :
  Lapackroutine dgesv: System ist genau singulär: U[3,3] = 0
Ruft auf: solve -> solve.default
Ausführung angehalten
```

10.6 Determinanten, Eigenwerte und -vektoren

Wollen wir vor dem Lösen des Gleichungssystems prüfen, ob es überhaupt lösbar ist, können wir mit `det()` die Determinante der Koeffizientenmatrix bestimmen.

```
A <- matrix(1:4, nrow=2)
cat('A:\n')
print(A)

det.A <- det(A)
cat('\nDeterminante(A):\n')
print(det.A)

N <- matrix(1:9, nrow=3)
cat('\nN:\n')
print(N)

det.N <- det(N)
cat('\nDeterminante(N):\n')
print(det.N)
```

Ergebnis:

```
A:
     [,1] [,2]
[1,]    1    3
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
[2,]    2    4

Determinante(A):
[1] -2

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

Determinante(N):
[1] 0
```

Hinweis: Da das Berechnen der Determinante fast genauso aufwendig ist wie das Lösen eines Gleichungssystems, ist es nur selten sinnvoll, diese explizit zu berechnen.

Mit der Funktion `eigen()` werden die Eigenwerte und -vektoren der Matrix berechnet.

```
A <- matrix(1:4, nrow=2)
cat('A:\n')
print(A)

eigen.A <- eigen(A)
cat('\nEigenwerte:\n')
print(eigen.A$values)

cat('\nEigenvektoren:\n')
print(eigen.A$vectors)

# Probe für ersten Eigenwert und Eigenvektor:
# A·v1 = e1*v1
v1 = eigen.A$vectors[,1]
e1 = eigen.A$values[1]
cat('\nA·v1:\n')
print(c(A %*% v1)) # einspaltige Matrix -> Vektor
cat('\ne1*v1:\n')
print(e1 * v1)
```

Ergebnis:

```
A:
     [,1] [,2]
[1,]    1    3
[2,]    2    4

Eigenwerte:
[1]  5.3722813 -0.3722813

Eigenvektoren:
           [,1]       [,2]
[1,] -0.5657675 -0.9093767
[2,] -0.8245648  0.4159736

A·v1:
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
[1] -3.039462 -4.429794

e1*v1:
[1] -3.039462 -4.429794
```

Wir werden Eigenwertzerlegungen im Kurs eher nicht benötigen und wollen deshalb auch nicht genauer darauf eingehen.

10.7 Aufgaben

Aufgabe 10.1 Erzeugen Sie eine Matrix aus zwei gleichlangen Zeilenvektoren ohne die Verwendung der Funktionen `rbind()` bzw. `cbind()`.

Aufgabe 10.2 Schreiben Sie ein R-Script, mit dem Sie lineare Gleichungssysteme mit n Unbekannten lösen können. Die Eingabe der notwendigen Parameter soll aus einer Datei erfolgen. Die Zeile i enthält die Koeffizienten a bis a und die rechte Seite b. Der Dateiname der Koeffizientenmatrix soll vom Script abgefragt und interaktiv eingegeben werden können.

Beispiel:

3 x1 + 4 x2 = 13 5 x1 – 2 x2 = 13

Zugehörige Eingabedatei:

```
3  4 13
5 -2 13
```

Das Script soll den Lösungvektor ausgeben und damit auch die Probe ausführen.

Verbesserungsmöglichkeiten: Nicht lösbare und nicht eindeutig lösbare Gleichungssysteme korrekt behandeln.

KAPITEL 11

Daten mit einer Nominalskala

11.1 Grundlagen

Nachdem wir bisher schon einige Auswertungen über Daten mit Nominalskala durchgeführt haben, wollen wir diese Erkenntnisse hier nochmal zusammenfassen und erweitern.

Daten dieser Art erhalten wir, wenn wir von unseren Beobachtungen (als den erhobenen Daten) eine Variable (Attribut, Eigenschaft) betrachten, die nur eine bestimmte Menge von Werten annehmen kann:

- Hauptstudiengang eines Studenten
- Qualitätskontrolle: Erzeugnis ist in Ordnung, nachbesserbar oder Ausschuss.
- Geworfene Zahl beim Roulette.

Man spricht auch von Klassen, in die sich die beobachteten Objekte bzw. deren Daten einteilen lassen.

Der erste Schritt der Auswertung ist eine Auszählung nach Kategorien oder Klassen.

Absolute Häufigkeit Anzahl der Daten in jeder Klasse.

Absolute Häufigkeitsverteilung (Tabellarische) Darstellung der Häufigkeit aller Klassen.

Relative Häufigkeit einer Klasse Anzahl der Beobachtungen einer Klasse im Verhältnis zur Gesamtzahl der Beobachtungen.

11.2 Gleichverteilung

Häufig stellt sich die Frage, ob die Klassen gleichberechigt sind, also ob jede Klasse mit gleicher Wahrscheinlichkeit angenommen wird. Dies ist für den fairen Verlauf von Glücksspielen (Würfel, Roulette) wichtig, aber z. B. noch viel entscheidender für die Schlüsselverteilung bei kryptographischen Verfahren. In [hatz14] findet man die Fragestellung, ob sich die Krankentage von Beschäftigten an Montagen häufen.

Um zum Experimentieren einfach Beobachtungsdaten erzeugen zu können, betrachten wir zunächst einen fairen Würfel, den wir n-mal werfen:

data <- **sample(**wertvektor, anzahl, **replace=T***[*, **prop=**vektor*]***)**

Der erste Parameter gibt einen Vektor der zulässigen Werte an, der zweite die Anzahl der Würfe. Mit dem dritten Parameter erlauben wir, dass die Werte mehrfach im Ergebnisvektor auftauchen dürfen. Der optionale vierte Paramter ist ein Vektor mit den Wahrscheinlichkeiten für das Ziehen eines bestimmten Wertes.

```
# fairer Würfel:
dice <- sample(1:6, 20, replace=T)
# gezinkter Würfel:
dice2 <- sample(1:6, 20, replace=T, prob=(c(rep(0.1, 5), 0.5))
```

Wenn wir Resultate später mit den gleichen Werten nachvollziehen zu können, ist es sinnvoll, diese Daten zu speichern:

write(vektor, **file="**dateiname**")**

Das Einlesen ist mit der Funktion `scan()` möglich:

vektor <- **scan("**dateiname**")**

Um sicherzustellen, dass wir wirklich mit einer Nominalskala arbeiten, wandeln wir die Zahlen in Faktoren um. Es ist sinnvoll, hier den Vektor aller Klassen anzugeben, falls in unseren Beobachtungsdaten bestimmte Klassen nicht auftauchen, die dann fehlen würden:

fdata <- **factor(**data, klassenvektor**)**

Für den Würfel verwenden wir

```
fdice <- factor(dice, 1:6)
```

Die absolute Häufigkeit einer Klasse im Vektor erhalten wir über die `length()`-Funktion über den Teilvektor, der nur diese Werte enthält:

numClass <- **length(**data**[**data==wert**])**

Häufiger interessiert uns aber nicht ein Einzelwert, sondern die Häufigkeitsverteilung:

table(vector)

Benötigen wir die relative Häufigkeitsverteilung, teilen wir entweder durch die Gesamtzahl

tabel**lsum(**tabelle**)**

oder benutzen die vorbereitete Funktion `prop.table()`

prop.table(tabelle**)**

Beispiel:

```
# Ausgabe Würfel, Häufigkeitstabelle
dice <- sample(1:6, 50, replace=T)
fdice <- factor(dice, 1:6)
tab <- table(fdice)
tab
# relativ
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
cat('\nAnteile:\n')
prop.table(tab)
# oder manuell
cat('\nmanuell:\n')
tab/sum(tab)
```

Ausgabe:

```
fdice
 1  2  3  4  5  6
 7  6  8 11  6 12

Anteile:
fdice
   1    2    3    4    5    6
0.14 0.12 0.16 0.22 0.12 0.24

manuell:
fdice
   1    2    3    4    5    6
0.14 0.12 0.16 0.22 0.12 0.24
```

Die Häufigkeit der Augenzahlen sehen wir graphisch recht deutlich mit einem Barplot, in den wir zusätzlich eine waagerechte Linie mit dem Erwartungswert für jede Augenzahl einfügen:

```
barplot(tab)
abline(h=sum(tab)/6)
```

Optisch ansprechend, aber bei annähernder Gleichverteilung weniger aussagekräftig ist ein Pie-Chart:

```
pie(tab)
```

11.3 Test auf Gleichverteilung

Um zu prüfen, ob ein Würfel «gezinkt» bzw. allgemein, ob die Verteilung unserer Daten von der Gleichverteilung abweicht, verwenden wir einen sogenannten *Hypothesentest*. Wir stellen ein Nullhypthtese H_0 auf (die Daten sind gleichverteilt), der die Alternativhypothese H_1 (die Daten sind nicht gleichverteilt) gegenübersteht. Nun bestimmen wir mit einem passenden Test die Wahrscheinlichkeit, dass die Nullhypothese für unsere Daten zutrifft. Ist diese kleiner als ein vorgegebener Schwellwert (z. B. 5%), lehnen wir unsere Nullhypothese ab.

Achtung: Wir erhalten nur eine Aussage darüber, mit welchem Fehler wir die Nullhypotese (Gleichverteilung) ablehnen, obwohl sie doch gilt, haben aber keinerlei Aussage, mit welcher Wahrscheinlichkeit wir fälschlicherweise Gleichverteilung annehmen, obwohl sie nicht gilt (Fehler zweiter Art). Je kleiner wir die Schranke für die Ablehnung der Nullhypothese machen, um so größer wird im allgemeinen der Fehler zweiter Art.

Für den Test auf Gleichverteilung verwenden wir den χ^2-Anpassungstest. Wir stellen dazu unseren beobachteten Häufigkeiten für jede Klasse o die erwarteten Häufigkeiten e gegenüber, bei Gleichverteilung gilt dabei e_i = *Klassenzahl/Beobachtungen*. Aus dieser Größe berechnen wir den Wert

$$X^2 = \sum_{i=1}^{K} \frac{(o_i - e_i)^2}{e_i}$$

Dieser Wert X^2 ist eine Schätzung für eine Zufallsgröße mit einer χ^2-Verteilung mit K-1 Freiheitsgraden. Je größer die Abweichung von den erwarteteten Häufigkeiten ist, um so größer ist auch der Wert X^2 und umso kleiner ist seine Wahrscheinlichkeit.

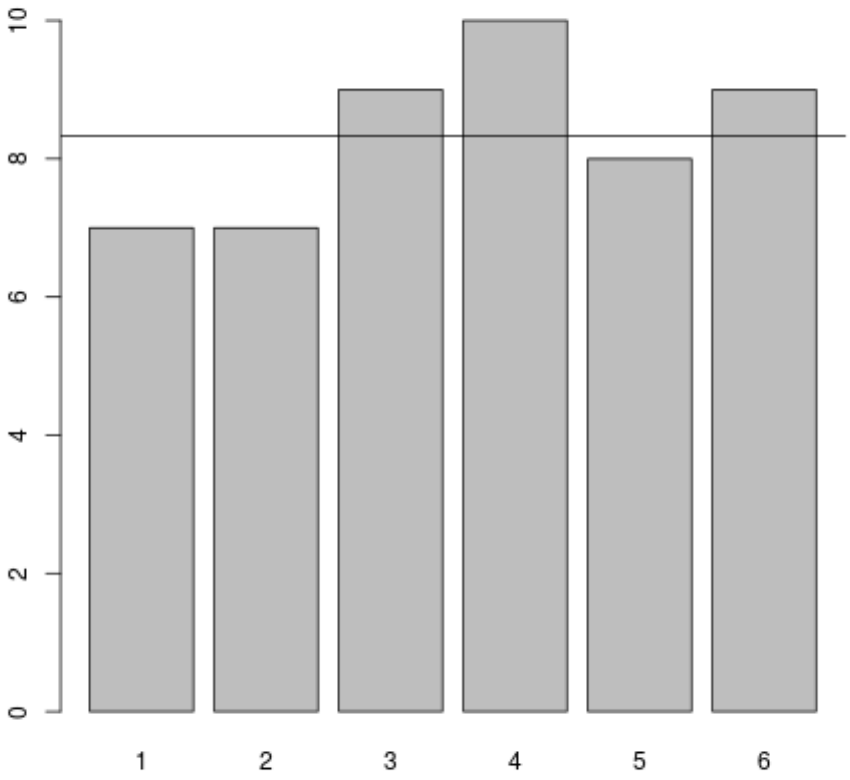


Abb. 11.1: Barplot Würfelergebnis mit Mittelwert

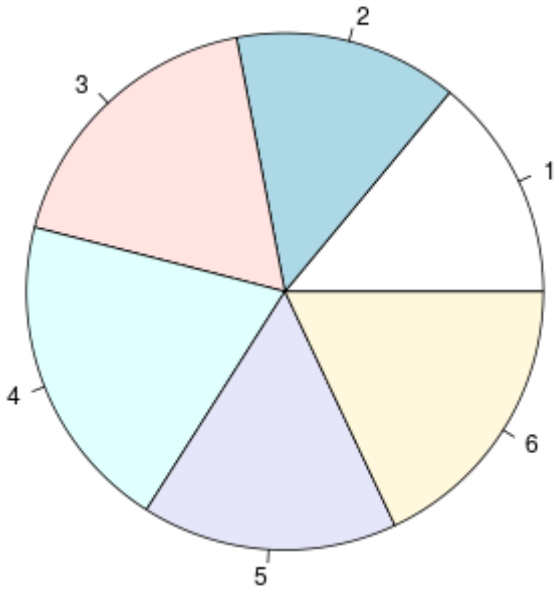


Abb. 11.2: Piechart Würfelergebnis

In *R* können wir diesen Test einfach nutzen, indem wir der Funktion `chisq.test()` die Tabelle bzw. einen Vektor der beobachteten Häufigkeiten übergeben. Die erwarteten Häufigkeiten werden automatisch nach einer Gleichverteilung berechnet.

Beispiel: Korrekter Würfel.

```
rm(list=ls())
dice <- sample(1:6, 50, replace=T)
fdice <- factor(dice, 1:6)
tab1 <- table(fdice)
cat("Verteilung\n")
print(tab1)
cat("chi^2-Test\n")
print(chisq.test(tab1))
```

Ergebnis:

```
Verteilung
fdice
 1  2  3  4  5  6
11 10  8  6  8  7
chi^2-Test

        Chi-squared test for given probabilities

data:  tab1
X-squared = 2.08, df = 5, p-value = 0.838
```

Wir lesen hier die Werte X^2, die Zahl der Freiheitsgrade (5, da wir 6 Kategorien haben) und die Wahrscheinlichkeit ab, dass die Nullhypothese (Gleichverteilung) gilt.

Beispiel: (extrem) gezinkter Würfel.

```
rm(list=ls())
dice <- sample(1:6, 50, replace=T,prob=c(rep(0.1,5), 0.5))
fdice <- factor(dice, 1:6)
tab1 <- table(fdice)
cat("Verteilung\n")
print(tab1)
cat("chi^2-Test\n")
print(chisq.test(tab1))
```

Ergebnis:

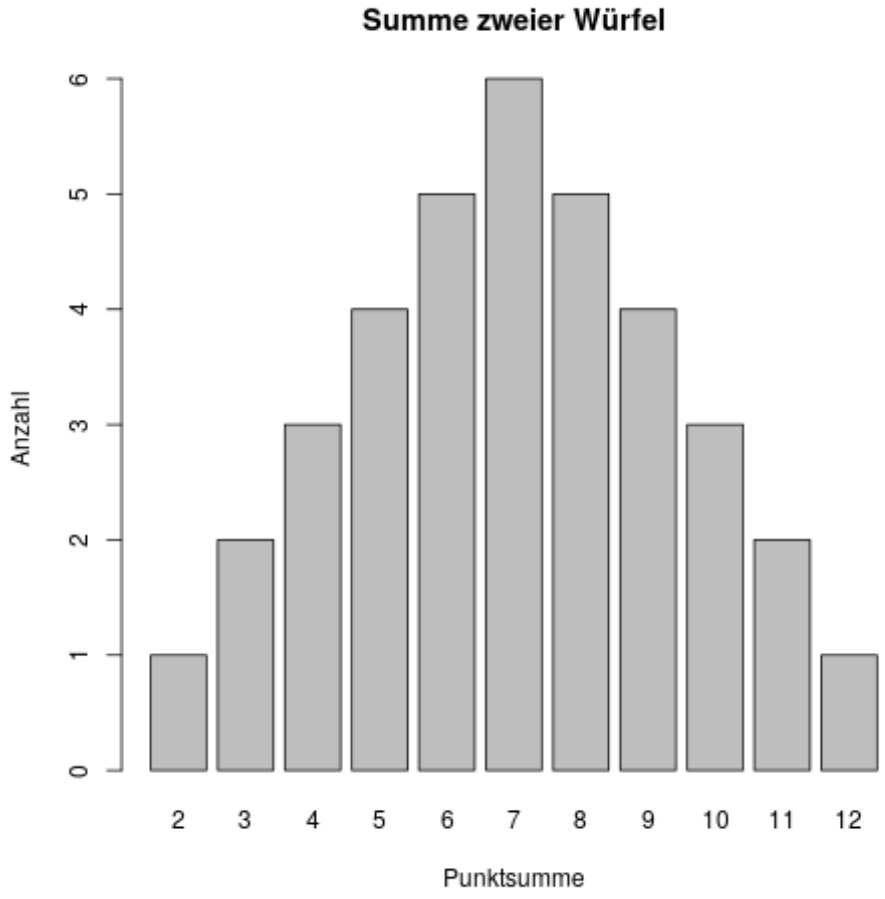
```
Verteilung
fdice
 1  2  3  4  5  6
 6  8  8  4  8 16
chi^2-Test

        Chi-squared test for given probabilities

data:  tab1
X-squared = 10, df = 5, p-value = 0.07524
```

Die Wahrscheinlichkeit für das Zutreffen der Nullhypothese ist hier extrem klein, wir werden diese deshalb ablehnen und davon ausgehen, dass der Würfel gezinkt ist.

11.4 Test auf andere Verteilungen

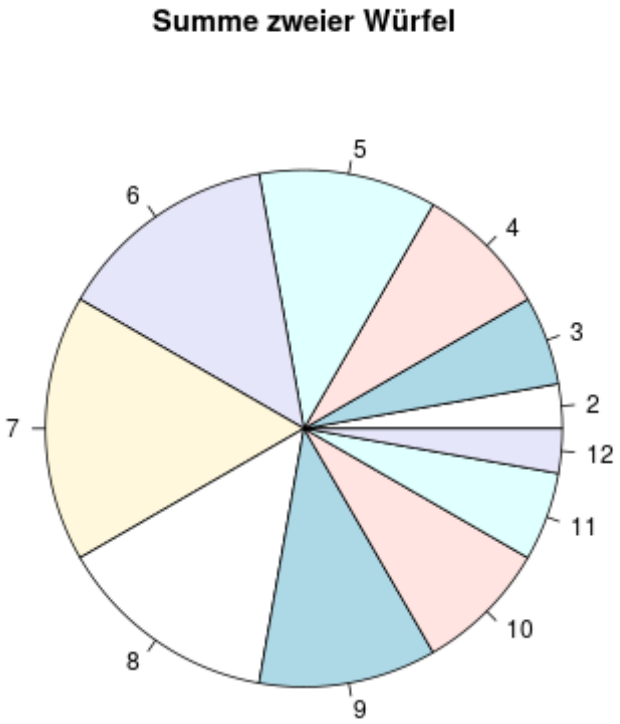
Würfeln wir mit zwei Würfeln und betrachten die Summe als Zufallsgröße, kann diese die Werte 2 bis 12 annehmen, aber sie ist mit Sicherheit nicht gleichverteilt: Wir haben 36 Elementarereignisse {1, 1}, {1, 2}, … {6, 6}, erhalten aber die Augenzahl 12 nur für ein einziges Ereignis, also mit Wahrscheinlichkeit 1/36, während wir z. B. die Augenzahl 5 für die Ereignisse {1, 4}, {2, 3}, {3, 2} und {4, 1}, also mit Wahrscheinlichkeit 4/36 erhalten. Um nicht alle Wahrscheinlichkeiten per Hand berechnen zu müssen, verwenden wir ein kleines R-Script:

```
# Generate all possible results of sum of 2 dices
setwd("~/lehre/R/samples/nominal")
k <- 0
results <- numeric() # create empty vector
for (i in 1:6) {
  for (j in 1:6) {
    k <- k+1
    results[k] <- i+j
  }
}
```



Und als Pie-Chart:

Um zu prüfen, ob unsere beobachteten Werte von der erwarteten Verteilung abweichen, können wir wieder den χ^2-Test verwenden, müssen aber nun die erwartete Häufigkeitsverteilung angeben. *Achtung*: Im Gegensatz zur obigen Formel für X^2 benötigen wir nicht die erwarteten Häufigkeiten, sondern deren Wahrscheinlichkeiten. Für unser Beispiel haben wir 36 verschiedene Elementarereignisse, wir berechnen also


Summe zweier Würfel
5
4
3
2
12
11
10
9
8
7
6

```
p.expected <- table(results)/length(results)
# oder
p.expected <- prop.table(table(results))
```

Wir erhalten die Tabelle

```
results
         2          3          4          5          6          7          8
0.02777778 0.05555556 0.08333333 0.11111111 0.13888889 0.16666667 0.13888889
         9         10         11         12
0.11111111 0.08333333 0.05555556 0.02777778
```

Wir werfen nun 200-mal mit einem korrekten und einem gezinkten Würfel (mit weniger Würfen erhalten wir meist keine klare Aussage):

```
x <- sample(1:6, 200, replace=T)
y <- sample(1:6, 200, replace=T, prob=c(rep(0.14,5),0.30))
fsum <- factor(x+y, 2:12)
```

Führen wir den χ^2-Test für `table(y)` (mit Gleichverteilung) aus, sollten wir als Ergebnis eine deutliche Ablehnung der Nullhypothese erhalten. Uns interessiert jedoch die Summe. Dazu erhält der Test außer der Tabelle der Beobachtungen als weiteren Parameter die erwarteten Wahrscheinlichkeiten:

```
chisq.test(table(fsum), p = p.expected)
```

Das Ergebnis könnte je nach den gewürfelten Werten etwa so aussehen:

```
        Chi-squared test for given probabilities

data:  table(fsum)
X-squared = 20.773, df = 10, p-value = 0.02273
```

Je nach den tatsächlich gewürfelten Werten erhalten wir ein mehr oder weniger klares Ergebnis, ob die Abweichung von der erwarteten Verteilung durch den Zufall bedingt ist oder auf eine Manipulation zurückgeführt werden muss.

11.5 Test auf Gleichverteilung zweier Gruppen

Wollen wir zwei Stichproben mit Nominalskala darauf vergleichen, ob diese die gleiche Verteilung haben, gehen wir ähnlich vor. Wir tabellieren beide Größen nebeneinander und bestimmen die Zeilen- und Spaltensummen (Randverteilung). Die Einzelwerte werden nun gegen die aus der Randverteilung erwarteten Werte verglichen.

Beispiel: Vergleich der AuP-Ergebnisse der Studiengänge B_In und B_Ma `aup-ws2017.dat`. Da jede Punktzahl nur sehr selten auftritt, fassen wir diese zu 4 gleichen Gruppen zusammen, die wir `zensur` nennen.

```
aup <- read.table("aup-ws2017.dat", header=T)
aup <- aup[aup$Teiln=="j",]
# Zusammenfassen
aup$zensur[aup$Summe <= 7] <- 4
aup$zensur[aup$Summe >  7 & aup$Summe <= 14] <- 3
aup$zensur[aup$Summe > 14 & aup$Summe <= 21] <- 2
aup$zensur[aup$Summe > 21] <- 1
# Auswahl B_In, B_Ma
aup2 <- aup[aup$StudGang == "B_In" | aup$StudGang == "B_Ma",]
```

Wir extrahieren nun die beiden Spalten `StudGang` und `zensur` und erzeugen daraus eine Kreuztabelle:

```
zensur <- factor(aup2$zensur, 1:4)
studg <- factor(aup2$StudGang) # muss neu erfolgen
tab.aup <- table(zensur, studg)
```

Anzeige von `tab.aup`:

```
      studg
zensur B_In B_Ma Sum
   1      0    3   3
   2      8    4  12
   3      5    1   6
   4      3    2   5
   Sum   16   10  26
```

Zum besseren Vergleich der unterschiedlich großen Gruppen zeigen wir die Daten besser relativ an. Wir bestimmen die Verhältnisse bezogen auf die Spaltensumme.

`prop.table`(tabelle, index)

Der Index gibt die Dimension an, 1 bedeutet zeilenweise, 2 spaltenweise Skalierung. Ohne Index werden die Werte relativ zur Summe aller Tabellenwerte berechnet.

```
proptab.col <- prop.table(tab.aup, 2)
```

Wir können hier einfach die Randverteilung (also alle Zeilen- und Spaltensummen) hinzufügen:

```
addmargins(proptab.col)
```

Ergebnis:

```
Relativ spaltenweise:
      studg
zensur   B_In   B_Ma    Sum
   1   0.0000 0.3000 0.3000
   2   0.5000 0.4000 0.9000
   3   0.3125 0.1000 0.4125
   4   0.1875 0.2000 0.3875
   Sum 1.0000 1.0000 2.0000
```

Zum optischen Vergleich erstellen wir einen Barplot, der die relativen Ergebnisse beider Gruppen (Studiengänge) nebeneinander darstellt:

```
# Spalten skalieren
barplot(proptab.col, col=1:4)
```

Ergebnis:

Für eine statistische Auswertung verwenden wir wieder den χ^2-Test, nun aber für unsere zweidimensionale Tabelle (Originaldaten, ohne hinzugefügte Randverteilung). Unsere Nullhypothese ist die gleiche Verteilung beider Daten, der ermittelte Wert gibt die Wahrscheinlichkeit an, mit der diese Hypothese gilt.

`chisq.test`(tabelle)

Für unser Beispiel erhalten wir (Anwenden auf `tab.aup`):

```
        Pearson's Chi-squared test

data:  tab.aup
X-squared = 6.1425, df = 3, p-value = 0.1049
```

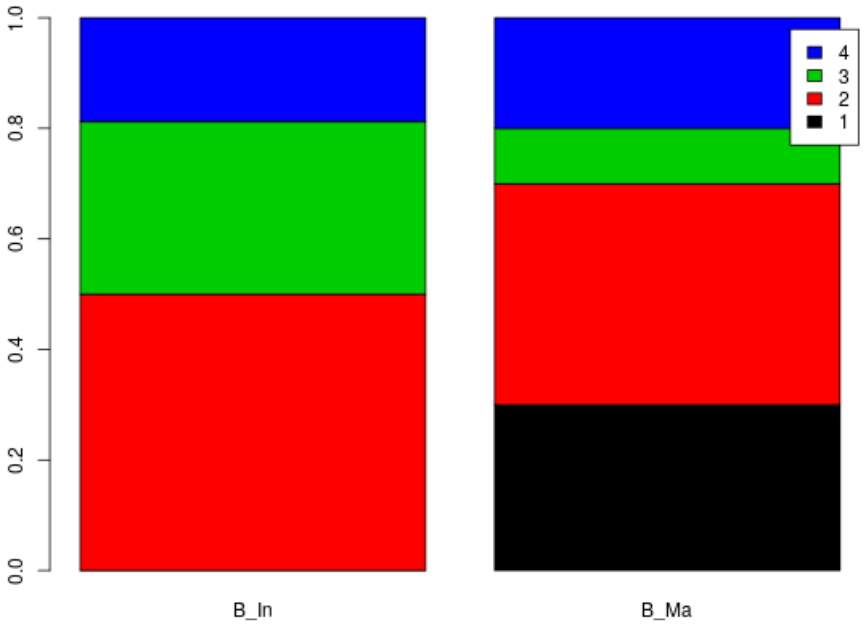


Abb. 11.3: Barplot Gruppenvergleich

Wir erhalten außerdem die Warnung:

```
Warning message in chisq.test(tab.aup):
"Chi-squared approximation may be incorrect"
```

Die Warnung erscheint aufgrund der recht kleinen Gruppengrößen, wir müssen die Testgenauigkeit mit Vorsicht genießen. Ein genaueres Ergebnis liefert der *Fisher*-Test für Zählgrößen mit kleinem Umfang:

```
fisher.test(tabelle)
```

Wir erhalten einen ähnlichen p-Wert:

```
        Fisher's Exact Test for Count Data

data:  tab.aup
p-value = 0.1374
alternative hypothesis: two.sided
```

Der p-Wert ist mit über 10% relativ groß, die Abweichungen der Punktverteilung in den Gruppen ist also nicht signifikant und kann auf zufällige Effekte zurückgeführt werden.

Notebook

Download: `aup-vergl.ipynb`.

Als weiteres Beispiel wollen wir zwei «gezinkte» Würfel vergleichen. Wir erzeugen zunächst zwei Vektoren der Würfelergebnisse, fügen die Nummer des Würfels hinzu und bauen daraus einen Datenframe. Jede Beobachtung enthält also die Würfelnummer und das Ergebnis.

```
count <- 50
dice1 <- sample(1:6, count, replace=T, p=c(0.15, 0.15, 0.15, 0.15, 0.15, 0.25))
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
dice2 <- sample(1:6, count, replace=T, p=c(0.25, 0.15, 0.15, 0.15, 0.15, 0.15))
fd1 <- factor(dice1, 1:6)
fd2 <- factor(dice2, 1:6)
# Data frame bauen
data1 <- cbind(rep(1, count),fd1)
data2 <- cbind(rep(2, count), fd2)
frame <- as.data.frame(rbind(data1, data2))
names(frame) <- c("Nr", "Res")
```

Wir erzeugen daraus analog eine Tabelle und wenden den χ^2-Test an:

```
tab <- table(frame$Nr, frame$Res)
chisq.test(tab)
```

Alternativ können wir uns den Umweg über den Datenframe sparen und die Tabelle direkt erzeugen, indem wir die Würfelergebnisse zusammenfassen und diese Spalten direkt zu einer Matrix zusammenfassen.

```
tab.dice1 <- table(fd1)
tab.dice2 <- table(fd2)
tab.dices <- cbind(tab.dice1, tab.dice2)
```

Wenn wir diese Würfel mit je 50 Würfen testen, erhalten wir z. B.

```
        Pearson's Chi-squared test

data:  tab
X-squared = 3.6167, df = 5, p-value = 0.6058
```

Wir erkennen also die unterschiedliche Verteilung nicht. Testen wir dagegen mit 500 Würfen, erhalten wir für die Tabelle und das Testergebnis z. B.

```
> tab.dices

  tab.dice1 tab.dice2
1        63       134
2        68        63
3        84        70
4        86        70
5        77        85
6       122        78

> chisq.test(tab.dices)

        Pearson's Chi-squared test

data:  tab.dices
X-squared = 38.768, df = 5, p-value = 2.644e-07
```

Die Wahrscheinlichkeit für eine gleiche Verteilung beider Würfel ist minimal, wir können nun recht sicher davon ausgehen, dass sie unterschiedliche Wahrscheinlichkeiten für die Augenzahlen haben.

11.6 Schätzung der Anteile, Konfidenzintervalle

Oft stehen wir vor der Aufgabe, aus einer Stichprobe einer Gesamtheit die tatsächlichen Anteile jedes Wertes zu schätzen:

- Anteil des Ausschusses der Gesamtproduktion, wenn in einer Stichprobe von 50 Produkten 2 fehlerhaft sind.
- Vorhersage des Wahlergebnisses aus einer Umfrage von wenigen Personen.
- Wirksamkeit eines Medikaments aus einer Studie an wenigen Patienten ableiten.

Beispiel: Zahl der in *Project Gutenberg Distributed Proofreaders* (pgdp.net) digitalisierten Bücher nach Genres `dp-bookgenre-2020.dat` (Januar–Oktober 2020) und `dp-bookgenre-2020.dat` (Januar–Oktober 2021).

Die Dateien enthalten zusätzlich noch die insgesamt digitalisierten Bücher bis zum jeweiligen Zeitpunkt, die wir zum Bewerten unserer Schätzung nutzen wollen.

Wir lesen die Daten ein, verwerfen die veraltete Gesamtzahl von 2020 und vereinigen die beiden Data Frames.

```
data.2020 <- read.table('dp-bookgenre-2020.dat', header=T)
data.2021 <- read.table('dp-bookgenre-2021.dat', header=T)

data.2020$Total <- NULL
data <- merge(data.2020, data.2021)
```

Wir beschränken uns auf die 3 häufigsten Genres.

```
data <- data[data$X2020 > 80,]
```

Ergebnis:

```
             Genre X2020 X2021 Total
29 General Fiction   176   144  5722
35         History    83    79  3030
40        Juvenile   186   122  4143
```

Zur Darstellung als Barplot und Vergleich der Verteilungen benötigen wir die Daten von 2020 und 2021 als Matrix. Wir berechnen weiterhin die relativen Anteile der Genres (spaltenweise).

```
mat <- as.matrix(data[,2:3])
rownames(mat) <- data$Genre

rmat <- prop.table(mat, 2)
# Kontrollanzeige
addmargins(rmat, 1)
```

Ergebnis:

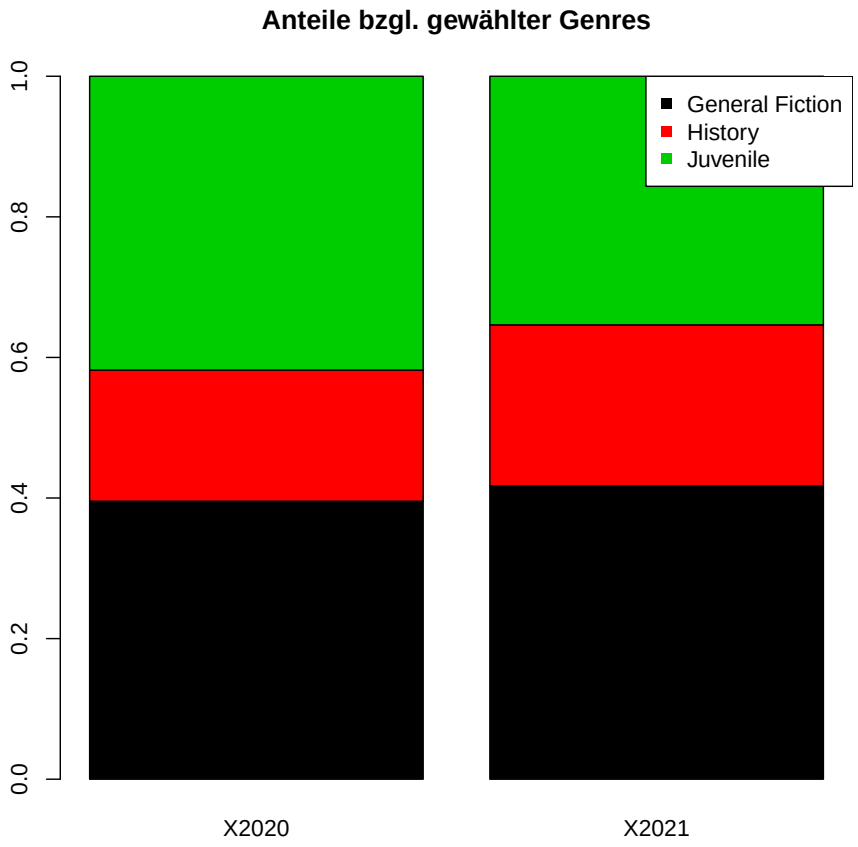
```
                    X2020     X2021
General Fiction 0.3955056 0.4173913
History         0.1865169 0.2289855
Juvenile        0.4179775 0.3536232
Sum             1.0000000 1.0000000
```

Diese stellen wir nun als Barplot dar:

```
barplot(rmat, col=1:nrow(mat),
  main='Anteile bzgl. gewählter Genres')
legend('topright', legend=rownames(rmat), col=1:nrow(rmat),
  pch=15, bg='white')
```

Auf den ersten Blick sieht die Verteilung der Genres relativ ähnlich aus.

Auch ein χ^2- bzw. Fisher-Test erlauben uns nicht, die Nullhypthese (Verteilung beider Jahre identisch) abzulehnen.


Anteile bzgl. gewählter Genres
1.0
0.8
0.6
0.4
0.2
0.0
General Fiction
History
Juvenile
X2020
X2021

```
chisq.test(mat)
```

Ergebnis:

```
        Pearson's Chi-squared test

data:  mat
X-squared = 4.0034, df = 2, p-value = 0.1351
```

```
fisher.test(mat)
```

Ergebnis:

```
        Fisher's Exact Test for Count Data

data:  mat
p-value = 0.1354
alternative hypothesis: two.sided
```

Hinweis: Beziehen wir in unsere Auswahl weitere Genres ein, verändert sich das Testergebnis, der exakte Fisher-Test scheitert komplett.

Unser Ziel ist nun, aus der Auswahl der 2020 erstellten Bücher (= Stichprobe) auf die Verteilung dieser Genres in der Gesamtmenge zu schließen.

Wir schätzen die Häufigkeit einfach als Quotient Anzahl der Beobachtungen mit dem gewünschten Merkmal und dem Umfang der Stichprobe.

$$\hat{p_i} = \frac{N_i}{N}$$

Diese Schätzung ist jedoch mit einer großen Unsicherheit behaftet, wie wir auch schon aus unseren Daten beim direkten Vergleich der relativen Häufigkeiten sehen können:

```
             Genre X2020 X2021 Total
29 General Fiction   176   144  5722
35         History    83    79  3030
40        Juvenile   186   122  4143
```

Wir versuchen deshalb einen Vertrauensbereich (Konfidenzintervall) zu schätzen: Dieses Intervall enthält mit einer vorgegebenen Wahrscheinlichkeit den tatsächlichen Anteil dieses Genres. Wenn wir die Wahrscheinlichkeit z. B. mit 90% festlegen und aus 10 gleichartigen, unabhängigen Stichproben das Konfidenzintervall berechnen, so wird im Mittel in 9 von diesen Intervallen der tatsächliche Mittelwert tatsächlich enthalten sein.

Dieses Intervall wird umso kleiner, je größer die Stichprobe ist und je besser diese den tatsächlichen Anteil des Genres widerspiegelt.

Wir können dieses Konfidenzintervall vereinfacht aus der geschätzten Varianz der Stichprobe ermitteln und erhalten die Formel:

$$\text{ci} = [\hat{p} - c, \hat{p} + c] \tag{11.1}$$

$$c = z_{\alpha/2} \cdot \sqrt{\frac{\hat{p} \cdot (1-\hat{p})}{n}} \tag{11.2}$$

Dabei beschreibt c die Schwankungsbreite, die sich aus dem geschätzen relativen Wert des Anteils $\hat{p}$, der Größe der Stichprobe n und einem Faktor z ergibt, der vom gewünschten Konfidenzniveau (also der Eintrittswahrscheinlichkeit) abhängt. Wenn wir von normalverteilten Daten ausgehen (was hier für die Zahlgröße von Nominaldaten nicht exakt gilt), verwendet man dafür das entsprechende Quantil der Normalverteilung, für 90% also $\alpha = 0.1/2 = 0.05$:

```
qnorm(0.05, 0.95)
```

Ergebnis:

```
[1] -1.644854  1.644854
```

Davon benutzen wir den positiven Wert (der Betrag ist natürlich aufgrund der Symmetrie der Normalverteilung gleich). Für ein Konfidenzlevel von 95% ($\alpha = 0.025$ bzw. 0.975) erhalten wir für z den Wert von 1.96.

Wir berechnen nun nach dieser Formel die Konfidenzintervalle für alle gewählten Genres (bzgl. aller Bücher) für die Auswahl von 2020:

```
ci.range <- 1.96 * sqrt(data$r2020 * (1 - data$r2020)/sum.2020)
ci.l <- data$r2020 - ci.range
ci.r <- data$r2020 + ci.range
# Zusammenfügen
ci.2020 <- data.frame(data$Genre, data$r2020, ci.l, ci.r)
names(ci.2020) <- c('Genre', 'X2020', 'ci.l', 'ci.r')
```

Ergebnis:

```
            Genre     X2020      ci.l      ci.r
1 General Fiction 0.3955056 0.3500750 0.4409362
2         History 0.1865169 0.1503251 0.2227086
3        Juvenile 0.4179775 0.3721504 0.4638046
```

Zum Vergleich können wir die Funktion `binom.test()` für eine genauere Schätzung des Konfidenzintervalls nutzen.

`binom.test`(treffer, gesamtZahl *[*, testwert, **`conf.level=`**prozent*]*)

Eigentlich testen wir die Nullhypthese, ob der Anteil der Treffer einem vorgegebenen Anteil entspricht. Uns interessiert hier aber nur das vom Test berechnete Konfidenzintervall. Wir berechnen es für das erste Genre:

```
binom.test(data$X2020[1], sum.2020, conf.level=0.95)
```

Ergebnis:

```
        Exact binomial test

data:  data$X2020[1] and sum2020
number of successes = 176, number of trials = 445, p-value = 1.208e-05
alternative hypothesis: true probability of success is not equal to 0.5
95 percent confidence interval:
 0.3497792 0.4426307
sample estimates:
probability of success
             0.3955056
```

Die Abweichung ergibt sich aus unserer vereinfachten Annahme einer Normalverteilung.

Wir stellen nun unsere Schätzung der Anteile der Genres einschließlich der Konfidenzintervalle graphisch als Balkendiagramm dar. Zum Darstellen der Bereiche benutzen wir die Funktion `segments()`

`segments`(x1, y1, x2, y2)

Wir übergeben vier Vektoren, (x1, y1) enthalten den Startpunkt, (x2, y2) den Endpunkt der Linie.

Diese Linien schließen wir noch mit einem Pfeil ab:

`arrows`(x1, y1, x2, y2*[*, **`code=`**wert, **`angle=`**winkel*]*)

Die Parameter (x1, y1) und (x2, y2) sind wieder die Segmentenden. Ohne weitere Parameter wird ein Pfeil bei den Endpunkten (x2, y2) gezeichnet. Mit dem Parameter `code=3` erhalten wir Pfeile an beiden Enden. Mit dem Winkel geben wir den Öffnungswinkel an, bei Angabe von 90 erhalten wir also eine gerade Linie. Zur Bewertung unserer Schätzung tragen wir noch die tatsächlichen Anteile der Genres für 2021 und für alle Bücher ein.

```
sum2021 <- sum(data$X2021)
sumtotal <- sum(data$Total)
r2021 <- data$X2021/sum2021
rtotal <- data$Total/sumtotal

centers <- barplot(ci.2020$X2020, ylim=c(0, 0.20))
segments(centers, ci.2020$ci.l, centers, ci.2020$ci.r)
arrows(centers, ci.2020$ci.l, centers, ci.2020$ci.r, code=3, angle=90)
points(centers, r2021, col='red')
points(centers, rtotal, col='blue')
```

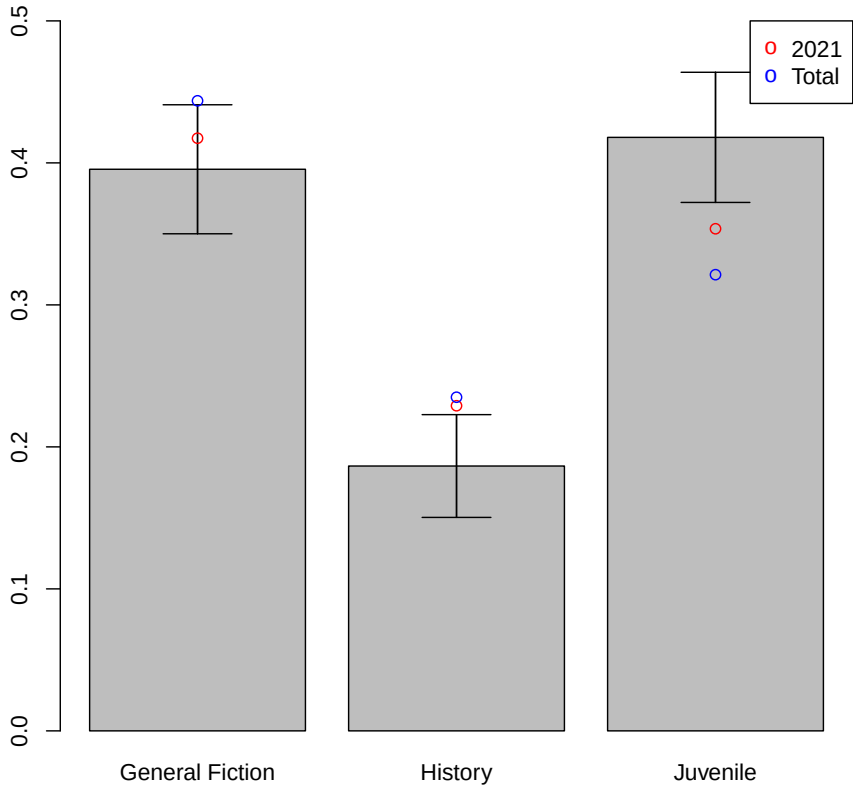


Abb. 11.4: Konfidenzintervalle aus den Anteilen von 2020. Tatsächliche Anteile 2021 und für alle Bücher.

Wir sehen hier, dass unsere Konfidenzintervalle für ein Genre die Anteile des Folgejahres und für alle Bücher enthalten, während die Schätzung für die beiden folgenden Kategorien leicht daneben liegt. Büchern enthalten, die Verteilung der Genres also Die Ursache ist hier ganz klar, dass die Auswahl der Projekte eines Jahres keine zufällige Stichprobe darstellen, da sich die Interessen und damit die Anteile der Genres im Laufe der Zeit (das Projekt besteht seit 2000) verschieben.

Notebook

Download: `ci-dp-genre.ipynb`. Das Notebook betrachtet zusätzlich die Anteile der 3 gewählten Genres bzgl. aller Buchkategorien, indem alle nicht betrachteten Genres zu einer neuen Kategorie zusammengefasst werden. Hier erkennt bereits der χ^2-Test, dass die Verteilungen von 2020 und 2021 nicht als gleich betrachtet werden können.

Untersuchung mit synthetischen Daten

Um zu demonstrieren, dass die Konfidenzintervalle tatsächlich eine gute Schätzung für den tatsächlichen Anteil darstellen, simulieren wir eine Produktion von Erzeugnissen mit einer Ausschussquote von 8%, der wir mehrere Stichproben von je 40 Objekten entnehmen und den Ausschuss zählen. Wir ermitteln jeweils das 90%-Konfidenzintervall und untersuchen, wie oft es den tatsächlichen Ausschußanteil überdeckt.

Da die R-Standardinstallation keine Funktionen für die Bernoulli-Verteilung (0-1-Verteilung mit Parameter p) enthält, ziehen wir Werte mit einer Gleichverteilung im Intervall $[0, 1]$ und setzen das Ergebnis auf 1, wenn wir einen Wert kleiner p, und 0, wenn wir einen Wert größer p erhalten:

```
val.unif <- runif(size, 0, 1)
val.bern <- as.integer(val.unif < p)
discard <- sum(val.bern)
```

Da wir eigentlich nur die Zahl der Einsen benötigen, die wir dann später selbst zählen, können wir mit Hilfe der Binomialverteilung auch gleich eine zusammengefasste Stichprobe erzeugen:

```
discard <- rbinom(1, size, p)
```

Bei der Berechnung der Konfidenzintervalle nach der vereinfachten Formel ist zu beachten, dass dieses nur dann ermittelt werden kann, wenn die Stichprobe mindestens 1 Ausschussteil enthält.

Wie dies genau geschehen kann, betrachten wir im folgenden Kapitel (*Programmieren in R*).

Schauen wir uns die berechneten Konfidenzintervalle an, so stellen wir fest, dass der linke Rand manchmal einen negativen Wert annimmt. Dieser Effekt kommt wieder durch die vereinfachte Annahme der Normalverteilung zustande.

Notebook

Download: `ci-discard.ipynb`.

11.7 Aufgaben

Aufgabe 11.1 Experimentieren Sie mit dem «gezinkten» Würfel mit verschiedenen Wahrscheinlichkeiten und verschiedenen Beobachtungszahlen. Wie viele verschiedene Augenzahlen treten tatsächlich auf? Wie sehr kann ich «schummeln», ohne das der χ^2-Test Auffälligkeiten bemerkt?

Aufgabe 11.2 Schreiben Sie ein Programm, dass aus einer (UTF-8-codierten) Textdatei einen Datenframe mit der Liste aller Wörter und den entsprechenden Wortlängen erzeugt. Beispiel:

```
Arbeit ist der Fluch der trinkenden Klasse. Oskar Wilde.
```

Daraus soll entstehen:

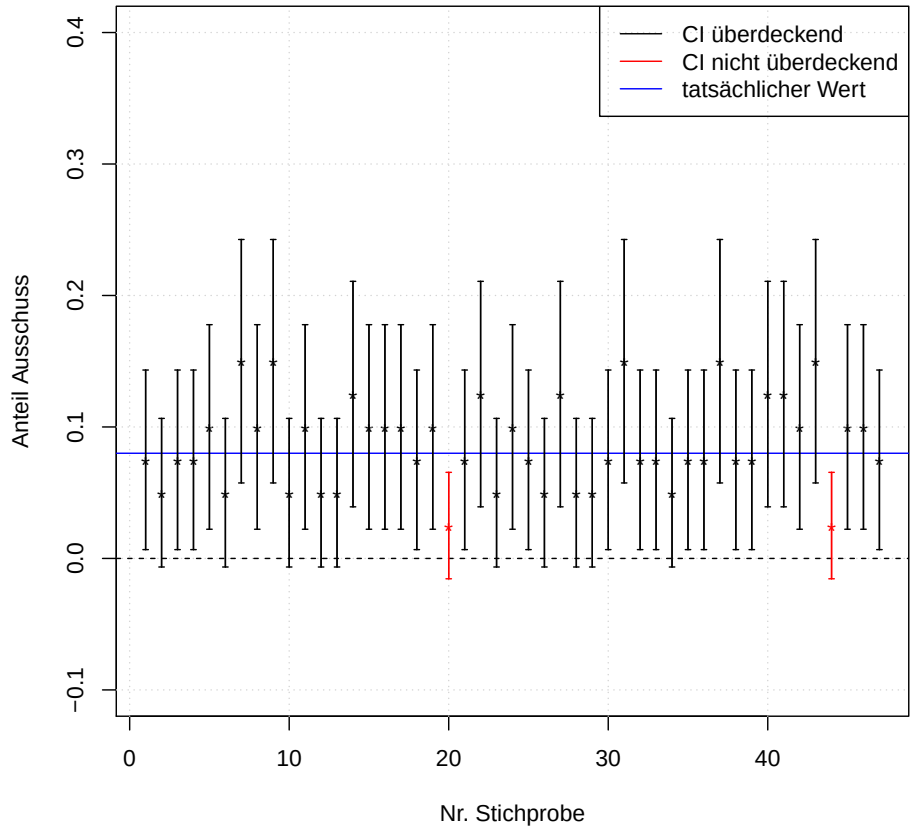


Abb. 11.5: Konfidenzintervalle von 50 Stichproben und deren Überdeckung des wahren Wertes.

```
Wort    Länge
Arbeit      6
ist         3
der         3
Fluch       5
der         3
trinkenden 10
Klasse      6
Oskar       5
Wilde       5
```

Hinweis: Das Programm kann in einer beliebigen Programmiersprache geschrieben sein, die im PC-Pool unter Linux verfügbar ist. Umlaute müssen allerdings korrekt behandelt werden.

Testen Sie nun einen deutschen Prosatext und einen deutschen wissenschaftlichen Text auf gleiche Verteilung der Wortlängen. Wiederholen Sie den Test mit dem deutschen und einem englischen Prosatext.

Hinweise zur Abgabe:

- Geben Sie die UTF-8-codierten Textdateien, das Programm zum Zerlegen der Texte mit ab. Die Namen der Eingabe- und Ausgabedatei sollten durch Programmparameter oder interaktive Eingaben festgelegt werden können.
- Die Texte sollten nicht zu kurz sein, um sinnvolle Ergebnisse zu erhalten.
- Ein getrenntes R-Programm soll die Auswertungen durchführen. Hier können die Dateinamen fest gewählt für Ihre Daten sein.
- Erstellen Sie auch eine graphische Auswertung der relativen Wortlängen (alle drei Texte gemeinsam) als Barplot und geben Sie das Diagramm (als PNG oder PDF) und auch den R-Code, mit dem es erzeugt wurde, ab.
- Geben Sie ebenfalls eine Datei mit Angaben zu den Textquellen und einer kurzen Interpretation Ihrer Ergebnisse mit ab.

Aufgabe 11.3 Tauchen in Statistiken metrische Zahlenwerte auf, zeigt sich sehr oft, dass deren erste Ziffern nach dem *Benford-Gesetz* verteilt sind. Überzeugen Sie sich davon, indem Sie einige Wochen Ihre Ausgaben notieren (sofern Sie nicht sowieso ein Kassenbuch führen):

```
Datum       Artikelgruppe    Summe Erste.Ziffer
2013-03-05  Lebensmittel     13.80      1
2013-03-07  Computer         34.34      3
...
```

Untersuchen Sie nun, ob die Verteilung der ersten Ziffer der Beträge diesem Gesetz folgt.

Hinweis: Verwenden Sie keine Einzelbeträge, sondern Summen über mehrere Artikel oder die Tagesausgaben, dann sollte der Effekt deutlich zu sehen sein.

Aufgabe 11.4 Erfassen Sie von Ihrem Lieblings-Reiseland (oder einem anderen Land) die 50 einwohnerstärksten Städte bzw. Orte sowie die Einwohnerzahlen. Eine dritte Spalte soll die erste Ziffer der Einwohnerzahl enthalten (entweder manuell erfassen oder später in *R* berechnen).

- Stellen Sie die Einwohnerzahlen der Städte in geeigneter, übersichtlicher Weise graphisch dar.
- Erstellen Sie eine Tabelle der Häufigkeiten der ersten Ziffer, stellen Sie diese textuell und in geeigneter Form graphisch dar.
- Testen Sie, ob die Verteilung der ersten Ziffer dem Benford-Gesetz entspricht.

Geben Sie das R-Script, Ihre Datendatei sowie die erzeugten Diagramme ab. In der R-Datei ist Ihre Quelle Ihrer Daten als Kommentar anzugeben.

Aufgabe 11.5 *Konfidenzintervalle*

Wählen Sie einen kurzen deutschen Text aus, zerlegen in Buchstaben, wandeln Sie alle Buchstaben in Kleinbuchstaben um und bestimmen Sie für die 5 häufigsten Buchstaben die erwarteten Häufigkeiten und Konfidenzintervalle (95%). Untersuchen Sie nun mit Hilfe eines längeren deutschen Textes, ob die Konfidenzintervalle die hier bestimmten Buchstabenhäufigkeiten überdecken. Vergleichen Sie nun auch mit den Buchstabenhäufigkeiten eines längeren englischen oder andersprachigen Textes.

KAPITEL 12

Programmieren in R

12.1 Ausführen von R-Scripten

```
source(file="filename")
```

12.2 Funktionen

Zum Strukturieren größerer Scripte und zur Wiederverwendung bestimmter Algorithmen verwenden wir Funktionen:

funktionsname <- **function(**</k>parameterliste**) {**
    Anweisungsfolge
[
    return(ausdruck**)**
]
}

Parameter werden als symbolische (lokal gültige) Namen einfach aufgeführt, eine Typspezifikation erfolgt nicht im Funktionskopf. Mit einer **return**-Anweisung wird die Funktion beendet und der Wert des berechneten Ausdrucks zurückgegeben. Fehlt die <code>return</code>-Anweisung, wird der Wert des zuletzt berechneten Ausdrucks zurückgegeben.

Beispiel: Funktion für 6-Sigma (sechsfache Standardabweichung, bei normalverteilten Werten liegen nur 1 Millionstel der Beobachtungen außerhalb des 6-Sigma-Intervalls um den Mittelwert):

```
sigma.6 <- function(x) { sd(x)*6; }
```

Die Parameter können auch Vorgabewerte erhalten:

parameterkw:= vorgabewert

Wird beim Aufruf der Funktion dieser Parameter weggelassen, wird statt dessen der Vorgabewert benutzt. Beispiel: Ziehen einer Bernoulli-verteilten Stichprobe mit Vorgabeumfang von 50 und p-Wert von 0.05.

```
getData <- function(size=50, p=0.05) {
  data <- runif(50) # Bernoulli-Verteilung nicht im Standard
  return(as.integer(data < p))
}
```

Diese Funktion kann nun mit oder ohne Paramtern aufgerufen werden:

```
x1 <- getData()                # size=50, p=0.05
x2 <- getData(20, 0.1)         # size=20, p=0.1
x3 <- getData(p=0.1, size=20)  # ditto, umgekehrte Reihenfolge
x4 <- getData(30)              # size=30, p=0.05
x5 <- getData(p=0.2)           # size=50, p=0.2
```

Da Funktionen ebenfalls Objekte sind, können auch diese als Parameter übergeben werden. Beispiel: Anwenden einer Funktion auf alle Elemente eines Vektors (eine solche Funktion steht mit `lapply()` in *R* zur Verfügung).

```
myApply <- function(x, fun) { fun(x) }

 x <- 1:4
 # Alle Werte addieren
 myApply(x, sum)
 # Funktion auf jeden Wert anwenden</c>
 myApply(x, sqrt)
 # Anonyme Funktion auf jeden Wert anwenden</c>
 myApply(x, function(x) {x*x})
```

12.3 Verzweigungen

Einfache Verzweigungen werden ähnlich wie in C realisiert:

if (bedingung) **{**
    dann_Anweisungen
[
} else {
    sonst_Anweisungen
]
}

Wird nur eine Anweisung pro Zweig benutzt, können die geschweiften Klammern entfallen.

Beispiel: Eine Stichprobe einer Bernoulli-Verteilung aus einer Stichprobe der Gleichverteilung erzeugen: Ist der Wert kleiner p wählen wir 1, sonst 0.

```
sample.unif <- runif(1)
if (sample.unif < p) {
  sample.bern <- 1
} else {
  sample.bern <- 0
}
```

Eine alternative funktionale Form entspricht dem <code>?:</code>-Operator.

ifelse(test, dannAusdruck, sonstAusdruck)

Beispiel: Stichprobe vom Umfang `size` einer Bernoulli-Verteilung aus der Stichprobe einer Gleichverteilung erzeugen.

```
sample.unif <- runif(size)
sample.bernoulli <- ifelse(sample.unif < p, 1, 0)
```

Eine Mehrfachverzweigung kann entweder mit einer <code>if-else if</code>-Kette oder der `switch()`-Funktion realisiert werden. Letztere erwartet als ersten Parameter entweder einen ganzzahligen oder einen Stringausdruck.

```
switch(intAusdruck1,
       cmd1,
       cmd2,
       ...
)
```

Wird der Ausdruck zu 1 ausgewertet, dann wird <code>cmd1</code> ausgeführt etc. Das Kommando kann dabei ein in `{}` geklammerter Block sein.

Beispiel: Einteilen Buchstaben (a...e) in Vokale und Konsonanten.

```
classifyLetter <- function(ch) {
  switch(as.integer(ch),
         print("vokal"),       # a
         print("konsonant"),   # b
         print("konsonant"),   # c
         print("konsonant"),   # d
         print("vokal")        # e
  )
}

txt <- c("a", "b", "e", "a", "b")
txt <- factor(txt, c("a", "b", "c", "d", "e"))
print(txt[1])
classifyLetter(txt[1])
print(txt[2])
classifyLetter(txt[2])
```

Ergebnis:

```
[1] a
Levels: a b c d e
[1] "vokal"
[1] b
Levels: a b c d e
[1] "konsonant"
```

Ist der Ausdruck dagegen ein String, hat die `switch()`-Funktion folgenden Aufbau:

```
switch(stringAusdruck,
       "wert1" = anw1,
       ...
[
       defaultAnweisung
]
)
```

12.4 Schleifen

Für wiederholte Vorgänge stellt *R* verschiedene Schleifentypen zur Verfügung. Die `while`-Schleife ist eine abweisende Schleife und verhält sich wie in den meisten Programmiersprachen.

```
while (bedingung) {
    Anweisungen
}
```

Zum Durchlaufen aller Elemente eines Vektors verwendet man die `for`-Schleife.

```
for (name in vektor) {
    Anweisungen
}
```

Beispiel: Erzeugen von maximal 20 Stichproben der Bernoulli-Verteilung, wobei jede Stichprobe mindestens einen «Treffer» (Wert 1) enthält. Die Zahl der Treffer werden in einem Vektor gespeichert. Die Stichproben, die ausschließlich aus 0-Werten bestehen, sollen ausgesondert werden. Wir verwenden die Funktion `getData()` von oben.

```
samples <- c() # leerer Vektor
for (i in 1:20) {
  sample <- getData(size=40, p=0.08)
  ones <- sum(sample)
  if (ones > 0) {
    samples <- c(samples, ones) # Trefferzahl anhängen
  }
}
```

Hinweis: Dieser Code kann durch die *R*-Funktion `rbinom()` ersetzt werden.

Eine weitere Schleifensteuerung ist mit `break` bzw. `next` möglich. Mit der ersten Anweisung wird die Schleife sofort verlassen, mit der zweiten wird der weitere Schleifenkörper übersprungen und zum Schleifenkopf mit dem Test zurückgekehrt. Die

Beide Steueranweisungen sind nur in Zusammenhang mit einer bedingten Anweisung (`if()`) sinnvoll.

Unsere Stichprobenauswahl kann also auch wie folgt formuliert werden:

```
samples <- c() # leerer Vektor
for (i in 1:20) {
  sample <- getData(size=40, p=0.08)
  ones <- sum(sample)
  if (ones == 0) next
  samples <- c(samples, ones) # Trefferzahl anhängen
}
```

R hat eine eigene Anweisung für Endlosschleifen, die sich auch mit `while(T)` realisieren ließe. Das Verlassen der Schleife ist nur mit der Schleifensteuerung `break` möglich:

```
repeat {
    Anweisungen
    ...
    if (...) break
    Anweisungen
}
```

Beispiel: Schlechter Code für die Suche nach der nächsten Primzahl unter Verwendung von `next` und `break`.

```
nextPrime <- function(n) {
  repeat {
    n <- n+1                  # Teste Nachfolger
    for (d in 2:n) {          # Teste Teiler
      if (n %% d == 0) break # Falls Primzahl, dann bei d==n erfolgreich
    }
    if (d != n) next          # Teiler gefunden -> weitersuchen
    return(n)                 # Ergebnis gefunden
  }
}
```

Warnung: Die Schleifensteuerungsanweisungen <code>break</code> und <code>next</code> sollten nur dann benutzt werden, wenn dadurch der Code besser lesbar wird. Im angegebenen Beispiel ist das nicht der Fall!

KAPITEL 13

Metrische Größen

13.1 Verteilung

Für eine metrische Variable interessiert uns im allgemeinen deren Verteilung:

- Über welches Intervall erstrecken sich die Werte?
- Bei welchen Werten konzentrieren sich die Daten?
- Gibt es ein Zentrum, um das sich die Werte konzentrieren?
- Wie stark schwanken die Werte um dieses Zentrum?
- Liegen die Daten symmetrisch zum Zentrum?

Wir gehen davon aus, dass unsere Daten als Vektor vorliegen, z. B. eine Spalte eines Datenframes.

Die einfachsten Kenngrößen sind Mittelwert, Varianz bzw. Standardabweichung sowie die Quartile:

```
# Zusammenfassung
summary(x)
# Mittelwert
mean(x)
# Standardabweichung, Varianz
sd(x)
var(x)
# Quartile:
quantile(x)
```

Statt der *Quartile* (0% = Minimum, 25%, 50% = Median, 75%, 100% = Maximum) können wir auch *Quantile* für beliebige Anteile berechnen.

Einen optischen Eindruck über die Verteilung liefert uns ein *Stem-Diagram*, das uns unter anderem über die Balkenlänge die Datenverteilung darstellt.

```
stem(x)
```

Beispiel: Statistische Kenngrößen der Verteilung der Eruptionsdauer der Ausbrüche des *Faithful*-Geysirs.

```
erup <- faithful$eruptions
cat('Mittel: ', mean(erup), '\n')
cat('Standardabw.: ', sd(erup), '\n')
cat('Varianz: ',var(erup), '\n')
cat('Quartile: ', quantile(erup), '\n')
cat('10%-Quantile: ', quantile(erup, seq(0, 1, 0.1)))
cat('Stem-Diagramm:\n')
cat(stem(erup),'\n')
```

Ergebnis:

```
Mittel:  3.487783
Standardabw.:  1.141371
Varianz:  1.302728
Quartile:  1.6 2.16275 4 4.45425 5.1
10%-Quantile:  1.6 1.8517 2.0034 2.3051 3.6 4 4.167 4.3667 4.533 4.7 5.1Stem-Diagramm:

  The decimal point is 1 digit(s) to the left of the |

  16 | 070355555588
  18 | 000022233333335577777777888822335777888
  20 | 00002223378800035778
  22 | 0002335578023578
  24 | 00228
  26 | 23
  28 | 080
  30 | 7
  32 | 2337
  34 | 250077
  36 | 0000823577
  38 | 2333335582225577
  40 | 000000335778888800223355557778
  42 | 033355557788002333335555777 8
  44 | 022223355577800000002333335777 8888
  46 | 0000233357700000023578
  48 | 00000022335800333
  50 | 0370
```

Eine graphische Darstellung der Verteilung ist mit der Funktion `hist()` möglich. Mit dem Boolean-Parameter `freq` können wir zwischen absoluter und relativer Anzahl unterscheiden. Zur Darstellung wird der Wertebereich in Intervalle zerlegt. Das Diagramm zeigt die Anzahl der Werte, die in dieses Intervall fallen.

```
hist(x[, freq=F])
```

Beispiel: Verteilung Eruptionsdauern *Faithful*-Geysir.

```
erup <- faithful$eruptions
hist(erup, freq=F)
```

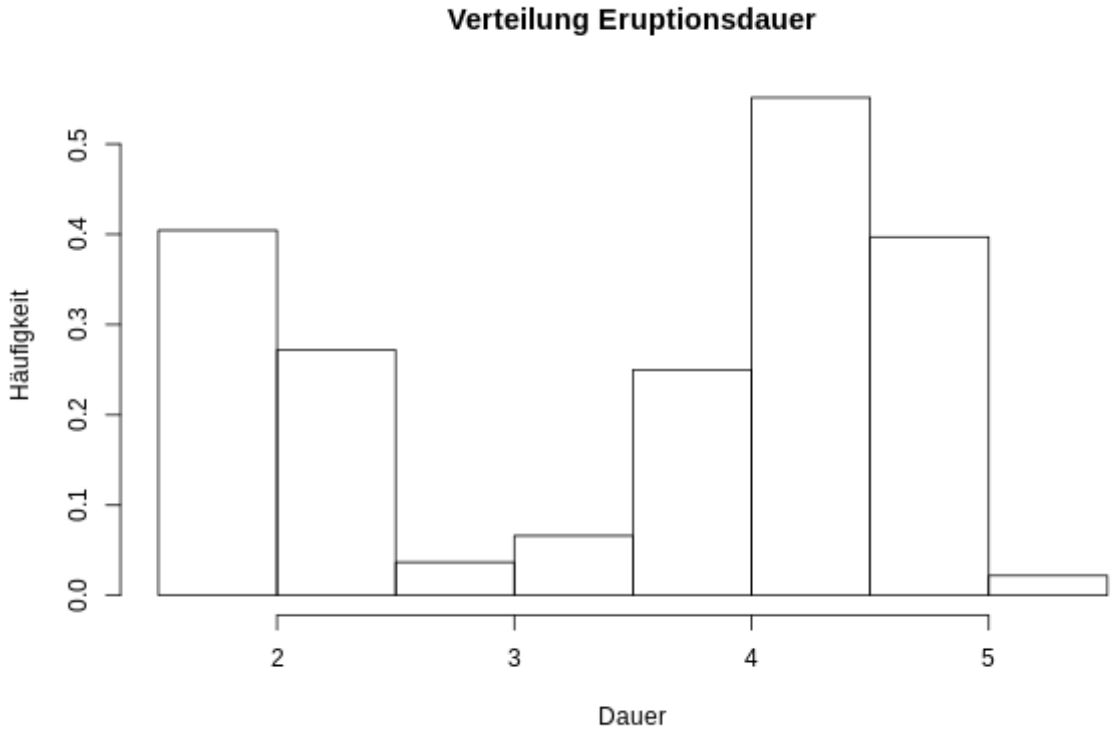


Abb. 13.1: Verteilung Eruptiondauer

Ein Boxplot zeigt uns die Streuung der Werte gut an.

```
boxplot(x)
```

Beispiel: Streuung der Eruptionsdauer.

```
boxplot(erup)
```

Wir können auch die Quantile graphisch darstellen, indem wir die Werte sortieren und danach jedem Wert seine relative Position innerhalb aller Werte zuordnen. Wir erzeugen uns dazu eine x-Scala von 0 bis 1 mit ebensoviel Werten, wie unser Datenvektor enthält.

```
sortVal <- sort(x)
x <- 1:length(x)/length(x)
plot(x, sortVal, ty='l', xlab='Anteil', ylab='Quantil')
```

Auf der x-Achse stellen wir diese Anteile (0 bis 1 = 100%) dar, auf der y-Achse können wir das Quantil ablesen: das ist der Wert unter dem dieser Anteil der Werte liegt. In der Graphik ist das Quantil für 0.2 hervorgehoben: wir lesen im Diagramm den Wert 2 ab. Mit dem Funktionsruf `quantile(x, 0.2)` vergleichen wir mit dem berechneten Wert.

Wir erkennen auch hier am Verlauf der Quantilkurve, dass wir zwei Häufungen haben, einmal für relativ kleine Werte um die Dauer 2, dann erfolgt ein steiler Anstieg bis ca. 3.5, ehe die Kurve wieder abflacht, was auf eine große Zahl von Werten in diesem Bereich hinweist.

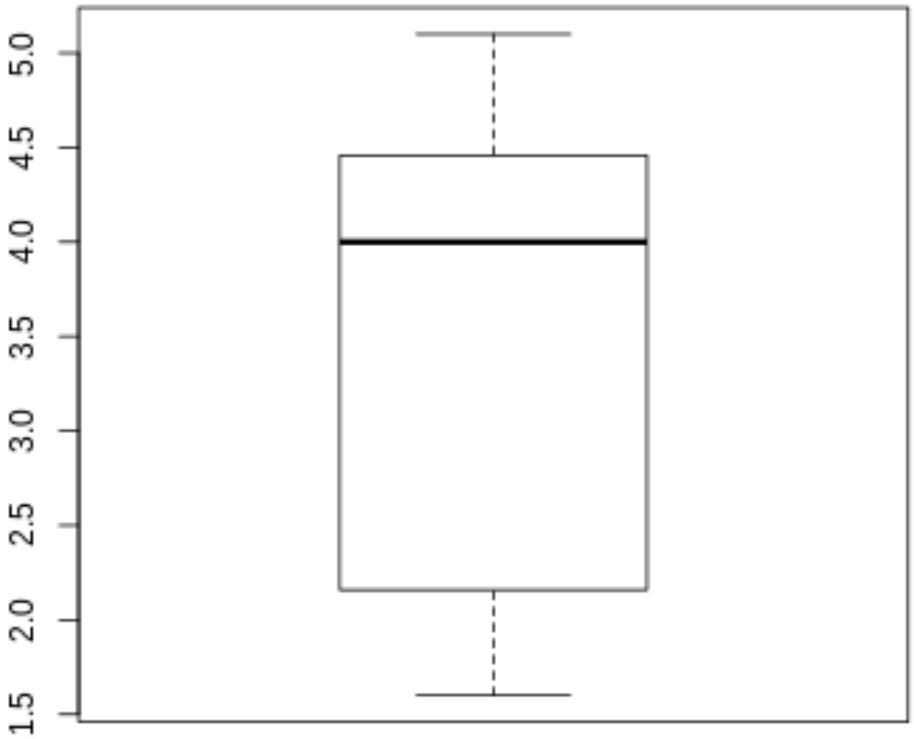


Abb. 13.2: Streuung Eruptiondauer

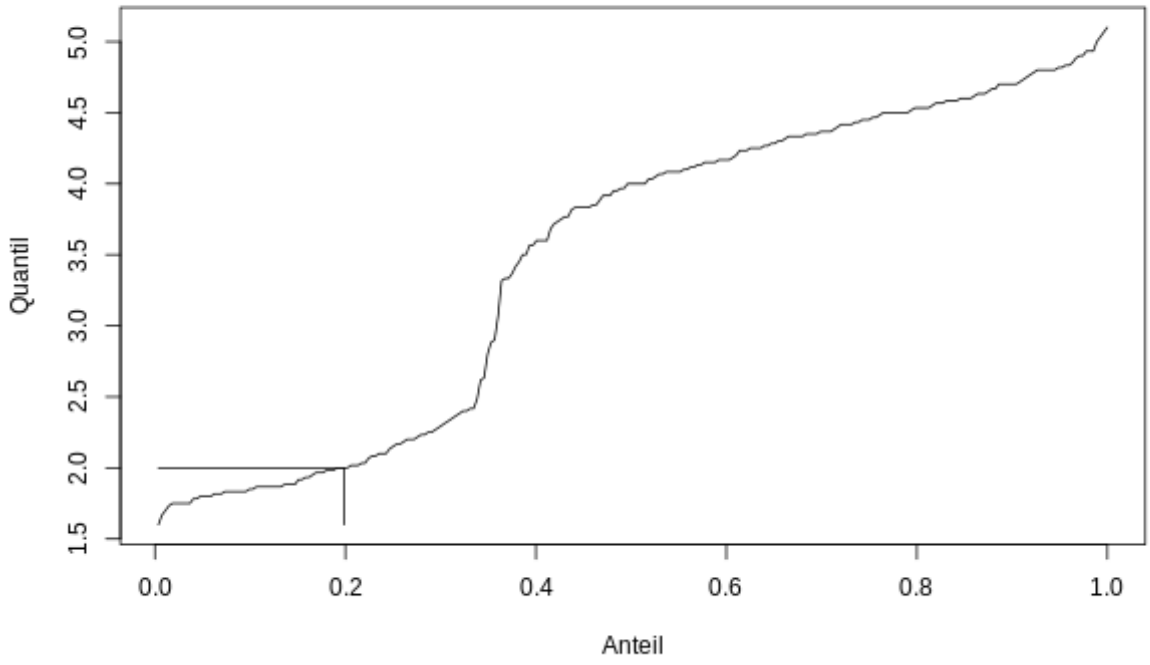


Abb. 13.3: Quantile Eruptiondauer

13.2 Vergleich mit einer Verteilung

Wollen wir aus den Daten Modelle bilden, um weitere Vorhersagen zu treffen, ist es sinnvoll zu prüfen, ob die Daten unserer Beobachtung einer bestimmten Wahrscheinlichkeitsverteilung folgen. Dies können wir zunächst optisch untersuchen, indem wir die Verteilung unserer Daten mit der zu erwartenden Datenverteilung nach einer bestimmten Verteilungsfunktion vergleichen.

Wir wollen hier zunächst auf Normalverteilung prüfen. Da unsere *Faithful*-Daten zwei Maxima haben, ist hier eher keine Normalverteilung zu erwarten, wir wählen uns deshalb einen weiteren Datensatz, und zwar die Krankenstunden der letzten Jahre in Deutschland, die wir auf deutschlandinzahlen.de finden: `krankenstunden.csv`. Die heruntergeladenen Daten wurden vorher etwas aufbereitet (Kopfzeilen zu Kommentaren umgewandelt, leere Spalten gelöscht).

```
kr <- read.csv2('krankenstunden.csv', header=T, comment='#')
names(kr) <- c('Jahr', 'Stunden')
print(summary(kr$Stunden))
cat('Std-Abw. ', sd(kr$Stunden), '\n')
```

```
 Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
48.68   55.16   62.57   64.25   68.44   89.44

Std-Abw.  12.38592
```

Wir plotten nun wiederum das Histogramm dieser Daten (als Relativwerte) und legen darüber die Kurve einer Normalverteilung mit dem geschätzten Mittelwert und der geschätzten Varianz. Als Intervall verwenden wir den Bereich zwischen minimalem und maximalem Beobachtungswert. Die Dichte der Normalverteilung erhalten wir mit dem Funktionsruf `dnorm()`:

dichte <- **dnorm**(x**[, mean=**mittelwert**, sd=**stdAbweichung**]**)

Für den Krankenstand gehen wir wie folgt vor:

```
# x-Werte für Kurve erzeugen
x <- seq(min(kr$Stunden), max(kr$Stunden), 0.1)
# Histogramm (relative Werte)
hist(kr$Stunden, freq=F)
# Kurve Normalverteilung darüberlegen
curve(dnorm(x, mean=mean(kr$Stunden), sd=sd(kr$Stunden)), add=T)
```

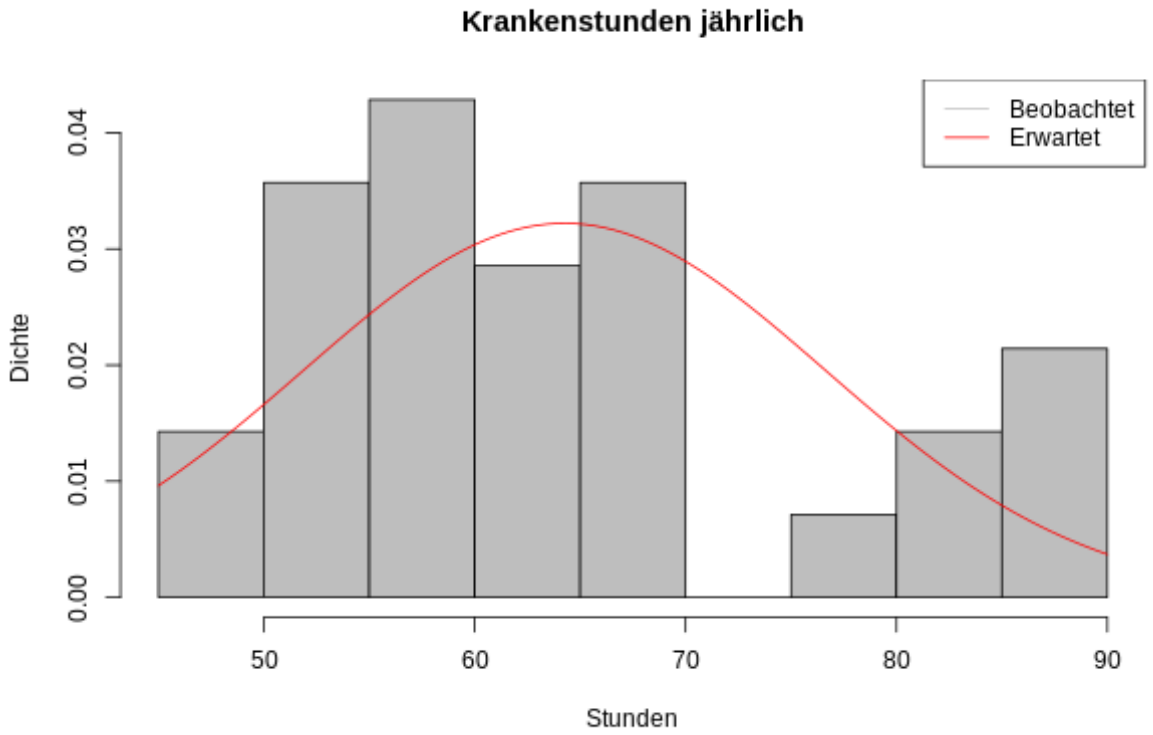


Abb. 13.4: Krankenstunden beobachtet/erwartet nach Normalverteilung

---

Hinweis: Für den Funktionsruf `curve()` ist der Name des Funktionsparameters zwingend `x`.

Beim Aufruf von `curve()` muss der Parameter `add=T` angegeben werden, damit die Kurve über dem vorigen Diagramm und nicht als neues Diagramm ausgegeben wird.

Aussagekräftiger als der Vergleich des Histogramms mit der erwarteten Dichtefunktion ist der Vergleich der *Quantile* zwischen Beobachtung und erwarteter Verteilung. Dazu sortieren wir einfach unsere Daten, die direkt den beobachteten Quantilen entsprechen. Die Wahrscheinlichkeiten berechnen wir dazu an den Punkten 1/n, 2/n, … n/n, wobei wir diese ein wenig nach links verschieben, da der Punkt 1 uninteressant ist (Wahrscheinlichkeit 1 umfasst alle Werte).

Wir können für alle gängigen Wahrscheinlichkeitsverteilungen die Quantile mit der Funktion `qverteilung` bestimmen.

quantile <- `q`verteilung`(`wahrscheinlichkeitsvektor, verteilungsparameter`)`

Für die Normalverteilung verwenden wir `qnorm()` mit den Parametern `mean` und `sd`, die wir aus den beobachteten Daten schätzen.

```
sortkr <- sort(kr$Stunden)
# Beobachtete Anteile = p
q <- (1:length(kr$Stunden)-0.5)/length(kr$Stunden)
# Erwartete Quantile
expect <- qnorm(q, mean=mean(sortkr), sd=sd(sortkr))
# Beobachtung
plot(q, sortkr, ty='l', col='red', xlab='p', ylab='Quantil')
# Erwartung
lines(q, expect, ty='l', col='blue')
legend('topleft', legend=c('Beobachtet', 'Erwartet'), col=c('red', 'blue'),
       lty=1)
```

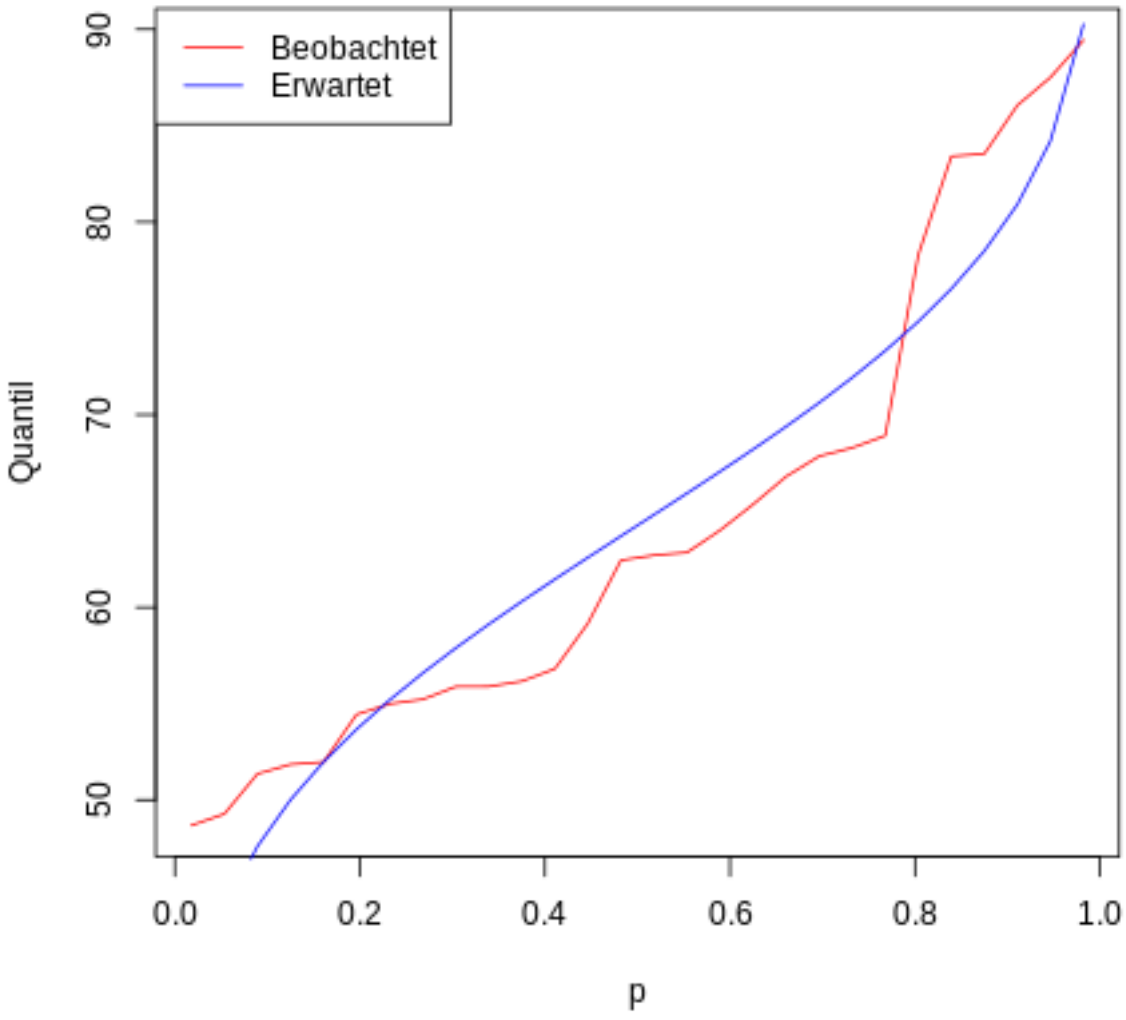


Abb. 13.5: Quantile Krankenstunden gegen Quantile Normalverteilung)

Der Vergleich wird einfacher, wenn wir einfach die beobachteten und erwarteten Quantile als x- und y-Werte betrachten und diese darstellen. Bei Übereinstimmung liegen diese alle auf einer Linie. Diese können wir einfach zum Vergleich ergänzen.

`qqplot(`erwarteteQuantile, beobachtete Quantile`)`

```
# Vergleichsgerade
abline(0, 1)
```

Beispiel: Krankenstand.

```
qqplot(expect, sortkr)
abline(0,1)
```

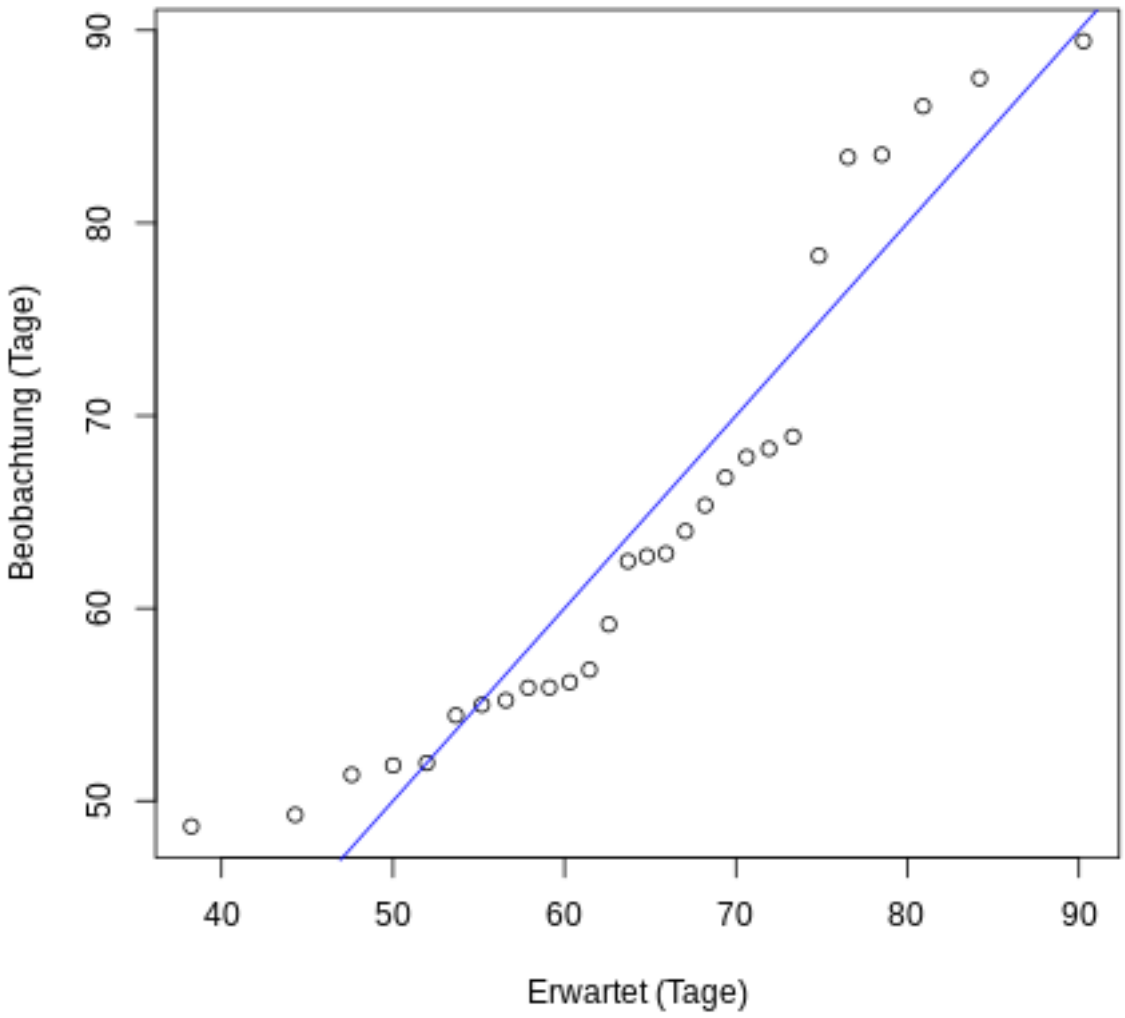


Abb. 13.6: QQ-Plot Krankenstunden (nach Normalverteilung)

Alternativ kann für Normalverteilung auch die Funktion `qqnorm()` benutzt werden. Hier können wir das Sortieren der Daten weglassen. Da hier mit einer standardisierten Normalverteilung verglichen wird, führt unsere Modellgerade nicht mehr durch die Punkte (0,0) und (1,1), sondern kann statt dessen mit der Funktion `qqline()` erzeugt werden.

```
qqnorm(vektor)
# Modellgerade
qqline(vektor)
```

Beispiel: Eruptionsdauer *Faithful*.

```
qqplot(faithful$eruptions)
qqline(faithful$eruptions)
```

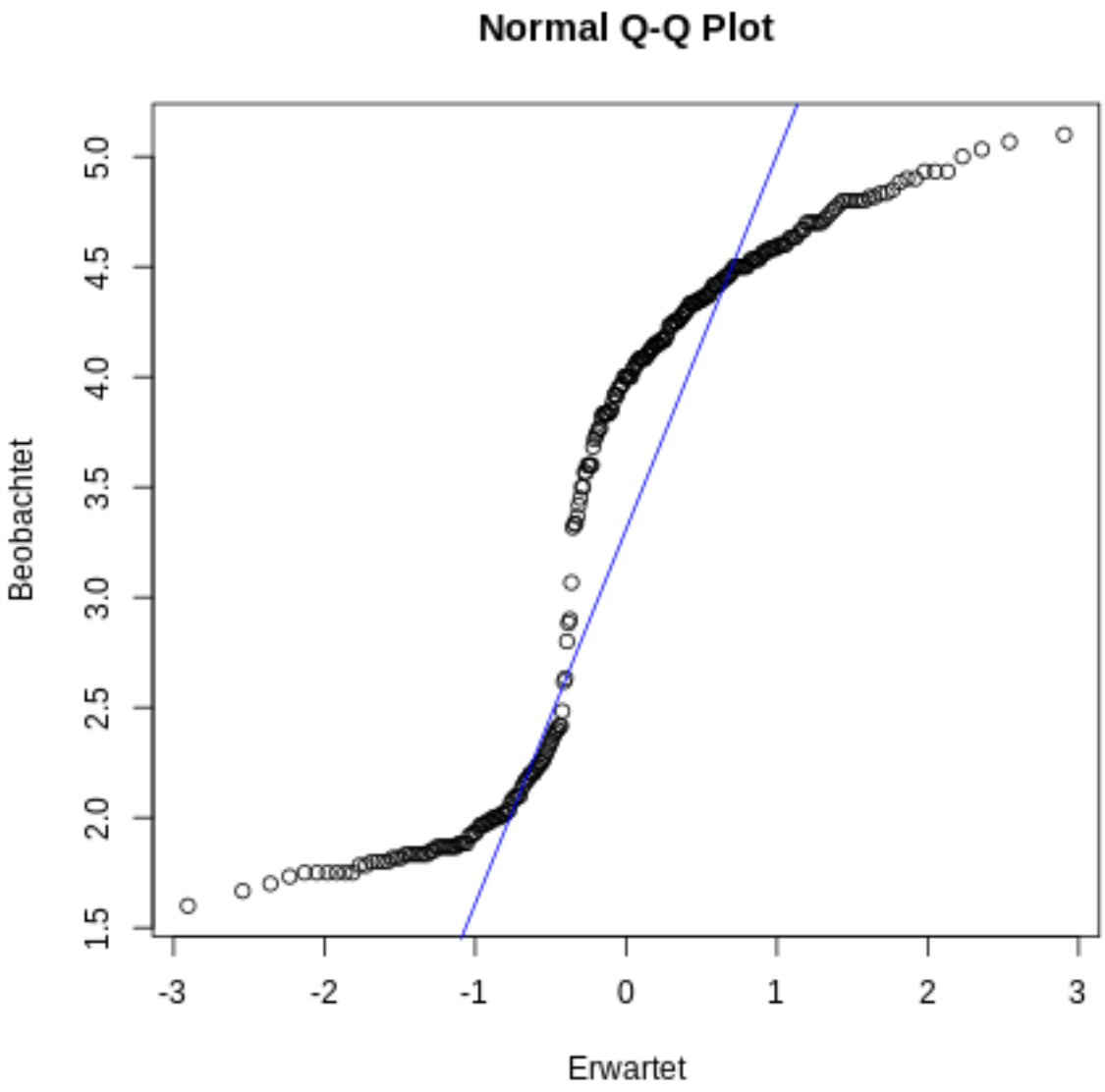


Abb. 13.7: QQ-Plot Eruptionsdauer Faithful (nach Normalverteilung)

Wir sind also recht weit von einer Normalverteilung entfernt.

Für andere Verteilungen müssen wir den ersten Weg wählen. Dazu betrachten wir den mitgelieferten Datensatz *rivers* mit den Längen wichtiger amerikanischer Flüsse.

Wir sortierten diese Werte zunächst und stellen ein Histogramm dar.

```
rivs <- sort(rivers)
hist(rivs)
```

Das Histogramm (s. u.) deutet auf eine Exponentialverteilung hin (viele kleine Werte, mit wachsender Länge nimmt die Zahl der Werte exponentiell ab). Wir bestimmen den Mittelwert und vergleichen mit einer Exponentialfunktion mit gleichem Mittelwert: der Mittelwert einer Exponentialfunktion ist gerade $1/\lambda$, dem Parameter der Exponentialverteilung.

```
rivs.m <- mean(rivs)
x <- seq(1, max(rivs), 10)
# Dichteplot
hist(rivs, freq=F)
# Dichtekurve der Exponentialverteilung
curve(dexp(x, 1/rivs.m), add=T)
```

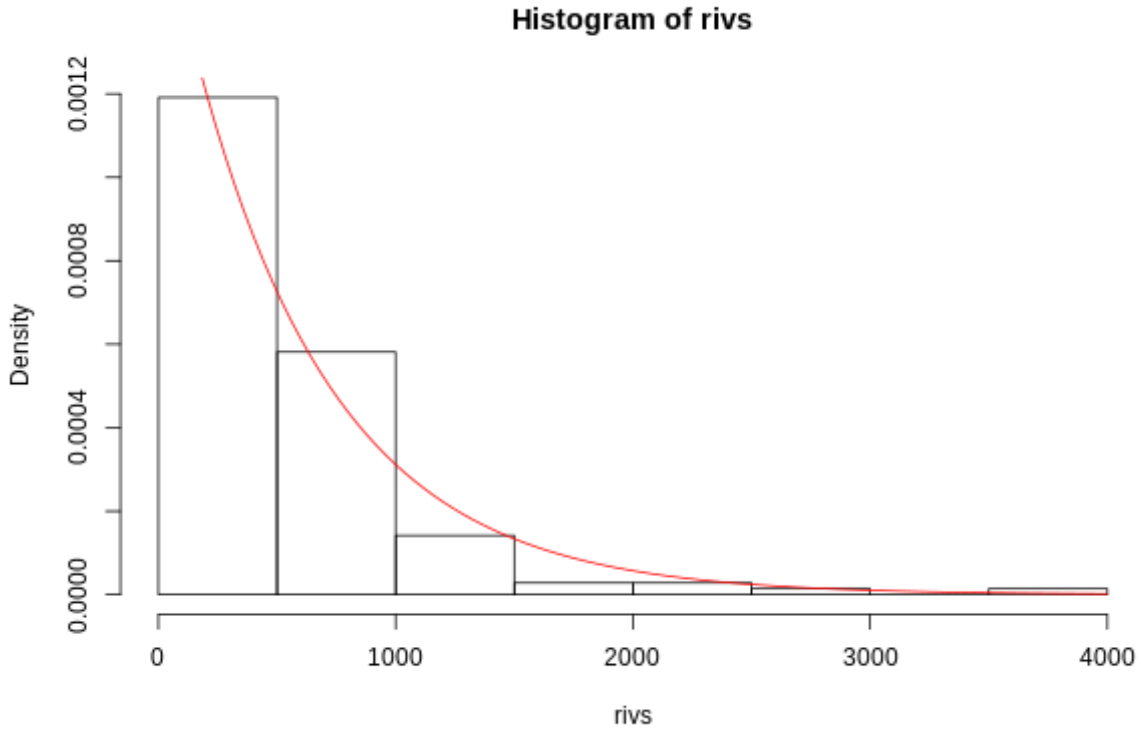


Abb. 13.8: Histogramm Flusslängen und Dichte Exponentialverteilung

Optisch ist eine gute Übereinstimmung zu vermuten. Wir untersuchen dies genauer mit dem *QQ-Plot*. Wir berechnen die Quantile unserer Exponententialfunktion an genausoviel Stellen, wir wir Flusslängen haben, und vergleichen im Scatterplot.

```
# Wahrscheinlichkeiten, für die wir Quantile berechnen
pq <- (1:length(rivs)-1)/length(rivs)
# Erwartete Quantile
qexpect <- qexp(pq, 1/rivs.m)
# QQ-Plot und Modellinie
qqplot(qexpect, rivs)
abline(0, 1)
```

Hier sehen wir schon optische deutliche Abweichungen von der Exponentialverteilung. Größen dieser Art wie Flusslängen, Berghöhen, Unternehmensgrößen sind üblicherweise auch nicht exponential-, sondern *Pareto*-verteilt.

13.3 χ^2-Test

Im Gegensatz zu Nominalskalen haben wir bei hier für jeden Wert nur eine Beobachtung, der χ^2-Test will aber beobachtete mit erwarteten Häufigkeiten vergleichen. Wir müssen deshalb unseren Wertebereich in

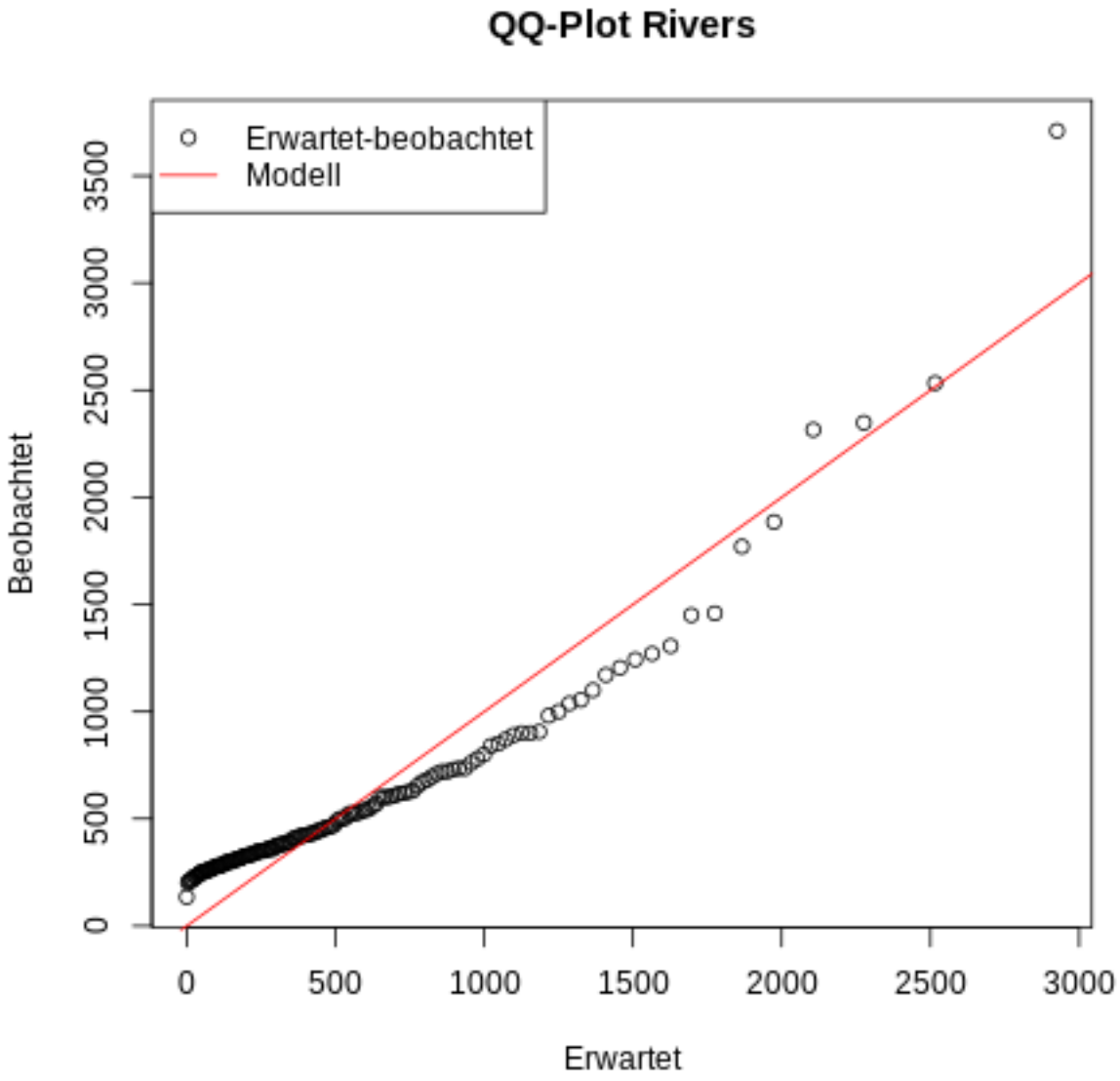


Abb. 13.9: QQPlot Flusslängen gegen Exponentialverteilung

Intervalle (*bins* = Behälter) zerlegen und anschließend die Werte pro Intervall mit den erwarteten Werten vergleichen.

Die Zerlegung (*Binning*) geschieht mit der Funktion `cut()`. Diese erwartet den Datenvektor und einen geordneten Vektor der Trennstellen. Jedem Wert wird durch das entsprechende Intervall zwischen zwei Trennstellen ersetzt, außerhalb liegende Werte werden zu `NA`. Im einfachsten Falle zerlegen wir also den kompletten Wertebereich in gleichgroße Abschnitte.

```
binGroesse <- max(vektor) - min(vektor)/binZahl
breaks <- seq(min(vektor), max(vektor), binGroesse)
bins <- cut(vektor, breaks)
```

Wir kontrollieren die Anzahl der Werte pro Intervall.

```
table(bins)
```

In jedem Intervall sollten mindestens 5 Werte liegen, um eine sinnvolle Testaussage zu erhalten. Dazu muss die Intervalleinteilung evt. auch ungleichmäßig vorgenommen werden.

Zur Berechnung der Wahrscheinlichkeiten für jedes Intervall berechnen wir die Wahrscheinlichkeitsfunktion mit den zu testenden Parametern an den Intervallgrenzen.

```
# Normalverteilung
p.breaks <- pnorm(bins, mean=mittel, sd=stdAbweichung)
# Exponentialverteilung
p.breaks <- pexp(bins, rate=lambdaWert)
```

Die Parameter schätzen wir aus den tatsächlichen Daten, wenn wir keine andere Hypothese haben, die wir prüfen wollen. Für die Normalverteilung verwenden wir die Funktionen `mean()` und `sd()`, der Parameter der Exponentialverteilung ist der reziproke Erwartungswert. Aus der Differenz der Wahrscheinlichkeitsfunktion am rechten und linken Intervallrand erhalten wir die Wahrscheinlichkeit des Intervalls. Diese können wir für alle Intervalle gemeisam berechnen, indem wir im Wahrscheinlichkeitsvektor einmal den ersten und einmal den letzten Wert weglassen und beide Vektoren subtrahieren.

```
p.bins < p.breaks[-1] - p.breaks[-length(p.breaks)]
```

Nun können wir den χ^2-Test durchführen:

```
chisq.test(table(bins), p.bins)
```

Beispiel: Krankentage. Wir testen auf Normalverteilung.

```
rm(list=ls())
data <- read.csv2('krankenstunden.csv', comment='#', header=T)
kr <- data$Krankenstand.in.Arbeitsstunden
# Erzeuge 5 bins
diff <- (max(kr) - min(kr))/5
breaks <- seq(min(kr), max(kr), diff)
kr.bin <- cut(kr, breaks)
# Berechne erwartete Wahrscheinlichkeiten normalverteilt
kr.mean <- mean(kr)
kr.sd <- sd(kr)
p.breaks <- pnorm(breaks, mean=kr.mean, sd=kr.sd)
p.bin <- p.breaks[-1] - p.breaks[-length(p.breaks)]
# chi^2-Test
print(chisq.test(table(kr.bin), p.bin))
```

Wir erhalten:

```
  Pearson's Chi-squared test

data:  table(kr.bin) and p.bin
X-squared = 10, df = 8, p-value = 0.265
```

Die Warnung besagt, dass in bestimmten Klassen zu wenige Werte vorliegen. Der p-Wert ist deutlich größer als 5%, wir können also die Hypothese einer Normalverteilung nicht ablehnen.

Beispiel: Flusslängen. Wir testen auf Exponentialverteilung. Da wir nur wenige sehr lange Flüsse haben, unterteilen wir nur die Längen bis 1500 (Meilen?) gleichmäßig und fassen alle längeren Flüsse zu einem Intervall zusammen.

```
rivs <- sort(rivers)
rivs.m <- mean(rivs)
breaks <- c(seq(0, 1500, 500), max(rivs))
rivs.bin <- cut(rivs, breaks)
cat('Bins:\n')
print(table(rivs.bin))
p.breaks <- pexp(breaks, 1/rivs.m)
p.bin <- p.breaks[-1] - p.breaks[-length(p.breaks)]
cat('chi²-Test:\n')
print(chisq.test(table(rivs.bin), p.bin))
```

Wir erhalten:

```
  Pearson's Chi-squared test

data:  table(rivs.bin) and p.bin
X-squared = 12, df = 9, p-value = 0.2133
```

Auch hier erhalten wir einen Wert, der deutlich über 5% liegt, aber auch wieder eine Warnung, dass der Test inkorrekt sein kann.

13.4 Shapiro-Wilk-Test

Wollen wir auf eine Normalverteilung testen, verwenden wir besser den *Shapiro-Wilk-Test*, der zum einen einfacher zu nutzen und zum anderen genauer ist.

shapiro.test(vektor)

Beispiel: Krankentage.

```
shapiro.test(kr)
```

Wir erhalten:

```
  Shapiro-Wilk normality test

data:  kr
W = 0.89124, p-value = 0.007176
```

Dieser Test liefert einen p-Wert von deutlich weniger als 5%, damit wäre die Hypthese einer Normalverteilung abzulehnen, während der χ^2-Test diese akzeptiert hat.

13.5 Aufgaben

Aufgabe 13.1 *Hausaufgabe (10 Punkte)*

Lesen Sie Ihren Gesamtdatenframe ein und erstellen Sie einen Vektor mit *normalisierten* Gesamtpreisen (ohne Versandkosten):

- Der Mittelwert der Preise je Produktgruppe ist 0.
- Die Varianz der Preise je Produktgruppe ist 1.

Ziehen Sie dazu bei jedem Angebot den entsprechenden Mittelwert der Produktgruppe ab und teilen anschließend durch die Standardabweichung der Produktgruppe.

Stellen Sie die beobachtete Dichteverteilung der normalisierten Preise und zum Vergleich die Dichtefunktion der Normalverteilung in einem gemeinsamen Diagramm graphisch dar.

Erstellen Sie einen QQ-Plot der normalisierten Preise in Bezug auf die Normalverteilung. Vergessen Sie die Vergleichsgerade nicht.

Schätzen Sie ein, ob Sie eine Normalverteilung erwarten.

Untersuchen Sie nun mit Hilfe des χ^2-Tests *und* des Shapiro-Tests, ob die Annahme einer Normalverteilung für die normalisierten Preise akzeptiert werden kann oder abgelehnt werden muss.

Untersuchen Sie graphisch, ob die Anzahl der Bewertungen für alle Angebote exponentialverteilt ist (die Rate ist dann der Reziprokwert des Erwartungswertes). Stellen Sie sowohl Histogramm als auch QQ-Plot dar.

- Falls ja, dann untersuchen Sie diese Annahme mithilfe des χ^2-Tests.
- Falls nein, dann untersuchen Sie den *Logarithmus* der Bewertungszahlen, ob dieser normalverteilt (das ist dann eine *Log-Normalverteilung*) oder exponentialverteilt (*Pareto-Verteilung*) ist. Stellen Sie dazu den QQ-Plot dar und führen Sie einen geeigneten Test durch.

Geben Sie neben Ihrem Jupiter-Notebook auch alle verwendeten Datendateien gemeinsam als ZIP-Datei ab.

KAPITEL 14

Metrische Größen, lineare Modelle

14.1 Übersicht

Werden für ein Objekt gleichzeitig verschiedene Daten beobachtet (z. B. Größe und Körpergewicht bei Personen, erreichte Punktzahlen bei verschiedenen Aufgaben einer Klausur, Außentemperatur und Anwesenheitsquote bei Vorlesung), stellt sich die Frage, ob diese Größen miteinander in Zusammenhang stehen. Man spricht von *Korrelation*.

Warnung: Eine starke Korrelation bedeutet noch lange nicht, dass die Größen in einem kausalen Zusammenhang stehen. Das bekannteste Beispiel ist der scheinbare Zusammenhang zwischen der Zahl der Störche und der Geburten in einem Ort. Ursache ist häufig eine oder mehrere *versteckte Variable*, die beide Größen beeinflusst.

14.2 Kovarianz und Korrelation

Die Kovarianz ist eine Erweiterung der Varianz auf zwei metrische Größen:

$$cov(X,Y) = \frac{\sum_{i=1}^{N}(x_i - \bar{X}) \cdot (y_i - \bar{Y})}{N-1}$$

Stehen die beiden Größen in einem linearen Zusammenhang, ist die Kovarianz größer, bei einem umgekehrten linearen Zusammenhang ist sie kleiner als 0. Besteht kein oder kein linearaer Zusammenhang, ist die Kovarianz nahe 0. Die Berechnung in *R* erfolgt mit der Funktion `var()` oder `cov()` mit zwei Parametern:

`var(`xVektor`,` yVektor`)`

Der absolute Zahlenwert der Kovarianz kann meist nur dann sinnvoll interpretiert werden, wenn beide Größen gleich skaliert sind, z. B. die x- und y-Koordinaten von Punkten darstellen. Sie gibt dann an, inwieweit die horizontale von der vertikalen Verteilung beeinflusst ist. Ist dies nicht der Fall, dann bietet sich eine *Normierung* an, wir erhalten den *Korrelationskoeffizienten*:

$$r_{X,Y} = \frac{cov(X,Y)}{\sqrt{var(X) \cdot var(Y)}}$$

Hier bedeutet 1 einen völlig positiv linearen Zusammenhang, -1 einen negativen linearen Zusammenhang (alle Werte liegen auf einer Geraden im X-Y-Diagramm). Werte nahe 0 bedeuten, dass es keinen linearen Zusammenhang gibt. Die Berechnung erfolgt mit der Funktion `cor()`:

`cor(`xVektor`,` yVektor`)`

Für Daten ordinaler Skalen existieren alternative Berechnungsmethoden des Korrelationskoeffizienten, die in *R* ebenfalls implementiert sind.

14.3 Lineare Modelle

Wollen wir einen linearen Zusammenhang genauer untersuchen, können wir versuchen, die Geradengleichung zu bestimmen. Dazu bilden erzeugen wir zunächst eine Formel und daraus ein Model:

formula <- abhängigeGröße `~` unabhängigeGröße *[`+ 0`]*
model <- `lm(`formula`)`

Für Datenframes ist es sinnvoller, das Modell wie folgt zu formulieren, da dann leichter Vorhersagen für neue Daten möglich sind:

model <- `lm(`Zielspalte `~` Quellspalte`, data=`datenframe`)`

Beispiel für `faithful`-Datensatz:

```
model.faithful <- lm(eruptions ~ waiting, data=faithful)
```

Der Ausdruck `+ 0` in der Formel bewirkt, dass die berechnete Gerade durch den Ursprung *(0,0)* geht. Ohne diese Ergänzung wird eine Konstante (*Intercept*) aufaddiert, um den Fehler möglichst klein zu halten.

Einen Überblick über das berechnete Modell gibt wiederum die Funktion `summary()`. Wir schauen uns dazu die Oesterreich-Wahl mit Intercept an, `Daten`.

```
rm(list=ls())
#setwd("~/afs/lehre/R")
oesi <- read.table("samples/testdata/oesiwahl.dat", header=T)
names(oesi) <- c("Name", "Talk", "Result")
oesi.form1 <- oesi$Result ~ oesi$Talk
oesi.model1 <- lm(oesi.form1)
cat("Modell mit Intercept:\n")
summary(oesi.model1)
```

Ergebnis:

```
Modell mit Intercept:

Call:
lm(formula = oesi.form1)

Residuals:
       1         2         3         4         5         6
-1.478805  1.998565  6.372385  0.179398  0.006865 -7.078408

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 9.017e+00  2.586e+00   3.487   0.0252 *
oesi$Talk   2.065e-04  4.393e-05   4.701   0.0093 **
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 4.923 on 4 degrees of freedom
Multiple R-squared:  0.8467,        Adjusted R-squared:  0.8084
F-statistic:  22.1 on 1 and 4 DF,  p-value: 0.009301
```

Die Übersicht enthält u. a. folgende Daten:

Residuals: Abweichungungen der Beobachtungen von den berechneten Modelldaten.

Coefficients: Berechnete Parameter, Intercept=Wert bei x=0, Anstieg. Der Wert `t` gibt den berechneten Wert des t-Tests für die Hypothese «keine lineare Abhängigkeit» an, `Pr(>|t|)` die Wahrscheinlichkeit für Fehler 1. Art (H_0 gilt, wird aber abgelehnt = kein linearer Zusammenhang vorhanden).

Ergebnisse ohne Intercept.

```
oesi.form0 <- oesi$Result ~ oesi$Talk + 0
oesi.model0 <- lm(oesi.form0)
summary(oesi.model0)
```

Ergebnis:

```
Bundespräsidentenwahl Oesterreich 2016, 1. Wahlgang


Call:
lm(formula = oesi.form0)

Residuals:
     1      2      3      4      5      6
-5.327  6.215 13.751  8.215  8.055  1.770

Coefficients:
           Estimate Std. Error t value Pr(>|t|)
oesi$Talk 3.029e-04  6.138e-05   4.935  0.00434 **
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 8.85 on 5 degrees of freedom
Multiple R-squared:  0.8297,        Adjusted R-squared:  0.7956
F-statistic: 24.36 on 1 and 5 DF,  p-value: 0.00434
```

Der Intercept-Wert wird nicht berechnet, die Differenzen der Beobachtungen zum Modell sind größer, der α-Wert ist dagegen noch kleiner.

Das lineare Modell kann einfach in einem Scatterplot mit der Funktion `abline()` ergänzt werden. Den oder die Koeffizienten der Linie erhalten wir mittels

```
coef(modell)
```

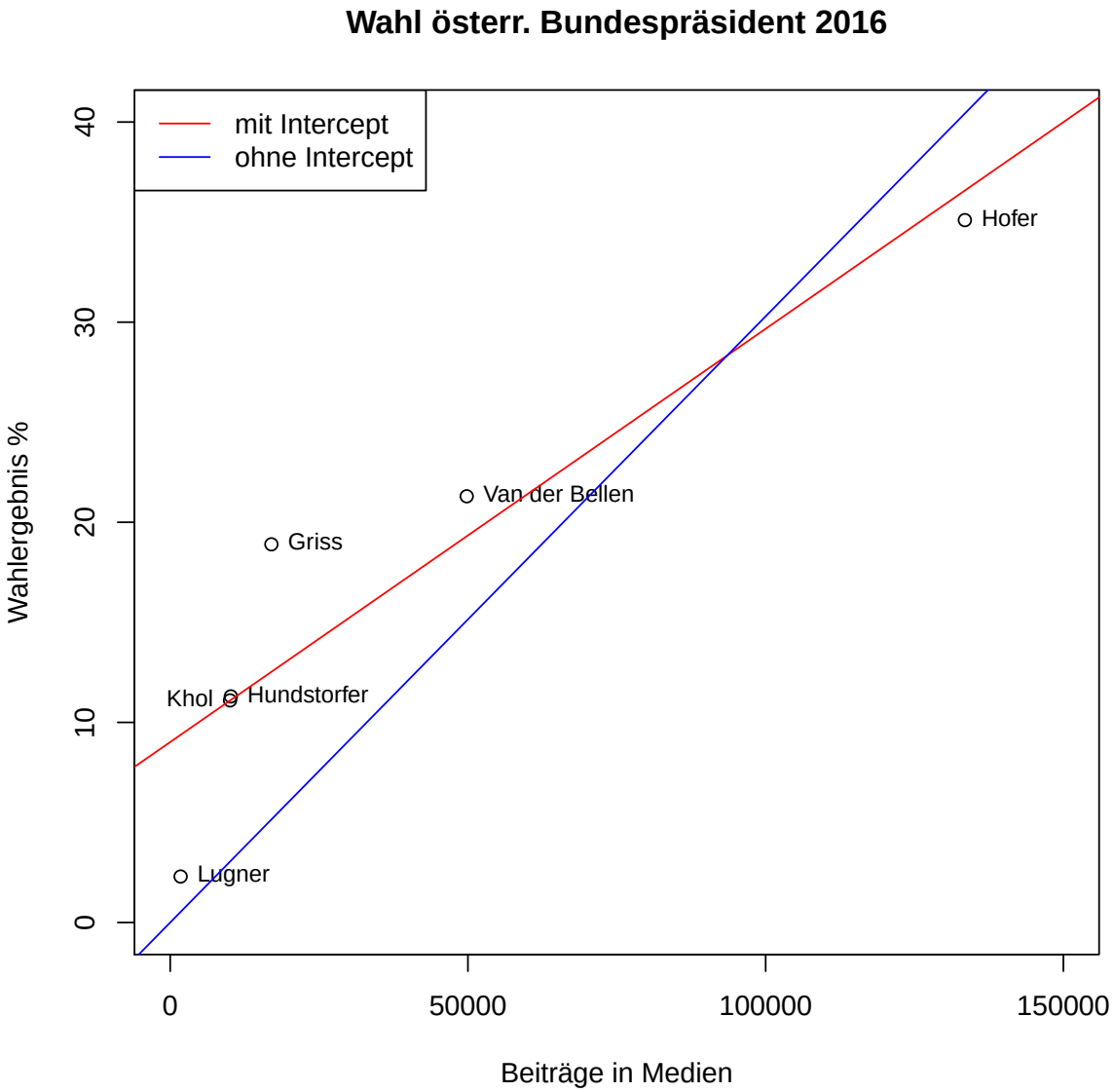
Mit Intercept verwenden wir den Aufruf

```
abline(coef=coef(modell))
```

ohne Intercept setzen wir den Parameter `a` auf 0 und den Parameter `b` auf den Modellkoeffizienten:

```
abline(a=0, b=coeff(modell))
```

Visueller Vergleich der Modelle mit und ohne Intercept:



In beiden Modellen ist der p-Wert für den Test der Nullhypothese (kein linearer Zusammenhang) für den Anstieg kleiner als 1%, die Nullhypothese wird abgelehnt, es besteht also mit hoher Wahrscheinlichkeit ein linearer Zusammenhang.

14.4 Angepasste Daten

Wurde ein Modell erzeugt, können daraus die nach dem Modell zu erwartenden Werte an den beobachteten Stellen berechnet werden:

fittedData <- **fitted**(modell)

Wir erhalten einen Vektor der abhängigen Werte. Wollen wir diesen in einen Scatterplot eintragen, benötigen wir zusätzlich den Vektor der unabhängigen x-Werte.

points(xWerte, **fitted**(modell) ...)

Diese Punkte liegen alle auf der Linie unseres linearen Modells. Mit diesen berechneten Punkten lassen sich nun auch einfach die Residuen (Differenz zwischen Beobachtungs- und Modelldaten) ergänzen. Wir benötigen dazu das Programmkonstrukt für eine Schleife, um alle Datenpunkte durchzugehen:

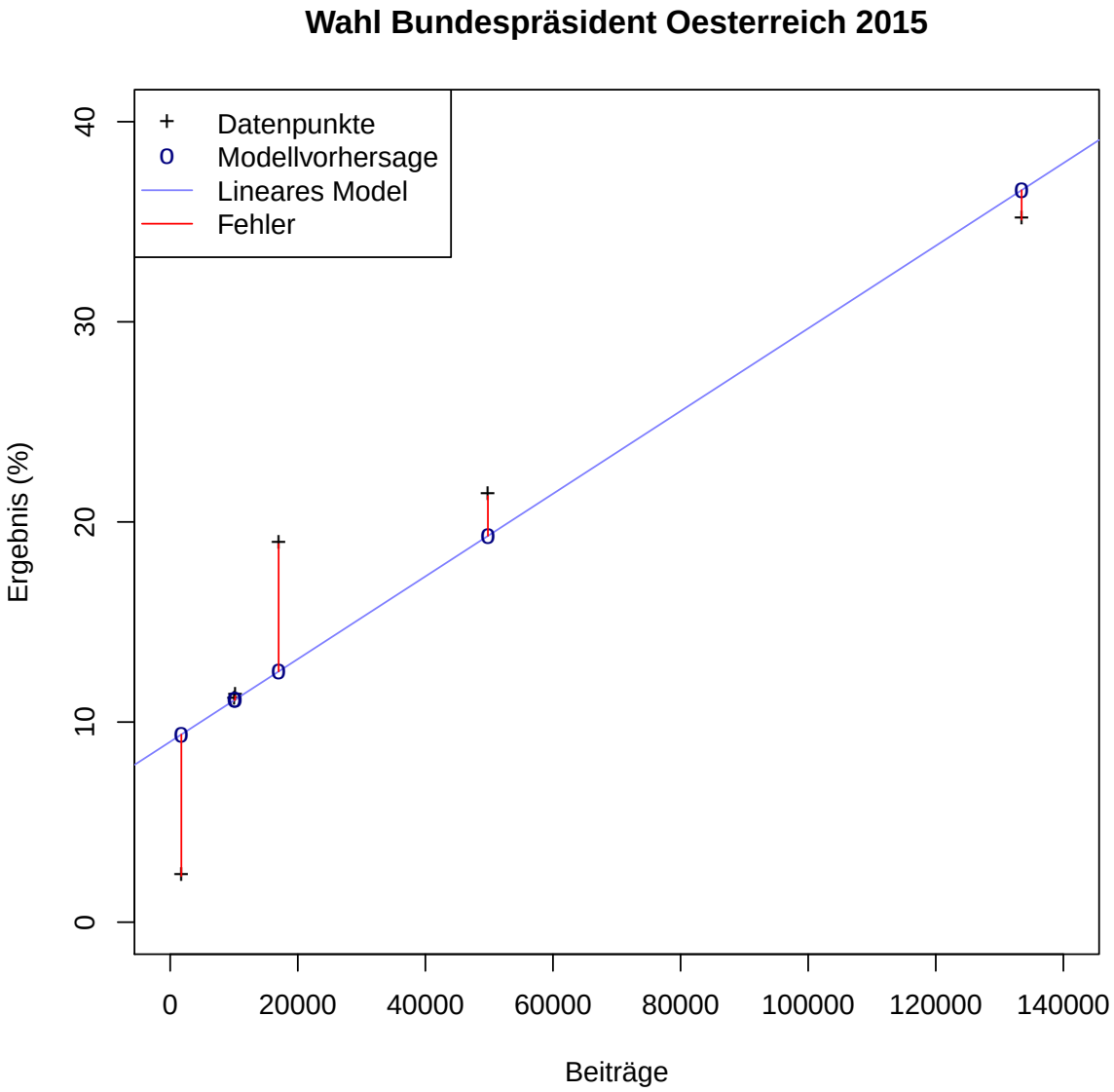
```
for (element in vektor) {
     # Verarbeite element
}
```

Da wir hier korrespondierende Werte aus zwei Vektoren benötigen, erzeugen wir uns eine Indexschleife:

```
for (index in 1:length(vektor) {
  # verarbeite index, vektor[index]
}
```

Beispiel: Lineares Modell (mit Intercept) und Residuen für die Oesterreich-Wahl.

```
setwd("~/lehre/R")
oesi <- read.table("samples/testdata/oesiwahl.dat", header=T)
names(oesi) <- c("Name", "Talk", "Result")
# Formel und Modell erzeugen
oesi.form <- oesi$Result ~ oesi$Talk
oesi.mod  <- lm(oesi.form)
# Vorhersagepunkte berechnen bzgl. Talk
oesi.fitted <- fitted(oesi.mod)
# Basisdiagramm: Scatterplot
plot(oesi.form, xlim=c(0, 140000), ylim=c(0,40), pch="+",
     main="Wahl Bundespräsident Oesterreich 2015",
     xlab="Beiträge", ylab="Ergebnis (%)")
# Gerade für lineares Modell
abline(coef=coef(oesi.mod), col="#7f7fff")
# Vorhersagepunkte eintragen
points(oesi$Talk, oesi.fitted, pch="o", col="#00007f")
# Fehler: Linien zwischen Vorhersage- und beobachteten Wert ziehen
for (i in 1:length(oesi$Talk)) {
  lines(rep(oesi$Talk[i], 2), c(oesi$Result[i], oesi.fitted[i]), col="red")
}
# Legende für 2 Punkt- und 2 Linienarten
legend("topleft", legend=c("Datenpunkte", "Modellvorhersage", "Lineares Model",
↪"Fehler"),
       pch=c("+", "o", NA, NA),
       lty=c(0, 0, 1, 1), col=c("black", "#00007f", "#7f7fff", "red"))
```


Wahl Bundespräsident Oesterreich 2015
Datenpunkte
Modellvorhersage
Lineares Model
Fehler
Ergebnis (%)
0
10
20
30
40
0
20000
40000
60000
80000
100000
120000
140000
Beiträge

14.5 Vorhersagen

Wollen wir die abhängige Größe für neue Daten vorhersagen, müssen wir bei der Modellerstellung den benutzten Datenframe angeben. Die neuen Daten packen wir ebenfalls ein einen Datenframe mit den gleichen Spaltennamen für die unabhängigen Variablen im Model.

Beispiel Faithful:

```
mod.faithful <- lm(eruptions ~ waiting, data=faithul)
newdata <- data.frame(waiting=c(50, 60, 70))
```

Die Vorhersage erfolgt nun mit der Funktion `predict()`.

datenVektor <- **`predict(`**modell, neuerDatenframe**`)`**

Beispiel:

```
res <- predict(mod.faithful, newdata)
```

14.6 Modell mit mehreren unabhängigen Größen, Simpson-Paradoxon

Ermittelt man statistische Größen über eine Gesamtheit, die sich aus Klassen mit unterschiedlichen Eigenschaften zusammensetzt, können diese stark von den statistischen Größen der Einzelklassen abweichen, ja sogar den entgegengesetzten Trend aufweisen. Dieser Effekt heißt *Simpson-Paradoxon*.

Wir betrachten dazu ein Beispiel aus [hesse14]. Es wird eine neue Diät beworben, die mit fortgesetzter Dauer eine hohe Gewichtsabnahme garantiert: `fiktive Daten`.

Wir stellen das Modell auf und sehen einen deutlichen negativen Trend.

```
mod1 <- diaet$Zunahme ~ diaet$Dauer
summary(mod1)
```

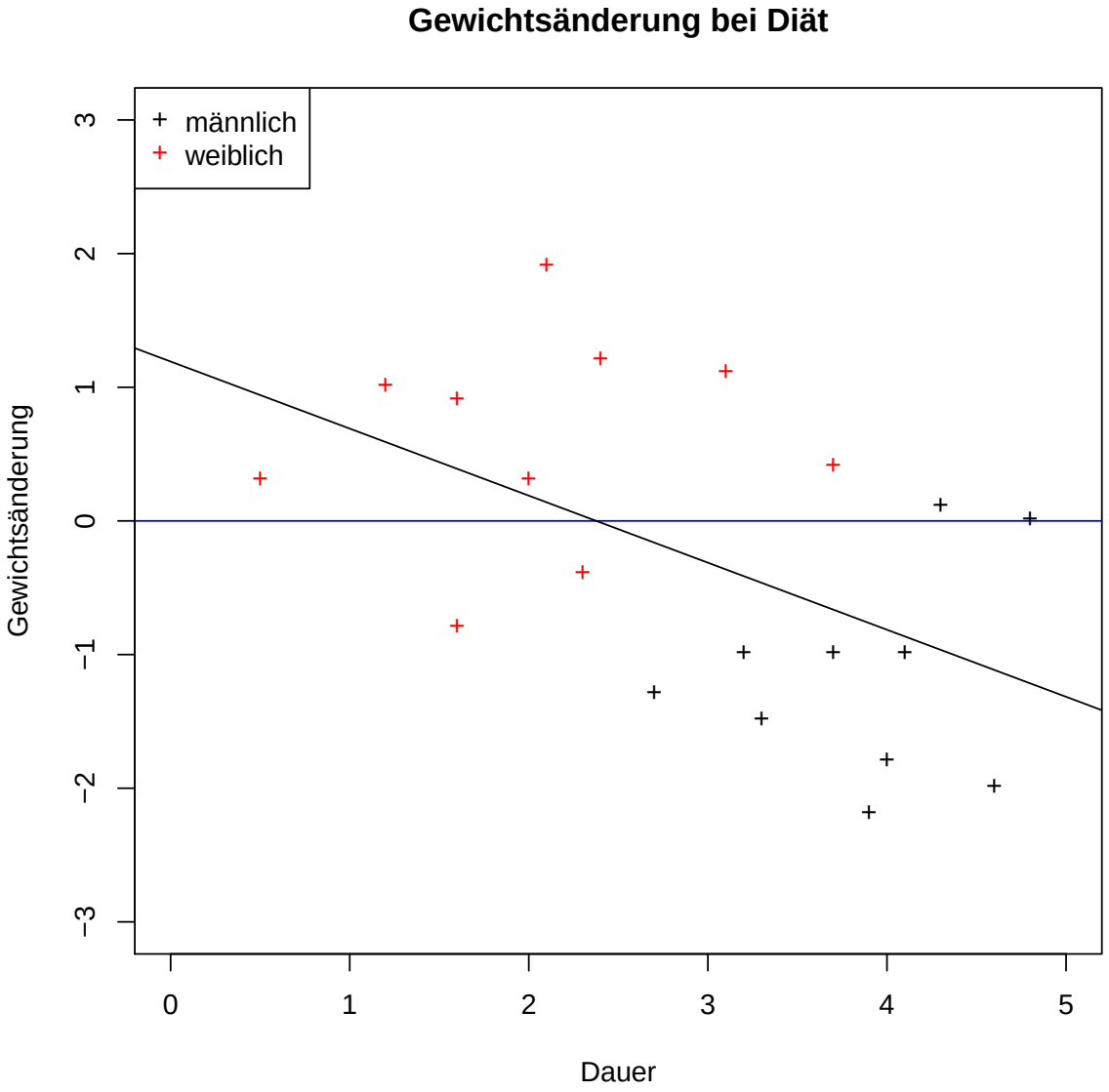
Ergebnis:

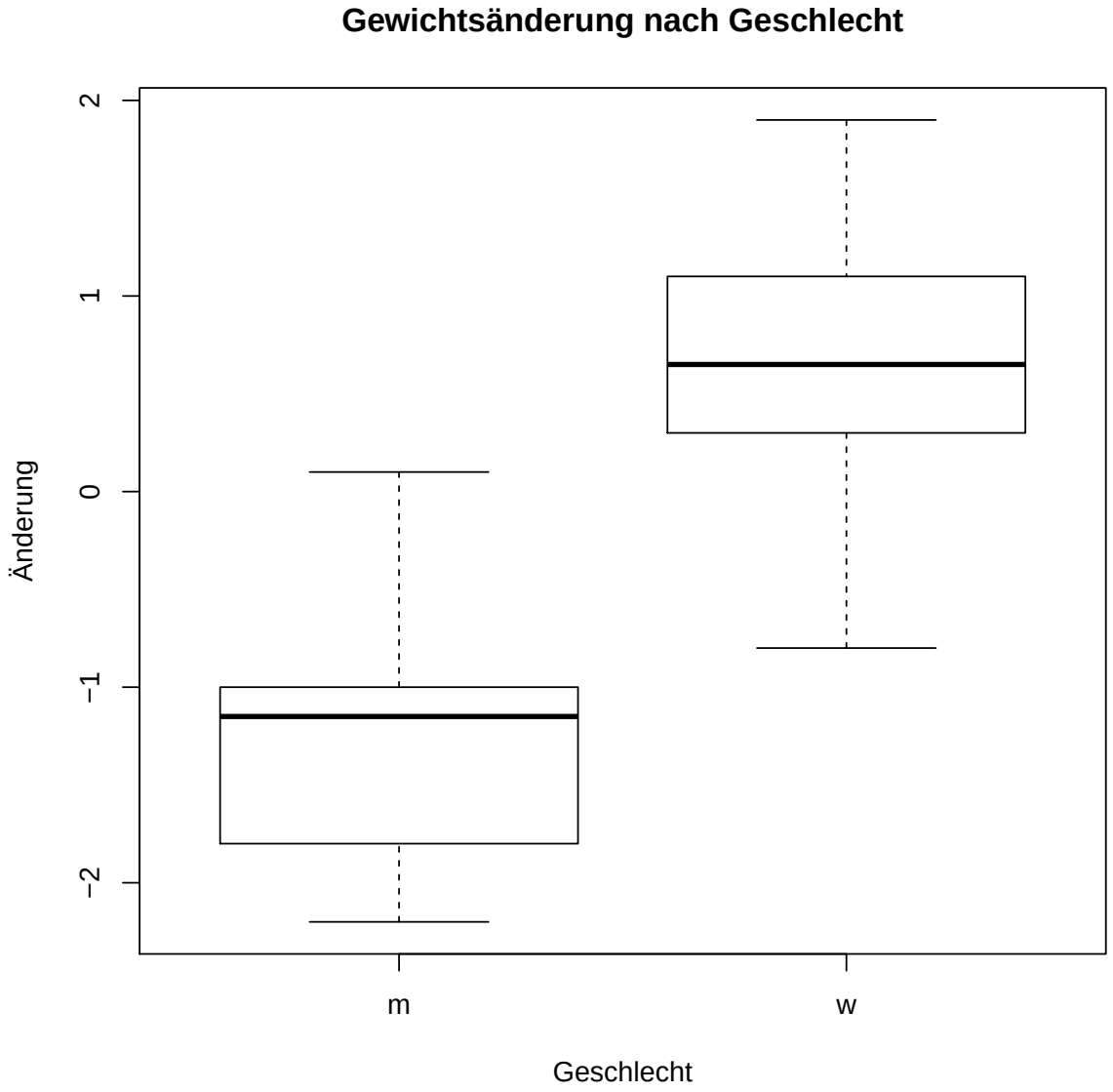
```
Call:
lm(formula = form1)

Residuals:
    Min      1Q  Median      3Q     Max
-1.4359 -0.9100 -0.2359  1.0640  1.7611

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   1.1924     0.6297   1.894   0.0745 .
diaet$Dauer  -0.5017     0.1980  -2.534   0.0208 *
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 1.043 on 18 degrees of freedom
Multiple R-squared:  0.263,        Adjusted R-squared:  0.222
F-statistic: 6.422 on 1 and 18 DF,  p-value: 0.02078
```

Auch die Graphik zeigt diesen Trend deutlich.

Nun betrachten wir zunächst die Verteilung der Gewichtsänderung getrennt nach Geschlecht mit einem Boxplot und sehen deutliche Unterschiede in der Verteilung.

Wir ermitteln deshalb getrennte lineare Modelle für beide Geschlechter:


Gewichtsänderung bei Diät
männlich
weiblich
Gewichtsänderung
Dauer
3
2
1
0
−1
−2
−3
0
1
2
3
4
5


Gewichtsänderung nach Geschlecht
Änderung
2
1
0
−1
−2
m
w
Geschlecht

```
diaet.w <- diaet[diaet$Geschlecht == "w", ]
diaet.m <- diaet[diaet$Geschlecht == "m", ]
```

Berechnen wir hier die Modelle, erhalten wir für beide einen positiven Trend (also Gewichtszunahme in jeder der Klassen):

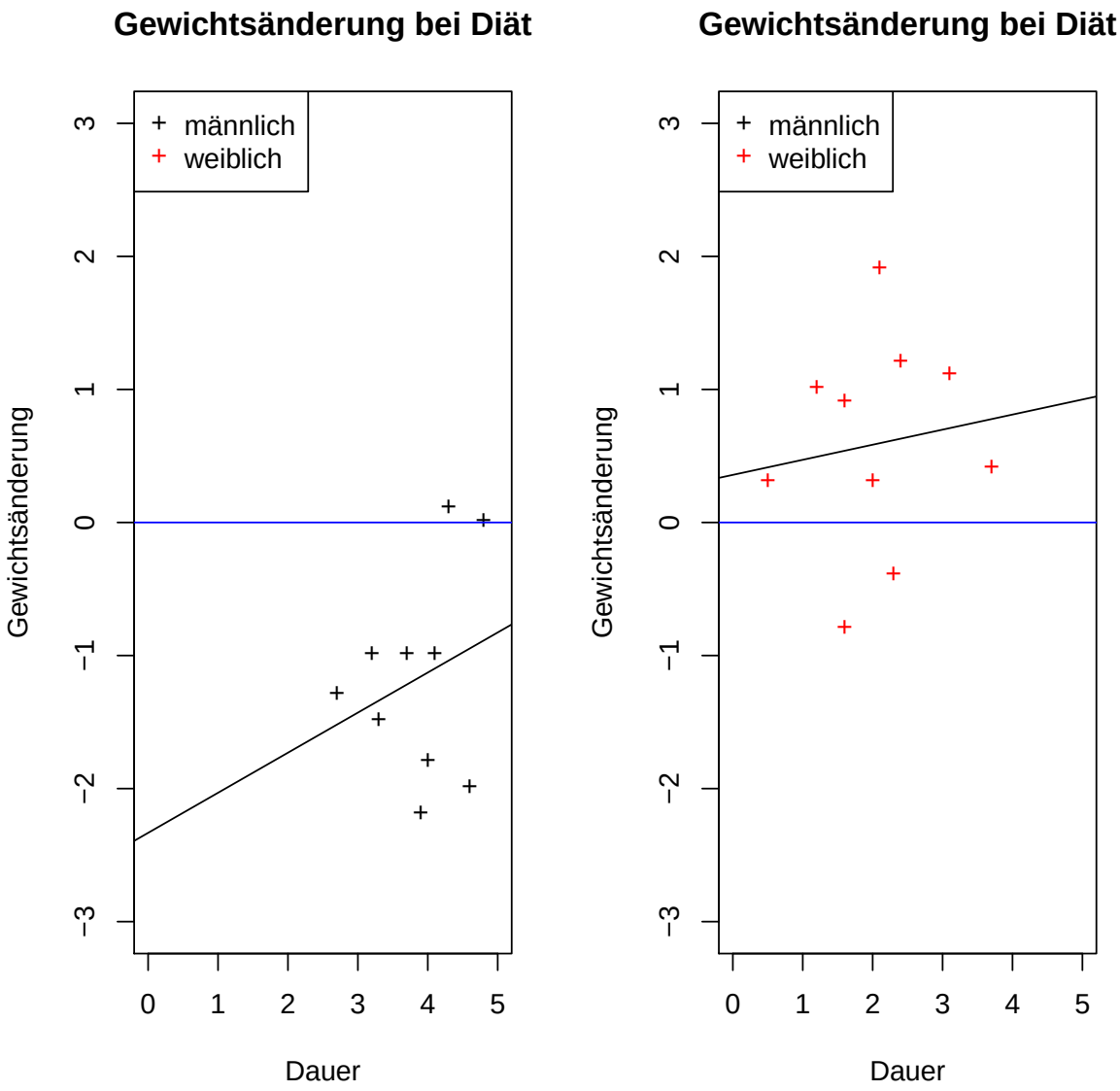
```
Modell männlich:
================


Call:
lm(formula = form.m)

Residuals:
    Min      1Q  Median      3Q     Max
-1.0529 -0.5445  0.1580  0.3315  1.1375

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)    -2.3328     1.5795  -1.477    0.178
diaet.m$Dauer   0.3013     0.4040   0.746    0.477

Residual standard error: 0.7901 on 8 degrees of freedom
Multiple R-squared:  0.06498,        Adjusted R-squared:  -0.0519
F-statistic: 0.5559 on 1 and 8 DF,  p-value: 0.4772

Model weiblich:
===============


Call:
lm(formula = form.w)

Residuals:
    Min      1Q  Median      3Q     Max
-1.3390 -0.3538  0.1233  0.4775  1.3043

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)     0.3577     0.6801   0.526    0.613
diaet.w$Dauer   0.1133     0.3055   0.371    0.720

Residual standard error: 0.8391 on 8 degrees of freedom
Multiple R-squared:  0.01691,        Adjusted R-squared:  -0.106
F-statistic: 0.1376 on 1 and 8 DF,  p-value: 0.7203
```

Graphische Darstellung:

Wir dürfen also die hier die Nominaldaten im Modell nicht vernachlässigen, sondern müssen diese mit einbeziehen. Dazu müssen diese in einen Faktor umgewandelt werden (was beim Einlesen mit `read.table()` automatisch geschehen ist). Ins Modell wird diese zweite Größe durch eine durch eine +-Verknüpfung zu den unabhängigen Daten hinzugefügt.

```
mod2 <- diaet$Zunahme ~ diaet$Dauer + diaet$Geschlecht
```

Die Anzeige dieses Modells zeigt deutlich, das der entscheidende und einzig signifikante Faktor für die abhängige Größe nicht die Dauer der Diät, sondern das Geschlecht ist. Auch in Abhängigkeit der Dauer sehen wir wieder einen positiven Trend (also Gewichtszunahme).


Gewichtsänderung bei Diät
männlich
weiblich
Gewichtsänderung
Dauer
Gewichtsänderung bei Diät
männlich
weiblich
Gewichtsänderung
Dauer

```
Call:
lm(formula = form2)

Residuals:
    Min      1Q  Median      3Q     Max
-1.3106 -0.5246  0.1012  0.4291  1.3012

Coefficients:
                  Estimate Std. Error t value Pr(>|t|)
(Intercept)        -1.8514     0.9429  -1.963   0.0662 .
diaet$Dauer         0.1765     0.2355   0.750   0.4637
diaet$Geschlechtw   2.0795     0.5547   3.749   0.0016 **
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.7939 on 17 degrees of freedom
Multiple R-squared:  0.5965,        Adjusted R-squared:  0.549
F-statistic: 12.57 on 2 and 17 DF,  p-value: 0.0004462
```

Unser Modell berechnet uns nun drei Parameter: `(Intercept)`, `diaet$Dauer` und `diaet$Geschlechtw`. Die Zunahme errechnet sich nun zu:

- zunahme = `(Intercept)` + `diaet$Dauer` * dauer + `diaet$Geschlecht`

  falls das Geschlecht weiblich,

- zunahme = `(Intercept)` + `diaet$Dauer` * dauer

  falls das Geschlecht männlich ist.

14.7 Codierung von Nominaldaten

Wie wir bereits gesehen haben, können Nominaldaten ebenfalls in einem linearen Modell berücksichtigt werden. Wir können diese allerdings nicht einfach als fortlaufende Zahlen codieren, da wir auf diese Weise zum einen eine Ordnung der Daten in unser Modell einfügen, zum anderen zwischen den Faktoren stets gleiche Abstände erzeugt würden, da unsere Formel ja wie folgt aussieht:

Statt dessen verwendet man eine sogenannte *Dummy-Codierung*: Für jedes Level außer dem ersten wird eine neue Variable `1` erzeugt, von denen nur eine den Wert 1 annehmen kann und alle anderen den Wert 0 haben. Für das erste Level haben alle Variablen den Wert 0. Diese Werte kann man mit der Funktion `contrasts()` anzeigen:

`contrasts(`factor-Vektor**`)`**

Beispiel: Spezies der Iris-Blüten aus dem Beispieldatensatz *iris*.

```
contrasts(iris$Species)
```

Ergebnis:

```
           versicolor virginica
setosa              0         0
versicolor          1         0
virginica           0         1
```

Die Spezies `setosa` würde also durch eine Null in beiden Variablen codiert, bei der Spezies `virginica` ist die erste Variable 0 und die zweite 1.

Unser Modell können wir nun so darstellen:

$$y = \mathrm{a} + \mathrm{b} \cdot x + \sum_{i=1}^{n} \mathrm{c}_i \cdot l_i$$

wobei l die Dummy-Variablen darstellen. Für jede Beobachtung geht nun höchstens einer der Parameter $\mathrm{c_i}$ in die Formel ein, da alle anderen Variablen den Wert 0 haben.

Mit dieser Information können wir nun ein Modell mit Nominaldaten interpretieren.

Beispiel: Länge Kelchblätter (Sepal) in Abhängigkeit der Länge der Blütenblätter (Petal) und der Iris-Spezies.

```
iris.formula <- Sepal.Length ~ Petal.Length + Species
iris.model <- lm(iris.formula, data=iris)
summary(iris.model)
```

Ergebnis:

```
Call:
lm(formula = iris.formula, data = iris)

Residuals:
     Min       1Q   Median       3Q      Max
-0.75310 -0.23142 -0.00081  0.23085  1.03100

Coefficients:
                  Estimate Std. Error t value Pr(>|t|)
(Intercept)        3.68353    0.10610  34.719  < 2e-16 ***
Petal.Length       0.90456    0.06479  13.962  < 2e-16 ***
Speciesversicolor -1.60097    0.19347  -8.275 7.37e-14 ***
Speciesvirginica  -2.11767    0.27346  -7.744 1.48e-12 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.338 on 146 degrees of freedom
Multiple R-squared:  0.8367,  Adjusted R-squared:  0.8334
F-statistic: 249.4 on 3 and 146 DF,  p-value: < 2.2e-16
```

Bei *Setosa* gehen in die Formel nur *Intercept* und *Petal.Length* als Parameter ein, bei den beiden anderen Spezies muss einer der beiden Werte *Speciesversicolor* bzw. *Speciesvirginica* subtrahiert werden, um die modellierte Kelchblattlänge zu erhalten.

Zur graphischen Darstellung trennen wir unsere Datenpunkte nach den Spezies auf und zeichnen jeweils die Modellgerade, deren y-Werte wir mit der Funktion `fitted()` erhalten.

yWerte = **`fitted(`**modell**`)`**

Aus diesen wählen wir jeweils y-Werte für die Spezies aus.

```
idx.1 <- which(iris$Species=='setosa')
idx.2 <- which(iris$Species=='versicolor')
idx.3 <- which(iris$Species=='virginica')

lines(iris$Petal.Length[idx.1], fitted(iris.model)[idx.1], col=1)
lines(iris$Petal.Length[idx.2], fitted(iris.model)[idx.2], col=2)
lines(iris$Petal.Length[idx.3], fitted(iris.model)[idx.3], col=3)
```

Zum Vergleich können wir auch das Modell ohne die Berücksichtigung der Spezies berechnen und ins Diagramm eintragen:

```
model0 <- lm(Sepal.Length ~ Petal.Length)
abline(coef=coef(model0), col=4)
```

Der graphische Vergleich zeigt deutlich, dass die Berücksichtigung der Spezies das Modell verbessert.

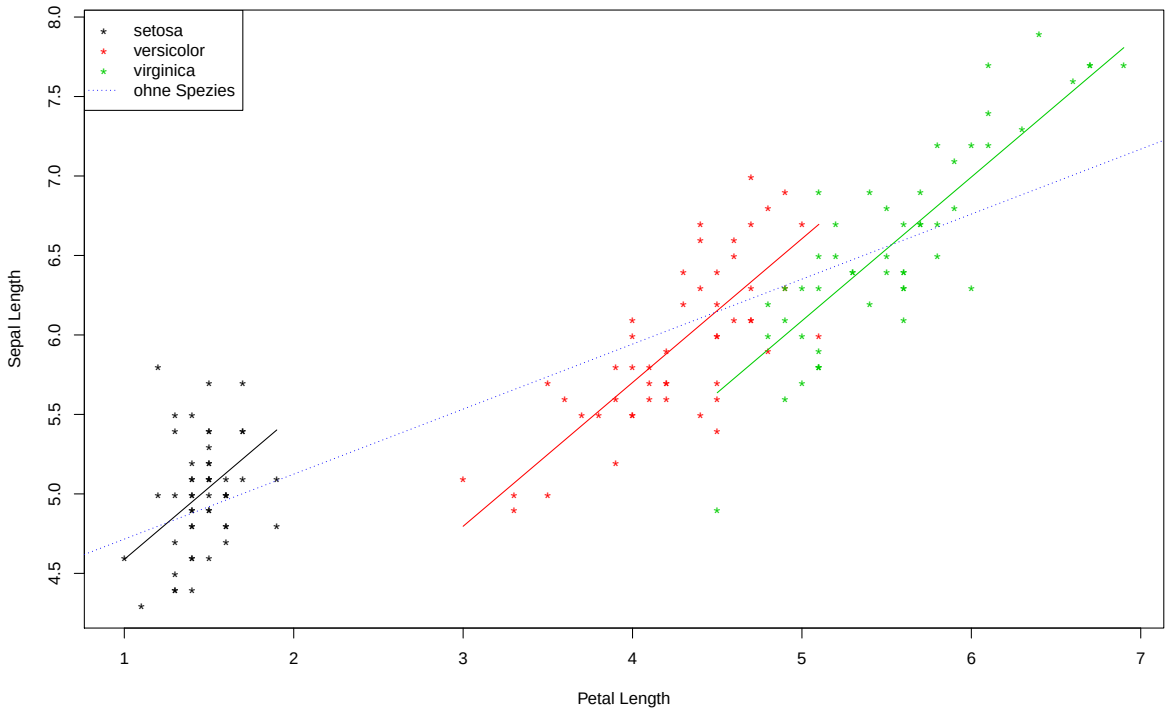


Abb. 14.1: Lineares Modell Iris

Hinweis: Sollen die Datenpunkte durch Linien verbunden werden, müssen die Daten *vor* der Modellbildung nach der x-Achse sortiert werden.

Hinweis: Je nach Art der Daten verwendet *R* statt der *Dummy-Codierung* auch andere Codierungen, wie dies z. B. beim Datensatz *diamonds* des Pakets *ggplot2* beobachtet werden kann.

Hinweis: Die Güte verschiedener linearer Modelle kann mit der *ANOVA*-Varianzanalyse bewertet werden. Darauf wird im Kurs nicht eingegangen.

14.8 Aufgaben

Aufgabe 14.1 Erstellen Sie ein lineares Modell für den Zusammenhang zwischen Umfang und Preis der Bücher Ihrer eigenen Erfassung bzw. der Beispieldaten. Stellen Sie das Modell graphisch dar, interpretieren Sie die Stärke der Abhängigkeit.

Aufgabe 14.2 *Hausaufgabe (10 Punkte)*

Lesen Sie die Data Frames Ihrer einzelnen Produkte sowie die Zusammenfassung aller Produkte ein. Erzeugen Sie wieder den normalisierten Preis (ohne Vesandkosten) für alle Angebote in Abhängigkeit vom Produkt in der Form, dass der Mittelwert der *Angebote eines Produkts* 0 und die Standardabweichung 1 ist.

- **Bestimmen Sie die Korrelation zwischen Preis und mittlerer Bewertung** zwischen den Angeboten einzeln für jedes Produkt.
- **Erstellen Sie für das Produkt mit der stärksten Korrelation je ein** Modell zwischen Preis und Bewertung mit bzw. ohne Berücksichtigung des Intercepts.
- Interpretieren Sie beide Modelle.
- **Stellen Sie die Daten dieses Produkts sowie die beiden Modelle in einem** Diagramm graphisch dar. Ergänzen Sie auch eine horizontale Linie, die die mittlere Bewertung für diese Produkt darstellt. Die Bewertungsachse sollte bei Null beginnen. Ergänzen Sie Achsenbeschriftung und Legende.
- **Interpretieren Sie anhand des Diagramms die beiden Modelle, auch im** Vergleich zum Mittelwert.
- **Erstellen Sie ebenfalls ein lineares Modell (mit Intercept) für alle** Angebote zwischen normalisierten Preis und mittlerer Bewertung. Stellen Sie auch hier die realen Daten, die Modelldaten sowie den Mittelwert der Bewertung graphisch dar. Bringt das Modell eine Verbesserung gegenüber einer einfachen Mittelwertbetrachtung (graphische Interpretation)?
- **Erstellen Sie nun für eines Ihrer Produkte ein lineares Modell,** das den Zusammenhang zwischen den Produktdaten und dem Preis darstellt.
- Interpretieren Sie das Modell.
- **Erzeugen Sie ein Diagramm, das graphisch den Zusammenhang zwischen** einer metrischen (als Achse) und einer nominalen (als Farbe) Produkteigenschaft und dem Preis darstellt. Stellen Sie die realen Angebotsdaten als Punkte, die Modelldaten dagegen als Geraden dar. Erzeugen Sie eine Legende, die alle Elemente erläutert.

Geben Sie neben dem Jupyter-Notebook auch die CSV-Dateien für die einzelnen Produkte und die zusammengefassten Daten gepackt als ZIP-Datei ab.

Aufgabe 14.3 Erstellen Sie ein lineares Modell für die Abhängigkeiten der Buchstabenhäufigkeiten zwischen englischer und deutscher Sprache. Verwenden Sie dabei relative Häufigkeiten. Stellen Sie den Scatterplot mit beschrifteten Datenpunkten und den beiden Modellgeraden (mit und ohne Intercept) dar.

Erfassen Sie z. B. aus Wikipedia die wichtigsten 15 deutschen Flüsse Flüsse mit Gesamtlänge, Gesamteinzugsgebiet und Wassermenge. Untersuchen Sie zwischen welchen beiden der der Größen Länge, Einzugsgebiet, Wassermenge der stärkste lineare Zusammenhang besteht und stellen Sie diesen graphisch dar.

KAPITEL 15

Zeitreihen

15.1 Überblick

Wollen wir zeitliche Veränderungen in Daten analysieren (Entwicklung Verkehrsdichte, Klimaänderung, Verkaufszahlen, …), stellt uns *R* mit den *Zeitreihen* (Time series) eine geeignete Datenstruktur zur Verfügung.

15.2 Erzeugen von Zeitreihen

Voraussetzung ist ein Datenvektor mit metrischer Skala. Dieser wird wie folgt in eine Zeitreihe umgewandelt:

timeSeries <- **`ts(`**vektor *`[, frequency=`*teilung*`] [, start=`*einheit*`]`***`)`**

Ohne weitere Angaben startet die Zeitreihe bei 1, die Zeitpunkte werden einfach durchnumeriert. Mit dem Parameter `start` kann dieser Wert verändert werden.

Beispiel: Jährliches Nilhochwasser (Datensatz `Nile`). Die Dokumentation verrät, dass die Messungen 1871 beginnen.

```
nil <- ts(Nile, start=1871)
```

Wichtiger ist die Möglichkeit der periodischen Teilung mit dem Parameter `frequency` z. B. in 4 (Quartale) oder 12 (Monate) Einheiten. Damit können saisonbedingte Schwankungen ausgewertet werden. Für den Start kann nun auch ein Vektor von zwei Werten (Jahr und Monat/Quartal) angegeben werden, bei einem Einzelwert wird die Periode auf 1 gesetzt. Es sind aber auch andere Unterteilungen möglich (Stunden des Tages).

Beispiel: Beispieldatensatz (Souvenirverkäufe Januar 1987 bis Dezember 1993) aus tsdl: Time Series Data Library, lokale Kopie: `fancy.dat`

```
data <- scan('fancy.dat')
souvenirs <- ts(data, freq=12, start=c(1987, 1))
```

R enthält ein Zeitreihenbeispiel für die monatlichen Fluggastzahlen einer Airline von Januar 1949 bis Dezember 1960.

```
print(str(AirPassengers))
print(frequency(AirPassengers)
```

```
Time-Series [1:144] from 1949 to 1961: 112 118 132 129 121 135 148 148 136 119 ...

[1] 12
```

Zeigen wir diese Daten an, werden sie gruppiert dargestellt. Für die Frequenzen 4 und 12 erhalten wir auch automatisch passende Spaltenbezeichnungen.

```
print(AirPassengers)
```

```
     Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
1949 112 118 132 129 121 135 148 148 136 119 104 118
1950 115 126 141 135 125 149 170 170 158 133 114 140
1951 145 150 178 163 172 178 199 199 184 162 146 166
1952 171 180 193 181 183 218 230 242 209 191 172 194
1953 196 196 236 235 229 243 264 272 237 211 180 201
1954 204 188 235 227 234 264 302 293 259 229 203 229
1955 242 233 267 269 270 315 364 347 312 274 237 278
1956 284 277 317 313 318 374 413 405 355 306 271 306
1957 315 301 356 348 355 422 465 467 404 347 305 336
1958 340 318 362 348 363 435 491 505 404 359 310 337
1959 360 342 406 396 420 472 548 559 463 407 362 405
1960 417 391 419 461 472 535 622 606 508 461 390 432
```

Für andere Periodenlängen außer 4 und 12 kann die tabellarische Darstellung mit dem Parameter `calendar=T` erzwungen werden.

Beispiel: Gruppierung von Zufallsbeobachtungen nach Wochentagen, Beginn Mittwoch der 1. Woche.

```
x <- ts(rnorm(50, 3, 1), freq=7, start=c(1,3)
print(x, calendar=T)
```

Beispielergebnis (Werte sind zufällig):

```
         p1        p2        p3        p4        p5        p6        p7
1                     3.6392840 2.3810939 1.3622274 1.6403343 1.0415396
2 4.0608673 3.2270464 3.7577422 2.8165191 2.3033485 3.0131014 2.6601728
3 2.5044734 3.4351101 3.1167032 4.4537715 3.2690506 3.6697011 3.4806759
4 2.8316070 4.5650075 2.5536139 1.7795909 2.2865992 3.0538514 3.4656119
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
5 3.2730226 5.2743154 3.2299693 4.7099090 3.9608770 1.6930929 2.6036927
6 1.3799664 1.0758574 2.1356351 2.8860400 4.1535094 4.6239069 2.9539195
7 0.9023984 3.9770955 3.4492774 2.9984157 2.4371954 4.3099190 2.5099039
8 1.9079987 2.2115869 2.1958905
```

Vertiefung

Die Änderung der Spaltennamen ist möglich, aber aufwendig (über die Funktion `.preformat.ts()`).

Werden Zeitreihen graphisch dargestellt, werden die Datenpunkte automatisch durch Linien verbunden.

Beispiel: Souvenirs.

```
plot(souvenirs)
```

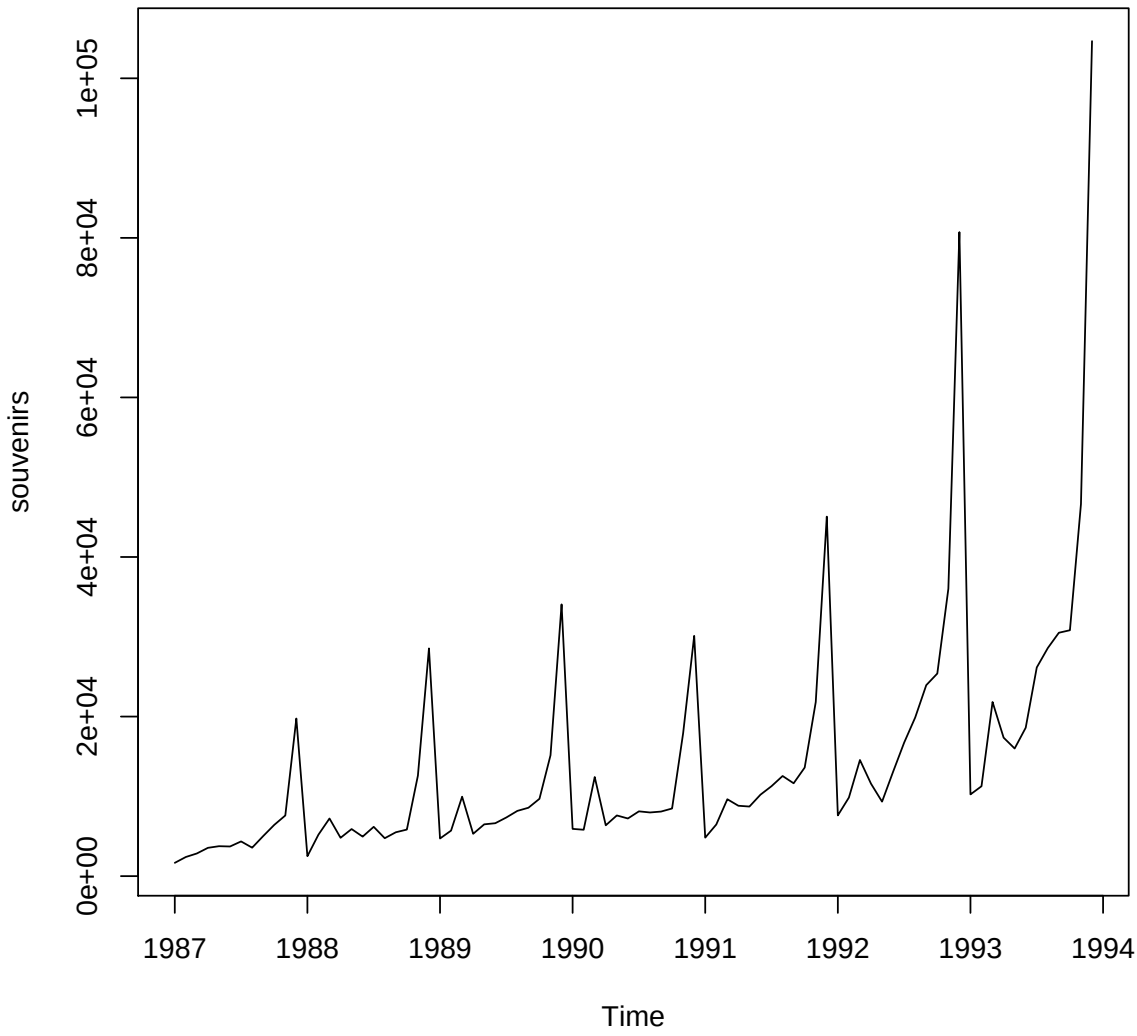


Abb. 15.1: Zeitreihe Souveniverkauf

Wir können Abschnitte aus Zeitreihen zwar wie bei Vektoren durch die Angabe eines Indexbereiches `zeitreihe[anfang:ende]` herauslösen, allerdings ist das Ergebnis ein Vektor, der wieder manuell in eine Zeitreihe umgewandelt werden muss. Deshalb verwendet man besser die Funktion `window()`.

abschnitt <- **window(**zeitreihe*[*, **start=**startwert*][*, **end=**endwert*]***)**

Die Start- und Enwerte können aus dem Index mit der Funktion `time()` berechnet werden.

zeitVektor <- **time(**zeitreihe**)**

Wir erhalten einen Vektor reeller Zahlen. Die Vorkommastelle durchläuft die Zahl der perioden, die Nachkommastelle ist die Position innerhalb der Periode. Diese Werte können für die Parameter `start` und `end` der `window()`-Funktion verwendet werden.

Beispiel: Flugpassagiere von 25 bis 75% des Zeitverlaufs.

```
idx1 <- length(AirPassangers) %/% 4
idx2 <- (length(AirPassengers)*3) %/% 4 # Achtung: Klammern erforderlich!
interval <- window(AirPassengers,
  start=time(AirPassengers)[idx1],
  end=time(AirPassengers)[idx2]))
print(interval)
```

Ergebnis:

```
     Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
1951                                             166
1952 171 180 193 181 183 218 230 242 209 191 172 194
1953 196 196 236 235 229 243 264 272 237 211 180 201
1954 204 188 235 227 234 264 302 293 259 229 203 229
1955 242 233 267 269 270 315 364 347 312 274 237 278
1956 284 277 317 313 318 374 413 405 355 306 271 306
1957 315 301 356 348 355 422 465 467 404 347 305 336
```

15.3 Zeitreihenzerlegung

Bestehen Zeitreihen aus gleichmäßigen Intervallen, z. B. Wochentagen einer Woche oder Monaten eines Jahres, sind regelmäßige Schwankungen innerhalb der Intervalle zu erwarten. Die Daten der Zeitreihe setzen sich dann häufig aus

- einem langfristigen Trend,
- einer periodischen (saisonalen) Schwankung,
- einer Zufallskomponente

zusammen. *R* bietet eine einfache Möglichkeit, die Zeitreihe in diese Komponenten zu zerlegen.

parts <- **decompose(**zeitreihe**)**

Das Ergebnis ist eine Liste, die die Zerlegung in mehrere Zeitreihen für die einzelnen Anteile enthält. Einen schnellen Eindruck liefert die `plot()`-Funktion.

plot(parts**)**

Beispiel. Für unsere Souvenirverkäufe erhalten wir:

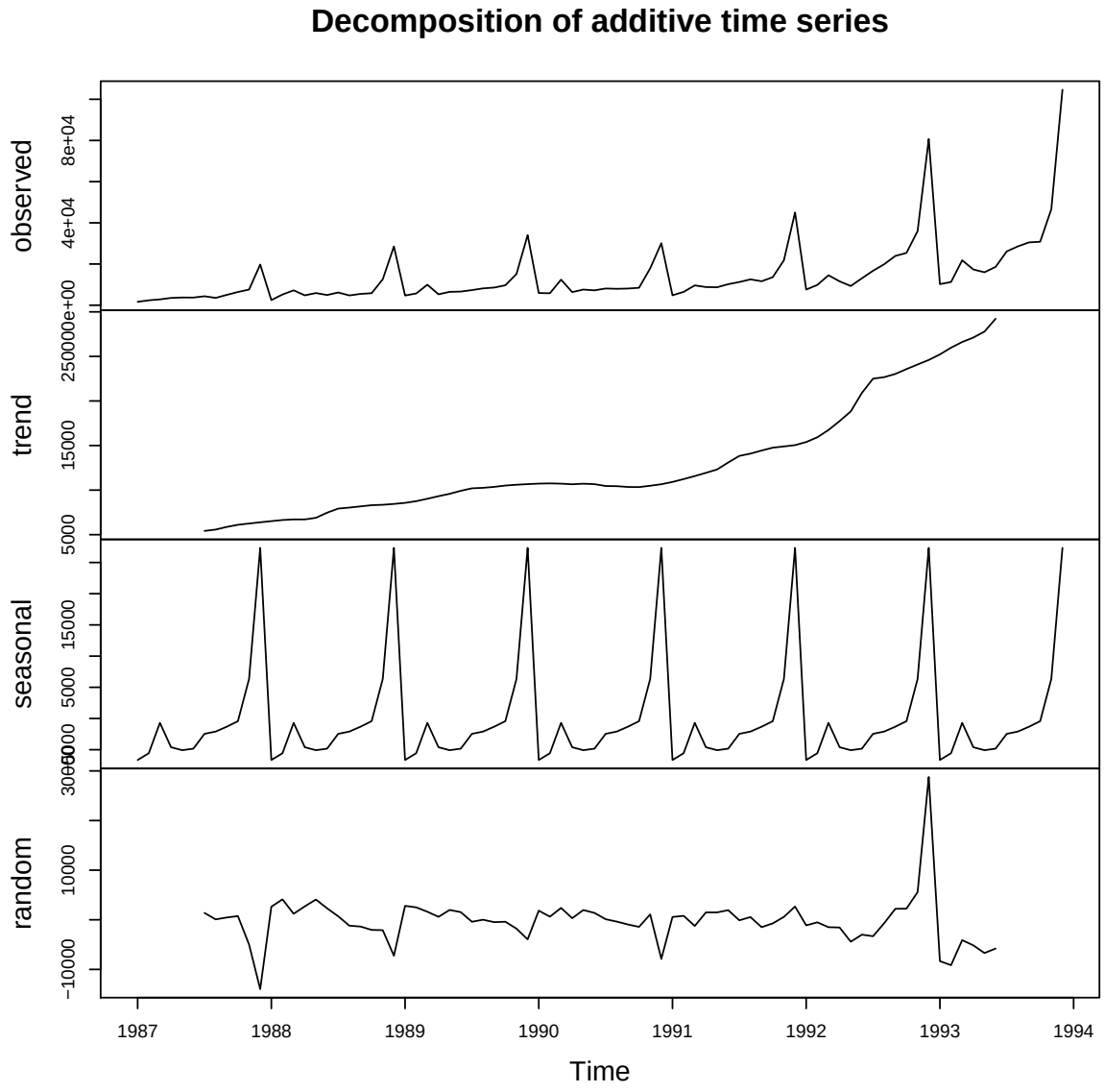


Abb. 15.2: Zeitreihenzerlegung

15.4 Linearer Trend

Bei Zeitreihen stellt sich häufig die Frage, ob diese einem Trend unterliegen (langfristiger Zuwachs oder Abnahme). Wir können dazu wiederum ein lineares Modell aufstellen. Der Zeitparameter ist ein Vektor der Zahlen von 1 bis zur Anzahl der Werte.

t <- **1:length(**zeitreihe**)**
model <- **lm(**zeitreihe **~** t**)**

Beispiel: Lineares Modell der Nilüberschwemmungen.

```
nil <- ts(Nile, start=1871)
t <- 1:length(nil)
model.nil <- lm(nil ~ t)
```

Mit `summary(model.nil)` erhalten wir:

```
Call:
lm(formula = nil ~ t)

Residuals:
    Min      1Q  Median      3Q     Max
-483.71  -98.17  -23.21  111.40  368.72

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 1056.4224    30.3377  34.822  < 2e-16 ***
t             -2.7143     0.5216  -5.204 1.07e-06 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 150.6 on 98 degrees of freedom
Multiple R-squared:  0.2165,        Adjusted R-squared:  0.2085
F-statistic: 27.08 on 1 and 98 DF,  p-value: 1.072e-06
```

Das Plotten einer Zeitreihe geschieht wieder einfach mit dem Befehl `plot()`. Wollen wir jedoch den Trend einzeichnen, funktioniert der Befehl `abline()` nur dann, wenn beide Parameter `start` und `frequency` den Wert 1 haben. Wir berechnen statt dessen die theoretischen Modellwerte, bilden daraus eine neue Zeitreihe und plotten diese:

modellWerte <- **fitted(**modell**)**
modellZeitreihe <- **ts(**modellWerte*[*, **frequency=**wert, **start=(**wert1, wert2**)***]***)
lines(modellZeitreihe ...**)**

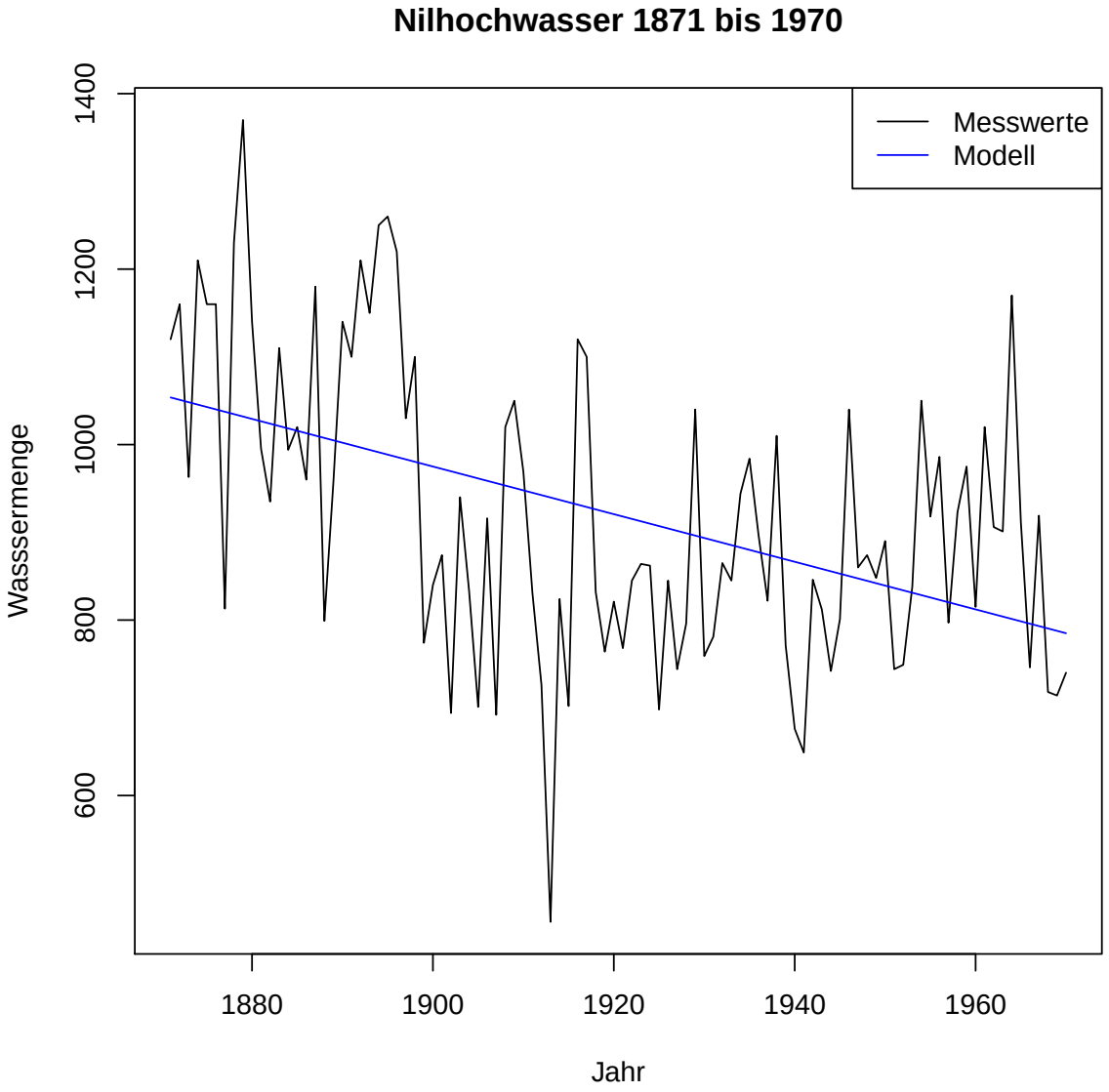
Beispiel Nil:

```
plot(nil)
nil.ts <- ts(fitted(model.nil), start=1871)
lines(nil.ts)
```

Ergebnis:

15.5 Exponentieller Trend

Häufig beobachten wir bei Zeitreihen auch einen exponentiellen Trend. Dieser wird modelliert, indem wir die abhängige Größe logarithmieren:


Nilhochwasser 1871 bis 1970
Messwerte
Modell
Wasssermenge
600
800
1000
1200
1400
1880
1900
1920
1940
1960
Jahr

expModell <- `log(`abhängigeGröße`)` ~unabhängigeGröße

Beispiel: Entwicklung Flugpassagierzahlen, lineares und exponentielles Modell.

```
ts.air <- ts(AirPassengers, frequency=12, start=c(1949,1))
t <- 1:length(ts.air)
model1.air <- lm(ts.air ~ t)
model2.air <- lm(log(ts.air) ~ t)
```

Lineares Modell:

```
Call:
lm(formula = ts.air ~ t)

Residuals:
    Min      1Q  Median      3Q     Max
-93.858 -30.727  -5.757  24.489 164.999

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 87.65278    7.71635   11.36   <2e-16 ***
t            2.65718    0.09233   28.78   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 46.06 on 142 degrees of freedom
Multiple R-squared:  0.8536,        Adjusted R-squared:  0.8526
F-statistic: 828.2 on 1 and 142 DF,  p-value: < 2.2e-16
```

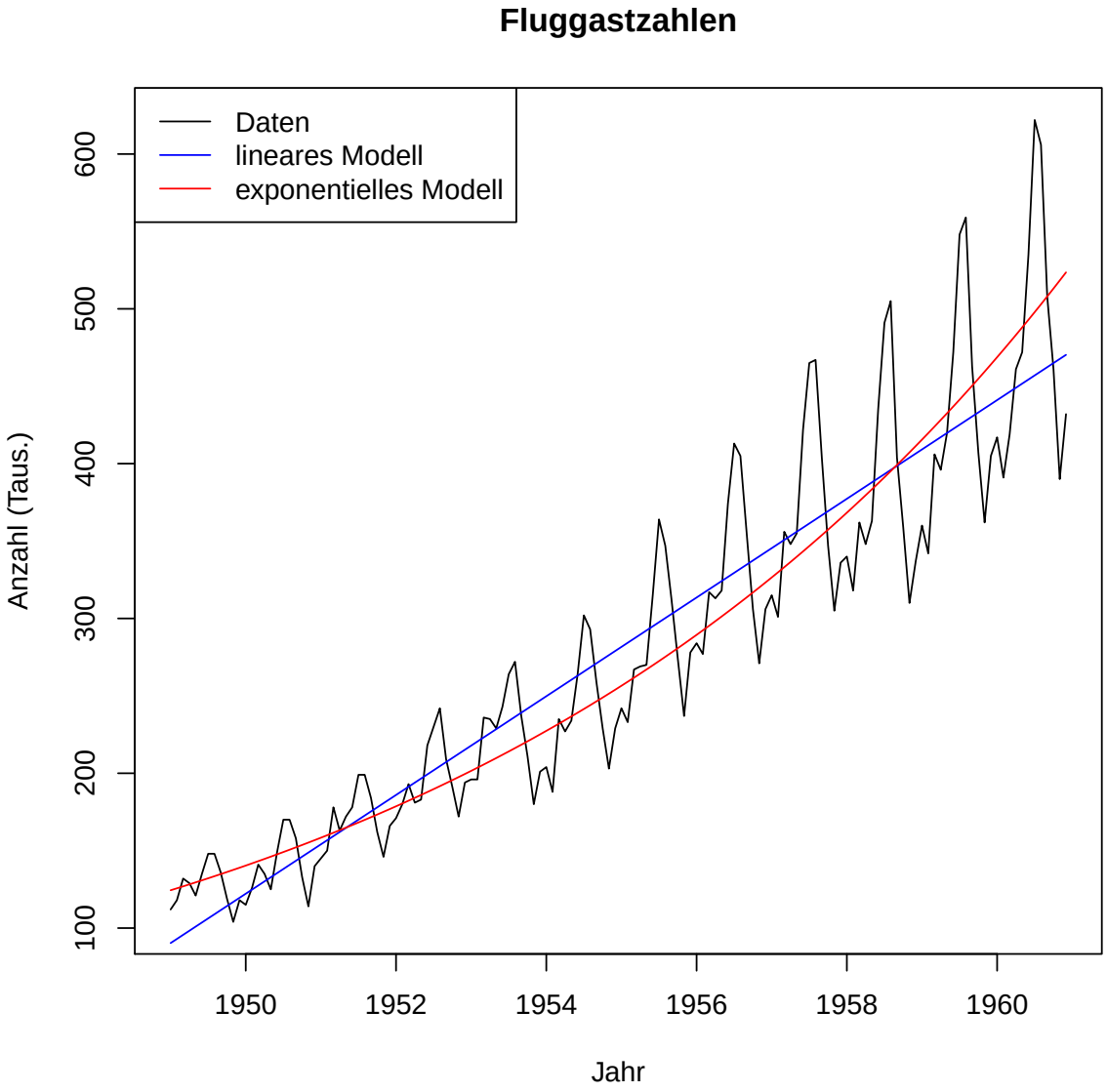
Exponentielles Modell:

```
Call:
lm(formula = log(ts.air) ~ t)

Residuals:
     Min       1Q   Median       3Q      Max
-0.30858 -0.10388 -0.01796  0.09738  0.29538

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 4.8136683  0.0232940  206.65   <2e-16 ***
t           0.0100484  0.0002787   36.05   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.139 on 142 degrees of freedom
Multiple R-squared:  0.9015,        Adjusted R-squared:  0.9008
F-statistic:  1300 on 1 and 142 DF,  p-value: < 2.2e-16
```

Wir erhalten jeweils Koeffizienten für ein lineares Modell mit niedrigen alpha-Werten (Fehler 1. Art), allerdings sind die Residuen beim zweiten Modell deutlich kleiner – allerdings wurden die Werte für das Modell logarithmiert und erscheinen dadurch kleiner. Deshalb bereichnen wir die Werte, die wir nach Modell beobachten würden:

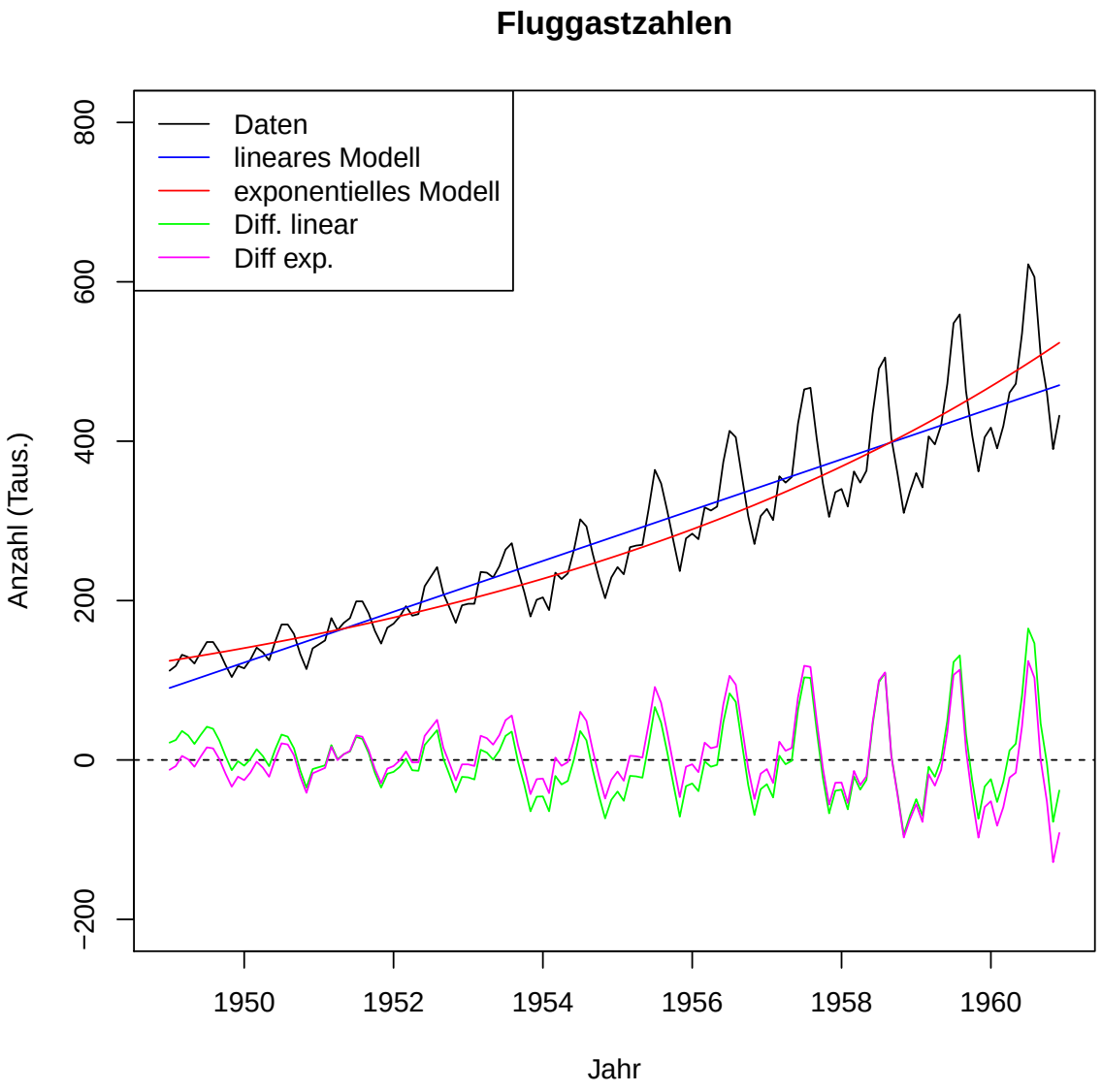
```
fit.model2 <- exp(fitted(model2.air))
ts.model2 <- ts(fit.model2, frequency=12, start=c(1949, 1))
```

Ein Plot des linearen und exponentiellen Modells lässt vermuten, dass das zweite Modell besser passt.


Fluggastzahlen
Daten
lineares Modell
exponentielles Modell
Anzahl (Taus.)
100
200
300
400
500
600
1950
1952
1954
1956
1958
1960
Jahr

Um die Abweichungen besser zu erkennen, können wir auch die Differenzen zwischen Modell und beobachteten Daten plotten:

```
lines(ts.air - ts.model2)
```



Beim Vergleich zwischen linearen und exponentiellen Modell ist optisch nicht zu erkennen, welches Modell besser passt.

15.6 Gleitender Durchschnitt

Oft ist überhaupt kein durchgängiger Trend erkennbar. Dann kann man versuchen, einen gleitenden Durchschnitt über mehrere Zeitpunkt zu bilden. Dazu werden ausgehend von jedem Zeitpunkt der Mittelwert eines bestimmten Intervalls entweder links vom oder symmetrisch zum Zeitpunkt gebildet.

```
gleitMittel <- filter(zeitReihe, gewichtsVektor, sides=1|2>)
```

Der Gewichtsvektor bestimmt die Größe des Intervalls, seine Werte sind die Gewichte, die sich zu 1 summieren sollten und im einfachsten Fall alle gleichgroß gewählt werden. Bei periodischen Zeitreihen

sollte die Intervallänge gleich oder ein Vielfaches der Periodenlänge sein. Das Ergebnis ist wieder eine Zeitreihe.

Beispiel: Glättung der Nilhochwasser. Keine Periode erkennbar, wir wählen 5- und 10-Jahres-Mittel, nur Vergangenheit berücksichtigen. Plot der originalen und gefilterten Werte.

```
m5.nil <- filter(nil, rep(1/5, 5), sides=1)
m10.nil <- filter(nil, rep(1/10, 10), sides=1)
plot(nil)
lines(m5.nil, col='blue')
lines(m10.nil, col='red')
```

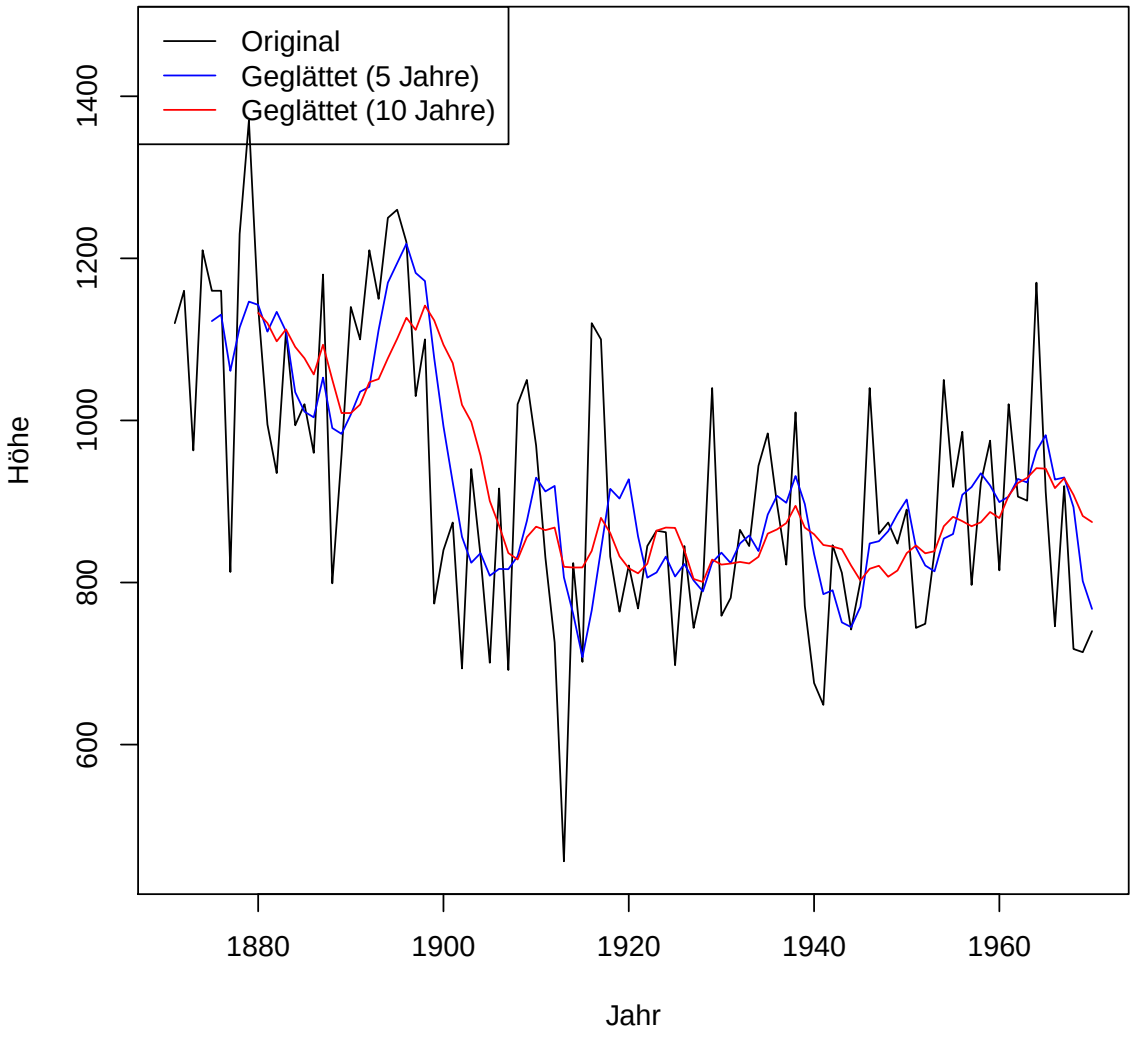


Abb. 15.3: Gleitender Durchschnitt

Wir können nun auch die Differenz zwischen Originalwerten und gleitenden Durchschnitt, die die Varianz der Daten beschreibt, weiter untersuchen.

Beispiel Souvenierverkauf. Vergleich von linearem und exponentiellem Modell für den Trend mit dem gleitenden Mittel (12 Werte). Wir wählen einen einseitigen Filter, damit wir auch für die aktuellen Werte eine Glättung erhalten, und verzichten dafür auf einige Anfangswerte.

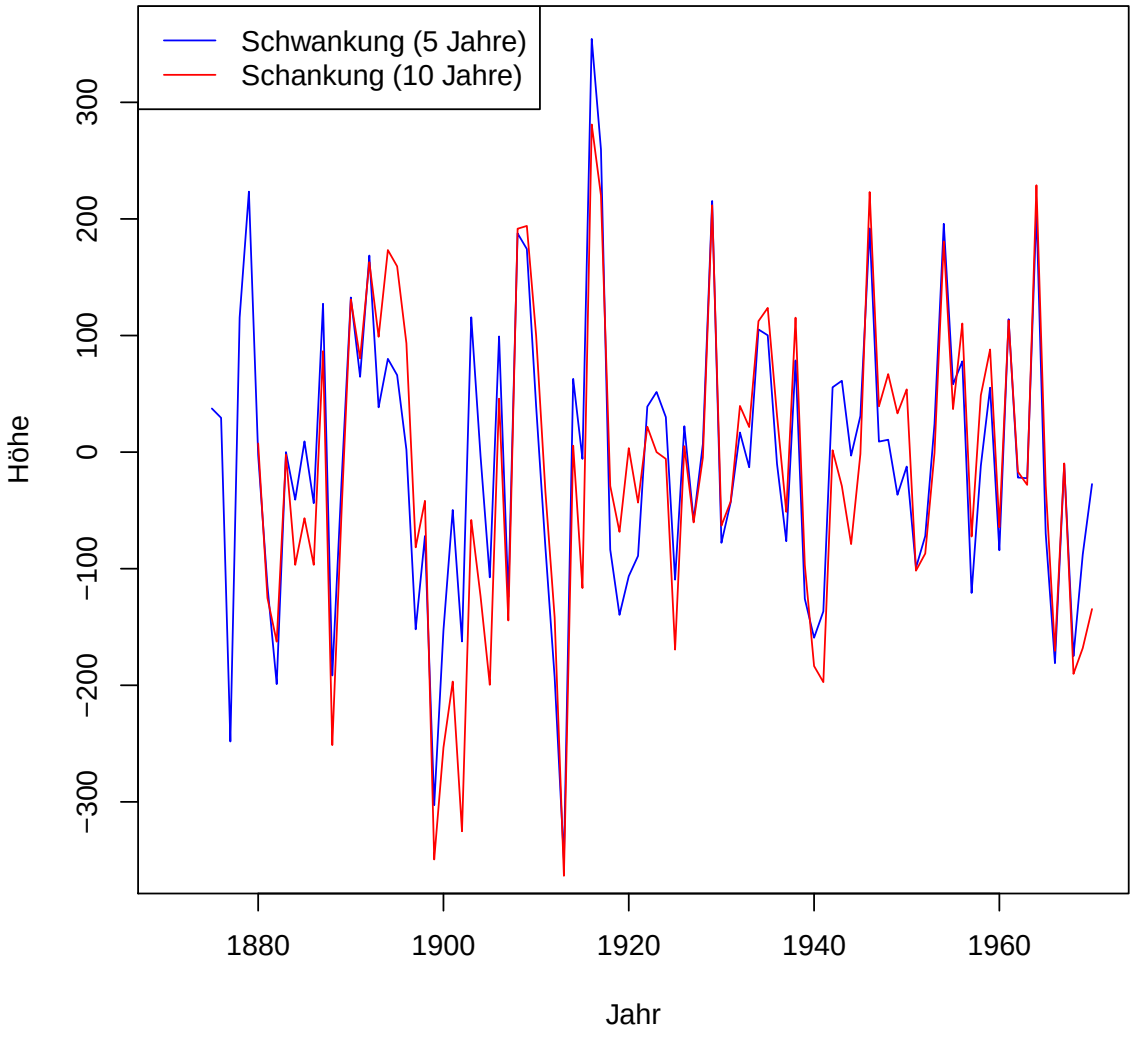


Abb. 15.4: Schwankung um Durchschnitt

```
data <- scan("fancy.dat")
souvenirs <- ts(data, freq=12, start=c(1987,1))

# Zeitachse
t <- 1:length(souvenirs)

# Lineares Modell
mo1 <- lm(souvenirs ~ t)
ts.mo1 <- ts(fitted(mo1), freq=12, start=c(1987, 1))

# Exponentielles Modell
mo2 <- lm(log(souvenirs) ~ t)
ts.mo2 <- ts(exp(fitted(mo2)), freq=12, start=c(1987, 1))

# Gleitendes Mittel
fi <- filter(souvenirs, rep(1/12, 12), sides=1)

# Plot
plot(souvenirs)
lines(ts.mo1, col='blue')
lines(ts.mo2, col='red')
lines(fi, col='green')

# Differenzen
diff1 <- souvenirs - ts.mo1
diff2 <- souvenirs - ts.mo2
diff3 <- souvenirs - fi

# Plot Differenzen
...
```

Wir sehen, dass auch die periodischen Schwankungen mit den Trendwerten und damit mit der Zeit anwachsen. In solchen Fällen ist eine multiplikative Zerlegung in die Kompenenten Trend, periodische und zufällige Schwankungen oft besser geeignet.

```
# Quotienten
quot1 <- souvenirs / ts.mo1
quot2 <- souvenirs / ts.mo2
quot3 <- souvenirs / fi

# Plot, ausblenden der ersten beiden Jahre
plot(window(quot1, start=c(1989, 1)))
...
```

Alternativ können wir statt der originalen Zeitreihe auch die logarithmierte Zeitreihe zerlegen.

```
parts.log <- decompose(log(souvenirs))
plot(parts.log)
```

Wir sehen, dass hier die Zufallskomponente nicht mehr mit der Zeit ansteigt, sondern gleichmäßig weit ausschlägt.

15.7 Saisonale Schwankungen

Um den Anteil der saisonalen Schwankungen besser untersuchen zu können, subtrahieren wir von der Zeitreihe den Trend.

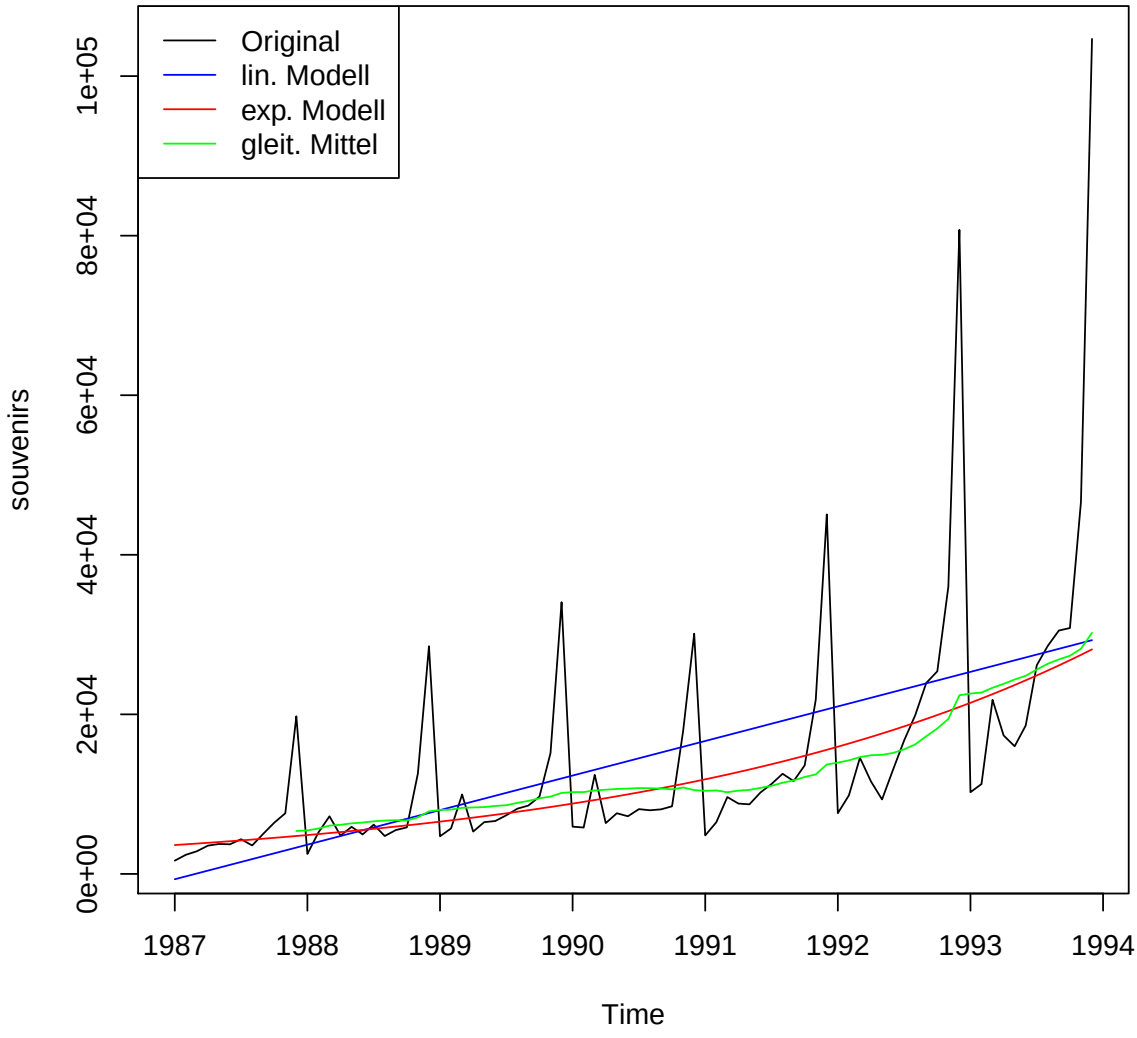


Abb. 15.5: Trend mit linear./exp. Modell und Filter

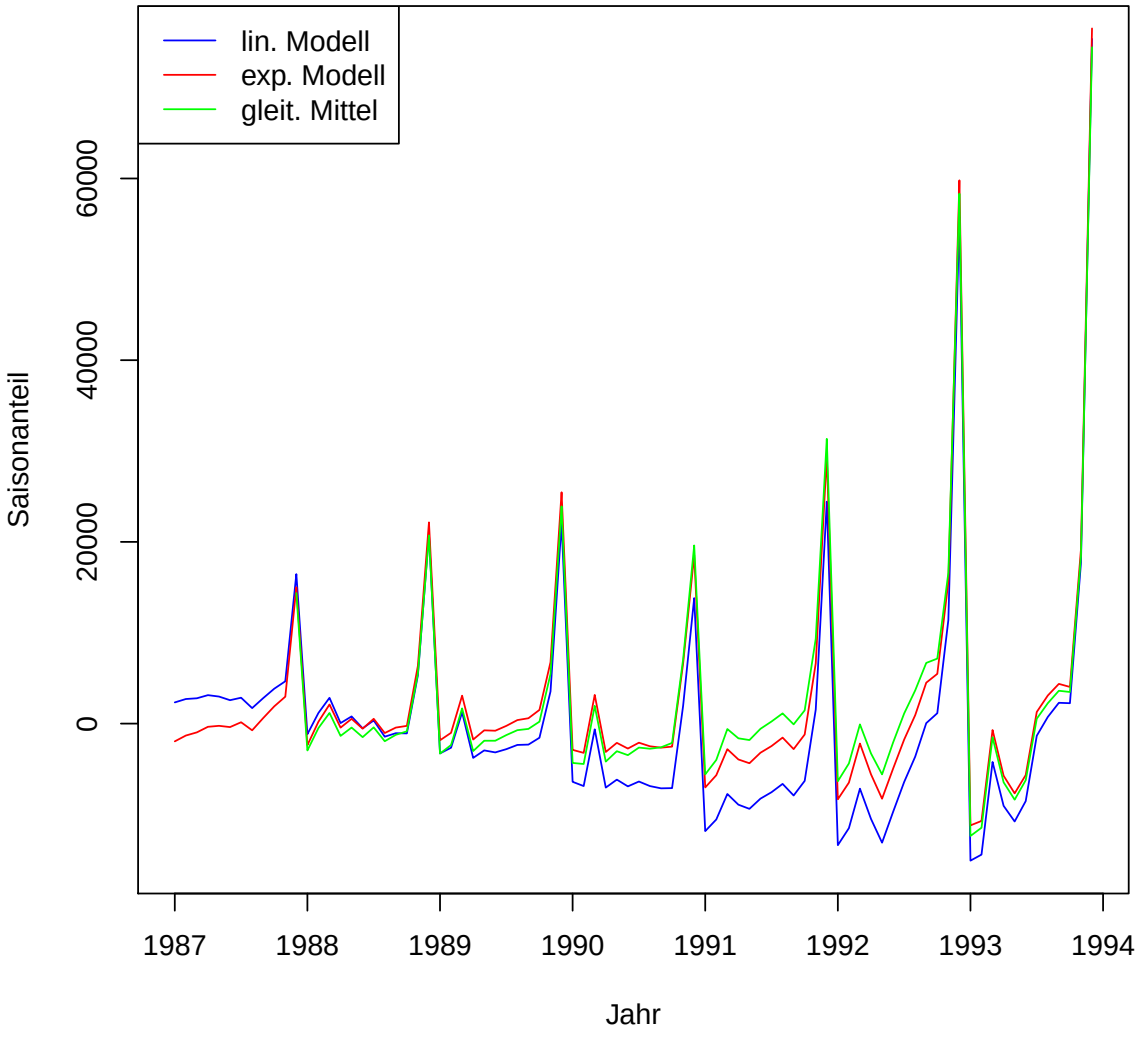


Abb. 15.6: Schwankung der Daten um den Trend

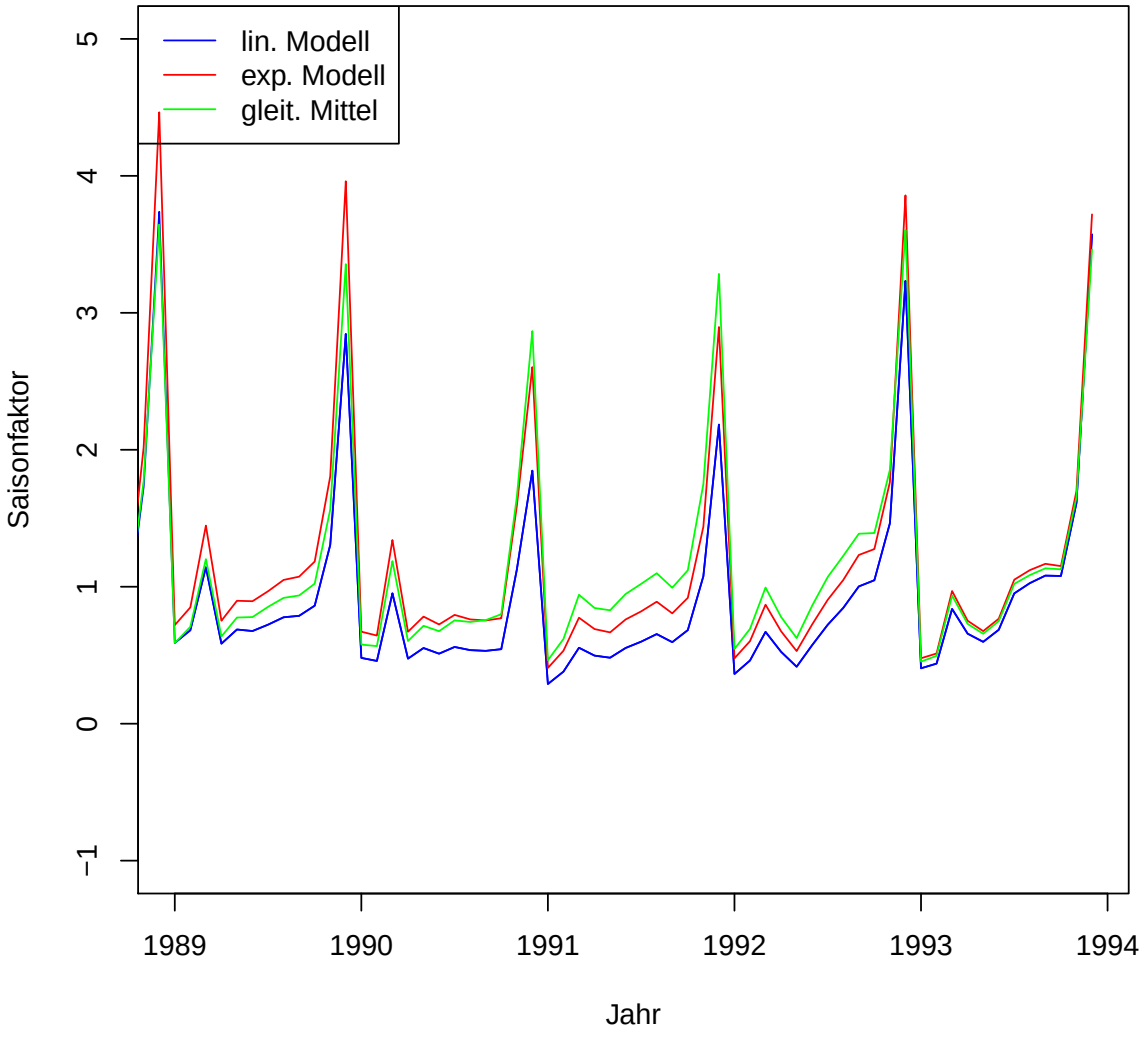


Abb. 15.7: Multiplikative Zerlegung

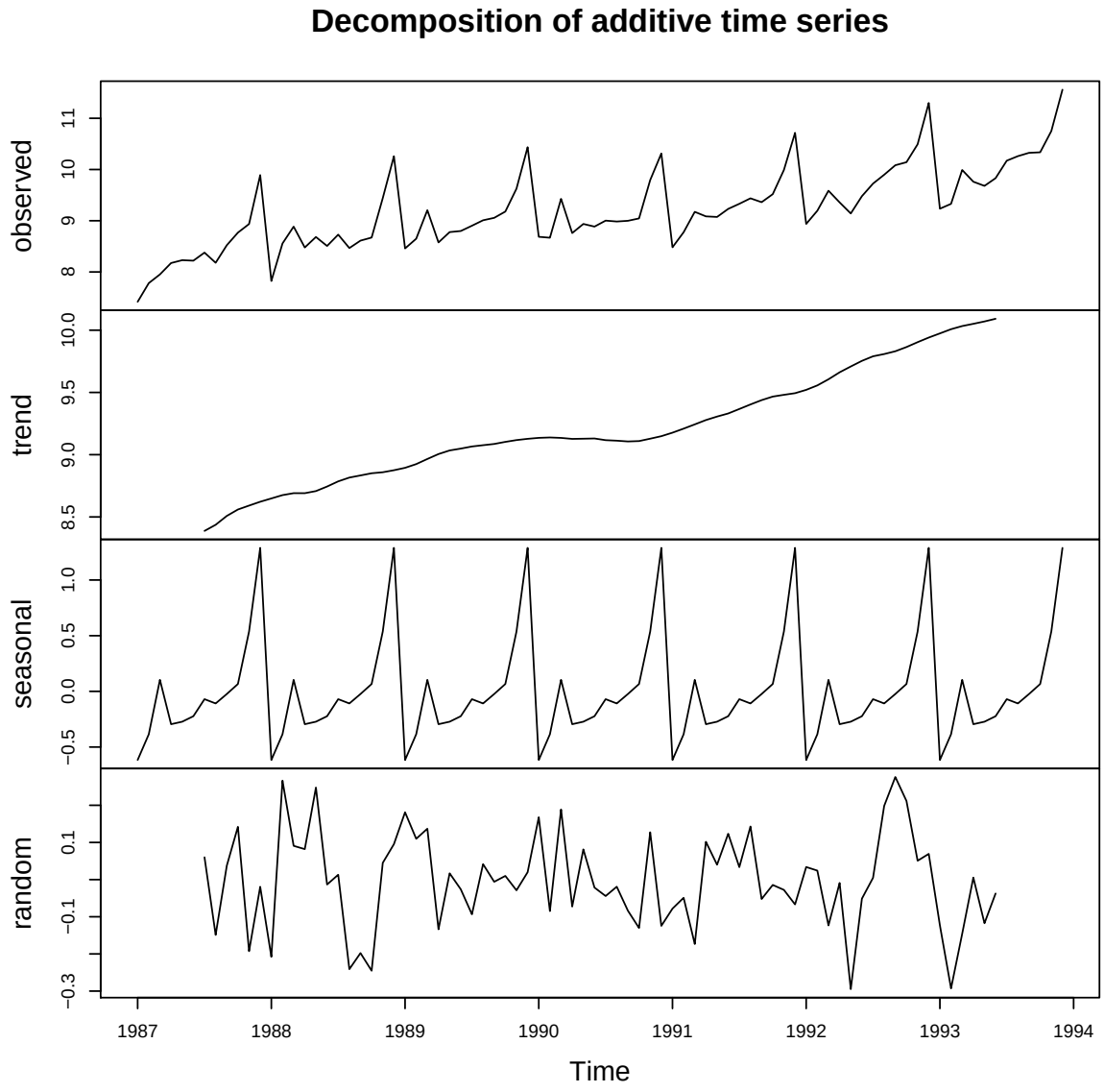


Abb. 15.8: Zerlegung der logarithmierten Verkaufszahlen

Beispiel: Souvenirverkäufe, Trend über exponentielles Modell.

```
t <- 1:length(souvenirs)
souv.mod <- lm(log(souvenirs) ~ t)
souv.trend <- ts(exp(fitted(souv.mod)), freq=12, start=c(1987,1))
souv.season <- souvenirs - souv.trend
```

Warnung: Wenn wir statt einer Modellvorhersage einen gleitenden Durchschnitt verwenden, enthält dieser `NA`-Werte an den Rändern. Diese müssen vor den weiteren Berechnungen gegebenenfalls abgeschnitten (Funktion `window()`) oder durch Default-Werte ersetzt werden.

Um diese Zeitreihe nach der Periodenunterteilung gruppieren zu können, verwenden wir die Funktion `cycle()`, die für jeden Wert der Zeitreihe zyklisch einen Faktor von 1 bis `frequency` erzeugt.

season <- **`cycle(`**zeitreihe**`)`**

Für unser Beispiel erhalten wir eine Zeitreihe der Länge 84, die die Werte 1 bis 12 zyklisch durchläuft. Damit können die Verteilung der Daten innerhalb der Periode untersuchen.

Beispiel: Boxplot.

```
months <- cycle(souvenirs)
boxplot(souv.season ~ months)
```

Da wir bereits gesehen hatten, dass hier eher ein multiplikatives Modell vorliegt, sollten wir besser den monatlichen Faktor untersuchen.

```
season.factor <- souvenirs / souv.trend
boxplot(season.factor ~ months)
```

Wir erkennen, dass der Dezember neben den höchsten Verkaufszahlen (australischer Sommer, Feiertage) auch die höchste Streuung hat.

Eine Zusammenfassung der Werte kann mit `tapply()` erfolgen. Wir können z. B. ermitteln, wie stark die jeweilige Saison vom Trend abweicht.

```
season.diff <- tapply(souv.season, months, FUN=mean)
barlot(season.diff)
```

Aufgrund der multiplikativen Zusammensetzung sollten auch hier besser die mittleren Faktoren statt der Differenzen dargestellt werden.

15.8 Autokorrelation

Interessant ist häufig die Frage, inwieweit die Beobachtungen einer Zeitreihe von der Vergangenheit abhängig sind, ob sich z. B. aus den Daten des Vormonats oder des Vorjahres die aktuellen Daten vorhersagen lassen. Zur Untersuchung dieser Frage dient die *Autokorrelation*, bei der die Zeitreihe um sogenannte *Lags* (das sind einfach Zeitschritte) verschoben und mit sich selbst korreliert wird. Haben wir wie bei den Verkaufszahlen regelmäßige monatliche Schwankungen, sollten bei einem Lag von 12 (also genau ein Jahr) ziemlich hohe Ähnlichkeiten auftreten.

Neben der Vorhersage können wir auf diese Weise auch bisher unbekannte periodische Effekte erkennen.

Eine einfache visuelle Analyse liefert die Funktion `lags.plot()`.

`lag.plot(`zeitreihe**`, lags=`**anzahl*`[, do.lines=F]`*)

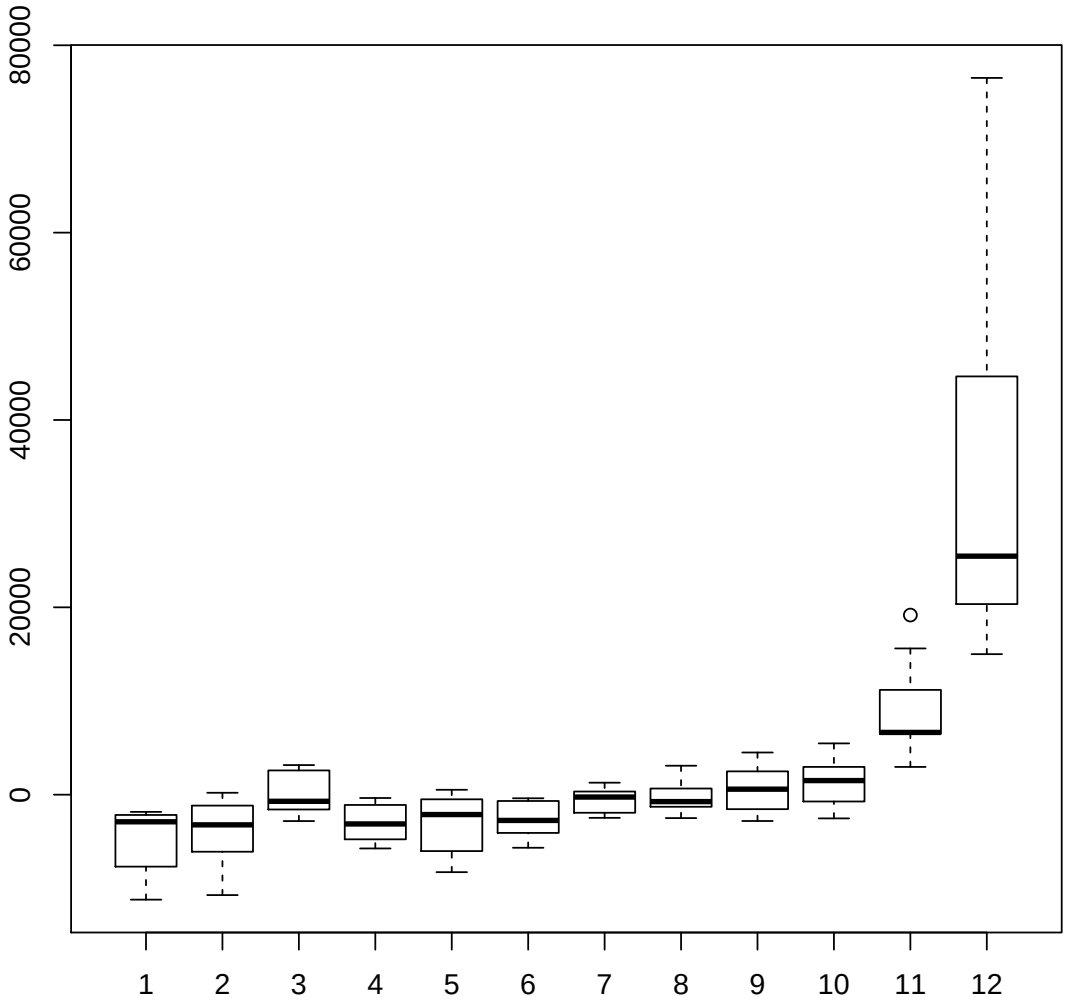


Abb. 15.9: Streuung der Monate

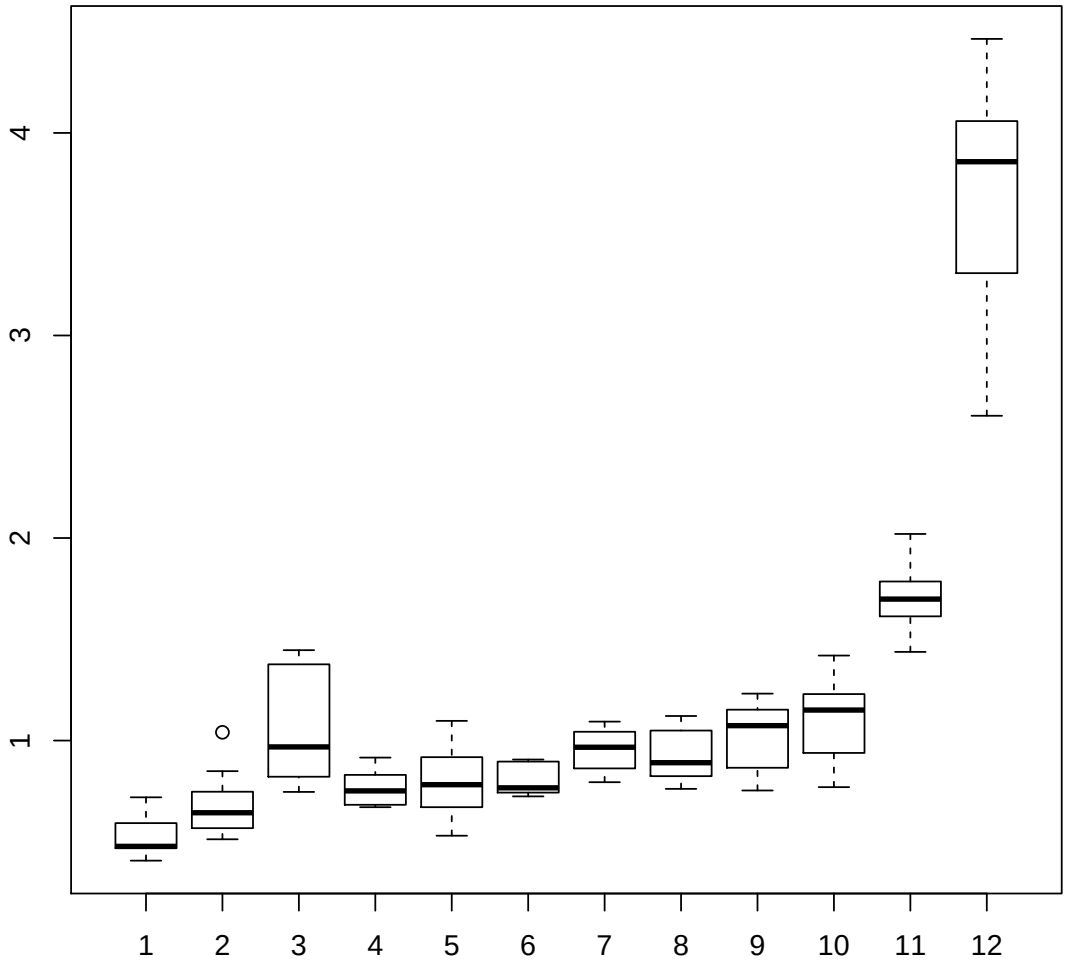


Abb. 15.10: Streuung des Monatsfaktors

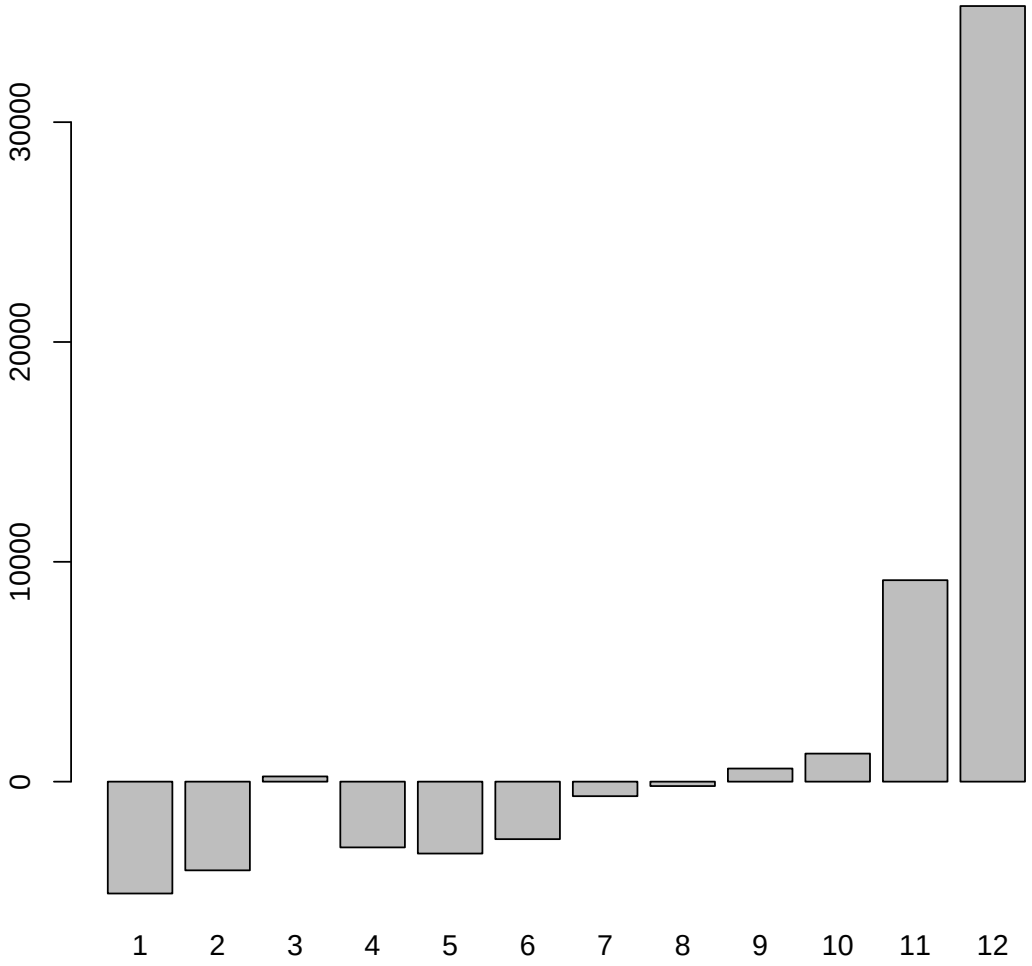
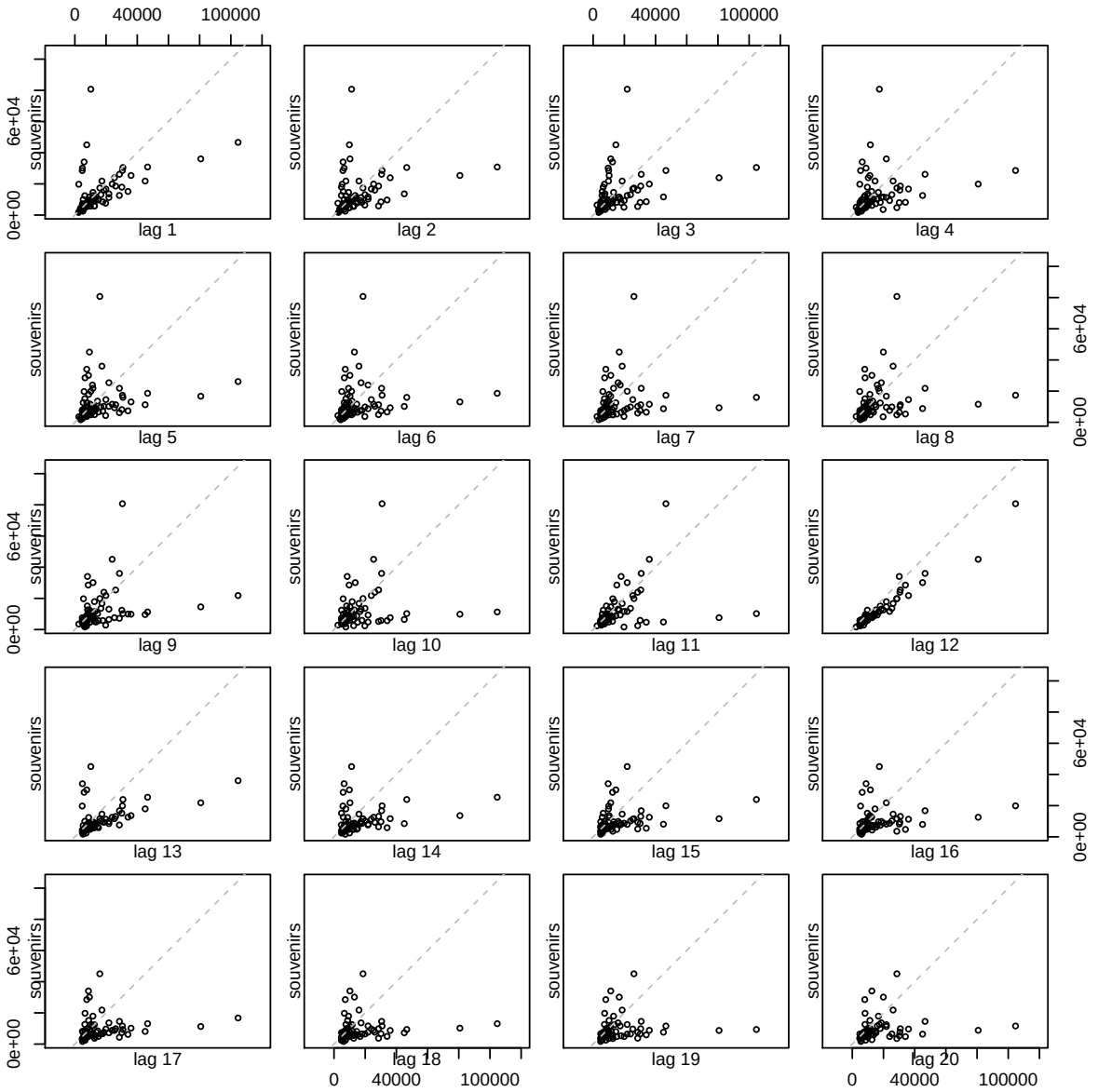


Abb. 15.11: Mittlere Abweichung vom Trend (absolut)

Die Zeitreihe wird dazu um 1, 2, …, `lags` Zeitpunkte verschoben und mit der beobachteten Zeitreihe jeweils ein Scatterplot erstellt. Bei hoher Korrelation eines Lags liegen die Punkte nahezu auf einer Geraden.

Beispiel: Lag-Plot für Souvenirverkauf.

```
lag.plot(souvnirs, lags=20, do.lines=F)
```



Eine genauere Analyse der Stärke dieser Korrelation liefert die *Autokorrelationsfunktion* `acf()`. Sie berechnet den Korrelationskoeffizienten zwischen jeder Lag und der originalen Zeitreihe und stellt diese graphisch dar.

`acf(`zeitreihe*`[`*`,lag.max=`**`]`**anzahl **`, plot=F`** *...`]`*`)`

Neben der Höhe des Korrelationskoeffizienten wird auch ein Signifikanzintervall dargestellt.

Beispiel: Autokorrelation für Souvenirs für 3 Jahre. Die maximale Zahl der Lags muss in tatsächlichen Zeitschritten erfolgen, die Beschriftung der x-Achse erfolgt dagegen in Intervallen (also 12 Monate entspricht der 1).

```
acf(souvenirs, lag.max=36)
```

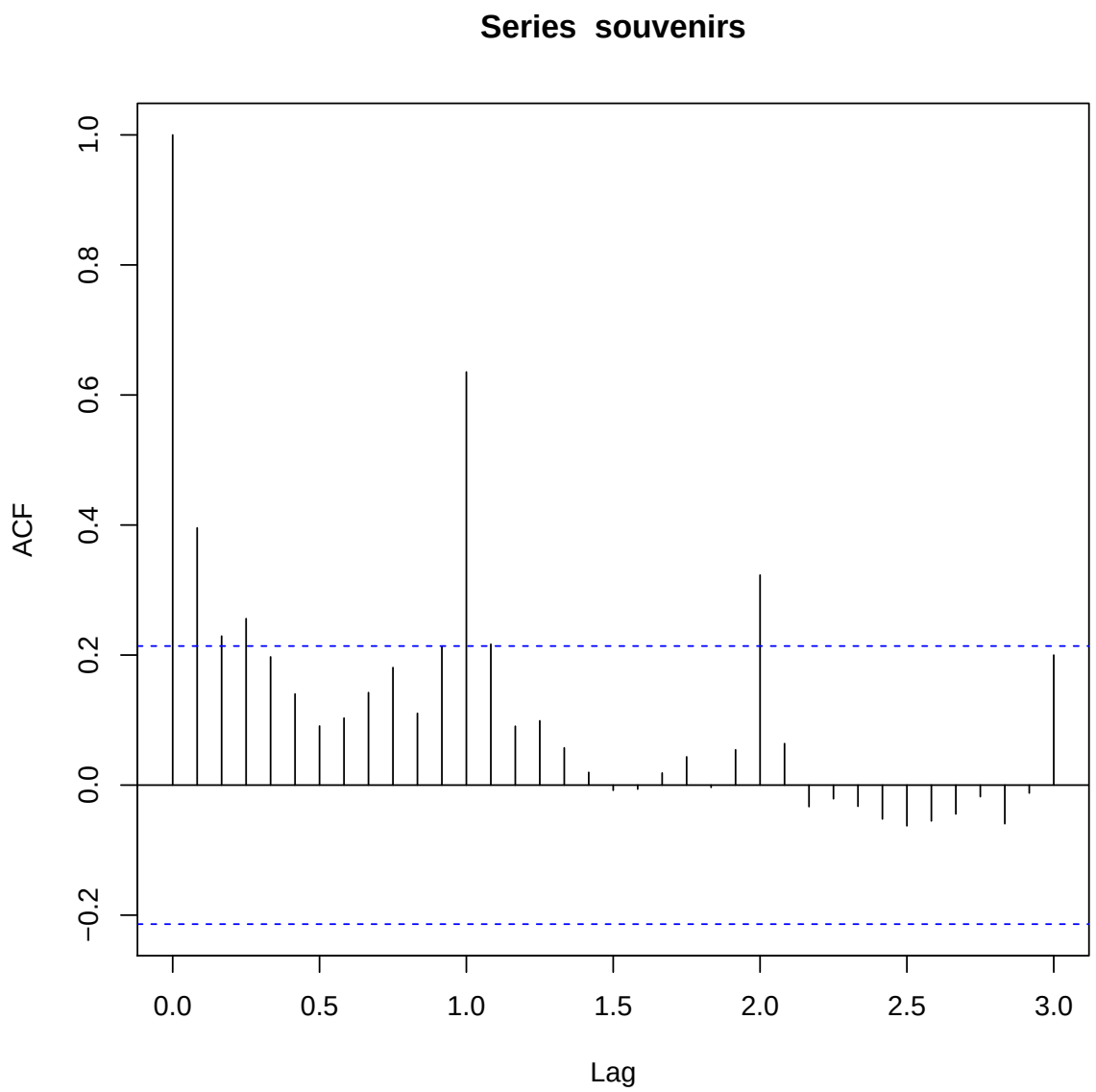


Abb. 15.12: Autokorrelation Souvenirs

Wir sehen, dass eine Korrelation zum Vormonat und deutlich stärker zum Vorjahr besteht. Über mehrere Jahre schwächt sich diese jedoch ab.

KAPITEL 16

Graphiken mit ggplot2

16.1 Grundlagen und Installation

Das Paket *ggplot* erzeugt leistungsfähigere Graphiken als die eingebaute Bibliothek und definiert dazu eine saubere Grammatik. Es ist nicht in der Grundinstallation enthalten und muss deshalb installiert werden. In der *Miniconda*-Umgebung ist diese Bibliothek im Metapaket `r-essentials` enthalten, als Einzelpaket kann es mit dem Befehl

```
conda install -c conda-forge r-ggplot2
```

installiert werden. Für den *SVG*-Export der Graphiken (wie für dieses Script benötigt), müssen wir noch das Paket `r-svglite` auf die gleiche Weise installieren.

Wird außerhalb der *Miniconda*-Umgebung z. B. mit *RStudio* gearbeitet, kann die Bibliothek zentral durch den Administrator installiert werden (*Debian*/*Ubuntu* etc.: `r-cran-ggplot2`). Als normaler Nutzer ist die Installation aus der *R*-Umgebung mit dem Befehl

```
install.packages("ggplot2")
```

möglich.

Wollen wir diese Bibliothek nutzen, müssen wir sie jedesmal vor ihrer Verwendung laden:

```
library(ggplot2)
```

Eine automatische Nachinstallation eines fehlenden Pakets kann mit folgendem Aufruf erfolgen:

```
if (!require(paketName)) install.packages("paketName")
```

Als Beispieldatensatz verwenden wir den Datensatz *Student Performance* des UCI Machine Learning Repository für den Mathematikkurs `student-mat.csv`. Er enthält verschiedene Daten wie Geschlecht, Alter, Wohnort sowie die Studienergebnisse nach der 1., 2. und 3. Prüfung (0 bis 20 Punkte).

```
student <- read.csv2('student-mat.csv', header=T, as.is=F)
```

Die ausführliche Dokumentation [wic09] verwendet den mitgelieferten Datensatz *Diamonds*. Dieser ist allerdings sehr groß, es bietet sich deshalb an, aus diesem eine Auswahl vorzunehmen:

```
smallIdx <- sample(1:nrow(diamonds), 1000)
dia.small <- diamonds[smallIdx,]
```

16.2 Grundlagen

Jeder Plot besteht aus 3 Komponenten:

- den Daten in Form eines Data Frame,
- einer Abbildung zwischen den Variablen (Spalten) des Data Frame und der entsprechenden Darstellung (Position, Farbe, Größe, ...), die *Aesthetics* genannt wird,
- einer Geometrieebene, in der die konkrete Darstellung (Punkte, Linien, Balken, ...) beschrieben wird.

Ein einfacher Plot-Befehl hat damit folgenden Aufbau:

```
ggplot(dataFrame, aes([x=]xAchse, [y=]yAchse ...))
    + geometrieFunktion()
```

Da wir fast immer eine x- und eine y-Achse benötigen, werden die beiden ersten Parameter von `aes()` als diese interpretiert, die Parameternamen dürfen deshalb entfallen.

Die Funktion `ggplot()` liefert ein Plot-Objekt zurück, zu dem die weiteren Komponenten hinzugefügt werden. Wir können statt dessen auch schreiben:

```
g <- ggplot(...)
g <- g + geometrieFunktion()
g <- g + ...
g
```

Ohne die letzte Zeile erfolgt keine Ausgabe der Graphik!

Einen Scatterplot erhalten wir mit der Geometriefunktion `geom_point()`. Wir wollen z. B. die Ergebnisse der 1. und der 3. Prüfung (Spalten `G1` und `G3`) vergleichen:

```
ggplot(student, aes(G1, G3)) + geom_point()
```

Wir können weitere Variablen des Data Frame in Form von Farbe, Form oder Punktgröße darstellen. Dazu erweitern wir unser Aesthetic Mapping `aes()`:

```
aes(x, y, [col=variable1,] [ size=variable2,] [shape=variable3])
```

In der Farbe und Form können Nominaldaten, in der Größe Ordinal- oder metrische Daten codiert werden.

Wir wollen zusätzlich das Geschlecht als Farbe, den Schultyp als Form und das Alter als Punktgröße darstellen.

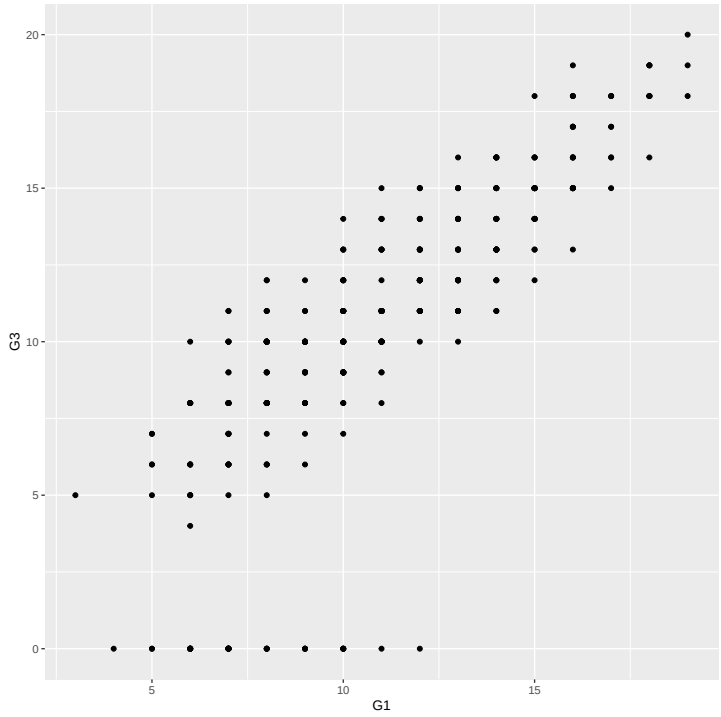


Abb. 16.1: Einfacher Scatterplot.

```
g <- ggplot(student, aes(G1, G3, col=sex, size=age, shape=school))
g <- g + geom_point()
g
```

Ein Linienplot wird mit der Geometriefunktion `geom_line()` erzeugt.

Da wir in unserem Datensatz keine Spalten mit einem linearen Zusammenhang haben, wollen wir die Dichtefunktion der erreichten Punktzahl im 1. Test (Spalte `G1`) darstellen. Wir berechnen zunächst die Dichte mit der Funktion `density()`. Aus dem Ergebnis erzeugen wir einen Data Frame (da das Listenobjekt nicht direkt verwendet werden kann).

```
dens <- density(student$G1)
df.dens <- data.frame(x=dens$x, y=dens$y)
g <- ggplot(df.dens, aes(x, y)) + geom_line()
g
```

Wollen wir das vorherige Erzeugen eines Data Frame vermeiden, können wir als ersten Parameter von `ggplot()` den Wert `NULL` verwenden. In der Funktion `aes()` geben wir nun statt der Spaltennamen Vektoren an:

```
dens <- density(student$G1)
g <- ggplot(NULL, aes(dens$x, dens$y)) + geom_line()
g
```

Bemerkung: Für die Darstellung der Dichte steht eine eigene Geometriefunktion `geom_density()` zur Verfügung. Wir können also auch direkt schreiben:

```
ggplot(student, aes(G1)) + geom_density()
```

Die y-Achse entfällt hier.

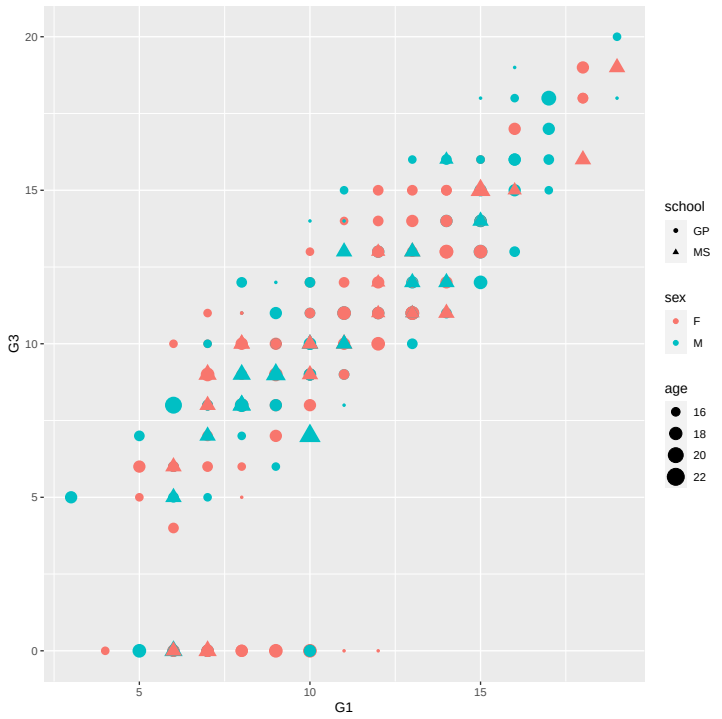


Abb. 16.2: Scatterplot mit zusätzlicher Farb- und Größenskala.

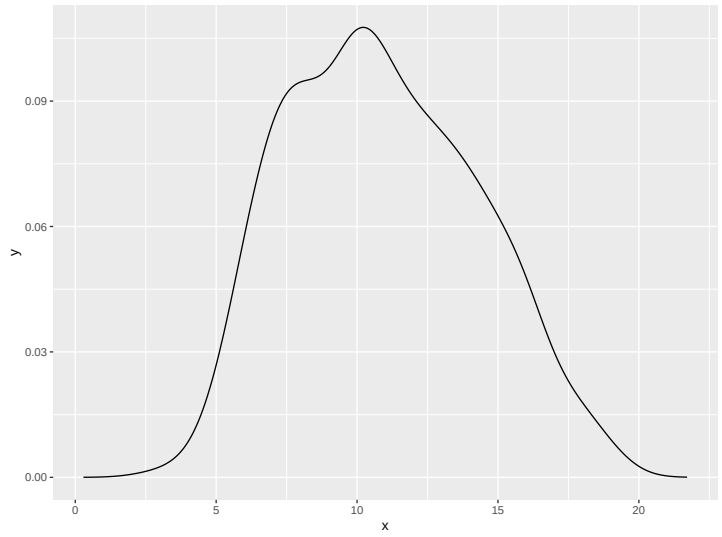


Abb. 16.3: Linienplot.

Wollen wir die Beschriftungen anpassen, fügen wir die Funktion `labs()` hinzu. Sie kennt u. a. folgende Parameter:

`caption`: Bildunterschrift

`title`: Haupttitel

`subtitle`: Untertitel

`x, y`: x- und y-Achse

`col`: Farbskala, falls vorhanden

`shape`: Formskala, falls vorhanden

`size`: Größenskala, falls vorhanden

Beispiel: Anpassung der Beschriftungen für den Scatterplot der Prüfungsergebnisse.

```
g <- ggplot(student, aes(G1, G3, col=sex, size=age, shape=school))
g <- g + geom_point()
g <- g + labs(caption='Scatterplot Mathe-Kurs',
  title='Vergleich Ergebnisse', subtitle='G1-G3',
  x='1. Prüfung', y='2. Prüfung',
  shape='Schulbildung', col='Geschlecht', size='Alter')
g
```

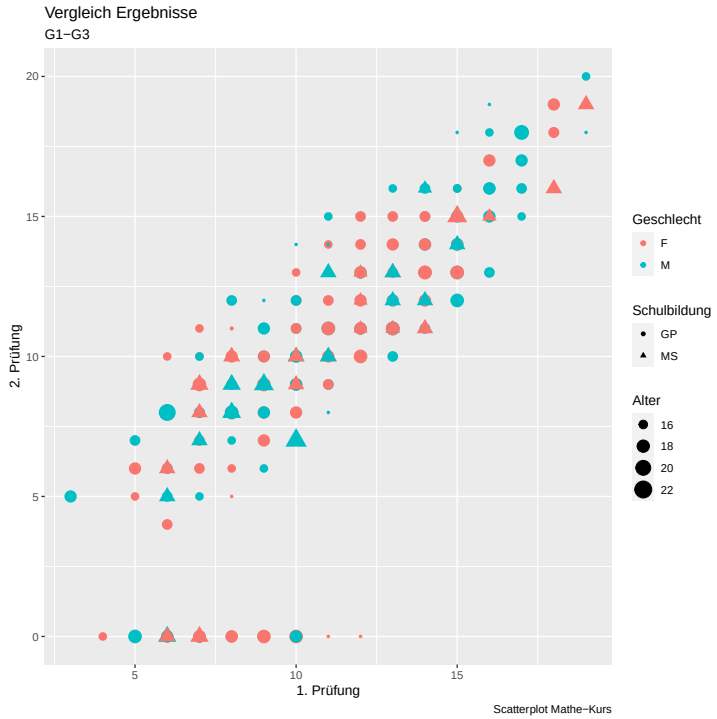


Abb. 16.4: Scatterplot mit angepassten Beschriftungen.

Das Speichern eines Plots geschieht mit der Funktion `ggsave()`:

```
ggsave('dateiName', plotObjekt, width=breite, height=höhe)
```

Der Graphiktyp wird anhand der Dateiendung ermittelt. Es stehen die üblichen Rasterformate wie `PNG` und `JPG` und die Vektorformate `PDF` und `SVG` (bei geladener Bibliothek `svglite`) zur Verfügung.

Notebook

Notebook der Beispiele: `student-base.ipynb`.

16.3 Diagramme für Nominaldaten

Da in den Beispieldaten die Gruppen sehr klein sind, erzeugen wir uns aus der numerischen Altersspalte einen weiteren Faktor:

```
student$f.age <- factor(student$age, levels=min(student$age):max(student$age))
```

Nominaldaten können im wesentlichen nur gezählt werden. In *ggplot2* wird dies durch die Verwendung von eingebauten Statistikfunktionen unterstützt. Einen einfachen Barplot erzeugen wir mit

ggplot(dataFrame, **aes**(*[***x=***]*faktorSpalte)) **+ geom_bar()**

Der y-Wert entfällt hier, da er automatisch von `geom_bar()` erzeugt wird.

Beispiel: Altersstruktur.

```
g <- ggplot(student, aes(f.age)) + geom_bar()
g
```

Ergebnis:

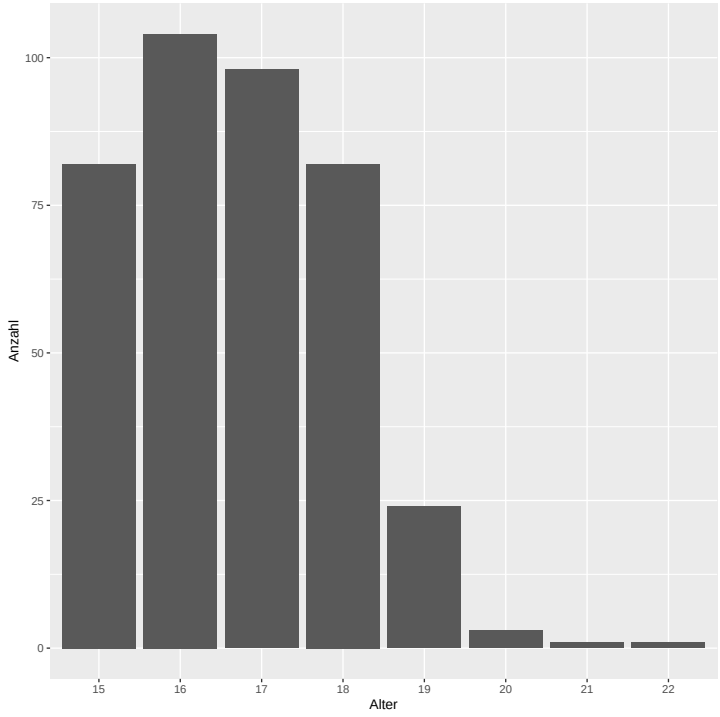


Abb. 16.5: Altersstruktur

Die Funktion `geom_bar()` besitzt einen Parameter `stat`, der den Vorgabewert `count` besitzt. Für das Diagramm werden also automatisch die Werte gruppenweise gezählt.

Sind die Werte bereits summiert, verwenden wir die Statistik `identity`, müssen nun aber den y-Wert spezifizieren.

Beispiel: Erzeugen eines gruppierten Data Frame und Anzeige als Barplot.

```
st.aggr <- aggregate(student[,1], list(student$f.age), FUN=length)
names(st.aggr) <- c('age', 'count')
g <- ggplot(st.aggr, aes(age, count)) + geom_bar(stat='identity')
g
```

Wir erhalten das gleiche Bild. Ähnlich gehen wir vor, wenn wir das Zählen der Gruppen mittels `table()` erledigt haben. Hier haben wir keinen Data Frame vorliegen. Wir erhalten die x-Werte aus den Namen der Tabellenspalten, die Tabellenwerte müssen noch in einen Vektor umgewandelt werden.

```
tab.age <- table(student$f.age)
ggplot(NULL, aes(names(tab.age), c(tab.age))) + geom_bar(stat='identity')
```

Wir können die Säulen auch übereinander stapeln. Wir verwenden für die x-Skala einen leeren String und spezifizieren die Gruppen mittels Aesthetics `fill`.

```
g <- ggplot(student, aes(x='', fill=f.age)) + geom_bar()
g
```

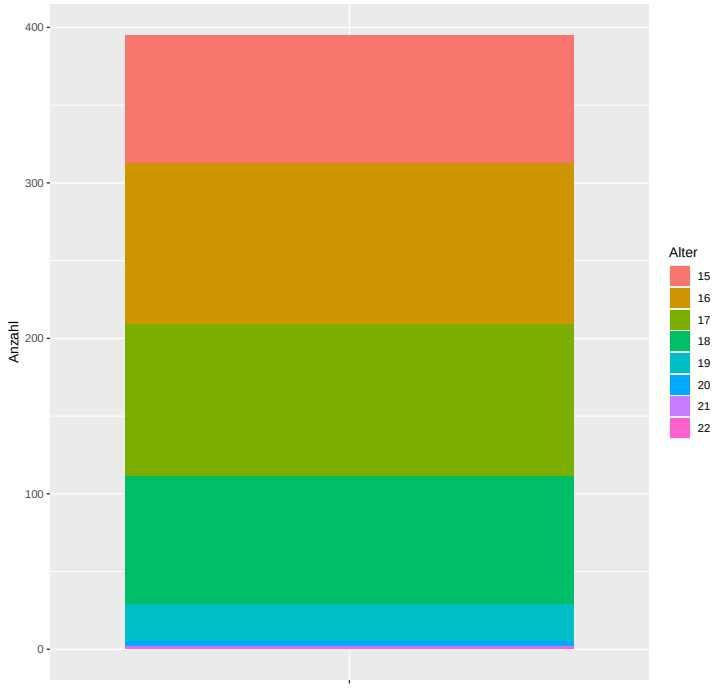


Abb. 16.6: Gruppen gestapelt.

Sind unsere Daten bereits gruppiert, verwenden wir wieder die Statistik `identity` und geben mit dem Parameter `y` die Werte (Gruppensummen) und mit `fill` wieder die Gruppe an.

```
g <- ggplot(st.aggr, aes(x='', y=age, fill=f.age)) + geom_bar(stat='identity')
g
```

Dieses Diagramme können wir nun einfach in ein Tortendiagramm umwandeln, indem wir eine Transformation des Koordinatensystems hinzufügen:

```
g <- g + coord_polar(theta='y')
```

Beispiel:

```
g <- ggplot(student, aes('', fill=f.age)) + geom_bar()
g <- g + coord_polar(theta='y')
g
```

bzw. für bereits gruppierte Daten:

```
g <- ggplot(st.aggr, aes('', y=count, fill=age)) + geom_bar(stat='identity')
g <- g + coord_polar(theta='y')
g
```

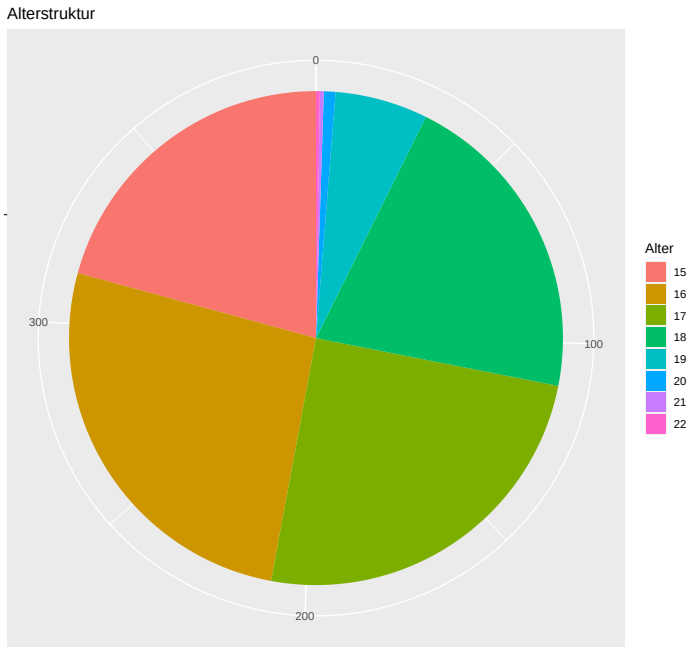


Abb. 16.7: Tortendiagramm.

Notebook

Download: `student-nominal.ipynb`.

16.4 Gruppierte Daten

Wollen wir Nominaldaten nach 2 Merkmalen gruppiert darstellen, können wir die erste Gruppe als Säulen, die zweite als farbige Säulenunterteilung darstellen. Wir verwenden dazu den Aesthetics-Parameter `fill`.

Beispiel: Altersstrukur nach Geschlecht.

```
g <- ggplot(student, aes(f.age, fill=sex)) + geom_bar()
g
```

Oder umgedreht:

```
g <- ggplot(student, aes(sex, fill=f.age)) + geom_bar()
g
```

Um die Verteilung metrischer Daten (Streumaße) zu visualisieren, stehen uns verschiedene Möglichkeiten zur Verfügung.

Ein Histogramm erzeugen wir mit der Funktion `geom_histogram()`.

> **`ggplot(`**dataFrame, **`aes(`***[`x=`]*spalte**`) + geom_histogram(binwidth=`**binBreite**`)`**

Beispiel: Verteilung der Punktzahlen für die letzte Prüfung (Intervallänge 4).

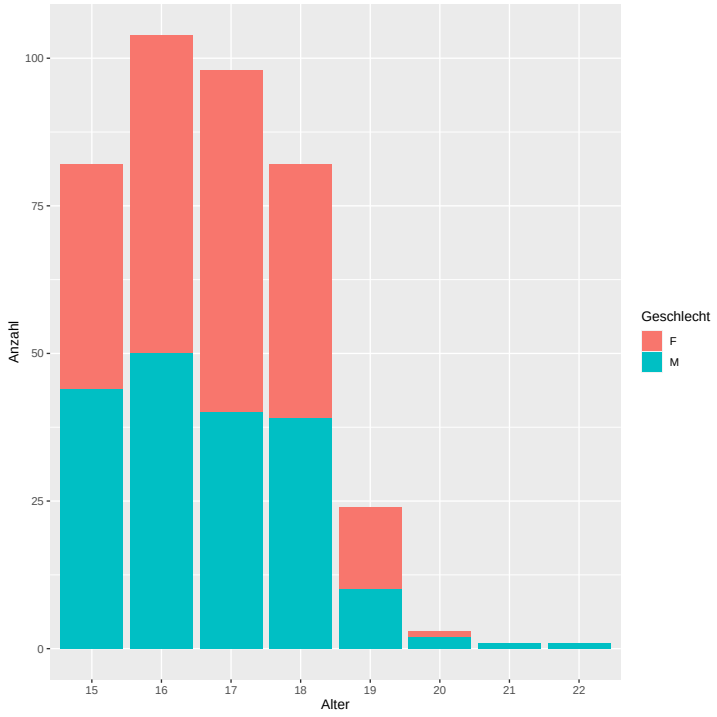


Abb. 16.8: Geschlechterverteilung nach Alter.

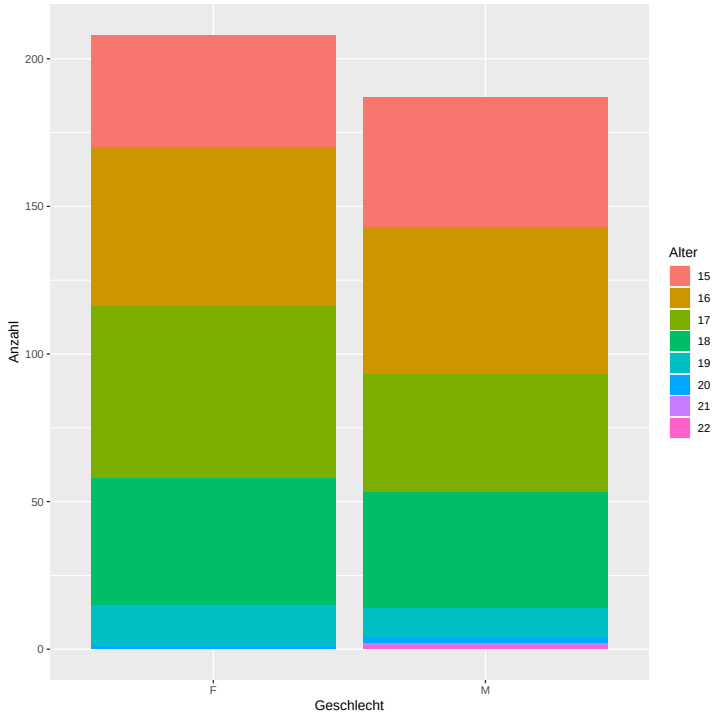


Abb. 16.9: Altersstruktur mit Geschlecht

```
g <- ggplot(student, aes(G3)) + geom_histogram(binwidth=4)
g
```

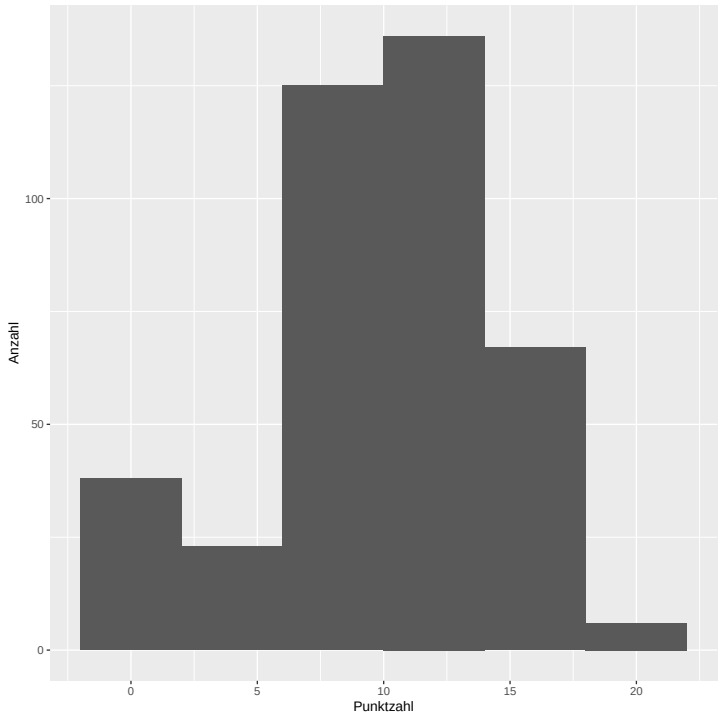


Abb. 16.10: Histogramm Punktverteilung.

Wir können auch dieses Histogramm nach Gruppen aufteilen, indem wir die Gruppe mit dem Aesthetics-Parameter `fill` spezifizieren:

```
g <- ggplot(student, aes(G3, fill=f.age)) + geom_histogram(binwidth=4)
g
```

Alternativ können wir auch einen Dichteplot nach Gruppen anzeigen, indem wir die Gruppe über den Aesthetics-Parameter `col` spezifizieren.

Beispiel: Punktverteilung G3 nach Geschlecht.

```
ggplot(student, aes(x=G3, col=sex)) + geom_density()
```

Warnung: Wenn eine Gruppe weniger als 2 Werte enthält, kann die Dichte nicht korrekt geschätzt werden, wir erhalten Warnungen beim Erstellen des Diagramms. Dies passiert im Beispiel bei der Gruppierung nach dem Alter.

Ein weiterere Darstellungsmöglichkeit für Streumaße ist der Boxplot.

`ggplot(`dataFrame, **`aes(`***`[x=]`*gruppenSpalte, *`[y=]`*wertSpalte**`)) + geom_boxplot()`**

Lassen wir die Gruppierung weg, erhalten wir einen einzelnen Boxplot. Wir können auch die Spalten für x und y vertauschen, um den Boxplot zu drehen.

Beispiel: Punktverteilung G3 nach Alter.

```
g <- ggplot(student, aes(x=f.age, y=G3)) + geom_boxplot()
g
```

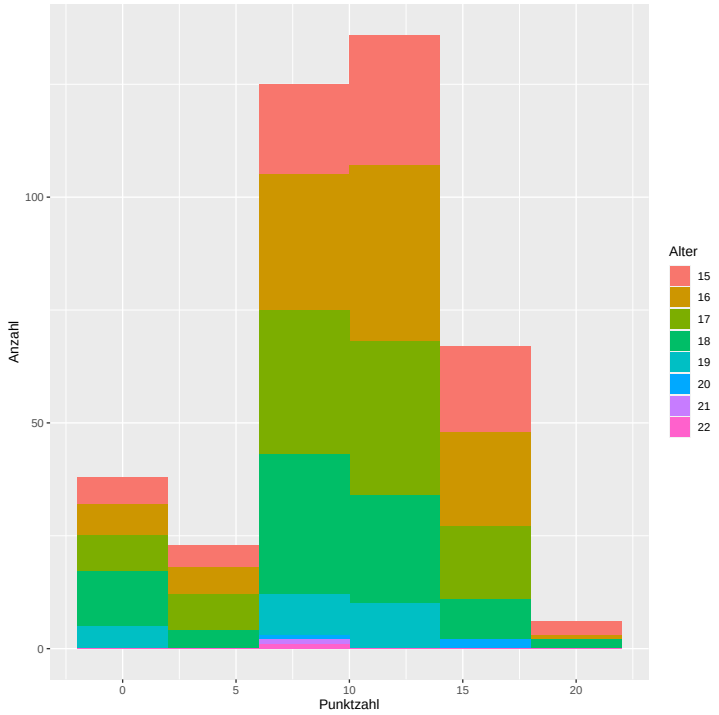


Abb. 16.11: Histogramm Punktverteilung.

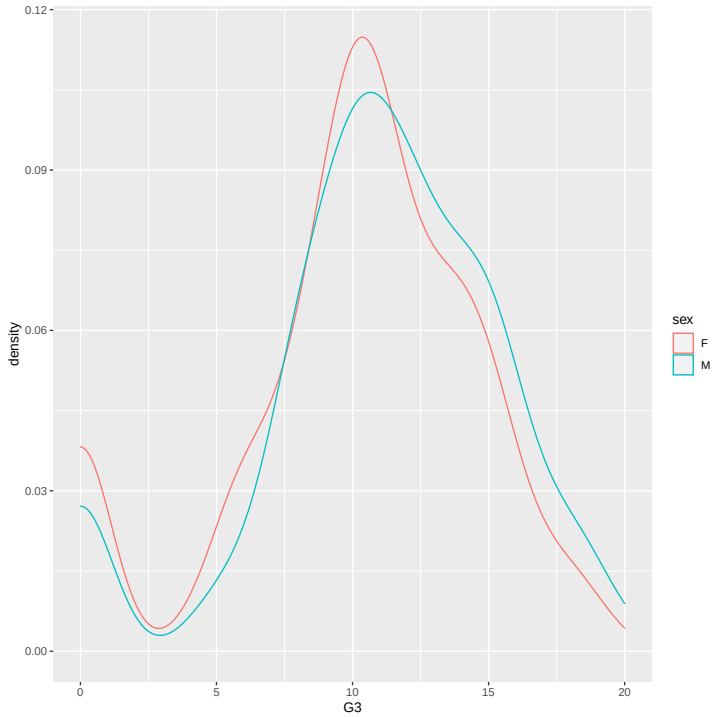


Abb. 16.12: Punktverteilung nach Geschlecht.

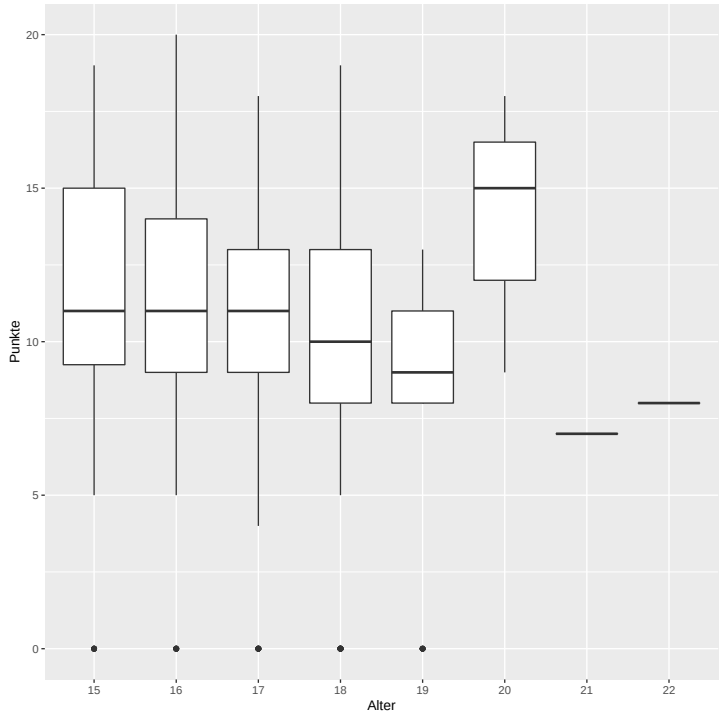


Abb. 16.13: Punktverteilung nach Alter.

Wollen wir die Streuung verschiedener Spalten vergleichen, erzeugen wir am besten einen neuen Data Frame, der eine Spalte für die jeweilige Gruppenbezeichnung und eine zweite Spalte für die zugehörigen Werte enthält. Wir gruppieren dann nach der ersten Spalte und stellen die Werte der zweiten Spalte dar.

Beispiel: Vergleich der Streuungen zwischen den 3 Prüfungen.

```
student.melt <- rbind(data.frame(id=rep('G1', nrow(student)), g=student$G1),
                      data.frame(id=rep('G2', nrow(student)), g=student$G2),
                      data.frame(id=rep('G3', nrow(student)), g=student$G3))
g <- ggplot(student.melt, aes(id, g)) + geom_boxplot()
g
```

Ein Violinplot ermöglicht uns die Kombination aus Box- und Dichteplot. Die Funktion `geom_violin()` wird genauso wie `geom_box()` verwendet, lediglich die Linie für den Median fehlt. Wir können diese einfach mit dem Parameter `draw_quantiles` ergänzen.

`geom_violin(`*[*`draw_quantiles=`*quantilVektor*]*`:kw:')'`

Beispiel: Violinplot für Ergebnisse G3 nach Geschlecht.

```
g <- ggplot(student, aes(x=sex, y=G3))
g <- g + geom_violin(draw_quantiles=c(0.25, 0.5, 0.75))
g
```

Warnung: Auch hier erhalten wir wieder Warnungen, wenn eine der Gruppen weniger als 2 Datenpunkte enthält.

Notebook

Download: `student-group.ipynb`.

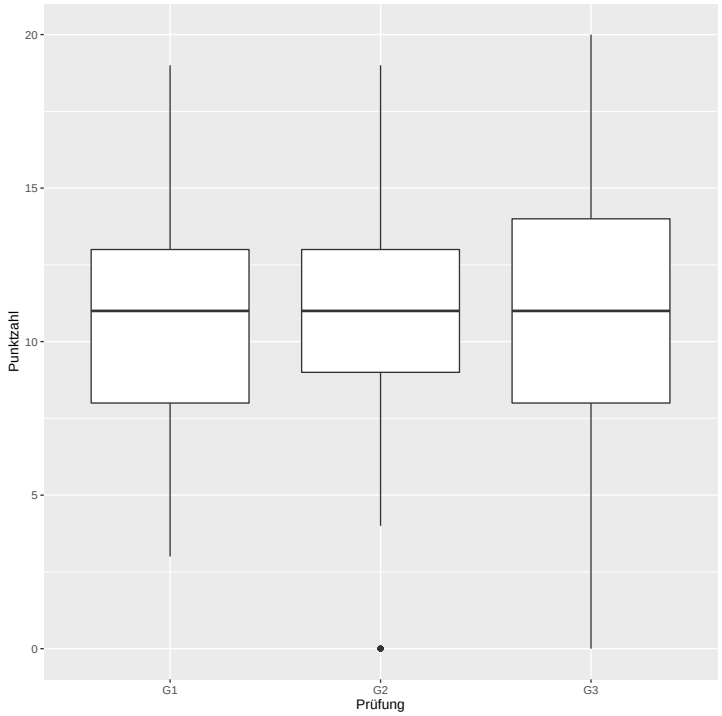


Abb. 16.14: Punktverteilungen der Prüfungen.

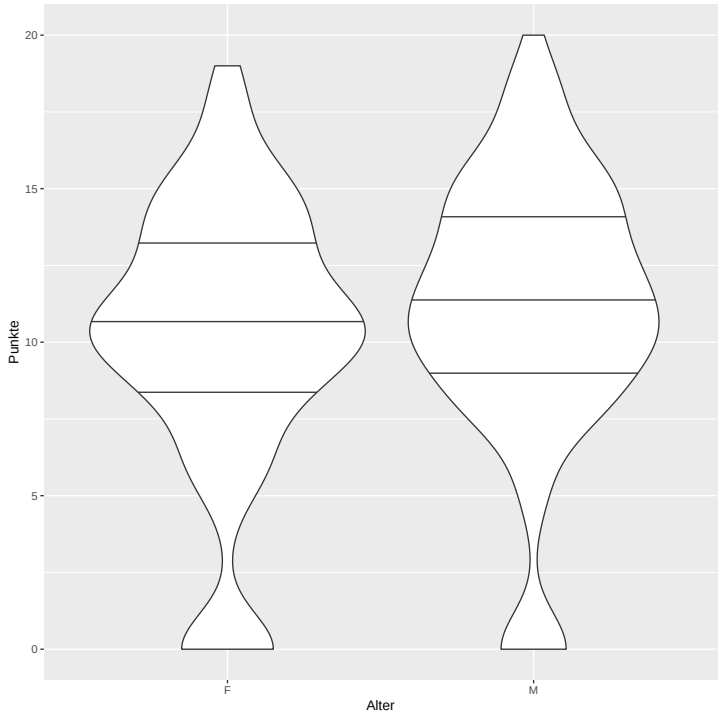


Abb. 16.15: Punktverteilungen Prüfung G3 nach Geschlecht.

16.5 Ebenen und Legende

Wollen wir in einem Diagramm mehrere Werte darstellen, können wir verschiedene Ebenen übereinander lagern, indem wir mehrfach z. B. `geom_line()` aufrufen und mit dem Parameter `aes` verschiedene Spalten für den y-Wert angeben. Sinnvoll ist auch die Angabe des Parameters `col`, dem wir einen Namen für diese Kurve zuordnen, und der dafür sorgt, dass jede Kurve eine verschiedene Farbe erhält.

Beispiel: Anzeige der drei Prüfungsergebnisse in Abhängigkeit vom Alter. Wir ermitteln dazu die minimale, mittlere und maximale Punktzahl für jede Prüfung gruppiert nach dem Alter.

```
age.mean <- aggregate(student[,c('G1', 'G2', 'G3')], by=list(student$age),
  FUN=mean)
names(age.mean) <- c('age', 'G1.mean', 'G2.mean', 'G3.mean')
age.min <- aggregate(student[,c('G1', 'G2', 'G3')], by=list(student$age),
  FUN=min)
names(age.min) <- c('age', 'G1.min', 'G2.min', 'G3.min')
age.max <- aggregate(student[,c('G1', 'G2', 'G3')], by=list(student$age),
  FUN=max)
names(age.max) <- c('age', 'G1.max', 'G2.max', 'G3.max')
stat <- merge(age.mean, age.min)
stat <- merge(stat, age.max)
```

Wir wollen nun die Mittelwerte der Prüfungsergebnisse von `G1`, `G2` und `G3` darstellen. Um diese unterscheiden zu können, ordnen wir ihnen eine Farbe zu.

```
# Nur Data Frame und x-Achse gewählt
g <- ggplot(stat, aes(age))
# G1
g <- g + geom_line(aes(y=G1.mean), col=1)
# G2
g <- g + geom_line(aes(y=G2.mean), col=2)
# G3
g <- g + geom_line(aes(y=G3.mean), col=3)
```

Um für diese Kurven eine Legende zu bekommen, definieren wir die Farbe innerhalb des Parameters `aes` und verwenden einen symbolischen Namen, wie er in der Legende auftauchen soll.

```
# Nur Data Frame und x-Achse gewählt
g <- ggplot(stat, aes(age))
# G1
g <- g + geom_line(aes(y=G1.mean, col='G1'))
# G2
g <- g + geom_line(aes(y=G2.mean, col='G2'))
# G3
g <- g + geom_line(aes(y=G3.mean, col='G3'))
```

Wollen wir einen Bereich wie z. B. die Streuung der Punktzahl in Abhängigkeit vom Alter darstellen, steht uns die Funktion `geom_ribbon()` zur Verfügung, die als Aestetics die Parameter `ymin` und `ymax` erwartet. Die Farbe wird mit dem Parameter *py:data* fill ausgewählt. Steht er innerhalb von `aes()`, geben wir einen symbolischen Namen an, der für die Legende übernommen wird, die Farbe wird automatisch zugeordnet (kann aber angepasst werden). Verwenden wir den Parameter außerhalb, geben wir einen Farbwert als String oder Zahl an.

Beispiel: Streuung Ergebnisse G3 nach Alter.

```
g <- ggplot(stat, aes(age))
g <- g + geom_ribbon(aes(ymin=G3.min, ymax=G3.max), fill='lightgray')
g
```

Hier können wir den Mittelwert, das Minimum und das Maximum als Linien ergänzen.

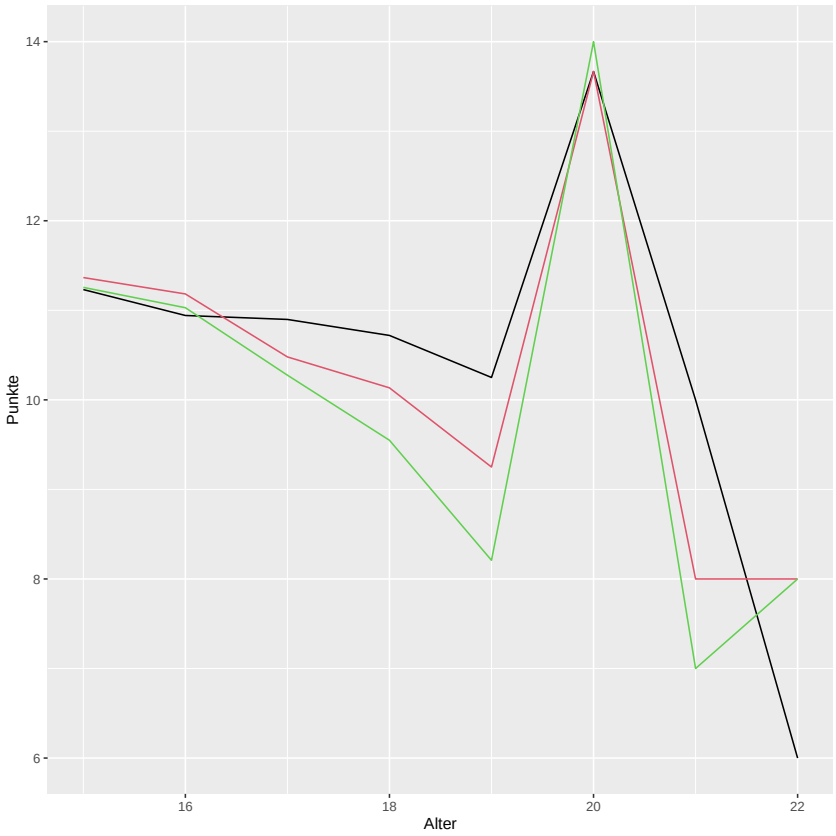


Abb. 16.16: Plot mit 3 Layern

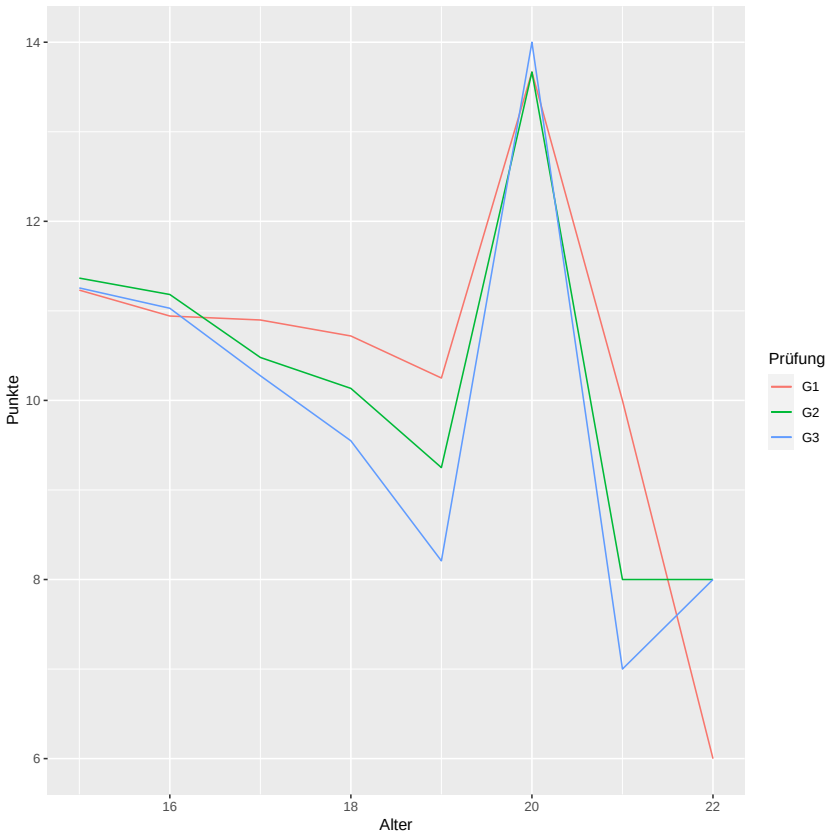


Abb. 16.17: Plot mit 3 Layern und Legende

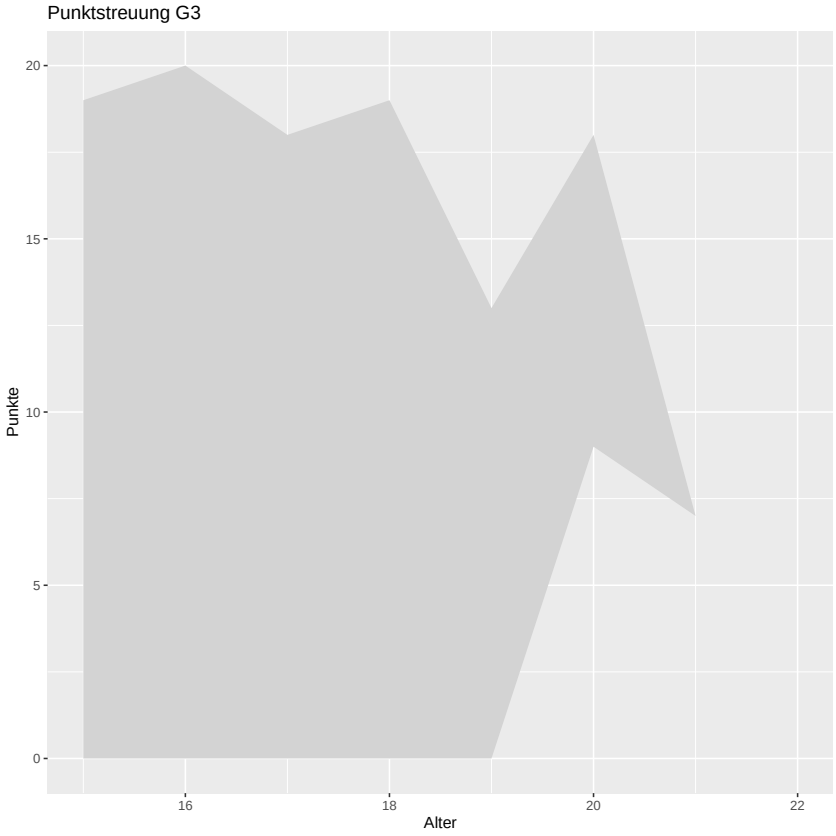


Abb. 16.18: Banddarstellung

```
g <- g + geom_line(aes(y=G3.mean, col='mean'))
g <- g + geom_line(aes(y=G3.min, col='min'))
g <- g + geom_line(aes(y=G3.max, col='max'))
```

Notebook

Download: `student-layers-legend.ipynb`.

16.6 Beschriftung der Daten

Wollen wir innerhalb der Diagrammfläche Beschriftungen ergänzen, verwenden wir die Funktion `geom_text()`, die als Aestetics neben der Position noch den Parameter `label` benötigt. Die Position muss nicht neu gesetzt werden, wenn sie sich auf vorher dargestellte Daten bezieht.

Beispiel: Für einen Scatterplot erzeugen wir wieder einen Data Frame mit der Anzahl der Personen in jeder Altersgruppe.

```
st.aggr <- aggregate(student[,1], list(student$f.age), FUN=length)
names(st.aggr) <- c('age', 'count')

ggplot(st.aggr, aes(age, count)) + geom_point()
```

Wir wollen diese Punkte nun mit der Anzahl der Personen dieses Alters (also dem y-Wert) beschriften.

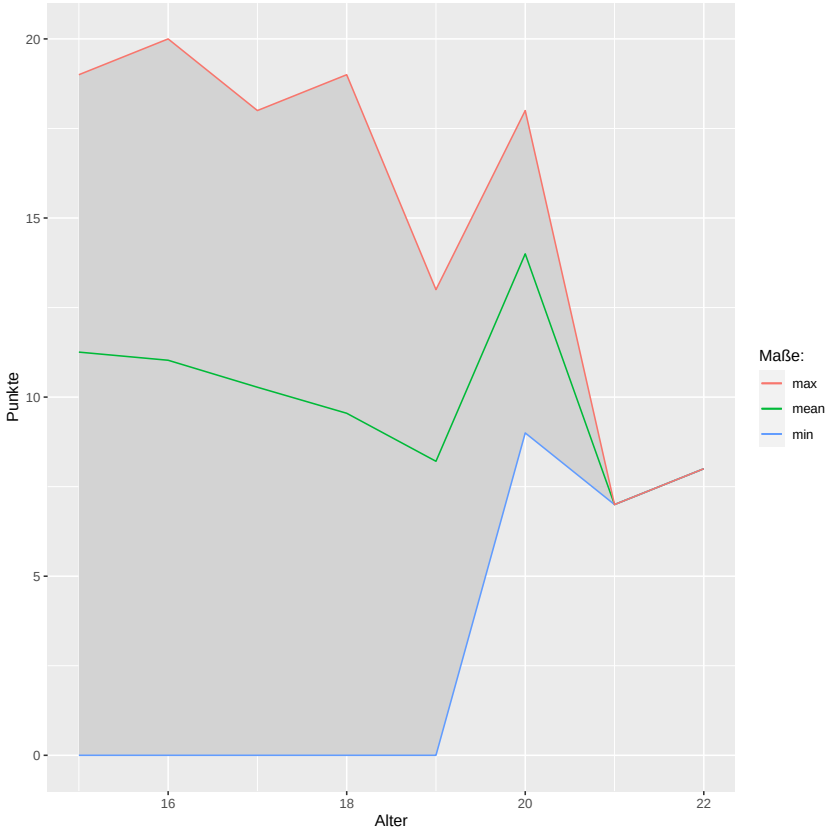


Abb. 16.19: Band mit beschrifteten Begrenzungslinien

```
g <- ggplot(st.aggr, aes(age, count))
g <- g + geom_point()
g <- g + geom_text(aes(label=count))
g
```

Wir sehen, dass die Beschriftung zentriert zur Position ausgegeben wird. Um diese zu verschieben, können wir mit den Parametern `vjust` und `vjust` die Position des Ankerpunktes verschieben, an dem der Text ausgerichtet wird. Dies ist entweder ein String (`top` bzw. `bottom` für `vjust`; `left` bzw. `right` für `hjust`) oder ein numerischer Wert sein (0 = zentriert).

Warnung: Die Parameter definieren den Ankerpunkt. Mit der Angabe von `left` erscheint der Text also *rechts* der Koordinate.

Wird der Text zusätzlich zu einem Symbol an dieser Stelle angegeben, wird er trotzdem noch das Symbol teilweise überdecken. Wir definieren deshalb mit den Parametern `nudge_x` bzw. `nudge_y` einen zusätzlichen Abstand, der in Einheiten der x- bzw. y-Achse angegeben wird. Für Texte unterhalb oder links der Position verwenden wir negative, rechts oder oberhalb positive Werte.

Beispiel: Beschriftung mit Abstand. Mit dem Aesthetic-Parameter `col` erzeugen wir eine Legende. Bei der Beschriftung unter dem Punkt verwenden wir einen neu erzeugten Label-Vektor.

```
g <- ggplot(st.aggr, aes(age, count))
g <- g + geom_point()
g <- g + geom_text(aes(label=count, col='oberhalb'), vjust='bottom',
  nudge_y=2)
g <- g + geom_text(aes(label=paste(age, ':', count),
```

(Fortsetzung auf der nächsten Seite)

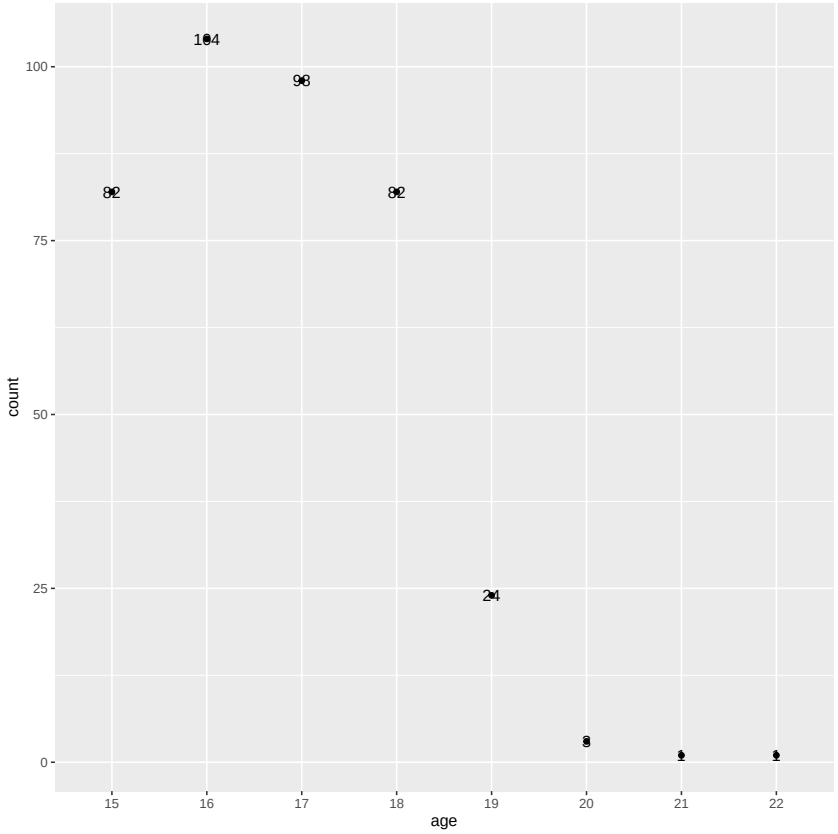


Abb. 16.20: Einfache Beschriftung

(Fortsetzung der vorherigen Seite)

```
  col='unterhalb (age+count)'), vjust='top', nudge_y=-2)
g <- g + geom_text(aes(label=count, col='rechts'), hjust='left', nudge_x=0.1)
g <- g + geom_text(aes(label=count, col='links'), hjust='right', nudge_x=-0.1)
g
```

Ebenso funktioniert die Beschriftung eines Balkendiagramms, das gruppierte Daten darstellt.

```
g <- ggplot(st.aggr, aes(age, count)) + geom_bar(stat='identity')
g <- g + geom_text(aes(label=count), vjust='bottom', nudge_y=1)
g
```

Versuchen wir jedoch, das Balkendiagramm aus den Originaldaten zu erzeugen (wenn `geom_bar` die Zahlung automatisch vornimmt), erhalten wir eine Fehlermeldung, dass die y-Koordinate fehlt. Der Grund ist, dass `geom_text()` diese Statistik nicht automatisch übernimmt. Wir müssen sie explizit mit dem Parameter `stat='count'` angeben. Für den Parameter `label` verwenden wir den Spezialwert `count..` (geben wir `f.age` an, erscheint das Alter, aber nicht die Gruppengröße).

```
g <- ggplot(student, aes(f.age)) + geom_bar()
g <- g + geom_text(aes(label=..count..), stat="count", vjust=-0.5)
```

Für einen gestapelten Barplot erhalten wir allerdings eine falsche Darstellung.

```
g <- ggplot(student, aes(x='', fill=f.age)) + geom_bar()
g <- g + geom_text(aes(label=..count..), stat='count')
```

Der Grund ist, dass sich die berechneten y-Werte immer bezüglich der 0-Achse dargestellt werden. Hier müssen diese jedoch kumulativ angewendet werden. Dies wird mit dem Parameter

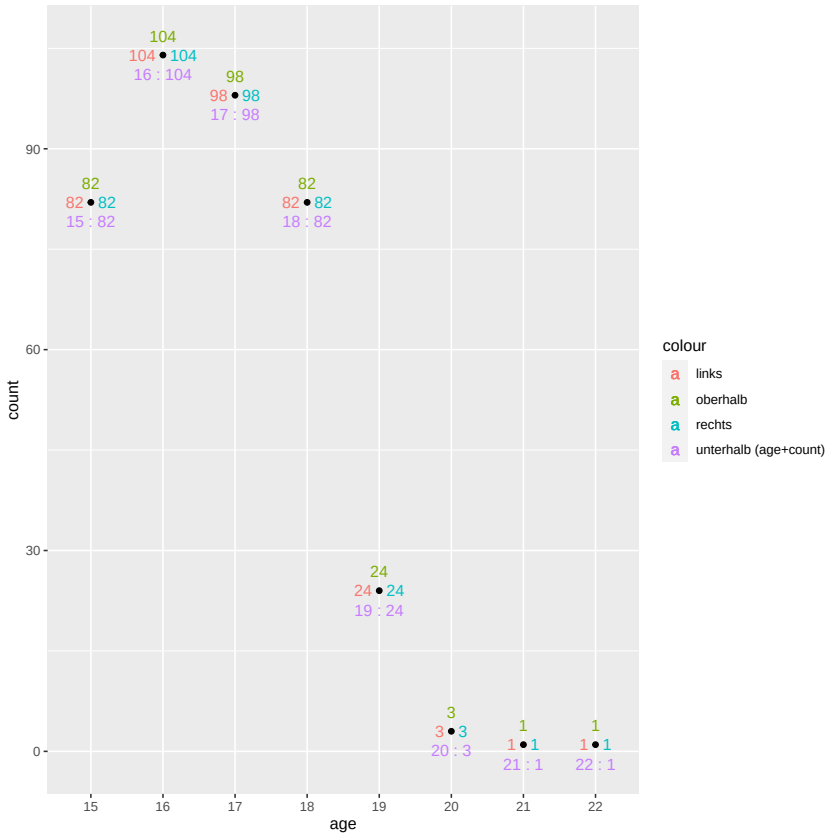


Abb. 16.21: Ausgerichtete Beschriftung

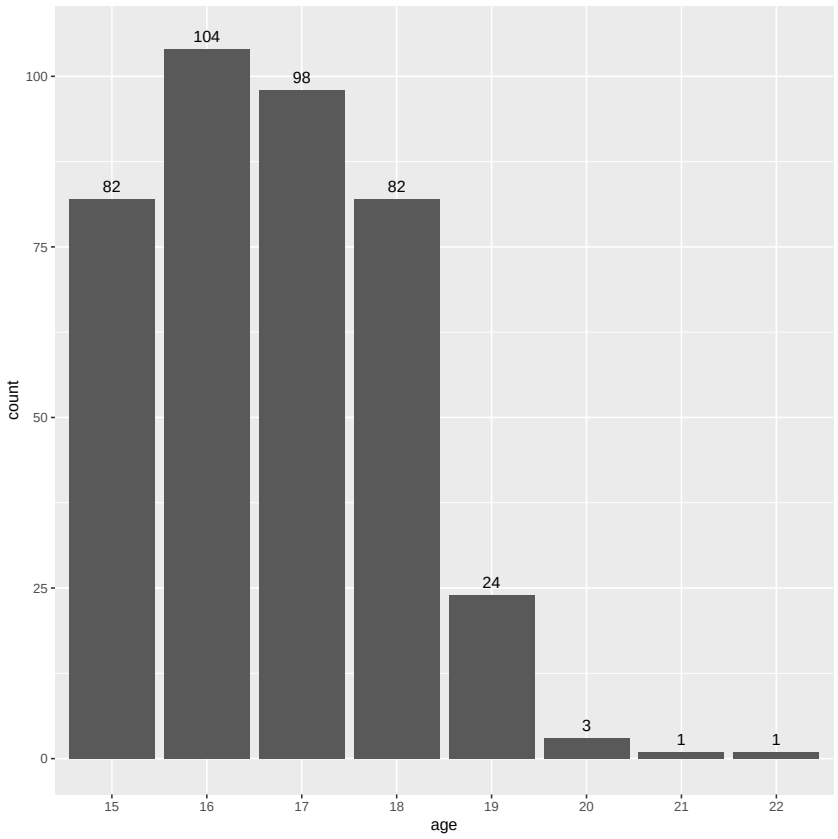


Abb. 16.22: Barplot mit gruppierten Daten

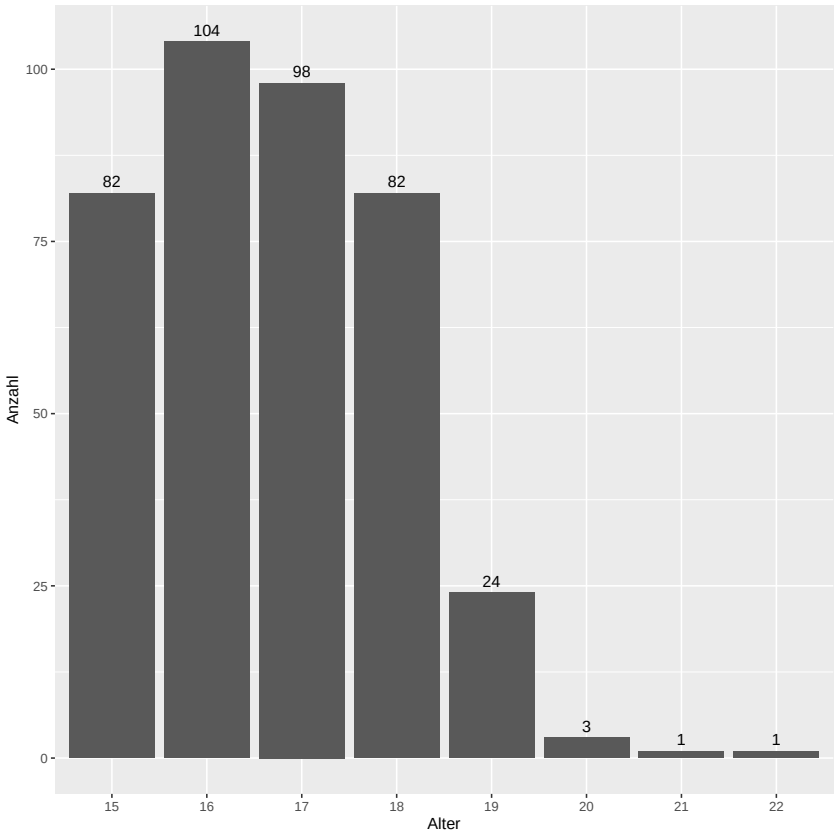


Abb. 16.23: Barplot, der selbst die Daten gruppiert (zählt)

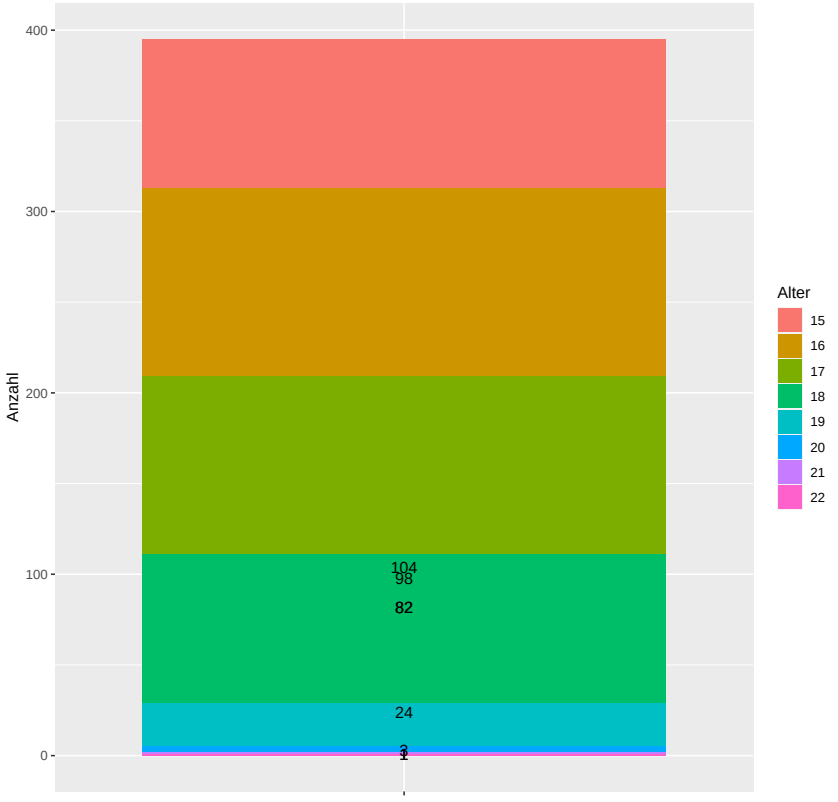


Abb. 16.24: Gestapelter Barplot mit falscher Anordnung der Beschriftung

`position=position_stack()` erreicht. Diese Funktion nimmt einen Parameter `vjust` entgegen, der allerdings numerisch angegeben werden muss. Ein Wert von 0.5 ordnet die Beschriftung in der Mitte, ein Wert nahe 1 oben im Balkenabschnitt an. Mit dem Parameter `check_overlay=T` können wir zusätzlich verhindern, dass sich Beschriftungen überdecken.

```
g <- ggplot(student, aes(x='', fill=f.age)) + geom_bar()
g <- g + geom_text(aes(label=..count..), stat='count',
                   position=position_stack(vjust=0.9), check_overlap=T)
```

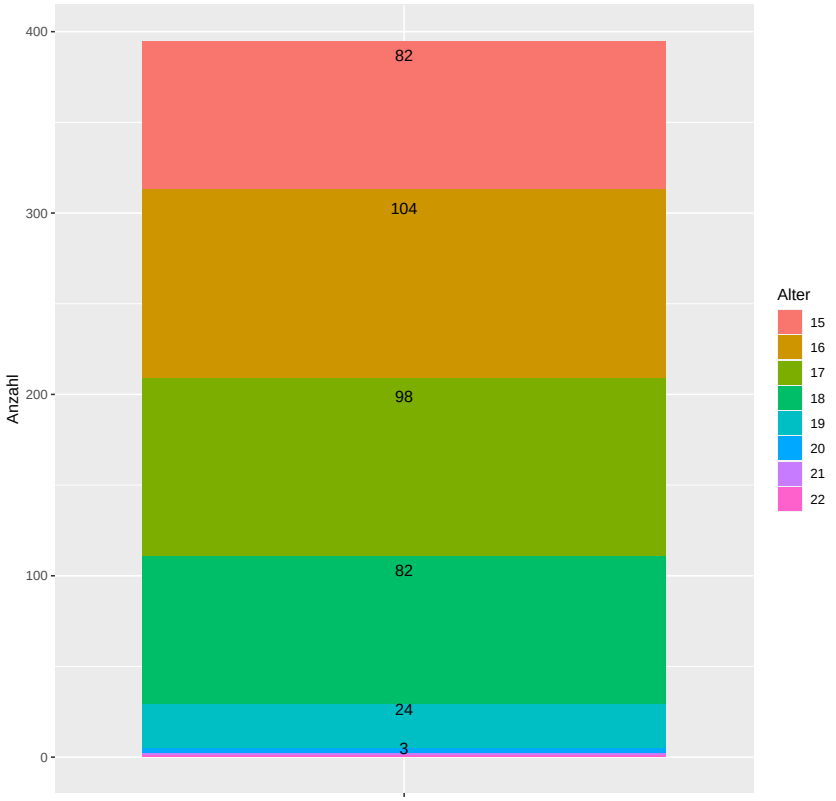


Abb. 16.25: Gestapelter Barplot mit korrekter Anordnung der Beschriftung

Genauso gehen wir bei der Darstellung eines Tortendiagramms vor, das ja ein gestapelter Barplot mit einer Koordinatentransformation ist. Hier sollte die Beschriftung in der Mitte des Tortenstückes stehen, wir verwenden deshalb den Parameter `vjust=0.5`.

```
g <- ggplot(student, aes('', fill=f.age)) + geom_bar()
g <- g + coord_polar(theta='y')
g <- g + geom_text(aes(label=paste('Anz.', ..count..)), stat='count',
                   position=position_stack(vjust=0.5),
                   check_overlap=T)
```

Notebook

Download: `student_text.ipynb`.

Anmerkung

Dieses Kapitel wird sicher noch erweitert.

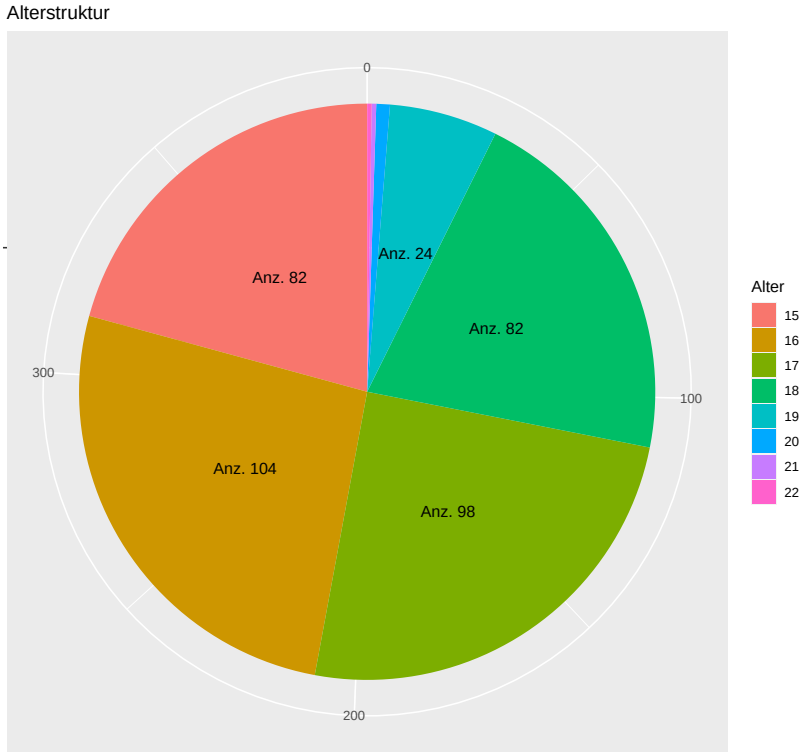


Abb. 16.26: Tortendiagramm mit eigener Beschriftung.

16.7 Aufgaben

KAPITEL 17

Abkürzungen

ADT Abstrakter DatenTyp

ASCII American Standard Code for Information Interchange

CSV Comma Separated Values

IDE Integrated Development Environment

JSON JavaScript Over Network

Jupyter Julia-Python-R

RGB Red-Green-Blue

SQL Structured Query Language

SVG Scalable Vector Graphics

URL Uniform Resource Locator

URZ UniversitätsRechenZentrum der TU Chemnitz

Literaturverzeichnis

[hatz14] Hatzinger, Hornik, Nagel, Maier: *R – Einführung durch angewandte Statistik*. Pearson 2014.

[braun07] Braun, Murdoch: *A First Course in Statistical Programming with R*. Cambridge 2007.

[rrzn11] Hain: *Statistik mit R*. RRZN (jetzt LUIS) 2011.

[adler10] Adler: *R in a Nutshell*. Deutsche Ausgabe. O'Reilly 2010.

[sachs15] Hedderich, Sachs: *Angewandte Statistik*. 15. Auflage. Springer 2015.

[wic09] Wickham: *ggplot2. Elegant Graphics for Data Analysis*. Springer 2009.
Online (Zugang von TU Chemnitz): https://link.springer.com/book/10.1007/978-0-387-98141-3

[r-home] https://cran.r-project.org/ R Home Page (Download von R)

[rstudio] https://www.rstudio.com/home/ RStudio Home Page (Download)

[miniconda] https://docs.conda.io/en/latest/miniconda.html Miniconda

[langref] https://cran.r-project.org/doc/manuals/R-lang.html R Language Definition

[coastal] http://ww2.coastal.edu/kingw/statistics/R-tutorials/ Tutorials by W. B. King, Coastal Carolina University

[oreilly-kurs] https://github.com/rstudio/Intro Course materials for "Introduction to Data Science with R"

[r-tut] http://www.r-tutor.com/ R-Tutorial

[r-stutorial] https://sites.google.com/site/tutorialrstudio/home Einführung in R mit RStudio

[StatJahrbuch] https://www.destatis.de/DE/Publikationen/StatistischesJahrbuch/StatistischesJahrbuch.html Statistisches Bundesamt: *Statistisches Jahrbuch*.

[dtl_in_zahlen] Deutschland in Zahlen.

[unstatistik] http://www.unstatistik.de

[hesse14] Hesse, *Wer falsch rechnet, den bestraft das Leben*. C. H. Beck, 2014.

Stichwortverzeichnis