Kommentierte R-Syntax für Schenderas Datenqualität mit SPSS [1]

Kapitel 3: Vollständigkeit

S. 32

```
missings.row <- apply(datensatz, 1, function(x) sum(is.na(x)))
```

Der Befehl `apply` berechnet mit der Funktion `sum(is.na(x))` die Summe der Missings in jeder Zeile von `datensatz` und speichert sie im Vektor `missing.row`. Würde man im `apply`-Befehl als zweites Argument eine 2 statt einer 1 angeben, würde man die Missings pro Spalte erhalten.

Schenderas Ansatz kann bei großen Datensätzen sehr mühsam werden, weil jede Variable einzeln angegeben werden muss.

COUNT sysmisnum=item1 item2 item3 (SYSMIS).
EXE.

COUNT missing=item1 item2 item3 (MISSING).
EXE.

oder

COMPUTE sysmisnum=SYSMIS(item1) + SYSMIS(item2) + SYSMIS(item3).
EXE.

Für den COUNT-Befehl kann man Abhilfe schaffen, indem man die Formulierung "item1 *to* item3" verwendet. Der COUNT-Befehl funktioniert aber nicht, wenn der Datensatz eine Mischung aus numerischen Variablen und Stringvariablen enthält.

Die deutliche Unterscheidung zwischen System-Missings und nutzerdefinierten Missings ist ein Vorteil von SPSS gegenüber R, wo Missings in der Regel nur durch `NA` gekennzeichnet sind. Für andere Werte als `NA` (z.B. `99`) könnte die R-Syntax so aussehen:

```
apply(datensatz, 1, function(x) sum(x == 99, na.rm = TRUE))
```

3.4 - Seite 33 f.

```
ID <- c(111, 111, 111, 111, 222, 222, 333, 333, 333)
PLZ <- c(NA, 20245, NA, NA, 60598, NA, 81669, NA, NA)
ORT <- c("Hamburg", "Hamburg", NA, "Hamburg", "Frankfurt", "Frankfurt",
"München", "München", "München")
PRODNR <- c(541, 655, NA, 652, 3412, 3221, 65464, 64623, 65435)
datensatz <- data.frame(ID, PLZ, ORT, PRODNR)
```

1 Christian F. G. Schendera, *Datenqualität mit SPSS* (München u.a.: Oldenbourg, 2007), http://deposit.d-nb.de/cgi-bin/dokserv?id=2912651&prov=M&dok_var=1&dok_ext=htm.

Erzeugt die Daten. Im Gegensatz zu Schenderas Syntax werden hier die Daten Variablenweise eingegeben und dann zu einem `data.frame` verknüpft.

Aufgefüllt werden die Postleitlzahlen (`PLZ`) mit der folgenden Syntax:

```
first <- rep(0, nrow(datensatz))
dat <- cbind(datensatz[order(ID,-PLZ),], first)
dat[!duplicated(ID), 5] <- 1
dat$PLZ <- rep(dat[which(dat$first == 1), 2], table(dat$ID))
dat
```

Zunächst wir dabei die Variable `first` erstellt, in der später die Zeilen markiert werden, in denen die jeweilige ID das erste mal auftaucht. Dann wird der Datensatz nach `ID` (aufsteigend) und `PLZ` (absteigend, gekennzeichnet durch das Minuszeichen) sortiert; `first` wird hinzugefügt und das Ganze wird als `dat` abgelegt. Im dritten Schritt wird `first` (5. Spalte im Datensatz `dat`) bei den Zeilen auf 1 gesetzt, in denen eine neue `ID` jeweils das erste mal auftaucht.

Jetzt werden die Postleitzahlen `(dat$PLZ)` aufgefüllt. Im ersten Teil des `rep`-Befehls wird eine Zahl oder Zahlenfolge angegeben, die wiederholt werden soll; im zweiten Teil wird angegeben, wie oft wiederholt werden soll. Mit `dat[which(dat$first == 1), 2]` werden die Postleitzahlen (zweite Spalte in `dat`) aus den Zeilen abgerufen, in denen `first` auf 1 gesetzt ist. `table(dat$ID)` gibt dann an, wie oft jede `ID` auftaucht und wie oft somit die jeweilige Postleitzahl wiederholt werden soll. Schließlich zeigt `dat` das Ergebnis an.

Nicht vergessen, die erstellten Objekt zu löschen:

```
rm(missings.row, mis.row, ID, PLZ, ORT, PRODNR, datensatz, first, dat)
```

Kapitel 4: Einheitlichkeit

4.1 - Seite 41 f.

Datensatz erzeugen (und als `dat` ablegen).

```
dat <- data.frame(c(1,2,3,4,5,6,7),
                  c("positiv", "+", "negativ", "-", "positiv",
                    "psosiiv", "+"),
                  stringsAsFactors = FALSE)
names(dat) <- c("ID", "Befund")
dat
```

Zunächst wird mit `data.frame` ein Datensatz erzeugt, der zwei Variablen für die ID und für die Befundergebnisse enthält. Beachten Sie das Argument `stringsAsFactors = FALSE`, das dafür sorgt, dass Stringvariablen nicht - wie in der Standardeinstellung vorgesehen - automatisch in Faktoren umgewandelt werden, sondern belassen werden wie sie sind.

```
BEFCODE <- rep(999, nrow(dat))
```

```
BEFCODE[dat$Befund == "positiv"] <- 1
BEFCODE[dat$Befund == "+"]       <- 1
BEFCODE[dat$Befund == "negativ"] <- 0
BEFCODE[dat$Befund == "-"]       <- 0
BEFCODE <- as.factor(BEFCODE)
levels(BEFCODE) <- c("negativ", "positiv")
```

Zunächst wird die Variable `BEFCODE` mit der gleichen Länge wir `dat` erstellt und mit "999" gefüllt. Dann werden in vier Einzelschritten "positiv" und "+" mit 1 sowie "negativ" und "-" mit 0 kodiert. Dem SPSS-Befehl IF entsprechen dabei in R die eckigen Klammern, innerhalb derer die Bedingung bestimmt wird, unter der der jeweilige Wert (1 bzw. 2) vergeben wird. Dann wird die Variable mittels `as.factor` in einen Faktor umgewandelt. Das ist die Objektklasse für kategorial skalierte Daten. Schließlich werden mit `levels` Wertelabel für `BEFCODE` gesetzt. In diesem Fall wird dadurch eine Fehlermeldung ausgelöst:

"...number of levels differs." zeigt an, dass die Anzahl der Levels nicht mit der Anzahl der angegebenen Wertelabels übereinstimmt. Woran das liegt, kann mit

```
levels(BEFCODE)
```

überprüft werden. Im Beispiel erhält man die Levels "0", "1", und "999". Man kann jetzt mit

```
dat$Befund[BEFCODE == 999]              # oder für größere Datensätze:
table(dat$Befund[BEFCODE == 999])
```

prüfen, welche weiteren Schreibweisen es gibt um dann die entsprechenden Kodierungen zuzuweisen. Oder man kann die Kodierung belassen wie sie ist und ein drittes Wertelabel hinzufügen:

```
levels(BEFCODE) <- c("negativ", "positiv", "999")
```

Die neue Variable besteht jetzt noch als eigenständiger Vektor und muss dem Datensatz angegliedert werden:

```
ls()
dat <- cbind(dat,BEFCODE)
dat
rm(BEFCODE)
```

`ls` zeigt die im Workspace vorhandenen Objekte an. Mit `cbind` wird `BEFCODE` dem Datensatz angehängt und mit `dat` wird das Ergebnis überprüft. Wenn das funktioniert hat, kann `BEFCODE` (als eigenständiger Vektor) mit `rm` aus dem Workspace gelöscht werden.

4.2 - Seite 43 f.

```
library(foreign)
dat  <- read.spss("C:/IHREDATEN.sav")
dat  <- data.frame(dat, stringsAsFactors = FALSE)
dat2 <- subset(dat, dat$GENDER != "m       " & dat$GENDER != "w       ")
```

Um SPSS-Dateien einzulesen wird hier mit `library` zunächst das Programmpaket `foreign`[2] geladen. Der Befehl `read.spss` liest dann die Daten in das Objekt `dat` ein. Dabei wird zunächst ein Objekt der Klasse "List" erzeugt, und im folgenden Befehl in einen `data.frame` umgewandelt. Dabei ist das Argument `stringsAsFactors = FALSE` zu beachten, damit Stringvariablen nicht automatisch in Faktoren umgewandelt werden. Mit `subset` wird jetzt eine Teilauswahl von `dat` getroffen, die nur Datenzeilen enthält, in denen `GENDER` weder mit "`m `" noch mit "`w `" angegeben ist. (Das Symbol für logisch nicht ist `!=`.) Dabei zeigt sich ein Problem mit dem Import von Stringvariablen aus SPSS mittels `read.spss`:

In SPSS wird eine feste Variablenbreite für jede Variable eingestellt. Standardmäßig ist das für Stringvariablen eine Breite von 8 Zeichen. Von `read.spss` wird das genau so übernommen und dem Stringinhalt werden Leerzeichen angefügt, so dass die vorgegebene Breite aufgefüllt wird. Zum Beispiel werden dem Inhalt "m" sieben Leerzeichen angefügt, wenn die Variablenbreite in SPSS auf acht Zeichen eingestellt war. Man muss diese Leerzeichen dann bei der Teilauswahl berücksichtigen. Speichert man die SPSS-Daten als tabulatorgetrennte Textdatei und importiert sie mit `read.table`, hat man dieses Problem nicht und kann außerdem auf das Programmpaket `foreign` verzichten. Der Nachteil dabei ist aber, dass dann in SPSS vergebene Variablenlabels nicht nach R übernommen werden.

```
dat$GENDER <- sub(" +$", "", dat$GENDER)
```

Mit diesem `sub`-Befehl werden endständige Leerzeichen gelöscht. Das `+` hinter dem Leerzeichen bedeutet, dass ein oder mehrere Zeichen dieser Art gesucht werden, das `$` am Ende des Ausdrucks bedeutet, dass das Leerzeichen am Ende des Strings gesucht wird. (`?regexp` gibt weitere Informationen zu regulären Ausdrücken, die im Umgang mit Strings verwendet werden können. Wenn Sie mit Stringvariablen arbeiten, lohnt es sich unbedingt, sich mit diesen Ausdrücken zu befassen (KASTEN?)) Jetzt funktioniert `subset` auch in der folgenden Ausführung:

```
subset(dat, dat$GENDER != "m" & dat$GENDER != "w")
```

Die folgende Syntax gibt das gleiche Ergebnis, ist aber handlicher, wenn viele Bedingungen aneinander gehängt werden:

```
dat[-grep("^(m|w)$", dat$GENDER),]
```

Der Befehl `grep` sucht mit "`(m|w)`" nach Zeilen, in denen `dat$GENDER` entweder "m" oder "w" enthält. Damit nur Zeilen gesucht werden, die ausschließlich "m" oder "w" enthalten und nicht auch solche die z.B. "*m*ännlich" oder "be*w*usst" enthalten, wird mit `^` sicher gestellt, dass der jeweilige Buchstabe am Anfang des Strings steht und mit `$`, dass er auch am Ende des Strings steht. Da `grep` (vor dem Komma) in den eckigen Klammern von `dat[...,]` steht, werden die Datenzeilen ausgegeben, in denen `grep` fündig wurde. Da wir aber an Datenzeilen interessiert sind, die eben nicht "m" oder "w" enthalten, steht ein Minuszeichen vor `grep`. Die folgende Befehlszeile funktioniert nach dem gleichen Prinzip:

---

2 R-core members u. a., *foreign: Read Data Stored by Minitab, S, SAS, SPSS, Stata, Systat, dBase, ...*, 2010, http://CRAN.R-project.org/package=foreign.

```
dat[-grep("^(ISEF|JIOJ|KERK|LPMS|MEDE|OJFW|OWJG|P2PC|PMKO|STST|TEKL|UHNS|
WERE)$", dat$statusg),]
```

Die multivariate Variante des Befehls erfordert ein etwas kompliziertes Vorgehen als das aneinander Ketten von zwei `grep`-Befehlen:

```
dat[-which(grepl("^(ISEF|JIOJ|KERK|LPMS|MEDE|OJFW|OWJG|P2PC|PMKO|STST|
TEKL|UHNS|WERE)$", dat$statusg) | grepl("^(m|w)$", dat$GENDER)),]
```

Zunächst einmal wird `grepl` an Stelle von `grep` verwendet. Der Unterschied besteht darin, dass `grep` einen Vektor erzeugt, der als Elemente die Zeilennummern enthält, in denen der Befehl fündig wurde, `grepl` dagegen erzeugt einen logischen Vektor, in dem jede Zeile mit `TRUE` bzw. `FALSE` gekennzeichnet wird, je nach dem ob das Stringmuster gefunden wird oder nicht. Die Länge (Anzahl der Elemente) von grep ist also abhängig davon, in wie vielen Zeilen das Muster gefunden wird, die Länge von `grepl` ist immer gleich der durchsuchten Datenzeilen. Wenn nun zwei `grep`-Befehle mit einander verknüpft werden (z.B mit `&`), müssen sie die gleiche Länge aufweisen und das ist nur mit der Variante `grepl` gewährleistet. Da `grepl` aber eine Abfolge von TRUE-FALSE zurück gibt und nicht die Zeilennummern, müssen diese über `which` abgefragt werden. Weil Datenzeilen ausgewählt werden sollen, die die entsprechenden Textmuster *nicht* enthalten, geht `which` ein Minuszeichen voraus. Beachten Sie, dass durch dieses Invertieren logisch UND (`&`) durch logisch ODER (`|`) ersetzt werden muss: NICHT A & NICHT B entspricht NICHT (A | B). Mit der folgenden Syntax können Sie es sich vergegenwärtigen:

```
daten <- data.frame(c("ja", "ja", "nein", "nein"), c("ja", "nein", "ja",
"nein"))
names(daten) <- c("A", "B")
daten

daten[which(grepl("ja", daten$A) | grepl("ja", daten$B)),]
daten[-which(grepl("ja", daten$A) | grepl("ja", daten$B)),]
daten[which(grepl("ja", daten$A) & grepl("ja", daten$B)),]
daten[-which(grepl("ja", daten$A) & grepl("ja", daten$B)),]

rm(daten)
```

Der Lösungsansatz mit den Wertelabels ist mit R nicht unmittelbar nachvollziehbar, weil Wertelabels hier nur für Vektoren der Klasse *factor* definiert werden können.

4.3 - Seite 47 f.

```
(TESTWORT <- "HELLO")
(Test_RP1 <- sub("L", "y", TESTWORT))
```

Der Befehl `sub` erledigt das Ersetzen und ablegen in `Test_RP1` in einem Schritt. (Die Klammern, die den gesamten Befehl einfassen, bewirken, dass das Ergebnis jeweils in der Konsole angezeigt wird.)

Die Variante `sub` ersetzt das "L" jeweils nur bei seinem ersten Auftreten im String. Das

Ergebnis ist "HEyLO". Würde man die Variante `gsub` verwenenden, würde das "L" bei jedem Auftreten im String ersetzt. Das Ergebnis wäre "HEyyO".

```
(TESTWORT <- "HEyLO")
(Test_RP2 <- gsub("L", "y", TESTWORT))

(TESTWORT <- "HEL:LO")
(Test_RP3 <- gsub(":", "-", TESTWORT))

(TESTWORT <- "    ")
(Test_RP4 <- gsub("    ", "_BLANK!_", TESTWORT))
```

Im Gegensatz zu SPSS legt R keine Stringlänge für Objekte der Klasse *character* fest. Der Wert ist jeweils genau so lang wie bei der Eingabe angegeben. Deshalb spielt es keine Rolle, ob ein String oder ein zu ersetzendes Muster kürzer oder länger als 8 Zeichen ist.

```
(TESTWORT <- "12345678901234567890123456789012345678901234567890")
(Test_RP5 <- gsub("12345678", "---", TESTWORT))

TESTWORT <- "12345678901234567890123456789012345678901234567890"
Test_RP6 <- TESTWORT
for(i in 1:2){
Test_RP6 <- sub("1234567890", "eins bis zehn ", Test_RP6)}
Test_RP6
```

Wenn allerdings - wie im Letzten Beispiel - das Stringmuster jeweils in einer bestimmten Anzahl (z.B. zwei mal) ersetzt werden soll, muss man die oben angegebene Variante mit einer Schleife anwenden. Zunächst wird dabei der Inhalt von `TESTWORT` in die neue Variable `Test_RP6` kopiert. Dann wir mit `for` eine Schleife angelegt. `i in 1:2` gibt dabei an, dass der Befehl in den geschweiften Klammern zwei mal ausgeführt wird. Im ersten Durchlauf wird die erste Folge `"1234567890"` durch `"eins bis zehn "` ersetzt und im zweiten Durchlauf die zweite Folge dieses Musters.

4.4 - Seite 47 ff.

```
dat <- data.frame(c(1,2,3,4,5,6,7,8,9,10),
                  c("Müller-Thurgau", "  Müller-Thurgau",
                    "Mueller-Thurgau", "Müller Thrugau",
                    "MÜLLER THURGAU", "Riesling", "RIESLING",
                    "Silvaner", "SILVANER", "Silvahner"),
                  stringsAsFactors = FALSE)
names(dat) <- c("ID", "Wein")
dat
```

Zunächst wird mit `data.frame` ein Datensatz erstellt und als `dat` gespeichert. Dann werden mit `names` die Variablennamen vergeben.

Für die Suche nach (und das Ersetzen von) ähnlichen Schreibweisen hat R die komfortable Funktion `agrep`. Damit gelingt das Vereinheitlichen der Schreibweisen im Beispiel mit jeweils nur einem Befehl pro Weinsorte.

```
dat$Wein[agrep("Müller-Thurgau", dat$Wein, ignore.case = TRUE)] <-
"Müller-Thurgau"
dat$Wein[agrep("Riesling", dat$Wein, ignore.case = TRUE)] <- "Riesling"
dat$Wein[agrep("Silvaner", dat$Wein, ignore.case = TRUE)] <- "Silvaner"
dat
```

Im ersten Befehl sucht `agrep` in `dat$Wein` nach dem Muster "Müller-Thurgau" oder ähnlichen Schreibweisen. Das Argument `ignore.case = TRUE` bewirkt, das Groß-/Kleinschreibung ignoriert wird. Im Beispiel funktioniert die Identifikation von ähnlichen Schreibweisen schon mit der Standardeinstellung sehr gut. Wenn das nicht der Fall ist, kann man mit dem Argument `max` genauer definieren, wie viele Auslassungen, Ergänzungen oder Ersetzungen man als ähnlich gelten lässt.

Das Ergebnis von `agrep` ist eine Folge von Index- bzw. Zeilennummern, in denen ein ähnliches Muster gefunden wurde. Dadurch, das der Befehl mit `dat$Wein[agrep(...)]` in eckige Klammern gesetzt wird, werden genau die Einträge von `dat$Wein` ausgewählt, die "Müller-Thurgau" ähneln. Mit `<- "Müller-Thurgau"` wird ihnen dann ein einheitlicher Eintrag zugewiesen. Die beiden folgenden Befehle funktionieren analog für "Riesling" und "Silvaner".

Darüber hinaus ist es auch möglich die bei Schendera beschriebenen Teilfunktionen in R abzubilden. Die Argumente `"^ +", ""` in `sub` bewirken das Löschen vorausgehender Leerzeiten. Das `^` bedeutet dabei, dass nur am Anfang des Strings gesucht wird und das `+`, dass ein oder auch mehrere Leerzeichen gesucht werden.

```
sub("^ +", "", "    vorangestellte Leerzeichen löschen")
```

Die Umwandlung in Groß- bzw. Kleinbuchstaben lässt sich durch `toupper` und `tolower` erreichen.

```
toupper("wandelt den String in Großbuchstaben um")
tolower("WANDELT DEN STRING IN KLEINBUCHSTABEN UM")
```

4.5 - Seite 51 f.

```
dat <- data.frame(c(1,2,3,4,5),
                  c("$100000", "$10000", "$1000", "$100", "$10"),
                  stringsAsFactors = FALSE)
names(dat) <- c("ID", "PREISE")
dat

dat$PREISE2 <- sub("[$]", "€", dat$PREISE)
```

Die Umwandlung des Dollarzeichens in das Eurozeichen wird wieder mit `sub` bewerkstelligt. Beachten Sie, dass `$` hier in eckige Klammern gesetzt wird, damit es wörtlich interpretiert wird und nicht als Metazeichen (Stelle am Ende des Strings).

```
PARAGRAF <- c(123, 456, 789)
PARAMIT <- paste("$", PARAGRAF, sep = "")
data.frame(PARAGRAF, PARAMIT)
```

Das Hinzufügen des Dollarzeichens wird hier mit dem Befehl `paste` bewirkt. Dabei bedeutet `sep = ""`, dass die Elemente direkt aneinander gehängt werden. Die Standardeinstellung (`sep = " "`) fügt ein Leerzeichen zwischen die Elemente. Die Umwandlung von numerischer zu Stringvariablen geschieht automatisch.

4.6 - Seite 52 f.

```
dat <- data.frame(c(1998, 1999, 2000, 2001, 2002, 2003, 2004),
                  c(1, 10, 100, 1000, 5000, 10000, 15000))
names(dat) <- c("JAHR", "PREIS")

dat$EPREIS <- dat$PREIS
dat$EPREIS[dat$JAHR < 2002] <- round(dat$PREIS[dat$JAHR < 2002]
                                     / 1.95583,
                                     2)
```

Nachdem der Datensatz erzeugt wurde, wird `EPREIS` als Kopie von `PREIS` erstellt. Die Auswahl der Variablen, die umgerechnet werden erfolgt über die eckigen Klammern (`[dat$JAHR < 2002]`). Diese Auswahl taucht dann auch wieder in der eigentlichen Berechnung `dat$PREIS[...] / 1.95583` auf. Das Ganze wird mit `round(..., 2)` auf zwei Nachkommastellen gerundet.

4.7 – Seite 53 ff.

```
dat <- data.frame(ID = c(1,2,3,4,5),
                  KONZERN = c("Daimler-Benz", "Mercedes-Benz",
                              "Daimler-Chrysler", "MB", "Benz-Daimler"))
KONZERN2 <- as.character(dat$KONZERN)
KONZERN2[grep("Daimler|Mercedes", dat$KONZERN)] <- "Daimler-Chrysler"
data.frame(dat, KONZERN2)
```

Im obigen und im folgenden Beispiel werden mit `grep` Einträge identifiziert, die eines der gesuchten Muster enthalten und dann jeweils vollständig durch einen einheitlichen Eintrag ersetzt. Die oder-Funktion „|“ ermögliche es dabei, innerhalb eines Befehls nach mehreren verschiedenen Mustern zu suchen.

```
KONZERN <-  c("IBM", "Industrial Business Machines", "IBM Ltd.",
              "Industrial Business Machines International", "MB",
              "Daimler-Benz", "DaimlerChrysler", "Mercedes-Benz",
              "Daimler-Chrysler")
AKRONYM <- KONZERN
AKRONYM[grep("Industrial Business Machines|IBM", KONZERN)] <- "IBM"
AKRONYM[grep("MB|Daimler|Benz|Mercedes", KONZERN)] <- "DC"
data.frame(KONZERN, AKRONYM)
```

4.8 – Seite 55 f.

In diesem Beispiel werden die abschließenden Nullen durch einen `gsub`-Befehl gelöscht. Der Zusatz „`+`“ hinter „`0`“ sorgt dafür, dass eine oder mehrere Nullen erstzt werden. der Zusatz „`$`“ sorgt dafür, dass nur Nullen am rechten Ende des Stings gelöscht werden.

```
STRING <- c("STRINGA", "STRINGB000", "STRINGC0000", "STRINGD00000")
STRING2 <- gsub("0+$", "", STRING)
data.frame(STRING, STRING2)
```

4.9.1 – Seite 56 f.

```
dat <- data.frame(ID = c(1,2,3,4,5),
                  KONZERN = c("Daimler", "Benz-Daimler",
                              "Daimler-Chrysler", "MB-Daimler",
```

```
                                  " Daimler-Benz"))
ZAEHLER <- rep(NA, 5)
ZAEHLER[grep("Daimler|Chrysler|MB", dat$KONZERN)] <- 1
data.frame(dat, ZAEHLER)
```

4.9.2 – Seite 57 f.

```
dat <- data.frame(ID = c(1,2,3,3,4,5,6),
                  KONZERN = c("Daimler", "Benz-Daimler", "Mercedes-Benz",
                              "Daimler-Chrysler", "MB-Daimler", "
DAIMLER-Benz", "    Benz"))
ZAEHLER <- rep(NA, 7)
ZAEHLER[grep("Daimler|Merceds|Benz|Chrysler", dat$KONZERN, ignore.case =
TRUE)] <- 1
data.frame(dat, ZAEHLER)
```

Der Unterschied zum vorherigen Beispiel besteht darin, dass dem `grep`-Befehl das Argument `ignore.case = TRUE` angefügt ist. Damit wird die Groß-/Kleinschreibung ignoriert.

4.10.1 – Seite 59

```
TNUMMER <- c("(491) 234-567", "(+4912) 3-4567", "+49/123-4567",
             "+49+123-4567 PRIV", "49 123 4567", "49-1234567",
             "+49 123-4567", "+491234567")
TNALT <- TNUMMER
TNUMMER <- gsub("[[:space:]]|[[:punct:]]|[[:alpha:]]", "", TNUMMER)
data.frame(TNALT, TNUMMER)
```

Die Telefonnummern werden in diesem Beispiel mit `gsub` vereinheitlicht. Dabei identifiziert `[[:space:]]` alle Leerzeichen (und ggf. auch Tabulatoren), `[[:punct:]]` alle Interpunktionszeichen (einschließlich Klammern, Plus- und Minuszeichen) und `[[:alpha:]]` alle Buchstaben.

4.10.2 – Seite 60 f.

```
TNALT <- c("(+49) 123-4567", "(+49)123-4567", "+49/123-4567",
           "123-4567 PRIV", "123-4567/8976", "123 4567", "1234567",
           "+49 123-4567", "491234567")
TNUMMER <- TNALT
TNUMMER <- gsub("49|/....$|[[:space:]]|[[:punct:]]|[[:alpha:]]", "",
                TNUMMER)
TNUMMER <- paste(substr(TNUMMER,1,3), "-", substr(TNUMMER,4,7), sep = "")
data.frame(TNALT, TNUMMER)
```

Auch in diesem Beispiel werden mit `gsub` zunächst alle überflüssigen Zeichen gelöscht. Zu beachten ist die Identifikation der alternativen Durchwahl in Zeile 5. Durch `/....$` wird hier jede Zeichenfolge mit einem „/“ an fünftletzter Stelle ausgewählt. In einem zweiten Befehl wird mit Hilfe von `paste` und `substr` der Bindestrich an die vierte Stelle eingefügt.

4.11.1 – Seite 63 f.

```
MIXDATEN <- c("1/7/99", "12/11/98", "1/12/01", "10/1/98", "12-8-2002",
              "10-7-00", "1.12.99")
DATEN2 <- gsub("[[:punct:]]", "/", MIXDATEN)
DATEUS <- strptime(DATEN2, "%d/%m/%y")
DATEUS[grep("/[[:digit:]]{4}$", DATEN2)]
       <- strptime(DATEN2[grep("/[[:digit:]]{4}$", DATEN2)], "%d/%m/%Y")
DATEEU <- strptime(DATEUS, "%Y-%d-%m")
data.frame(MIXDATEN, DATEUS, DATEEU)
```

In diesem Beispiel wird mit `strptime` eine Variable im Datum-Zeit-Format erzeugt. Zunächst werden mit `gsub` die Trennzeichen zwischen Tagen, Monaten und Jahren vereinheitlicht. Dann erfolgt die Umwandlung mit `strptime`. Das Argument `%d/%m/%y` gibt an, dass der String nach dem Muster Tag/Monat/Jahr interpretiert wird.

Ein besonderes Problem ergibt sich daraus, dass das Jahr in Zeile fünf abweichend von den anderen Zeilen vier- und nicht zweistellig angegeben wird. Diese Zeile wird mit `grep("/[[:digit:]]{4}$", DATEN2)` identifiziert und mit dem Format `%d/%m/%Y` umgewandelt. (Für zeistellige Jahreszahlen wird klein-y angegebn, für vierstellige Jahreszahlen groß-Y.)

In der Ausgabe werden Datumsvariablen in R (abweichend von SPSS) im Format Jahr(vierstellig)-Monat-Tag angezeigt.

Um die Daten im europäischen Format (Tag-Monat-Jahr) zu interpretieren, wird die Reihenfolge von Tagen und Monaten mit `strptime(DATEUS, "%Y-%d-%m")` noch einmal vertauscht. (Eigentlich muss es genau anders herum gemacht werden. Die Variable DATEEU ist eine amerikanische und die Variable DATEUS eine europäische Interpretation der Daten. In der Syntax wird Schenderas Fehler aber eins zu eins reproduziert.)

Auch in dem zweiten (folgenden) Beispiel müsste inhaltlich eigentlich ein Fehler angezeigt werden, wenn ZEIT1 größer, nicht kleiner ZEIT2 wäre, aber auch hier wurde Schenderas Syntax wortgetreu übersetzt. Die R-Syntax erzeugt hier allerdings eine logische Variable, in der Fehler mit `TRUE` statt mit 1 indiziert werden.

```
ZEIT1 = strptime(c("13/08/90"), "%d/%m/%y")
ZEIT2 = strptime(c("21/10/90"), "%d/%m/%y")
FEHLER = ZEIT1 <= dat$ZEIT2
```

4.12.1 – Seite 66 ff.

```
NOKOMMA <- c(1, 123, 12345, 12345678)
KOMMA <- NOKOMMA * 10^(3-nchar(NOKOMMA))
cbind(NOKOMMA, KOMMA)
```

In diesem Beispiel wird das Komma nicht wie im Buch über Stringfunktionen eingefügt, sondern mittels Algebra. `NOKOMMA` wird mit Zehnerpotenzen multipliziert, die von der Anzahl der Zeichen (`nchar`) bestimmt werden. Da drei Stellen vor dem Komma stehen sollen ist die Funktion *10 hoch (3-Anzahl der Zeichen)*.

Im folgenden Beispiel wird aus `NOKOMMA` (mittels `as.character` als String ausgegeben) durch den Befehl `formatC` zunächst `NSTRING2` mit führenden Leerzeichen erzeugt. Im zweiten Schritt werden die Leerzeichen dann mit `gsub` durch Nullen ersetzt.

```
NOKOMMA <- c(1, 123, 12345, 12345678)
NSTRING2 <- formatC(as.character(NOKOMMA))
```

```
NSTRING2 <- gsub(" ", "0", NSTRING2)
data.frame(NOKOMMA, NSTRING2)
```

Im nächsten Beispiel wird durch die Potenzierung 10 hoch 3-Zeichenzahl zunächst ein Dezimalpunkt nach der dritten Ziffer eingefügt. Dabei sorgt `formatC(... format = "f")` dafür, dass die Zahl nicht in wissenschaftlicher Schreibweise ausgegeben wird. Dann wird mit `paste` und s`ubstr` der zweiter Dezimalpunk nach der fünften Ziffer eingefügt.

Im nächsten Schritt werden mittels gsub die Punkte durch Doppelpunkte ersetzt. `fixed = TRUE` sorgt dafür, dass `"."` als Punkt und nicht als Wildcard interpretiert wird.

```
NOKOMMA <- c(1, 123, 12345, 12345678)
KOMMAS <- formatC(NOKOMMA * 10^(3 - nchar(NOKOMMA)),
                  digits = 5, format = "f")
KOMMAS <- paste(substr(KOMMAS, 1, 6), ".", substr(KOMMAS, 7, 8),
                sep = "")
KOMMAS2 <- gsub(".", ":", KOMMAS, fixed = TRUE)
data.frame(NOKOMMA, KOMMAS, KOMMAS2)
```

Mit paste werden im folgenden Beispiel die Inhalte zweier Variablen mit einander zu einer gemeinsamen Variable `LABEL2` zusammengefügt. Nachdem mit plot ein Streudiagramm der Variablen VARX und VARY erzeugt wurde, wird mit text eine Bezeichnung der einzelnen Datenpunkte erwirkt.

```
dat <- data.frame(VARX = 123:128, VARY = 123:128)
(LABEL2 <- paste(dat$VARX, ":", dat$VARY, sep = ""))
plot(dat$VARX, dat$VARY)
text(dat$VARX, dat$VARY + 0.1, lables = LABEL2,
     cex = 0.7, adj = c(0.5, 0))
```

Nächstes Beispiel:

```
dat <- matrix(c("AA", "AA", "AA", "BB",
                "BB", "BB", "CC", "CC",
                "CC", "AA", "BB", "CC",
                "AA", "BB", "CC", "AA",
                "BB", "BB", "BB", "CC",
                "AA", "AA", "BB", "AA",
                "AA", "AA", "AA", "BB",
                "BB", "BB", "CC", "CC",
                "CC", "AA", "BB", "CC",
                "AA", "BB", "CC", "AA",
                "BB", "BB", "BB", "CC",
                "AA", "AA", "BB", "AA"),
       ncol = 4, byrow = TRUE)
KODE <- factor(paste(dat[,1], ":", dat[,2], ":", dat[,3], ":", dat[,4],
                     sep = ""))
KODENUM <- unclass(KODE)
data.frame(KODE, KODENUM)
```

Hier werden die Daten mittels `matrix` erzeugt. Dabei wird durch `ncol` die Anzahl der Spalten festgelegt und `byrow = TRUE` bewirkt, dass die Daten zeilenweise eingefügt werden.

Die Daten jeweils einer Zeile werden dann mittels `paste` zu einer Variable zusammengefasst. In den eckigen Klammern wird dabei (weil die Zahl hinter dem Komma steht) jeweils die Spalte der Matrix bestimmt. Die so erzeugte Variable hätte das String-

Format. Um eine kategoriale Variable zu bekommen, wird der `paste`-Befehl noch mit `factor` bearbeitet. Das ganze wird als `KODE` abgelegt. Um schließlich die Kodenummern von `KODE` in `KODENUM` darzustellen, wird der Befehl `unclass` verwendet.

4.12.2 – Seite 73 ff.

```
KOMMA <- c("1:2.3456", "12.:3456", "1.23:45.6", "1.234:56", "1.23.45:6")
KOMMA2 <- KOMMA
KOMMA <- gsub("[[:punct:]]", "", KOMMA)
data.frame(KOMMA, KOMMA2)
```

In diesem Beispiel werden die Interpunktionszeichen mit dem `gsub`-Befehl entfernt. Die Erkennung der Interpunktionszeichen (verschiedenster Art) erfolgt durch `[[:punct:]]`.

Auch das folgende Beispiel kann ganz einfach durch `gsub` bewältigt werden.

```
KOMMA <- c("1:23456", "12:3456", "123:456", "1234:56", "12345:6")
OHNEKOMMA <- gsub(":", "", KOMMA)
data.frame(KOMMA, OHNEKOMMA)
```

Man kann aber auch Schenderas kompliziertem Weg folgen.

```
STELLE <- regexpr(":", KOMMA)
VORKOMMA <- substr(KOMMA, 1, STELLE-1)
NACHKOMMA <- substr(KOMMA, STELLE+1, nchar(KOMMA))
OHNEKOMMA <- paste(VORKOMMA, NACHKOMMA, sep = "")
data.frame(KOMMA, STELLE, VORKOMMA, NACHKOMMA, OHNEKOMMA)
```

Wenn es wie im folgenden Beispiel darum geht, das gesuchte Muster nur bei seinem ersten Auftreten im String zu ersetzen, ist die Lösung des Problems extrem einfach, in dem man `sub` an Stelle von `gsub` verwendet.

```
KOMMA <- c("1:23456", "12:3456", "123:456", "1:234:56", "12345:6")
OHNEKOMMA <- sub(":", "", KOMMA)
data.frame(KOMMA, OHNEKOMMA)
```

Geht es darum, das Zeichen, wenn es zwei mal auftritt, nur beim zweiten Mal zu löschen, wird die Sache etwas komplizierter.

```
MEHRERE <- grepl(":.*:", KOMMA)
STELLE <- regexpr(":", KOMMA)
STELLE[MEHRERE == TRUE] <- regexpr(":.*:", KOMMA[MEHRERE == TRUE]) +
                           attr(regexpr(":.*:", KOMMA[MEHRERE == TRUE]),
                                        "match.length") -1
OHNEKOMMA <- paste(substr(KOMMA, 1, STELLE-1),
                   substr(KOMMA, STELLE+1, nchar(KOMMA)), sep = "")
data.frame(KOMMA, MEHRERE, STELLE, OHNEKOMMA)
```

Zunächst werden mit `grepl` Einträge identifiziert, die zwei Doppelpunkte enthalten und unter `MEHRERE` abgelegt. Das Suchmuster `":.*:"` ist dabei so zu interpretieren, dass am Beginn und am Ende des Musters ein Doppelpunkt steht und dass dazwischen beliebige Zeichen (symbolisiert durch den Punkt) in beliebiger Häufigkeit (symbolisiert durch das nachfolgende Sternchen) stehen dürfen.

Dann wird mit `regexpr(":", KOMMA)` die `STELLE` des ersten Auftretens eines Doppelpunktes identifiziert. Im nächsten Schritt wird für die Einträge, die zwei Doppelpunkte enthalten die Stelle neu bestimmt (`STELLE[MEHRERE == TRUE]`). Dabei

macht man sich zu Nutze, dass `regexpr` nicht nur die Stelle des ersten Auftretens eines Musters ausgibt, sondern als `attr(..., "match.length")` auch die Länge des gefundenen Musters. Die Stelle des zweiten Auftretens ergibt sich dann als Stelle des ersten Auftretens + Länge -1.

Mit `paste` werden dann die einzelnen Teile des Strings unter Auslassung des jeweiligen Doppelpunktes zusammen gefügt und in `OHNEKOMMA` abgelegt.

4.12.3 – Seite 77 f.

Die Vereinheitlichung der Datumsangaben funktioniert wie schon unter 4.11 beschrieben mit `gsub` und `strptime`.

```
MIXDATUM <- c("14/8/2005", "14-8-2005", "14.8.2005", "14.8/2005",
              "14.8-2005", "14-8/2005", "14/8.2005", "14/8/2005",
              "14-8.2005", "14.8-2005", "14/8.2005")
DATUM <- strptime(gsub("[[:punct:]]", ".", MIXDATUM), "%d.%m.%Y")
```

Wenn man sich die Tage, Monate und Jahre einzeln ausgeben lassen möchte, funktioniert das wie folgt:

```
TAG   <- DATUM$mday
MONAT <- DATUM$mon +1
JAHR  <- DATUM$year + 1900
data.frame(MIXDATUM, TAG, MONAT, JAHR, DATUM)
```

4.13 – Seite 78 f.

In R werden Missings grundsätzlich als `NA` angegeben. Verschiedene Typen von Missings werden nicht unterschieden. Es gibt meines Wissens keinen Befehl, der es ermöglicht, andere Werte (z.B. 99, 88 etc.) so als Missings zu definieren, dass sie den entsprechenden Kode behalten. Statt dessen kann man eine Kopie des Datensatzes erstellen und dort die entsprechenden Kodes durch `NA` ersetzen. Möchte man später wissen, um welchen Missing-Kode es sich jeweils handelt, kann man das im Originaldatensatz nachsehen. *Achtung, das Rückverfolgen ist nur möglich, wenn die Datenzeilen mit einem eindeutigen ID-Kode identifiziert werden können!*

```
dat <- data.frame(ID = 1:4,
                  VAR_1 = c(99,2,98,4),
                  VAR_2 = c(1,99,3,4),
                  VAR_STRG = c("eins", "zwei", "", "vier"))
dat.orig <- dat
dat$VAR_1[dat$VAR_1 == 98 | dat$VAR_1 == 99] <- NA
dat$VAR_2[dat$VAR_2 == 99] <- NA
dat$VAR_STRG[dat$VAR_STRG == ""]  <- NA
cbind(dat.orig, dat)
```

Man kann die Missings auch für den ganzen Datensatz erzeugen. Das geht allerdings nur, wenn nicht der Missing-Kode von einer Variable (z.B. 99 für die Variable Geschlecht) ein Sinnvoller Wert in einer anderen Variable (z.B. Alter) ist.

```
dat <- data.frame(ID = 1:4,
                  VAR_1 = c(99,2,98,4),
                  VAR_2 = c(1,99,3,4),
                  VAR_3 = c(1,98,3,99),
                  VAR_STRG = c("eins", "zwei", "Missing", "vier"))
dat.orig <- dat
```

```
dat[dat == 99 | dat == 98 | dat == "Missing"]   <- NA
cbind(dat, dat.orig)
```

Im nächsten Beispiel wird die Datenauswahl zum einen durch `[grepl("^0*$",...) == TRUE` bestimmt. Zum anderen wird `, 2:3]` angegeben, damit nur die Einträge in den Variablen VAR_1 und VAR_2 (die die zweite und dritte Spalte der Datenmatrix bilden) auf `NA` gesetz werden.

```
dat <- data.frame(ID = 1:4,
                  VAR_1 = c(0,2,0,""),
                  VAR_2 = c(0,"",3,0))
dat.orig <- dat
dat[grepl("^0*$",dat$VAR_1) == TRUE &
    grepl("^0*$", dat$VAR_2) == TRUE, 2:3] <- NA
cbind(dat, dat.orig)
```

4.14 – Seite 81

Die Einstellungen können in R mit `options()` eingesehen und ggf. geändert werden.

4.12.1 – Seite 81 f.

Um ein Einheitliches Aussehen zu erhalten, sollten die Ergebnisse in ein Textverarbeitungs- oder Tabellenkalkulationsprogramm kopiert und dort einheitlch formatiert werden.

Die Pakete sweave und odfweave bieten die Möglichkeit R aus einem Latex- oder Open-Office-Writer-Dokument heraus anzusteuern und die Ergebnisse automatisch in diese Dokumente einfügen zu lassen.

Gibt man seine Ergebnisse als Grafik aus, kann ein Einheitliches Aussehen erzeugt werden, indem die Grafikausgabe als Funktion `function` programmiert wird.

5 Doppelte Werte und mehrfache Datenzeilen

5.2.1 – Seite 96

```
dat <- data.frame(ID = c(1, 2, 3, 4, 2, 3, 5),
                  GRUPPE = c("a", "a", "b", "b", "a", "b", "a"),
                  ALTER = c(8, 17, 23, 75, 17, 23, 65))
table(dat$ID)
barplot(table(dat$ID))
```

In diesem Beispiel wird über den Befehl `table(dat$ID)` ermittelt, wie oft die jeweiligen ID's im Datensatz vorkommen. Eine grafische Darstellung als Balkendiagramm wird mit `barplot` bewirkt.

5.2.2 – Seite 97

Das folgende Beispiel funktioniert genau so wie 5.2.1. Es wird aber in den `table`-Befehl als zweite Ebene noch `dat$GRUPPE` eingefügt.

```
dat <- data.frame(ID = c(1, 2, 3, 4, 1, 2, 3, 3, 4),
                  GRUPPE = c(rep("a", 4), rep("b", 5)),
                  ALTER = c(8, 17, 23, 74, 75, 17, 23, 65, 46))
table(dat$ID, dat$GRUPPE)
barplot(table(dat$ID, dat$GRUPPE), beside = TRUE)
```

Im `barplot`-Befehl sorgt das Argument `beside = TRUE` dafür, dass die Balken innerhalb der Gruppen nebeneinander stehen und nicht übereinander gestapelt werden.

5.3 – Seite 98 ff.

```
dat <- data.frame(ID   = c(1, 2, 3, 4, 4, 5),
                  VAR1 = c(2, 3, 4, 5, 6, 6),
                  VAR2 = c(3, 4, 5, 6, 7, 7),
                  VAR3 = c(4, 5, 6, 7, 8, 8),
                  VAR4 = c(5, 6, 7, 8, 9, 9))
```

Ansatz a und b1 im Buch bewirken das gleiche. Es wird jeweils die Zeile, bei der die ID zuerst auftritt behalten. Die Zeile, in der sie ggf. ein zweites Mal auftritt wird gelöscht. Mit R lässt sich dieser Effekt folgendermaßen erzielen:

```
dat[duplicated(dat$ID) == FALSE,]
```

Der Befehl `duplicated` erzeugt einen logischen Vektor, der im Falle eines wiederholten Auftreten eines Wertes `TRUE` ausgibt, sonst `FALSE`. Mit den eckigen Klammern werden nun alle Datenzeilen mit `FALSE` ausgewählt. Den gleichen Effekt hat die etwas kürzere Variante

```
dat[!duplicated(dat$ID),]
```

Will man wie in Ansatz b2 das letzte Duplikat behalten und das erste verwerfen, kann man die folgende Variante anwenden:

```
dat[!duplicated(dat$ID, fromLast = TRUE),]
```

Will man alle Zeilen entfernen, die keine einzigartige ID haben, wie in Ansatz III, kann man `table` als Auswahlwerkzeug verwenden.

```
dat[dat$ID != names(table(dat$ID))[table(dat$ID) > 1],]
```

Dabei werden mit `[table(dat$ID) > 1]` die ID's identifiziert, die öfter als ein mal auftreten. Mit `[dat$ID != names(table(dat$ID))[...],]` werden dann genau die Datenzeilen ausgewählt, die diese ID's nicht enthalten.

5.4 – Seite 102 f.

```
dat <- data.frame(ID   = c(1, 2, 3, 4, 4, 5),
                  VAR1 = c(2, 3, 4, 5, 6, 6),
                  VAR2 = c(3, 4, 5, 6, 7, 7),
                  VAR3 = c(4, 5, 6, 7, 8, 8),
                  VAR4 = c(5, 6, 7, 8, 9, 9))
dat[!duplicated(dat[1:4]),]
```

Die Auswahl der Variablen, die für die Suche nach Duplikaten berücksichtigt werden, wird hier durch Angabe der Spaltennummern 1-4 (`dat[1:4]`) vorgenommen. Wenn man die Auswahl über Angabe der Variablennamen vornehmen möchte, funktioniert die folgende Syntax mit Hilfe von `cbind`. Das Ergebnis ist das gleiche

```
dat[!duplicated(cbind(dat$ID, dat$VAR1, dat$VAR2, dat$VAR3)),]
```

Hier noch das Beispiel, bei dem Datensätze mit doppelten ID's nicht gelöscht, sondern angezeigt werden. Es unterscheidet sich von Ansatz III in 5.3 nur dadurch, dass statt `!=` (ist ungleich) == (ist gleich) eingesetzt wird.

```
dat[dat$ID == names(table(dat$ID))[table(dat$ID) > 1],]
```

5.5 - Seite 104 ff.

```
dat <- data.frame(ID       = c(1, 2, 3, 4, 2, 3, 5, 2, 3, 5),
                  GRUPPE   = c("a","a","b","b","a","b","a","a","b","a"),
                  ALTER    = c(8, 17, 23, 75, 17, 23, 65, 17, 23, 65),
                  VARNUM1  = c(1, 0, 1, 0, 0, 1, 1, 0, 1, 1),
                  VARNUM2  = c(0, 1, 1, 1, 1, 1, 0, 1, 1, 0),
                  VARNUM3  = c(1, 0, 0, 0, 0, 0, 0, 0, 0, 0),
                  VARCHAR1 = c("b","a","a","b","a","a","c","a","a","c"),
                  VARCHAR2 = c("c","b","b","a","b","b","b","b","b","b"),
                  VARCHAR3 = c("a","c","c","c","c","c","a","c","c","a"))
```

Um Duplikate im Datensatz zu löschen, kann die Syntax

```
dat[!duplicated(dat),]
```

angewendet werden. Zum gleichen Ergebnis kommt man noch einfacher mit

```
unique(dat)
```

Um sich einen schnellen Überblick über das Auftreten und die Häufigkeit duplizierter Datenzeilen zu verschaffen, hilft die folgende Syntax:

```
(DOPPELT <- table(paste(data.frame(t(dat))), dat$ID))
```

Dabei wird eine Tabelle erstellt, die einzigartigen Datenzeilen als Zeilen und die ID's als Spalten darstellt. Innerhalb der Tabelle wird dann die jeweilige Häufigkeit angegeben. Als Zwischenschritt wird dabei mit `paste(data.frame(t(dat)))` ein Vektor erzeugt, bei dem jeweils der Inhalt einer Datenzeile in einen Eintrag zusammen kopiert wird. (String-Angaben werden dabei in Zahlen dargestellt. Das macht die Darstellung etwas verwirrend.) Für kleine Datensätze ist das eine sehr gute Methode, um sich eine Übersicht zu verschaffen. Bei großen Datensätzen wird die Tabelle schnell unübersichtlich.

Mit der folgenden Syntax wird eine neue Variable `HÄUFIGKEIT` erzeugt, die für jede Zeile angibt, wie häufig sie im Datensatz auftritt.

```
dat$HÄUFIGKEIT <- table(paste(data.frame(t(dat))))[
                          paste(data.frame(t(dat)))]
```

Dabei wird eine Tabelle erzeugt, die die Häufigkeiten der zusammen kopierten Datenzeilen angibt. Bewirkt durch die eckigen Klammern wird dann jeweils die dem Zeilenmuster entsprechende Häufigkeit in den Vektor eingefügt.

Um anzuzeigen, wie viele Datenzeilen einmalig sind, genügt jetzt

```
table(dat$HÄUFIGKEIT == 1)
```

Die Häufigkeit einzigartiger Zeilen erhält man mit

```
table(duplicated(dat))
```

Eine Übersicht darüber, wie Häufig einmalige, doppelte, dreifache usw. Datenzeilen sind, erhält man mit

```
table(DOPPELT, exclude = 0)
```

Dabei wird auf die oben erstellte Tabelle `DOPPELT` zurück gegriffen, die einzigartige Datenzeilen in Bezug zur ID angibt. Von dieser Tabelle wird nun wieder eine Tabelle erstellt. Normalerweise würde dabei auch inhaltlich unsinnige Häufigkeit von „0“ dargestellt. Durch das Zusatzargument `exclude = 0` wird die Anzeige dieser unsinnigen Häufigkeit unterdrückt.

Die Summe dieser Tabelle ist nicht mehr die Anzahl aller Datenzeilen, sondern die der einzigartigen Datenzeilen. Interessiert man sich für die Häufigkeiten auf Basis aller Datenzeilen, kann man die folgende Syntax anwenden:

```
table(dat$HÄUFIGKEIT)
```

5.6 – Seite 111 ff.

Die Aufgaben aus diesem Abschnitt lassen sich mit einfachen Variationen von `duplicated` lösen.

```
dat <- data.frame(ID       = c(1, 2, 3, 4, 2, 3, 5),
                  GRUPPE   = c("a","a","b","b","a","b","a"),
                  ALTER    = c(8, 17, 23, 75, 17, 23, 65),
                  VARNUM1  = c(1, 0, 1, 0, 0, 1, 1),
                  VARNUM2  = c(0, 1, 1, 1, 1, 1, 0),
                  VARNUM3  = c(1, 0, 0, 0, 0, 0, 0),
                  VARCHAR1 = c("b","a","a","b","a","a","c"),
                  VARCHAR2 = c("c","b","b","a","b","b","b"),
                  VARCHAR3 = c("a","c","c","c","c","c","a"))

table(duplicated(dat))
dat[!duplicated(dat),]
dat[duplicated(dat),]
```

5.7 – Seite 113 ff.

```
dat <- data.frame(PATIENT = c(1,1,1,1,2,2,2,3,3,3),
                  RECORD  = c("A","B","B","C","A","B","C","A","B","C"),
                  DATA    = c(127,58,58,59,98,67,75,58,59,43))

data.frame(ID    = dat$PATIENT,
           IQ    = dat$DATA[dat$RECORD == "A"],
           ALTER = dat$DATA[dat$RECORD == "B"],
           SCL90 = dat$DATA[dat$RECORD == "C"])
```

Bei diesem Versuch des Einlesens der Daten wird die Warnmeldung ausgegeben, dass der Befehl unterschiedliche Zeilenlängen impliziere. Das ist auf den doppelten Fall `1B` zurück zu führen. Die folgende Syntax unter Verwendung von `unique` funktioniert dann problemlos.

```
data.frame(ID    = unique(dat$PATIENT),
           IQ    = unique(dat)$DATA[unique(dat)$RECORD == "A"],
           ALTER = unique(dat)$DATA[unique(dat)$RECORD == "B"],
           SCL90 = unique(dat)$DATA[unique(dat)$RECORD == "C"])
```

5.8 – Seite 115 ff.

Daten aus relationalen Datenbanken können u.a. mit Hilfe des Paketes RODBC in R eingelesen werden.

6. Kapitel – Missings

6.1 – Seite 129 f.

```
dat <- data.frame(wert1 = c(1,1,1,1,1), miss1 = c(1,NA,NA,NA,1),
                  wert2 = c(1,1,1,1,1), miss2 = c(1,1,NA,1,1))
```

Der im Buch beschriebene Fehler bei der Mittelwertberechnung wird von R im Regelfall nicht gemacht, weil hier das Ergebnis von Berechnungen mit Daten, die Missings enthalten standardmäßig `NA` ist.

```
apply(dat, 1, sum)
apply(dat, 1, sum) / 4
```

Anmerkung: Der `apply(dat, 1, ...)`-Befehl bewirkt, dass die entsprechende Funktion (hier `sum`) auf jede Zeile des Datensatzes angewandt wird.

Man sollte jetzt ausreichend gewarnt sein, dass Missings im Datensatz vorhanden sind und dass somit der Nenner in den Mittelwertberechnungen nicht in jedem Fall 4 ist. Um die Missings aus der Berechnung auszuschließen, kann man das Argument `na.rm = TRUE` anwenden.

```
apply(dat, 1, function(x) sum(x, na.rm = TRUE))
```

Anmerkung: Innerhalb des `apply`-Befehls greift man jetzt nicht mehr auf die Standardeinstellung von `sum` zurück, sondern definiert bestimmte Argumente. Deshalb muss man die etwas kompliziertere Form `function(x) sum(x, ...)` verwenden.

Versucht man es nun trotz aller Warnungen mit

```
apply(dat, 1, function(x) sum(x, na.rm = TRUE)) / 4,
```

reproduziert man den Fehler aus dem Buch. Das gleiche gilt für die folgende Syntax.

```
apply(dat, 1, function(x) sum(x, na.rm = TRUE) / length(x))
```

Leider kann man das Argument `na.rm = TRUE` nicht innerhalb des `length`-Befehls anwenden. Ein korrektes Ergebnis erhält man aber, wenn man `sum(table(x))` verwendet.

```
apply(dat, 1, function(x) sum(x, na.rm = TRUE) / sum(table(x)))
```

Noch viel einfacher ist jedoch die folgende Syntax in ihren Varianten mit oder ohne `na.rm = TRUE`.

```
apply(dat, 1, mean)
apply(dat, 1, function(x) mean(x, na.rm = TRUE))
```

6.2 – Seite 135 f.

Alle Missings sind in R als `NA` definiert. Es gibt nicht die Möglichkeit, wie in SPSS benutzerdefinierte Missings festzulegen. Solche benutzerdefinierten Kodes für Missings sollten im Originaldatensatz beibehalten werden und man sollte sich für die Berechnungen eine Kopie erstellen, in der die Kodes dann durch `NA` ersetzt werden.

6.3.2 – Seite 138 ff.

```
dat <- data.frame(wert1 = c(1,1,1,1,NA,NA,1),
```

```
                    miss1 = c(1,NA,NA,NA,NA,NA,1),
                    wert2 = c(1,1,1,1,NA,NA,1),
                    miss2 = c(1,1,NA,1,NA,NA,1))
N_MISS <- apply(dat, 1, function(x) sum(is.na(x)))
PROZENT <- N_MISS / apply(dat, 1, length) * 100
cbind(N_MISS, PROZENT)

dat[PROZENT < 100,]
```

Zunächst wird mit `sum(is.na(x))` die Anzahl der Missings pro Zeile aufsummiert und als `N_MISS` abgelegt. `PROZENT` wird dann bestimmt, indem `N_MISS` mit `length` durch die Länge der Zeilen geteilt wird.

Mit `dat[PROZENT < 100,]` werden schließlich die Datenzeilen ausgewählt, die weniger als 100% Missings aufweisen. Das ganze geht aber mit der folgenden Syntax auch einfacher:

```
dat[apply(dat, 1, function(x) sum(is.na(x))/length(x)) < 1,]
```

Ich empfehle, sich die Muster der Verteilung von fehlenden Werten noch genauer anzusehen, als In Schenderas Buch beschrieben. Diese Muster können Hinweise auf die Ursache von Missings geben und ggf. systematische Verzerrungen deutlich machen.

Das Paket mice bietet mit `md.pattern` eine Funktion, die die verschiedenen Verteilungsmuster von Missings über die Datenzeilen darstellt und deren Häufigkeiten angibt.

```
library(mice)
md.pattern(dat)
```

Bei großen Datensätzen kann es hilfreich sein, sich die Muster von Fehlenden Daten grafisch anzeigen zu lassen.

```
plot(PROZENT)
```

zeigt auf der y-Achse den prozentualen Anteil von Missings pro Datenzeile, auf der x-Achse die Datenzeilen.

Mit der folgenden Syntax kann man die Datenmatrix darstellen, wobei Fehlende Werte farblich gekennzeichnet werden.

```
image(1:ncol(dat), 1:nrow(dat), is.na(t(dat))== FALSE)
```

Um das gleiche etwas hübscher darzustellen und mit passenden Achsenbeschriftungen zu versehen, habe ich die folgende Funktion programmiert.

```
popp.miss.plot <- function(x, gridlines = TRUE, Farbe = "darkviolet",
                           Name = deparse(substitute(x)),
                           varlab = names(x), ...){
  image(1:ncol(x), 1:nrow(x), is.na(t(x)),
        col = c("grey80", Farbe),
        xaxt = "n", xlab = "", ylab = "Fallnummer",
        main = paste("Missings im Datensatz\n", Name))
  if(ncol(x) > 200) gridlines <- FALSE
  if(gridlines == TRUE) segments((1:ncol(x)-1)+0.5, 0,
                                 (1:ncol(x)-1)+ 0.5,
                                 nrow(x)+1, col = "grey90")
  ifelse(ncol(x) > 60, lascode <- 1, lascode <- 2)
  axis(1, at = 1:ncol(x), labels = varlab,
       las = lascode, cex.axis = 0.7, tick = gridlines)
```

```
}
```

6.4.1 – Seite 142

```
dat <- data.frame(VAR1 = c(NA,5,7), VAR2 = c(11,NA,15),
                  VAR3 = c(65,67,NA), VAR4 = c(NA,92,94))

dat[is.na(dat)]  <- 3
dat
```

Die Cold-Deck-Imputation wird in R erzielt, indem mit `[is.na(dat)]` die Missings ausgewählt werden und dann durch `3` ersetzt werden.

6.4.2 – Seite 143 ff.

```
dat <- data.frame(REFERENZ = c(21,31,41,21,23,12,11),
                  MIT_MISS = c( 2,NA, 3,NA, 2, 1,NA))
```

R bietet mit dem Befehl sample eine sehr einfache Möglichkeit zum Auffüllen der Missings mit einer Zufallsauswahl definierter Werte.

```
OHNE_MISS <- dat$MIT_MISS
OHNE_MISS[is.na(OHNE_MISS)] <- sample(1:3,
                               length(OHNE_MISS[is.na(OHNE_MISS)]),
                                replace = TRUE)
data.frame(dat, OHNE_MISS)
```

Zunächst wird eine Kopie von `MIT_MISS` als `OHNE_MISS` erstellt. Dann werden mit `[is.na(OHNE_MISS)]` die Missings in `OHNE_MISS` ausgewählt und mit Hilfe des `sample`-Befehls durch zufällig ausgewählte Zahlen ersetzt. Dabei bestimmt das erste Argument `1:3` das eine Zufallsauswahl aus den Zahlen von 1 bis 3 gezogen werden soll. Das zweite Argument bestimmt, wie viele Zahlen zufällig gezogen werden sollen. Da im Beispiel genau so viele Zufallszahlen benötigt werden wie Missings, wurde hier `length(OHNE_MISS[is.na(OHNE_MISS)])` angegeben. Schließlich bewirkt `replace = TRUE`, dass die ausgewählte Zahl für die nächste Zufallsauswahl wieder zur Verfügung steht. So hat jede der Zahlen für jedes der Missings die gleiche Wahrscheinlichkeit ausgewählt zu werden.

Etwas umständlicher kann man das gleiche Ergebnis auch erreichen, wenn man Schenderas Logik nachvollzieht.

```
OHNE_MISS <- dat$MIT_MISS
ZUFALL <- rep(NA, length(OHNE_MISS))
ZUFALL[is.na(OHNE_MISS)] <- runif(length(OHNE_MISS[is.na(OHNE_MISS)]))
OHNE_MISS[ZUFALL <= 0.333] <- 1
OHNE_MISS[ZUFALL > 0.333 & ZUFALL <= 0.666] <- 2
OHNE_MISS[ZUFALL > 0.666] <- 3
cbind(dat, OHNE_MISS, ZUFALL)
```

Zunächst wird `MIT_MISS` wieder nach `OHNE_MISS` kopiert. Außerdem wird mit `rep(NA, length(OHNE_MISS))` ein leerer Vektor `ZUFALL` mit der gleichen Länge wie `OHNE_MISS` erzeugt.

Dann wird für jede Zeile von `ZUFALL`, bei der `OHNE_MISS` ein Missing aufweist (`ZUFALL[is.na(OHNE_MISS)]`) eine Zufallszahl zwischen 0 und 1 berechnet. Dazu dient der Befehl `runif`, der Zufallszahlen mit gleichförmiger Auftrittswahrscheinlichkeit

(d.h. Jede Zahl hat die gleiche Wahrscheinlichkeit ausgewählt zu werden) erzeugt. Der Wertebereich von 0-1 ist die Standardeinstellung, so dass in diesem Beispiel Minimum und Maximum nicht angegeben werden müssen. Die Anzahl der berechneten Zufallszahlen wird durch `length(OHNE_MISS[is.na(OHNE_MISS)])` festgelegt.

Anschließend werden die Einträge von `OHNE_MISS`, bei denen `ZUFALL` kleiner oder gleich 0,333 ist, mit `OHNE_MISS[ZUFALL <= 0.333]` ausgewählt und auf 1 gesetzt. Analog werden Einträge, bei denen `ZUFALL` größer 0.333 bis 0.666 ist auf 2 gesetzt und Einträge mit `ZUFALL` größer 0.666 auf 3.

Das nachfolgende Beispiel auf Seite 144 fügt dem nichts wirklich neues hinzu. Die Werte von ERSATZ – einer gerundeten gleichförmigen Verteilung von 1-100 – werden statt der Missings in OHNE_MISS eingesetzt, unabhängig davon, welche Zufallszahl in ZUFALL berechnet wird. Diese Zufallszahl bestimmt zwar, bei welchem Syntaxschrit die Zahl von ERSATZ nach OHNE_MISS kopiert wird. Das Endergebnis wird davon in keiner Weise beeinflusst. (Obendrein wird bei der Berechnung von ERSATZ durch das Runden eine Verzerrung der gleichförmigen Verteilung produziert. Die Werte 1 und 100 haben nur eine halb so große Auftrittswahrscheinlichkeit wie die anderen Werte. Besser wäre es hier gewesen, das Minimum auf 0,5 und das Maximum auf 100,5 festzulegen.)

Die folgende Syntax bewältigt die Aufgabe problemlos.

```
OHNE_MISS <- dat$MIT_MISS
OHNE_MISS[is.na(OHNE_MISS)] <- sample(1:100,
                                      length(OHNE_MISS[is.na(OHNE_MISS)]),
                                       replace = TRUE)
```

Im nächsten Beispiel werden im sortierten Datensatz die Missings durch den Median ihres jeweiligen Vorgängers und Nachfolgers ersetzt.

```
dat <- data.frame(ITEM1 = c(1,1,1,4,4,4,1), ITEM2 = c(2,2,2,1,1,1,2),
                  ITEM3 = c(3,4,3,2,2,3,3), ITEM4 = c(5,5,5,NA,3,3,NA))

dat$VORHER <- dat$ITEM4
dat <- dat[order(dat$ITEM1, dat$ITEM2, dat$ITEM3),]
INDEX <- 1:length(dat$ITEM4)
dat$ITEM4[is.na(dat$ITEM4)] <-
  apply(cbind(dat$ITEM4[INDEX[is.na(dat$ITEM4)]-1],
              dat$ITEM4[INDEX[is.na(dat$ITEM4)]+1]),
        1, median)
dat
```

Nachdem eine Kopie der Variable `ITEM4`, unter dem Namen `VORHER` abgelegt wurde, wird der Datensatz mit `dat[order(dat$ITEM1, dat$ITEM2, dat$ITEM3),]` anhand der Variablen `ITEM1-ITEM3` sortiert.

Um die Einträge jeweils vor und nach einem Fehlenden Wert auswählen zu können, wird dann mit `1:length(dat$ITEM4)` die Hilfs(index)variable `INDEX` gebildet. Die Auswahl dieser Einträge erfolgt mit `cbind(dat$ITEM4[INDEX[is.na(dat$ITEM4)]-1], dat$ITEM4[INDEX[...]+1])`. Dabei wird eine Tabelle mit zwei Spalten erstellt, mit den Einträgen vor dem Missing in der ersten und den Einträgen nach dem Missing in der zweiten Spalte. Mit `apply(..., 1, median)` wird dann der Median berechnet und an der Stelle des Missings eingesetzt.

6.4.3 Seite 146 ff.

```
dat <- data.frame(ID = 1:5, SEX = c("M", NA, "M", NA, "W"),
```

```
                    SCHWANGER = c("NEIN", "JA", "NEIN", "JA", "NEIN"))
dat$SEX[dat$SCHWANGER == "JA"] <- "W"
```

Mit den Angaben in der eckigen Klammer werden Einträge ausgewählt, die auf Grund logischer Erwägungen ersetzt werden sollen und es wird der gewünschte Wert eingesetzt. Auch das folgende geringfügig komplexere Beispiel funktioniert nach diesem Prinzip.

```
OHNE_MISS <- c(21, 31, 41, 21, 23, 12, 11)
MIT_MISS  <- c("21-40", NA, "41-60", NA, "21-40", "0-20", NA)

ORIGINAL  <- MIT_MISS
MIT_MISS[is.na(MIT_MISS) & OHNE_MISS >  0 & OHNE_MISS <= 20]  <- "0-20"
MIT_MISS[is.na(MIT_MISS) & OHNE_MISS > 20 & OHNE_MISS <= 40]  <- "21-40"
MIT_MISS[is.na(MIT_MISS) & OHNE_MISS > 40 & OHNE_MISS <= 60]  <- "41-60"
data.frame(OHNE_MISS, ORIGINAL, MIT_MISS)
```

6.4.5 – Seite 149 f.

```
dat <- data.frame(ID = 1:5, ALTER = c(5, 10, NA, 60, 75))
dat$MEANALT <- dat$ALTER
dat$MEANALT[is.na(dat$ALTER)] <- mean(dat$ALTER, na.rm = TRUE)
dat$MEANLIN <- approx(dat$ALTER, n = length(dat$ALTER))$y
dat$MEANMED <- dat$ALTER
dat$MEANMED[is.na(dat$ALTER)] <- median(dat$ALTER, na.rm = TRUE)
dat
```

In diesem Beispiel werden die beiden Ansätze Schenderas zur univariaten Schätzung der Missings in etwa abegebildet[3]. Die Methode mit der `MEANLIN` erzeugt wird entspricht dabei in etwa der RMV-Lösung in SPSS. Der Befehl `approx` erstellt eine lineare Interpolation auf Basis der Daten in `ALTER`. Das Argument `n = length(dat$ALTER)` legt fest, dass genau so viele Einträge erzeugt werden, wie in `ALTER` enthalten. Automatisch werden vorhandene Einträge so wiedergegeben wie gehabt und Missings werden linear interpoliert. Ein `approx`-Objekt ist eine Liste mit den zwei Vektoren `$x` für Indexnummern und `$y` für interpolierte Werte. Da hier nur die interpolierten Werte interessieren, wird dem `approx`-Befehl `$y` angehängt.

Die Mittelwert- und Medianschätzungen in `MEANALT` und `MEANMED` folgen eher der AGGREGATE-Lösung in SPSS, obgleich dieser Ansatz in R viel einfacher zu realisieren ist. Zuerst wird `ALTER` in die jeweilige Zielvariable kopiert. Dann werden mit `[is.na(dat$ALTER)]` die Missings ausgewählt und mit `mean` bzw. `median` aufgefüllt. Dabei ist es wichtig, das Argument `na.rm = TRUE` anzugeben, weil damit Missings aus der Mittlewert- bzw. Medianberechnung ausgeschossen werden. Vergisst man dieses Argument wird wieder ein Missing eingesetzt.

6.4.6 – Seite 152 ff.

Das folgende Beispiel ähnelt sehr dem letzten Beispiel unter 6.4.2.

```
dat <- data.frame(ITEM1 = c(1,1,1,4,4,4,1), ITEM2 = c(2,2,2,1,1,1,2),
                  ITEM3 = c(3,3,3,2,2,2,3), ITEM4 = c(5,5,5,NA,3,3,NA))

dat <- dat[order(dat$ITEM1,dat$ITEM2, dat$ITEM3, dat$ITEM4),]
INDEX <- 1:length(dat$ITEM4)
```

3 Die Variablenbezeichnungen MEANLIN und MEANMED habe ich wörtlich übernommen. Sinnvoller wären Bezeichnungen wie LINALT und MEDALT.

```
dat$ITEM4[duplicated(dat[,1:3]) & is.na(dat$ITEM4)] <-
  dat$ITEM4[INDEX[duplicated(dat[,1:3]) & is.na(dat$ITEM4)]-1]
dat
```

Zunächst wird der Datensatz sortiert, wobei die Variable, deren Missings imputiert werden sollen (hier ITEM4) als letztes Sortierkriterium angegeben werden muss. Dann wird die Hilfsvariable INDEX erzeugt.

Die eigentliche Ersetzung wird durchgeführt, indem mit `[duplicated(dat[,1:3]) & is.na(dat$ITEM4)]` die Einträge von `INDEX4` ausgewählt werden, die sowohl Missings enthalten als auch Duplikate sind. Dabei ist wichtig, dass das Kriterium Duplikat die Variable, deren Missings ersetzt werden sollen nicht einbezieht, sonst wird die entsprechende Datenzeile evtl. gerade wegen des Missings einzigartig. Technisch gelöst wird das wieder durch eckige Klammern: `dat[,1:3]` wählt nur die ersten drei Spalten von `dat` aus und lässt `ITEM4` somit aus.

Die so ausgewählten Einträge werden dann durch `dat$ITEM4[INDEX[duplicated(dat[,1:3]) & is.na(dat$ITEM4)]-1]` mit Einträgen aus `ITEM4` aufgefüllt, die eine Zeile weiter oben stehen.

Bei dem folgenden komplexeren Beispiel auf den Seiten 153 f. sollen die Missings wahrscheinlich durch einen Wert aufgefüllt werden, der zufällig aus den vorhandenen Einträgen des gleichen Stratums ausgewählt wird. Ich bin mir da aber nicht ganz sicher, weil die im Buch angegebene Syntax offenbar nicht korrekt ist und gar keine Missings ersetzt. Eine nach dem angenommenen Prinzip funktionierende R-Syntax ist hier ohne ausführliche Kommentare angegeben:

```
# SPSS-Datensatz importieren
library(foreign)
dat <-
 data.frame(read.spss
 ("C:\\Programme\\SPSSInc\\Statistics17\\Samples\\German\\Employee
 data.sav"))
str(dat)

dat2 <- data.frame(respnr = dat$id, stratum = dat$ausbild,
                   x = dat$gehalt) # Kopie mit 3 Variablen
dat2$x[1:40] <- NA    # Missings erzeugen
with(dat2, cbind(N.gültig = length(x[!is.na(x)]),
                 Fehlend = length(x[is.na(x)]),
                 Min = min(x, na.rm = TRUE),
                 Max = max(x, na.rm = TRUE),
                 Mean = mean(x, na.rm = TRUE),
                 SD = sd(x, na.rm = TRUE)))

# Daten sortieren und die erste und zweite Zeile jedes Stratums mit 1
# bzw. 2 inder Variable "seqstr" markieren.
dat2$xnew <- as.numeric(is.na(dat2$x))
dat2$xmis <- as.numeric(is.na(dat2$x))
dat2$seqnbeg <- 1:length(dat[,1])
dat2 <- dat2[order(dat2$stratum, is.na(dat2$x), dat2$seqnbeg),]
strat <- c(NA, dat2$stratum[1:(length(dat2$stratum)-1)])
dat2$seqnstr[dat2$stratum != strat] <- 1
dat2$seqnstr[1] <- 1
volg <- c(NA, dat2$seqnstr[1:(length(dat2$seqnstr)-1)])
dat2$seqnstr[is.na(dat2$seqnstr)] <- volg[is.na(dat2$seqnstr)] +1

# Das Gleiche noch einmal mit nur 3 Zeilen R-Kode
```

```
dat2 <- dat2[order(dat2$stratum, is.na(dat2$x), row.names(dat2)),]
dat2$sq.st[!duplicated(dat2$stratum)] <- 1
dat2$sq.st[!is.na(c(NA, dat2$sq.st[1:(length(dat2$x)-1)]))] <- 2


with(dat2, cbind(Missing = table(stratum, is.na(x))[,2], N =
table(stratum)))
dat2$stratn <- rep(table(dat2$stratum), table(dat2$stratum))
dat2$stratm <- rep(table(dat2$stratum, is.na(dat2$x))[,2],
table(dat2$stratum))

# 4 Zufalls-Indexnummern in seqnstr erzeugen, die den jeweiligen Strata
entsprechen
dat2$seqnstr[is.na(dat2$x)] <-
  with(dat2,
       trunc(1 + runif(length(x), 0, stratn-stratm))[is.na(x)])

# oder in einem Befehl, ohne stratn und stratm erzeugen zu  müssen.
dat2$sq.st[is.na(dat2$x)] <-
  with(dat2,
       trunc(1 + runif(length(x), 0,
                 rep(table(stratum, is.na(x))[,1], table(stratum))
                 ))[is.na(x)])
with(dat2, cbind(stratn-stratm, x, seqnstr, sq.st)[is.na(x),])

# Wahrscheinlich war das gemeint: Für fehlende x wird ein Zufällig
ausgewählter Wert der vorhandenen x im entsprechenden Stratum eingesetzt.
dat2$xnew <- dat2$x
dat2$xnew[is.na(dat2$x)] <- with(dat2, x[seqnstr][is.na(x)])

# Zurück in die urspründliche Reihenfolge
dat2[order(as.numeric(row.names(dat2))),]
```

6.4.7.1 – Seite 156 f.

```
dat <- data.frame(NUM1 = c(1,1,0,0,1,0,0,1,0),
                  NUM2 = c(1,1,0,0,1,0,1,1,0),
                  NUM3 = c(1,1,0,1,1,1,0,1,0),
                  NUM4 = c(1,0,1,0,NA,NA,0,1,0))
library(epicalc)
alpha(c(NUM1,NUM2, NUM3, NUM4), dataFrame = dat)
VORHER <- dat$NUM4
dat$NUM4[is.na(dat$NUM4)] <-
     round(apply(dat[is.na(dat$NUM4),1:3], 1, mean), 0)
SKALA1 <- apply(dat, 1, sum)
SKALA2 <- apply(cbind(dat[,1:3], VORHER), 1,
                function(x) sum(x, na.rm = TRUE))
cbind(SKALA1, SKALA2)
```

Zunächst wird mit `alpha` aus dem Paket `epicalc` {{185 Chongsuvivatwong,Virasakdi 2008;}} Cronbach's Alpha bestimmt. (SPSS-17 liefert dabei Werte, die von R und von den Angaben im Buch abweichen. Ich weiß nicht warum.) Dann wird eine Kopie von `NUM4` in `VORHER` abgelegt.

Die Ersetzung der Missings durch den Zeilenmittelwert erfolgt mit `apply(dat[is.na(dat$NUM4)], 1, mean)`. Um daraus 1 oder 0 zu machen wird mit `round(..., 0)` auf null Nachkommastellen gerundet.

Die zeilenweise Summenberechnung für `SKALA1` und `SKALA2` erfolgt mit `apply(..., 1, sum)`. Bei `SKALA2` ist es ein bisschen komplizierter, weil der `sum`-Befehl mit `na.rm = TRUE` ergänzt werden muss.

6.4.7.2 – Seite 158

Das Paket mice bietet mit dem Befehl `mice` sehr umfangreiche Möglichkeiten zur multiplen Imputation von Missings. Bei den angegebenen Beispieldaten konvergiert der Algorhythmus aber leider nicht. Angehängt an die Hilfe-Seite des Befehls ist ein detaillierter Artikel über multiple Imputation mit verketteten Gleichungen im Allgemeinen und die Umsetzung im Paket mice {{413 Buuren,Stef van 2011;}} im speziellen.

```
dat <- data.frame(ITEM1 = c(1,1,1,4,4,4,1), ITEM2 = c(2,2,2,1,1,1,2),
                  ITEM3 = c(3,3,3,2,2,2,3), ITEM4 = c(5,5,5,NA,3,3,NA))
library(mice)
imp <- mice(dat)
dat.neu <- complete(imp)
```

Der Befehl `mice` erzeugt die imputierten Werte, mit `complete` werden sie in den Datensatz eingefügt.

Kapitel 7: Ausreißer

7.2.1 – Seite 171

```
dat <- data.frame(matrix(c(1,0,5,3,4,5,6,7,3,4,5,6,7,3,4,5,6,7,3,4,5,
                           2,5,6,7,3,4,5,6,7,3,4,5,7,3,4,5,6,8,8,9,9,
                           3,5,3,1,6,4,2,8,0,4,5,3,2,4,3,5,6,7,8,9,1,
                           4,1,2,3,4,5,6,7,8,9,0,4,3,5,6,7,8,9,0,1,4),
                         nrow = 4, byrow = TRUE))
colnames(dat) <- c("ID", paste("var", 1:20, sep = ""))

AUSREIS1 <- apply(dat[,-1], 1,
                  function(x) length(x[x==0 | x==1 | x==8 | x==9]))
AUSREIS2 <- apply(dat[,-1], 1, function(x) length(x[x<2 | x>7]))
cbind(dat$ID, AUSREIS1, AUSREIS2)
```

Die Anzahl der Extremwerte wird durch `length(x[x==...])` berechnet. Um das für jede Zeile zu berechnen, wird die Funktion in den `apply`-Befehl eingebaut. Dabei bewirkt `dat[,-1]`, dass die erste Spalte (ID) nicht berücksichtigt wird.

Tabelle 1: Streuungsmaße in R

Maß	**R-Syntax**
Spannweite	`diff(range(x))` (range alleine gibt das Minimum und das Maximum aus.)
Quartilen	`summary(x)` `quantile(x, c(0, 1/4, 2/4, 3/4, 1))`
Quartilenabstand	`IQR(x)`
MAD	Formel wie im Buch: `sum(abs(x - median(x))) / length(x)` `mad(x)` arbeitet mit einer anderen Formel
Varianz	`sum((x - mean(x))^2) / (length(x) -1)` `var(x)`
Standard-abweichung	`sqrt(sum((x - mean(x))^2) / (length(x) -1))` `sd(x)`
Variations-koeffizient	`sd(x) / mean(x)`

7.2.4 – Seite 176 ff.

```
dat <- data.frame(SEE = c("Bodensee", "Müritz", "Chimsee",
                          "Schweriner See", "Starnberger See",
                          "Ammersee", "Plauer See", "Kummerower See",
                          "Steinhuder Meer", "Großer Plöner See",
                          "Schaalsee", "Selenter See", "Kölpinsee"),
                  QKM = c(571.5, 109.2, 79.9, 61.5, 56.4, 46.6, 38.4,
                          32.5, 29.1, 30, 22.8, 22.4, 20.3))
```

Eine feine Quelle, um sich Anregungen für R-Grafiken zu holen ist die Internetseite R Graph Gallery. Dort findet man ein Vielzahl verschiedener Grafiken zusammen mit der Syntax, mit der sie realisiert wurden.

Boxplot

Ein einfaches Boxplot lässt sich mit dem Befehl `boxplot` realisieren. In der Standard-einstellung werden dabei solche Werte als Ausreißer dargestellt, die mehr als 1,5 Interquartilenabstände unter der 25%-Quartile oder über der 75%-Quartile liegen. Diese Spanne kann ggf. mit dem Argument `range` verändert werden.

```
boxplot(dat$QKM)
# Mit Beschriftung
text(1.02,
     boxplot(dat$QKM)$out,
             labels = dat$SEE[dat$QKM ==
                              boxplot(dat$QKM,
                                      ylab = expression(Km^2),
                                      xlab = "Natürliche Seen")$out],
     adj = c(0, 0.5))
```

Das beschriften der Ausreißer erfolgt dann mit `text`. Dabei ist der erste Wert die Position auf der x-Achse, das zweite Argument die Position auf der y-Achse. Diese Positionen der Ausreißer werden mit `boxplot(x)$out` abgerufen. Mit dem dritten Argument labels werden die anzuzeigenden Texte bestimmt. Durch `dat$SEE[dat$QKM == boxplot(dat$QKM, ...)$out]` werden die Namen in `SEE` ausgewählt, die den gemäß

`boxplot` bestimmten Ausreißern entsprechen. In dem dabei aufgerufenen `boxplot`-Befehl werden dann noch gleich die Achsenbeschriftungen `xlab` und `ylab` eingefügt. Beim `ylab` kann mit Hilfe von `expression(Km^2)` die hochgestellte 2 (von „Km²“) dargestellt werden.

Alternative

Eine sehr Alternative, mit der sich in R sehr einfach Ausreißer erkennen lassen ist

```
plot(dat$SEE, dat$QKM, las = 2)
```

Noch einfacher, aber dafür ohne Beschriftung ist der Indexplot

```
plot(dat$QKM)
```

Histogramm

```
hist(dat$QKM)
```

Ein einfaches Histrogramm kann man mit hist erzeugen. Will man wie bei Schendera eine Normalverteilungskurve hinzufügen wird es ein wenig komplizierter. Für die Identifizierung von Ausreißern ist das allerdings gar nicht unbedingt nötig.

```
lines(seq(min(dat$QKM), max(dat$QKM), length.out = 1000),
      dnorm(seq(min(dat$QKM), max(dat$QKM), length.out = 1000),
            mean(dat$QKM), sd(dat$QKM))
        * length(dat$QKM) * max(diff(hist(dat$QKM)$breaks)))
```

Der Befehl `lines` fügt einem Diagramm eine Linie hinzu. Das erste Argument enthält dabei einen Vektor mit den x-Koordinaten. Hier wird mit `seq(min(dat$QKM, max(dat$QKM, length.out = 1000)` eine Sequenz mit tausend Einträgen erzeugt, die vom Minimum bis zum Maximum von `QKM` reicht. Das zweite Argument bestimmt die y-Koordinaten. Hier ist es die normalverteilte Dichtefunktion der x-Koordinaten, die mit `dnorm` berechnet wird. Bei diesem Befehl werden zunächst die Ausgangsdaten (hier die x-Koordinaten) angebeben, dann der Mittelwert (hier `mean(dat$QKM)`) und dann die Standardabweichung (hier `sd(st$QKM)`). Bei einem Histogramm mit einer proportionalen y-Achse wäre das ausreichen. Da in diesem Beispiel aber die absoluten Häufigkeiten abgebildet werden, müssen die y-Koordinaten der Normalverteilungskurve noch mit der Sichprobengröße (`length(dat$QKM)`) und der Balkenbreite des Histogramms (`max(diff(hist(dat$QKM)$breaks))`) multipliziert werden.

Rücken-an-Rücken-Histogramm

```
Erziehungsberatung <- abs(rnorm(200,20,10)^3)/30
Kath.Träger <- sample(c("nein", "nein", "nein","ja"),200,replace = TRUE)

library(Hmisc)
out <- histbackback(split(Erziehungsberatung, Kath.Träger), las = 3)

barplot(-out$left, col = "red", horiz = TRUE, add = TRUE,
        space = 0, axes = FALSE)
barplot(out$right, col = "blue", horiz = TRUE, add = TRUE,
        space = 0, axes = FALSE)
```

Das Rücken-an-Rücken-Histogramm wird hier mit `histbacktoback` aus dem Paket Hmisc {{414 Harrell, Frank E Jr. 2010;}} realisiert. Innerhalb dieses Befehls bewirkt `split`

die Aufteilung von `Erziehungsberatung` in die Kategorien von `Kath.Träger`. Um das Diagramm farblich zu gestalten, wird das Ergebnis von `histbackback` zunächst als `out` gespeichert. Die beiden `barplot`-Befehle nutzen dann `out`, um das Diagramm farbig auszufüllen.

Fehlerbalken

```
errbar(1, mean(dat$QKM),
       yplus = mean(dat$QKM) + sd(dat$QKM),
       yminus = mean(dat$QKM) - sd(dat$QKM),
       xlab = "Natürliche Seen",
       ylab = expression(Km^2))
```

Fehlerbalkendiagramme können mit dem Befehl `errbar` aus dem Paket Hmisc (ggf. muss das mit `library(Hmisc)` erst geladen werden) erzeugt werden. Das erste Argument bestimmt die x-Koordinate(en), das zweite Argument die y-Koordinate(en) des Lagemaßes (hier `mean(dat$QKM)`). yplus und yminus bestimmen die y-Koordinaten des oberen bzw. unteren Balkens. In diesem Beispiel ist das mit `mean(dat$QKM) +/- sd(dat$QKM)` je eine Standardabweichung über bzw. unter dem Mittelwert.

Eindimensionale Streudiagramme

```
plot(dat$QKM, rep(1, length(dat$QKM)))
```

Oder mit Beschriftung:

```
plot(rep(1, length(dat$QKM)), dat$QKM)
text(1.02, dat$QKM, labels = dat$SEE, adj = c(0,0.5))
```

Stängel-Blatt-Diagramm

```
stem(dat$QKM)
stem(rnorm(100, 20, 5))
```

Eigener Vorschlag

Die folgende Funktion ist ein eigener Vorschlag zu univariaten Überprüfung intervallskalierter Variablen. Zunächst einmal zeigt sie die Anzahl der Fehlenden, Minimum, Maximum, Mittelwert, Quartilen und die Standardabweichung an. Weiterhin wird ein Histogramm und ein Boxplot ausgegeben, wobei die Zeilennummern von Ausreißern angegeben werden. Schließlich wird zur Prüfung auf Normalverteilung ein Q-Q-Plot und ein Shapiro-Wilk-Test ausgegeben.

```
# Univariate Darstellung intervallskalierter Daten
popp.cont <- function(x, col="grey", col.box = col, curve = FALSE, ...) {
  dev.par <- par()              # ursprüngliche Grefikparameter speichern
  layout(rbind(c(5, 5),         # Einrichtung der Ausgabebereiche
               c(4, 1),         # 1 = Histogramm, 2, Boxplot, 3 = QQ-Plot,
               c(4, 2),         # 4 = Text, 5 = Überschrift
               c(4, 3)),
         heights = c(4, 10, 2, 12))
  par(mar = c(0,4,3,2))         # Histogram
  p <- hist(x, axes = FALSE, xlab = NULL, col = col, main = "Histogramm")
  if(curve == TRUE){            # Normalverteilungskurve
    vx <- seq(min(x, na.rm = TRUE), max(x, na.rm = TRUE),
```

```
            length.out = 1000)
    lines(vx, length(x[is.na(x) == FALSE]) * max(diff(p$breaks)) *
     dnorm(vx, mean(x, na.rm = TRUE), sd(x, na.rm = TRUE)))
    }
  q <- boxplot(x, plot = FALSE)      # nötig um Ausreißer zu definieren
  axis(c(2), at = NULL)
  rug(x, side = 3)
  if(length(q$out) > 0) {            # Index von Ausreißern anzeigen
    text(q$out, max(p$counts), adj = c(0, 0.3),
         labels = which(x < q$stats[1] | x > q$stats[5]),
         cex = 0.7, srt = 270)
  }
  par(mar = c(3,4,0,2), xpd = NA)                    # Boxplot
  q <- boxplot(x, horizontal = TRUE, ylim = par("usr")[1:2],
               yaxs = "i", axes = FALSE, boxwex = 2.5, at = 1.2,
               col = col.box)
  axis(1, at = NULL, line = 0.1)                     # QQ-Plot
  par(mar = c(3, 4, 4, 2), xpd = FALSE)
  qqnorm(x)
  qqline(x)
  par(mar = c(3, 4, 0, 2))
  plot.new()                                         # Text
    shw <- shapiro.test(x)
    if (shw$p.value < 0.001) pval <- "<0.001"
    if (shw$p.value >=0.001) pval <- round(shw$p.value, 3)
    text(0, 1,
         labels = paste("N:", "Fehlende:", "Gültige:\n", "Minimum:",
                        "Maximum:\n", "Mittelwert:", "SD:\n",
                        "Quartilen:", "   25%:", "   Median:",
                        "   75%:\n\n",
                        "Shapiro-Wilk-Test auf Normalverteilung",
                        paste("W = ", round(shw$statistic, 3),
                        ",   p-Wert: ", pval, sep = ""), sep = "\n"),
         adj = c(0,1))
    text(0.8, 1,
         labels = paste(length(x), length(x[is.na(x) == TRUE]),
                        length(x[is.na(x) == FALSE]),
                        "", summary(x)[1], summary(x)[6], "",
                        summary(x)[4], round(sd(x, na.rm = TRUE), 2), "",
                        "", summary(x)[2], summary(x)[3], summary(x)[5],
                        sep = "\n"),
         adj = c(1, 1))
 plot.new()                                          # Überschrift
 par(mar = c(0,0,0,0))
 text(0.5, 0.5,
      labels = paste("Verteilung von ", deparse(substitute(x))), cex = 2)
 layout(1)                           # Grafikeigenschaften zurück setzen
 par(mar = c(5.1, 4.1, 4.1, 2.1), xpd = TRUE)
}
```

7.3.1 – Seite 183 ff.

Diskrepanz- und Hebelwerte von (multivariaten) Modellen lassen sich in R extrem einfach mit `plot(model)` darstellen.

```
model <- lm(circumference ~ age, data = Orange)
par(mfrow = c(2,2))
plot(model)
```

Zunächst wird hier mit `lm` ein einfaches lineares Modell aus Beispieldaten erzeugt und als `model` abgelegt. Dann wird mit `par(mfrow = c(2, 2))` dafür gesorgt, dass die nachfolgend erzeugten vier Diagramme alle in einer Abbildung dargestellt werden. (Man kommt allerdings auch ohne diesen Befehl aus.)

Wenn man den `plot`-Befehl auf ein Modell anwendet, werden automatisch vier Diagramme ausgegeben. Zur Identifikation von Ausreißern eignet sich hier besonders das vierte Diagramm, in dem studentisierte Residuen vs. Leverage angezeigt werden und Cook's Distanz als Linie eingetragen wird. Werte, die hinter dieser Linie liegen gelten als sehr auffällig und einflussreich. (In den anderen drei Diagrammen werden (in zwei Varianten) Residuen vs. Vorhergesagte dargestellt, um die Varianzhomogenität einzuschätzen und ein Q-Q-Diagramm gezeigt, um die Normalverteilung der Residuen zu prüfen.)

Die Residuen kann man sich absolut, standardisiert oder studentisiert ausgeben lassen:

```
resid(model)
rstandard(model)
rstudent(model)
```

Hebelwerte werden standardmäßig als Hat-Values angegeben.

```
hatvalues(model)
```

Leverage wie bei Cohen {{410 Cohen,Jacob 2003;/a/f, S. 394}} basiert auf der Formel $\frac{1}{n}+\frac{(x-\bar{x})^2}{\sum(x-\bar{x})^2}$ und kann mit der folgenden Funktion berechnet werden.

```
lev <- function(x){
  1/length(x[!is.na(x)]) +
  (x - mean(x, na.rm = TRUE))^2 /
  (sum((x - mean(x, na.rm = TRUE))^2, na.rm = TRUE))}
lev(model$model$age)
```

In SPSS (Version 17) werden beim Befehl RERESSION allerdings sogenannte zentrierte Hebelwerte berechnet, bei denen nicht 1/n addiert wird. Der Befehl UNIANOVA erzeugt jedoch die unzentrierten Hebelwerte.

Die Einflusstatistiken DfFit, DfBeta und Cook-Distanzen können mit influence.measures abgerufen werden.

```
influence.measures(model)
```

7.3.3.1 – Seite 189

```
dat <- data.frame(WEEK = rep(1:280, each = 7),
                  crestpr = seq(2.1, 1.8, length.out = 280*7) +
                            rnorm(280*7, 0, 0.1))
dat$crestpr[1850] <- 25
plot(tapply(dat$crestpr, dat$WEEK, mean), type = "l")
```

Zunächst wird ein Beispieldatensatz `dat` erzeugt. `WEEK` enthält die Zahlen von 1 bis 280 je sieben mal. `crestpr` ist eine leicht abfallende Kurve von 2,1 bis 1,8 mit Zufallsschwankungen mit einer Standardabweichung von 0,1. Der 1850ste Eintrag wird auf 25 gesetzt.

Da im Diagramm jeweils die Mittelwerte der Wochen dargestellt werden sollen kommt

`tapply` zum Einsatz. Hier wird für jeden Wert von `WEEK` der Mittelwert von `crestpr` berechnet. Mit `plot` wird das Ganze dargestellt. Damit ein Liniendiagramm ausgegeben wird, ist das Argument `type = "l"` angegeben.

7.3.3.2 – Seite 191

Regelkarten[4] und andere Diagramme zur statistischen Qualitätskontrolle können mit Hilfe des Pakets qcc {{411 Scrucca,Luca 2004;}} erzeugt werden.

```
library(qcc)
data(pistonrings)
diameter <- qcc.groups(pistonrings$diameter, pistonrings$sample)

qcc(diameter[1:25,], type="xbar")
qcc(diameter[1:25,], type="xbar", newdata=diameter[26:40,])
```

Zunächst wird das Paket `qcc` und der Beispieldatensatz `pistonrings` geladen. Alle Durchmesser sind hier in der Variable `diameter` gespeichert. Die Variable `sample` gruppiert diese Einträge in einzelne Stichproben. Als Vorbereitung für die Erstellung der Regelkarte wird `diameter` mit Hilfe von `qcc.groups` in eine Matrix überführt, bei der jede Zeile einem `sample` mit einer entsprechenden Zahl von Spalten für die einzelnen Durchmesser entspricht.

Eine X-Quer-Regelkarte wird dann mit `qcc(diameter[1:25,], type="xbar")` erstellt. Dabei werden zunächst nur die ersten 25 Stichproben verwendet. Im zweiten `qcc`-Befehl werden mit `newdata=diameter[26:40,]` die mit den ersten Stichproben ermittelten Grenzwerte auf die folgenden Ergebnisse angewandt.

7.3.4 – Seite 192 ff.

Andrews Plots lassen sich mit Hilfe des Befehls `andrews(x)` aus dem Paket tourr {{415 Cook,Dianne 2011;}} erstellen. Sie sollen allerdings schwierig zu interpretieren sein. Deshalb geht es hier weiter mit bivariaten Streudiagrammen.

```
dat <- data.frame(Thrombo = round(rnorm(100, 170,60), 2),
                  Leuko = round((rgamma(100, 2) + 0.5) * 3, 2),
                  pO2 = trunc(rnorm(100, 86, 5)),
                  Sex = sample(c("männlich", "weiblich"), 100,
                               replace = TRUE))

plot(dat$Leuko, dat$Thrombo, main = "Bivariate Exploration")
```

Ein zweidimensionales Streudiagramm kann ganz einfach mit `plot(x, y)` erzeugt werden. Dreidimensionale Streudiagramme erhält man mit `cloud` aus dem Paket lattice.

```
library(lattice)
cloud(Thrombo ~ pO2 * Leuko, data = dat)
```

Mit `scatter3d` aus dem Paket car lässt sich ein dreidimensionales Streudiagramm erzeugen, dass man im Ausgabefenster frei drehen kann.

```
library(car)
scatter3d(Thrombo ~ pO2 * Leuko, data = dat)
```

Eine weitere sehr gute Möglichkeit mehrdimensionale Daten grafisch darzustellen ist das Open-Source-Programm GGobi (http://www.ggobi.org/). Mit Hilfe des Pakets rggobi lassen

4 Regelkarten findet man auch unter dem Begriff „Shewhart chart“.

sich Daten einfach mit R austauschen. Ich habe leider noch einige Schwierigkeiten das zu installieren.

Mit plot lassen sich ebenso bivariate Streudiagramme mit Kategorialen Daten darstellen.

```
plot(dat$Leuko, dat$Sex)
```

Kapitel 8: Plausibilität

8.2.2.1 – Seite 214 f.

```
dat <- data.frame(LK.unt = rbeta(500, 1, 5)*60,
                  LK.pos = rnorm(500, 25, 7),
                  Menopause = sample(c("prä-menopausal",
                                       "post-menopausal",
                                       "unbekannt"),
                                     500, replace = TRUE))
dat$LK.unt[239] <- 55
dat$LK.pos[c(331, 492, 90, 282)] <- c(127, 50, 49, 47)
levels(dat$Menopause) <- levels(dat$Menopause)[c(2,1,3)]
dat$Menopause[c(331, 492)]  <- "prä-menopausal"
dat$Menopause[c(90, 282)] <- "post-menopausal"
```

Ein mit Zeilennummern beschriftetes Streudiagramm wird durch eine Kombination von `plot` und `text` erzeugt. Diese Befehle müssen nur mit so vielen Argumenten bestückt werden, um hübsch auszusehen. Für einen schnellen Überblick reicht die Angabe der Variablen.

```
plot(dat$LK.pos, dat$LK.unt, col = "grey", pch = 20,
     ylab = "Anzahl untersuchter LK", xlab = "Anzahl positiver LK")
text(dat$LK.pos, dat$LK.unt, adj = c(0, 0.5), cex = 0.8)
```

Für die Darstellung einer Kombination von kontinuierlicher und kategorialer Variabel eignet sich ein Boxplot.

```
boxplot(dat$LK.pos ~ dat$Menopause,
        ylab = "Anzahl positiver Lymphknoten")
text(dat$Menopause, dat$LK.pos, cex = 0.7, pos = 4)
```

Bei diesem Plot werden allerdings alle Datenwerte von LK.pos beschriftet, was zu einem nicht ganz so hübschen Bild führt (für einen Überblick aber vollkommen ausreichend ist). Möchte man nur die Ausreißer beschriften ist die Syntax ein wenig komplizierter.

```
b <- boxplot(dat$LK.pos ~ dat$Menopause,
             ylab = "Anzahl positiver Lyphknoten")
text(b$group, b$out,
     labels = sapply(b$out, function(x) which(x == dat$LK.pos)),
     pos = 4, cex = 0.7)
```

Zunächst wird der `boxplot` als `b` gespeichert, um im Nächsten Schritt einfach auf einzelne Werte des Plots zurück greifen zu können. In `text` werden dann mit `b$group` die x-Koordinaten und mit `b$out` die y-Koordinaten der Ausreißer definiert. Die anzugebenden Zeilennummern der Ausreißer werden mit `labels = sapply(b$out, function(x) which(x == dat$LK.pos))` bestimmt. Dabei sorgt `which(...)` dafür, die Zeilennummern zu bestimmen, bei denen `b$out` mit `dat$LK.pos` übereinstimmt. Da bei diesem Vergleich aber immer nur jeweils ein Wert (von `b$out`) mit einem Vektor

(`dat$LK.pos`) verglichen werden kann, wird das Ganze in einen sapply-Befehl eingebaut, der die `which`-Funktion für jeden Wert von `b$out` einzeln durchrechnet.

Befehl	Argumente	Beschreibung	siehe S.
$	Objekt$Subobjekt Datensatz$Variablenname	Wählt Unterobjekte anhand ihres Namens aus Objekten aus, z.B. Variablen aus Datensätzen	2, 3, 6
[]	Objekt[Zeilen, Spalten, ...]	Wählt bestimmte Werte, Zeilen oder Spalten eines Objekts aus	2, 3, 3, 6, 8, 13
abs	abs(x)	Gibt die absolute Zahl von x ohne Vorzeichen wieder	27
agrep	agrep("Stringmuster", Objekt, ...)	Sucht ähnliche Stringmuster. vergl. grep	6
alpha	alpha(c(Variablennamen), Datensatz)	Berechnet Chronbach's Alpha. Befehl aus dem Paket epicalc	24
apply	apply(matrix, Zeile(1)/Spalte(2), Funktion)	Wendet eine Funktion auf die Zeilen(1) oder Spalten(2) einer Matrix oder Data.frames an (vergl. sapply, tapply)	1, 18, 19, 24, 25
approx	approx(x, y = NULL, n = 50, …)	Erzeugt lineare Interpolationen der Länge n aus den Daten x und ggf. y.	**22**
as.character	as.character(objekt)	Gibt ein Objekt als String wieder	8, 10
as.factor	as.factor(objekt)	Wandelt ein Objekt in einen Faktor um	3
as.numeric	as.numeric(Objekt)	Gibt ein Objekt als Zahl wieder	23
attr	attr(Objekt, „Attributsbezeichnung“)	Gibt ausgewählte Attribute von Objekten aus	12
barplot	barplot(x, …)	Erzeugt ein Balkendiagramm von x	14, 27
boxplot	bocplot(x)	Erzeugt ein Boxplot von x	**26**
c	c(x, y, z, ...)	Verknüpft eine Reihe von Daten	1, 2, 3, 6, 7, 8, 25, 27, 31
cbind	cbind(var1, var2, ...)	Verknüpft Vektoren spaltenweise zu einer Matrix bzw. einem Datensatz	2, 3, 10, 15, 21, 23, 24, 25, 32
cloud	cloud(z ~ x * y, data = …)	Erzeugt ein 3D-Streudiagramm mit den Variablen x, y und z. Teil des Pakets lattice	31
colnames	colnames(x)	Gibt die Spaltennamen eines Datensatzes oder einer Matrix aus	25
complete	complete(x)	Fügt imputierte Daten in den Datensatz ein. Teil des Paketes mice	25
data	data(Datensatzname)	Läd einen Beispieldatensatz	31
data.frame	data.frame(var1, var2, ...)	Verknüpft Vektoren zu einem Datensatz. Stringvariablen werden automatisch in Faktoren umgewandelt, es sei denn man ergänzt "..., stringsAsFactors = FALSE"	1, 2, 3, 6, 7, 8, 16, 23, 25
diff	diff(x)	Differenzen von x	27
dnorm	dnorm(x, Mittelwert, SD)	Berechnet die Dichtefunktion von x bei gegebenem Mittelwert und Standardabweichung	27
duplicated	Datensatz[duplicated(var1, var2, ...)]	Erzeugt ein logisches Objekt, das Duplikate mit dem Wert TRUE markiert. !duplicated markiert die Nicht-Duplikate also den ersten Fall bei dem die Wertekombination auftritt (vergl. unique)	2, 15, 16, 23, 24

errbar	errbar(x, y, yplus, yminus)	Erzeugt ein Fehlerbalkendiagramm. Angegeben werden die x- und y-Koordinate(n) sowie mit yplus und yminus die Enden der Fehlerbalken. Teil des Pakets Hmisc	28
expression	expression(Mathematischer Ausdruck)	Stellt Symbole und besondere Zeichen dar	**26**
factor	factor(Objekt)	Wandelt ein Objekt in einen Faktor (kategoriale Variable) um. Wie as.factor aber mit mehr Variationsmöglichkeiten.	11
for	for(i in 1:x){funktion}	Führt eine Funktion x mal aus	6
formatC	formatC(Objekt)	Objekt im C-Format wiedergeben	10
function	function(x, ...){Funktionsdefinition}	Erstellt eigene Funktionsaufrufe	1, 14, 18, 19, 24, 25, 30, 32
grep, grepl	grep("Stringmuster", Objekt)	Sucht Stringmuster in Objekten. grep gibt die jeweiligen Indexnummern aus, grepl einen logischen Vektor. vergl. sub und agrep	4, 5, 8, 9, 10, 12, 14
gsub		siehe "sub"	
hatvalues	hatvalues(Modell)	Gibt die Hat-Values eines Modells aus	30
hist	hist(x)	Erzeugt ein Histogramm	**27**
histbackback	histbackback(a, b,...)	Erzeugt ein Rücken-an-Rücken-Histogramm der (Teil-)Variablen a und b (z.B. eine Bevölkerungspyramide)	27
influence.measures	influence.measures(Modell)	Berechnet die Einflussstatistiken DfFit, DfBeta, Kovarianzratio, Hat-Value und Cook-Distanz eines Modells	30
is.na	is.na(x)	Ergibt ein logisches Objekt mit TRUE für Missings(NA) und False für gültige Werte	1, 19, 20, 21, 22, 23, 23, 24, 30
length	length(Objekt)	Gibt die Länge eines Objekts bzw. die Anzahl der Einträge aus	18, 19, 20, 21, 22, 23, 25, 27, 28, 30
levels	levels(Faktor)	Zeigt die Wertelevels eines Faktor-Objektes an. Mit der Ergänzung "<- c('level1', 'level2', ...)" können Wertelevels gesetzt werden	3, 32
library	library(Paketname)	Läd ein zusätzliches Programmpaket. Das Paket muss vorher installiert werden.	3, 23, 24, 25, 27, 31, 31
lines	lines(x-Koordinaten, y-Koordinaten)	Fügt einem Diagramm eine Linie hinzu	**27**
lm	lm(y ~ x1 + x2 +..., data = Datensatz)	Berechnet ein lineares Modell	29
ls	ls()	Zeigt die im Workspace vorhandenen Objekte an	3
matrix	matrix(Daten, ncol = …, byrow = FALSE)	Erstellt eine Datenmatrix	11, 25
max	max(x)	Bestimmt das Maximum von x	23, 27

mean	mean(x)	Bildet den arithmetischen Mittelwert	18, 22, 23, 24, 27, 28, 30
median	median(x)	Bildet den Median von x	21, 22
mice	mice(Datensatz, ...)	Führt eine multiple Imputation von Fehlenden Werten durch. Teil des Paketes mice	**25**
min	min(x)	Bestimmt das Minimum von x	23, 27
names	names(objekt)	Gibt die Namen von Subobjekten aus (z.B. Variablennamen in Datensätzen)	2, 6, 7, 8
nchar	nchar(Objekt)	Gibt die Anzahl der Zeichen in den einzelnen Einträgen eines Objekts zurück	10, 11
nrow	nrow(Datensatz)	Gibt die Anzahl der Zeilen zurück	2, 2
order	Datensatz[order(Var1, Var2, ...)]	Sortiert Vektoren und Datensätze. Ein Minuszeichen vor der sortierenden Variable erzeugt eine absteigende Sortierung	2, 21, 23
par	par(...)	Mit par werden die Parameter der Grafikausgabe festgelegt	29
paste	paste(Element 1, Element 2, ...)	Fügt verschiedene Elemente zu einem zusammen.	7, 9, 11, 12, 16
plot	plot(Objekt(e), ...)	Erzeugt eine Grafik, bei zwei numerischen Variablen ein Streudiagramm	11, 27, 28, 29, 30, 31, 32
qcc	qcc(Matrix, type = „...“, …)	Erstellt ein Regelkartendiagramm aus einer mit qcc.groups erstellten Matrix. Teil des Pakets qcc	31
qcc.groups	qcc.groups(x, f)	Erstellt eine Matrix aus den Daten x, gruppiert durch f (vergl. qcc). Teil des Pakets qcc	31
rbeta	rbeta(n, shape1, shape2)	Erzeugt beta-verteilte Zufallszahlen	32
read.spss	read.spss("Dateipfad")	Importiert SPSS-Datendateiten (.sav). Das Ergebnis ist eine Liste. Der Befehl ist Teil des Pakets „foreign“	3, 23
regexpr	regexpr(„Muster“, Objekt)	Sucht ein Muster in einem String und gibt die Stelle an, an der es innerhalb des Stings steht.	12
rep	rep(Wert(e), Anzahl(en) der Wiederholungen)	Wiederholt einen Wert oder eine Wertefolge	2, 2, 9, 20, 24, 28, 30
resid	resid(Modell)	Gibt die Residuen eines Modells aus	30
rgamma	rgamma(n, shape)	Erzeugt gammaverteilte Zufallszahlen	31
rm	rm(objekt1, objekt2, ...)	Löscht Objekte aus dem Workspace	2, 3
rnorm	rnorm(n, Mittelwert, SD)	Erzeugt n normalverteilte Zufallszahlen mit angegebenem Mittelwert und Standardabweichung	27, 28, 30, 31, 32
round	round(objekt, Nachkommastellen)	Rundet ein Objekt auf eine bestimmte Anzahl von Nachkommastellen	8, 24, 31
row.names	row.names(Datensatz)	Gibt die Zeilennamen eines Datensatzes aus. (In der Regel 1-n)	24

rstandard	rstandard(Modell)	Berechnet die standadisierten Residuen eines Modells	30
rstudent	rstudent(Modell)	Berechnet die studentisierten Residuen eines Modells	30
runif	runif(min = 0, max = 1, n)	Brerechnet n Zufallszahlen zwischen min und max mit gleichförmiger Wahrscheinlichkeitsverteilung	**20**, 24
sample	sample(Grundgesamtheit, n, replace = FALSE, ...)	Zieht eine Zufallsstichprobe der Größe n aus einer Grundgesamtheit. Replace bestimmt, ob ein ausgewählter Eintrag erneut gezogen werden kann (TRUE) oder nicht (FALSE)	**20**, 21, 27, 31, 32
sapply	sapply(x, Funktion)	Wendet eine Funktion auf jeden Eintrag von x an. (Vergl. apply, tapply)	32
scatter3d	scatter3d(z ~ x * y, data = ...)	Erzeugt ein frei drehbares 3D-Streudiagramm. Teil des Pakets cat	31
sd	sd(x)	Bestimmt die Standardabweichung von x	23, 27, 28
seq, :	seq(min, max, Schrittweite = 1), Kurz: min:max	Erstellt eine Zahlenfolge von min bis max. Ein Kurzform ist min:max	**6**, 13, 14, 15, 19, 20, 21, 23, 27, 30
split	split(x, f)	Trennt Variable x gemäß der Kategorien von Variable f auf	27
strptime	strptime(Objekt, „Format“)	Erzeugt aus einem String ein Datum-Zeit-Objekt in vorgegebenem Format	10, 13
sub, gsub	sub("Muster", "Ersetzung", "in Objekt")	Sucht und ersetzt Stringmuster. vergl. grep. sub ersetzt das Muster jeweils nur beim ersten Auftreten im String; gsub ersetzt jedes Auftreten.	4, 5, 6, 6, 7, 8, 9, 10, 11, 12, 12, 13
subset	subset(Objekt, Bedingung)	Ergibt eine Teilauswahl eines Objektes (ähnlich wie eckige Klammern)	3
substr	substr(Objekt, erstes Zeichen, letztes Zeichen)	Gibt einen Teil eines Strings wieder. Der Bereich wird durch die Angabe des ersten und des letzten Zeichens definiert	9, 11, 12
sum	sum(x, y, z, ...)	Summiert	1, 18, 19, 24, 30
stem	stem(x)	Erzeugt ein Stängel-Blatt-Diagramm von x	28
t	t(Objekt)	Transponiert eine Matrix oder einen Datensatz; d.h. Zeilen und Spalten werden miteinader vertauscht	16
table	table(var1, var2, ...)	Erzeugt eine ein- bis multidimensionale Häufigkeitstabelle	2, 3, 14, 16, 18, 24
tapply	tapply(x, f, Funktion)	Berechnet eine Funktion von x für jede Kategorie von f (vergl. apply, sapply)	30
text	text(x-Koordinate, y-Koordinate, labels = Textinhalt)	Fügt Text in eine bestehende Grafik ein	11, 26, 28, 32
tolower	tolower("String" oder character-Objekt)	Wandelt einen String in Kleinbuchstaben um	7

toupper	toupper("String" oder character-Objekt)	Wandelt einen String in Großbuchstaben um	7
trunc	trunc(x)	Schneidet die Nachkommastellen ab	31
unclass	unclass(Objekt)	Gibt ein Objekt wieder, ohne seine Klassen-eigenschaften (z.B. factor, numeric etc)	11
unique	unique(Objekt)	Gibt ein Vektor, Array oder Datensatz ohne Duplikate aus (vergl. duplicated)	16, 17
which	which(Bedingung)	Gibt den Index der Elemente wieder, die die Bedingung erfüllen	2, 5, 32
with	with(Datensatz, Befehl)	Fügt den Datensatz vorübergehend in den Suchpfad ein. Im Befehl muss man dann bei der Angabe der Variablen nicht jedes mal den Datensatz mit angegeben.	23