R kompakt

Der schnelle Einstieg in die Datenanalyse - Zusatzkapitel
Datensätze aufbereiten & Diagramme erstellen mit dem Basisumfang von R

Daniel Wollschläger

22. August 2021

Kapitel 4

Datensätze aufbereiten und aggregieren mit dem Basisumfang von R

Dieses Kapitel zeigt alternativ zum gedruckten Kapitel 4 (*Datensätze aufbereiten und aggregieren mit* ***dplyr***), wie dieselben Aufgaben mit dem Basisumfang von R bewältigt werden können.

Bevor Datensätze analysiert werden können, müssen sie häufig in eine andere als ihre ursprüngliche Form gebracht werden – etwa um neue Variablen zu bilden, Beobachtungen bzw. Variablen auszuwählen sowie unterschiedliche Datensätze zusammenzuführen. Der folgende Abschnitt demonstriert diese Auswertungsschritte mit dem Basisumfang von R. Als Datengrundlage dient der in Abschn. 3.8 erstellte Datensatz `myDf1`.

```
> head(myDf1)
  id sex group age  IQ rating
1  1   f     T  26 112      1
2  2   m    CG  30 122      3
3  3   m    CG  25  95      5
4  4   m     T  34 102      5
5  5   m    WL  22  82      2
6  6   f    CG  24 113      0
```

4.1 Variablen umbenennen

Variablen lassen sich umbenennen, indem der entsprechende Variablenname im von `names()` erzeugten Ergebnisvektor ausgewählt und über eine Zuweisung ersetzt wird. Dabei kann entweder die Position direkt angegeben oder durch Vergleich mit dem gesuchten Variablennamen implizit ein logischer Indexvektor verwendet werden.

```
> names(myDf1)                              # vorhandene Variablen
[1] "id" "sex" "group" "age" "IQ" "rating"

> names(myDf1)[3] <- "fac"                  # Variable 3 umbenennen
> names(myDf1)
[1] "id" "sex" "fac" "age" "IQ" "rating"

# neue Variable fac wieder zurück nach group umbenennen
> names(myDf1)[names(myDf1) == "fac"] <- "group"
```

```
> names(myDf1)
[1] "id" "sex" "group" "age" "IQ" "rating"
```

4.2 Teilmengen von Daten auswählen

Bei der Analyse eines Datensatzes möchte man häufig eine Auswahl der Daten treffen, etwa nur Personen aus einer bestimmten Gruppe untersuchen, nur Beobachtungsobjekte berücksichtigen, die einen bestimmten Testwert überschreiten, oder aber die Auswertung auf eine Teilgruppe von Variablen beschränken. Die Auswahl von Variablen auf der einen und Beobachtungsobjekten auf der anderen Seite unterscheidet sich dabei konzeptuell nicht voneinander.

Die Funktion `subset()` bietet eine Alternative zur Auswahl von Beobachtungen und Variablen mit den ebenfalls möglichen Methoden zur Indizierung eines Datensatzes (Abschn. 3.8.2), die aber weniger übersichtlich sind. `subset()` gibt einen Datensatz mit der gewünschten Teilmenge von Variablen und Beobachtungen zurück.

`subset(x=⟨Datensatz⟩, subset=⟨Auswahl Zeilen⟩, select=⟨Auswahl Spalten⟩)`

4.2.1 Variablen auswählen

Das Argument `select` dient dazu, eine Teilmenge der Spalten auszuwählen, wofür ein Vektor mit auszugebenden Spaltenindizes oder -namen benötigt wird. Dabei müssen Variablennamen ausnahmsweise nicht in Anführungszeichen stehen.[1] Fehlt ein Vektor für `select`, werden alle Spalten ausgegeben.

```
> subset(myDf1, select=c(group, IQ))    # Variablen group und IQ
   group  IQ
1      T 112
2     CG 122
3     CG  95
4      T 102                            # Ausgabe gekürzt ...
```

Um alle bis auf einige Variablen auszugeben, eignet sich ein negatives Vorzeichen vor dem numerischen Index oder vor dem Namen der wegzulassenden Variablen.

```
> subset(myDf1, select=c(-sex, -IQ))    # ohne sex und IQ
   id group age rating
1   1     T  26      1
2   2    CG  30      3
3   3    CG  25      5                  # Ausgabe gekürzt ...
```

Alle Variablen, deren Namen einem bestimmten Muster entsprechen, können mit den in Abschn. 3.12.3 vorgestellten Funktionen zur Suche nach Zeichenfolgen ausgewählt werden.

[1] Dafür kommt die Technik des *non-standard evaluation* zum Einsatz, die Wickham (2019a, Kap. 19, 20) erläutert.

```
# Variablen, deren Name mit i / I beginnt und weitere Zeichen enthält
> (colIdx <- grep("^i.+", names(myDf1), ignore.case=TRUE))
[1] 1 5

> subset(myDf1, select=colIdx)
   id  IQ
1   1 112
2   2 122
3   3  95                                  # Ausgabe gekürzt ...
```

4.2.2 Beobachtungen auswählen

Das Argument `subset` ist ein Indexvektor zum Indizieren der Zeilen des Datensatzes `x`. Dieser Vektor ergibt sich häufig aus einem logischen Vergleich der Werte einer Variable mit einem Kriterium.[2] Die Variablennamen des Datensatzes sind innerhalb von `subset()` bekannt, ihnen muss also im Ausdruck zur Bildung des Indexvektors nicht ⟨`Datensatz`⟩`$` vorangestellt werden. Fehlt ein Vektor für `subset`, werden alle Zeilen ausgegeben.

```
> subset(myDf1, sex == "f")             # Daten weibliche Personen
   id sex group age  IQ rating
1   1   f     T  26 112      1
6   6   f    CG  24 113      0
12 12   f     T  21  98      1

> subset(myDf1, id == rating)           # Wert id = Wert rating
  id sex group age  IQ rating
1  1   f     T  26 112      1
```

Auch die Auswahl nach mehreren Kriterien gleichzeitig ist möglich, indem der Indexvektor durch entsprechend erweiterte logische Ausdrücke gebildet wird (Abschn. **??**). Dabei kann es der Fall sein, dass mehrere Bedingungen gleichzeitig erfüllt sein müssen (logisches UND, `&`), oder es ausreicht, wenn bereits eines von mehreren Kriterien erfüllt ist (logisches ODER, `|`).

```
# alle männlichen Personen mit einem Rating größer als 2
> subset(myDf1, (sex == "m") & (rating > 2))
   id sex group age  IQ rating
2   2   m    CG  30 122      3
3   3   m    CG  25  95      5
4   4   m     T  34 102      5          # Ausgabe gekürzt ...

# alle Personen mit einem eher hohen ODER eher niedrigen IQ-Wert
> subset(myDf1, (IQ < 90) | (IQ > 110))
   id sex group age  IQ rating
1   1   f     T  26 112      1
```

[2] Fehlende Werte behandelt `subset()` als `FALSE`, sie müssen also nicht extra vom logischen Indexvektor ausgeschlossen werden. Um die Stufen der Faktoren auf die in der Auswahl noch tatsächlich vorhandenen Ausprägungen zu reduzieren, ist `droplevels(`⟨`Datensatz`⟩`)` zu verwenden (Abschn. **??**).

```
2   2   m    CG  30 122      3
5   5   m    WL  22  82      2          # Ausgabe gekürzt ...
```

Für die Auswahl von Fällen, deren Wert auf einer Variable aus einer Menge bestimmter Werte stammen soll (logisches ODER), gibt es eine weitere Möglichkeit: Mit dem Operator ⟨`Menge1`⟩ `%in%` ⟨`Menge2`⟩ als Kurzform von `is.element()` kann ebenfalls ein logischer Indexvektor zur Verwendung in `subset()` gebildet werden. Dabei prüft ⟨`Menge1`⟩ `%in%` ⟨`Menge2`⟩ für jedes Element von ⟨`Menge1`⟩, ob es auch in ⟨`Menge2`⟩ vorhanden ist und gibt einen logischen Vektor mit den einzelnen Ergebnissen aus.

```
# Personen aus Wartelisten- ODER Kontrollgruppe
> subset(myDf1, group %in% c("CG", "WL"))
   id sex group age  IQ rating
2   2   m    CG  30 122      3
3   3   m    CG  25  95      5
5   5   m    WL  22  82      2
6   6   f    CG  24 113      0          # Ausgabe gekürzt ...
```

4.3 Variablen entfernen, hinzufügen und transformieren

Variablen eines Datensatzes werden gelöscht, indem ihnen die leere Menge `NULL` zugewiesen wird – im Fall mehrerer Variablen gleichzeitig in Form einer Liste mit der Komponente `NULL`.

```
> dfTemp       <- myDf1                # Kopie erstellen
> dfTemp$group <- NULL                 # eine Variable löschen
> head(dfTemp, n=3)
  id  sex  age   IQ  rating
1  1    w   26  112       1
2  2    m   30  122       3
3  3    m   25   95       5

> dfTemp[c("sex", "IQ")] <- list(NULL)  # mehrere Variablen löschen
> head(dfTemp, n=3)
  id  age  rating
1  1   26       1
2  2   30       3
3  3   25       5
```

Wie bei Listen (Abschn. 3.9.1) können einem bestehenden Datensatz neue Variablen mit den Operatoren ⟨`Datensatz`⟩`$`⟨`neue Variable`⟩ und ⟨`Datensatz`⟩`["`⟨`neue Variable`⟩`"]` hinzugefügt werden. Analog zum Vorgehen bei Matrizen kann an einen Datensatz auch mit `cbind(`⟨`Datensatz`⟩`, `⟨`Vektor`⟩`, ...)` eine weitere Variable passender Länge als Spalte angehängt werden.[3]

[3] Dagegen ist das Ergebnis von `cbind(`⟨`Vektor1`⟩`, `⟨`Vektor2`⟩`)` eine Matrix. Dies ist insbesondere wichtig, wenn numerische Daten und Zeichenketten zusammengefügt werden – in einer Matrix würden die numerischen Werte automatisch in Zeichenketten konvertiert.

Im Beispiel soll der Beziehungsstatus der Personen dem Datensatz hinzugefügt werden.

```
> married <- sample(c(TRUE, FALSE), nrow(myDf1), replace=TRUE)
> myDf2   <- myDf1                                    # erstelle Kopie
> myDf2$married1    <- married                        # Möglichkeit 1
> myDf2["married2"] <- married                        # Möglichkeit 2
> myDf3 <- cbind(myDf1, married)                      # Möglichkeit 3
> head(myDf3, n=3)
  id  sex  group  age   IQ  rating  married
1  1    f      T   26  112       1     TRUE
2  2    m     CG   30  122       3     TRUE
3  3    m     CG   25   95       5    FALSE
```

Alternativ kann auch die Funktion `transform()` Verwendung finden. Sie überschreibt einen Datensatz nicht, sondern gibt einen veränderten Datensatz zurück, der einem neuen Objekt zugewiesen werden kann.

`transform(⟨Datensatz⟩, ⟨Variablenname1⟩=⟨Ausdruck1⟩, ...)`

Unter ⟨`Ausdruck`⟩ ist anzugeben, wie sich die Werte der neuen Variable ergeben, die unter ⟨`Variablenname`⟩ gespeichert und an ⟨`Datensatz`⟩ angehängt wird. Es können mehrere solcher Zuweisungen durch Komma getrennt vorgenommen werden. Die Variablennamen des Datensatzes sind zur Verwendung innerhalb von ⟨`Ausdruck`⟩ bekannt. Trägt man links des `=` den Namen einer schon bestehenden Variable ein, wird diese überschrieben.

Im Beispiel soll das Quadrat des Ratings angefügt und zwei Gruppen anhand des IQ gebildet werden.

```
> myDf4 <- transform(myDf1,
+                    rSq=rating^2,
+                    IQgrp=cut(IQ, breaks=c(0, 100, Inf)))

> head(myDf4, n=3)
  id sex group age  IQ rating married rSq     IQgrp
1  1   f     T  26 112      1   FALSE   1 (100,Inf]
2  2   m    CG  30 122      3   FALSE   9 (100,Inf]
3  3   m    CG  25  95      5   FALSE  25   (0,100]
```

4.4 Doppelte und fehlende Werte behandeln

Doppelte Werte können in Datensätzen etwa auftreten, nachdem sich teilweise überschneidende Daten aus unterschiedlichen Quellen in einem Datensatz integriert wurden. Alle später auftretenden Duplikate mehrfach vorhandener Zeilen werden durch `duplicated(⟨Datensatz⟩)` identifiziert und durch `unique(⟨Datensatz⟩)` ausgeschlossen (Abschn. **??**). Lediglich die jeweils erste Ausfertigung bleibt so erhalten.

```
# Datensatz mit doppelten Werten herstellen
> myDfDouble <- rbind(myDf1, myDf1[sample(seq_len(nrow(myDf1)), 4), ])

# doppelte Zeilen identifizieren (alle Ausfertigungen)
> duplicated(myDfDouble) | duplicated(myDfDouble, fromLast=TRUE)
[1] FALSE FALSE FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE FALSE
[12] FALSE TRUE TRUE TRUE TRUE

> myDfUnique <- unique(myDfDouble)    # doppelte Zeilen ausschließen
> any(duplicated(myDfUnique))         # Kontrolle: noch doppelte?
[1] FALSE
```

Fehlende Werte werden in Datensätzen weitgehend wie in Matrizen behandelt (Abschn. 3.11.4). Auch hier müssen also `is.na()` und `anyNA()` benutzt werden, um das Vorhandensein fehlender Werte zu prüfen.[4]

```
> myDfNA           <- myDf1          # Kopie
> myDfNA$IQ[2]     <- NA             # auf missing setzen
> myDfNA$rating[3] <- NA             # auf missing setzen
> is.na(myDfNA)[1:3, ]
     id   sex group   age    IQ rating
1 FALSE FALSE FALSE FALSE FALSE  FALSE
2 FALSE FALSE FALSE FALSE  TRUE  FALSE
3 FALSE FALSE FALSE FALSE FALSE   TRUE

# prüfe jede Variable, ob sie mindestens ein NA enthält
> apply(myDfNA, 2, anyNA)
   id    sex  group    age    IQ  rating
FALSE  FALSE  FALSE  FALSE  TRUE    TRUE
```

Selbst definierte Funktionen (Abschn. 12.2) helfen in Kombination mit `apply()` dabei, pro Variable die Anzahl fehlender Werte bzw. die Anzahl nicht fehlender Beobachtungen zu zählen.

```
# Anzahl fehlender Werte je Variable
> apply(myDfNA, 2, function(x) { sum(is.na(x)) })
    id    sex  group    age     IQ rating
     0      0      0      0      1      1

# Anzahl nicht fehlender Werte je Variable
> apply(myDfNA, 2, function(x) { length(na.omit(x)) })
    id    sex  group    age     IQ rating
    12     12     12     12     11     11
```

Eine weitere Funktion zur Behandlung fehlender Werte ist `complete.cases(⟨Datensatz⟩)`. Sie liefert einen logischen Indexvektor zurück, der für jedes Beobachtungsobjekt (jede Zeile) angibt, ob fehlende Werte vorliegen. Die vollständigen Fälle, oder die Fälle mit fehlenden

[4] Das Paket `naniar` (Tierney, Cook, McBain & Fay, 2020) hilft dabei, das Ausmaß fehlender Werte und ihr Verteilungsmuster in den Variablen eines Datensatzes durch verschiedene Diagramme zu analysieren.

Werten können mit `subset()` ausgegeben werden. Für die Zeilen mit fehlenden Werten ist dabei der Indexvektor aus der logischen Negation des Ergebnisses von `complete.cases()` zu bilden.

```
> complete.cases(myDfNA)
[1] TRUE FALSE FALSE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE

# zähle vollständige und unvollständige Zeilen
> table(complete.cases(myDfNA))
FALSE  TRUE
    2    10

> subset(myDfNA, !complete.cases(myDfNA)) # Zeilen mit fehlenden Werten
  id sex group age IQ rating
2  2   m    CG  30 NA      3
3  3   m    CG  25 95     NA
```

Weiterhin können wie bei Matrizen alle Zeilen mit `na.omit()` gelöscht werden, in denen Werte fehlen.

```
> head(na.omit(myDfNA), n=4)              # nur vollständige Zeilen
  id sex group age  IQ rating
1  1   f     T  26 112      1
4  4   m     T  34 102      5
5  5   m    WL  22  82      2
6  6   f    CG  24 113      0
```

4.5 Datensätze sortieren

Datensätze werden ebenso wie Matrizen mit `order()` sortiert (Abschn. 3.7.4).

`order(⟨Vektor⟩, na.last=TRUE, decreasing=FALSE)`

Unter ⟨`Vektor`⟩ ist die Variable (Spalte) eines Datensatzes anzugeben, die in eine Reihenfolge zu bringen ist. `na.last` ist per Voreinstellung auf `TRUE` gesetzt und sorgt ggf. dafür, dass Indizes fehlender Werte zum Schluss ausgegeben werden. Die Voreinstellung `decreasing=FALSE` bewirkt eine aufsteigende Reihenfolge. Zeichenketten werden in alphabetischer Reihenfolge sortiert. Die Reihenfolge bei Faktoren wird dagegen von der Reihenfolge der Stufen bestimmt, die nicht deren alphabetischer Reihenfolge entsprechen muss (Abschn. 3.5.4).

`order()` gibt einen Indexvektor aus, der die Zeilenindizes des Datensatzes in der Reihenfolge der zu ordnenden Variablenwerte enthält. Soll der gesamte Datensatz in der Reihenfolge dieser Variable angezeigt werden, ist der ausgegebene Indexvektor zum Indizieren der Zeilen des Datensatzes zu benutzen.

```
> (idx1 <- order(myDf1$rating))              # sortiere nach rating
[1] 6 1 11 12 5 8 2 7 9 3 4 10
```

```
> head(myDf1[idx1, ])                          # sortierter Datensatz
   id  sex  group  age   IQ  rating
6   6    f     CG   24  113       0
1   1    f      T   26  112       1
11 11    m     CG   20   92       1
12 12    f      T   21   98       1
5   5    m     WL   22   82       2
8   8    m     WL   35   90       2           # Ausgabe gekürzt
```

Soll nach zwei Kriterien sortiert werden, weil die Reihenfolge durch eine Variable noch nicht vollständig festgelegt ist, können weitere Datenvektoren in der Rolle von Sortierkriterien als Argumente für `order()` angegeben werden.

```
# sortiere myDf1 primär nach group und innerhalb jeder Gruppe nach IQ
> (idx2 <- order(myDf1$group, myDf1$IQ))
[1] 11 3 6 2 7 12 4 1 10 5 9 8

> head(myDf1[idx2, ])                          # sortierter Datensatz
   id  sex  group  age   IQ  rating
11 11    m     CG   20   92       1
3   3    m     CG   25   95       5
6   6    f     CG   24  113       0
2   2    m     CG   30  122       3
7   7    m      T   28   92       3
12 12    f      T   21   98       1           # Ausgabe gekürzt
```

4.6 Datensätze aufteilen

Wenn die Beobachtungen (Zeilen) eines Datensatzes in durch die Stufen eines Faktors festgelegte Gruppen aufgeteilt werden sollen, kann dies mit `split()` geschehen.

`split(x=⟨Datensatz⟩, f=⟨Faktor⟩)`

Für jede Zeile des Datensatzes `x` muss der Faktor `f` die Gruppenzugehörigkeit angeben und deshalb die Länge `nrow(x)` besitzen. Auch wenn `f` Teil des Datensatzes ist, muss der Faktor vollständig mit ⟨`Datensatz`⟩`$`⟨`Faktor`⟩ angegeben werden. Sollen die Zeilen des Datensatzes in Gruppen aufgeteilt werden, die sich aus der Kombination der Stufen zweier Faktoren ergeben, sind diese in eine Liste einzuschließen. In diesem Fall sorgt das Argument `drop=TRUE` dafür, dass nur jene Stufen-Kombinationen berücksichtigt werden, für die auch Beobachtungen vorhanden sind.

Das Ergebnis ist eine Liste, die für jede Faktorstufe eine Komponente in Form eines Datensatzes besitzt. Diese Datensätze können etwa dazu dienen, Auswertungen getrennt nach Gruppen vorzunehmen (Abschn. 4.11).

```
> split(myDf1, myDf1$group)
$CG
```

```
   id  sex  group  age   IQ  rating
2   2    m     CG   30  122       3
3   3    m     CG   25   95       5
6   6    f     CG   24  113       0
11 11    m     CG   20   92       1

$T
   id  sex  group  age   IQ  rating
1   1    f      T   26  112       1
4   4    m      T   34  102       5
7   7    m      T   28   92       3
12 12    f      T   21   98       1

$WL
   id  sex  group  age   IQ  rating
5   5    m     WL   22   82       2
8   8    m     WL   35   90       2
9   9    m     WL   23   88       3
10 10    m     WL   29   81       5

# teile Beobachtungen nach sex und group auf
> split(myDf1, list(myDf1$sex, myDf1$group))      # Ausgabe gekürzt ...
```

`split()` akzeptiert auch einen Vektor `x`, dessen Elemente entsprechend der im Faktor `f` definierten Gruppenzugehörigkeit aufgeteilt werden. Das Ergebnis ist wieder eine Liste mit einer Komponente je Faktorstufe, wobei jede Komponente ein Vektor ist. Um eine Matrix `x` analog zu einem Datensatz bzgl. ihrer Zeilen aufzuteilen, muss man explizit die Methode `split.data.frame()` aufrufen (Abschn. 12.2.3). In der erzeugten Liste ist dann jede Komponente eine Matrix.

4.7 Datensätze zeilen- oder spaltenweise verbinden

Wenn zwei oder mehr Datensätze `df` vorliegen, die hinsichtlich ihrer Variablen identisch sind, so fügt die Funktion `rbind(⟨df1⟩, ⟨df2⟩, ...)` die Datensätze analog zum Vorgehen bei Matrizen zusammen, indem sie sie untereinander anordnet. Auf diese Weise könnten z. B. die Daten mehrerer Teilstichproben kombiniert werden, an denen dieselben Variablen erhoben wurden.[5] Die Reihenfolge der Variablen muss in den Datensätzen nicht übereinstimmen. Wenn die Datensätze Gruppierungsfaktoren enthalten, ist zunächst sicherzustellen, dass alle dieselben Faktorstufen in derselben Reihenfolge besitzen (Abschn. **??**, 3.5.4).

```
> (dfNew <- data.frame(id=13:15,             group=c("CG", "WL", "T"),
+                      sex=c("f", "f", "m"), age=c(18, 31, 21),
+                      IQ=c(116, 101, 99),   rating=c(4, 4, 1)))
  id  sex  group  age   IQ  rating
```

[5] `bind_rows()` aus dem Paket `dplyr` kann auch Datensätze miteinander verbinden, die sich bzgl. der Variablen unterscheiden (Abschn. **??**).

```
1 13    f     CG   18  116       4
2 14    f     WL   31  101       4
3 15    m      T   21   99       1

> dfComb <- rbind(myDf1, dfNew)
> dfComb[11:15, ]
   id  sex  group  age   IQ  rating
11 11    m     CG   20   92       1
12 12    f      T   21   98       1
13 13    f     CG   18  116       4
14 14    f     WL   31  101       4
15 15    m      T   21   99       1
```

Beim Zusammenfügen mehrerer Datensätze besteht die Gefahr, Fälle doppelt aufzunehmen, wenn es Überschneidungen hinsichtlich der Beobachtungsobjekte gibt (Abschn. 4.4).

Liegen von denselben Beobachtungsobjekten zwei Datensätze `df1` und `df2` aus unterschiedlichen Variablen vor, können diese analog zum Anhängen einzelner Variablen an einen Datensatz mit `cbind(df1, df2)` so kombiniert werden, dass die Variablen nebeneinander angeordnet sind. Dabei ist sicherzustellen, dass die zu demselben Beobachtungsobjekt gehörenden Daten in derselben Zeile jedes Datensatzes stehen.

4.8 Datensätze zusammenführen

`merge()` kann flexibel zwei Datensätze zusammenführen, die sich nur teilweise in den Variablen oder in den Beobachtungsobjekten entsprechen. Auf diese Weise lassen sich einander ergänzende Informationen aus verschiedenen Datenquellen integrieren.

```
merge(x=⟨Datensatz 1⟩, y=⟨Datensatz 2⟩,
      by.x, by.y, by, all.x, all.y, all)
```

Zunächst sei die Situation betrachtet, dass die unter `x` und `y` angegebenen Datensätze Daten derselben Beobachtungsobjekte beinhalten, die über eine eindeutige ID identifiziert sind.[6] Dabei sollen einige Variablen bereits in beiden, andere Variablen hingegen nur in jeweils einem der beiden Datensätze vorhanden sein. Die in beiden Datensätzen gleichzeitig vorhandenen Variablen enthalten dann dieselbe Information, da die Daten von denselben Beobachtungsobjekten stammen. Ohne weitere Argumente ist das Ergebnis von `merge()` ein Datensatz, der jede der in `x` und `y` vorkommenden Variablen nur einmal enthält, identische Spalten werden also nur einmal aufgenommen.[7]

Beispiel sei ein Datensatz mit je zwei Messungen eines Merkmals an drei Personen mit eindeutiger ID. In einem zweiten Datensatz sind für jede ID die Informationen festgehalten, die

[6] Dabei zu beachtende Aspekte der Datenqualität bespricht Abschnitt **??**. Hinweise für den Fall uneindeutiger IDs geben Abschn. 3.12, Fußnote **??** und Abschn. **??**, Fußnote **??**.

[7] Zur Identifizierung gleicher Variablen werden die Spaltennamen mittels `intersect(names(x), names(y))` herangezogen. Bei Gruppierungsfaktoren ist es wichtig, dass sie in beiden Datensätzen dieselben Stufen in derselben Reihenfolge besitzen.

konstant über die Messwiederholungen sind – hier das Geschlecht und eine Gruppenzugehörigkeit.[8]

```
# Datensatz mit 2 Messwerten pro ID
> (IDDV <- data.frame(ID=factor(rep(1:3, each=2)),
+                     DV=round(rnorm(6, mean=100, sd=15))))
  ID  DV
1  1  93
2  1 105
3  2 112
4  2 101
5  3 109
6  3 101

# Datensatz mit Informationen, die pro ID konstant sind
> (IV <- data.frame(ID=factor(1:3),
+                   IV=factor(c("A", "B", "A")),
+                   sex=factor(c("f", "f", "m"))))
  ID IV sex
1  1  A   f
2  2  B   f
3  3  A   m
```

Als Ziel soll ein Datensatz erstellt werden, der die pro Person konstanten und veränderlichen Variablen integriert, wobei die Zuordnung von Informationen über die ID geschieht. Die pro Person über die Messwiederholungen konstanten Werte werden dabei von `merge()` automatisch passend oft wiederholt.

```
> merge(IDDV, IV)
  ID  DV IV sex
1  1  93  A   f
2  1 105  A   f
3  2 112  B   f
4  2 101  B   f
5  3 109  A   m
6  3 101  A   m
```

Verfügen `x` und `y` im Prinzip über teilweise dieselben Variablen, jedoch mit abweichender Bezeichnung, kann über die Argumente `by.x` und `by.y` manuell festgelegt werden, welche ihrer Variablen trotz ungleichen Namens übereinstimmen und deshalb nur einmal aufgenommen werden sollen. Als Wert für `by.x` und `by.y` muss jeweils ein Vektor mit Namen oder Indizes der übereinstimmenden Spalten eingesetzt werden. Beide Argumente `by.x` und `by.y` müssen gleichzeitig verwendet werden und müssen dieselbe Anzahl von gleichen Spalten bezeichnen. Haben beide Datensätze dieselben Variablennamen, kann auch auf das Argument `by` zurückgegriffen werden, das sich dann auf die Spalten beider Datensätze gleichzeitig bezieht. Es empfiehlt sich,

[8] Diese Datenstruktur entspricht einer *normalisierten* Datenbank mit mehreren *tables* zur Vermeidung von Redundanzen.

`by` in jedem Aufruf von `merge()` explizit zu setzen, um nicht zum Abgleich versehentlich gleich benannte Spalten zu verwenden, die eigentlich andere Informationen tragen.

Im folgenden Beispiel besitzt die in beiden Datensätzen eigentlich übereinstimmende ID-Variable unterschiedliche Namen. Die Spalte der Initialen wird künstlich als nicht übereinstimmende Variable gekennzeichnet, indem sie nicht an `by.x` und `by.y` übergeben wird.

```
> (dfA <- data.frame(ID=1:4, initials=c("AB", "CD", "EF", "GH"),
+                    IV1=c("-", "-", "+", "+"), DV1=c(10, 19, 11, 14)))
  ID  initials  IV1  DV1
1  1        AB    -   10
2  2        CD    -   19
3  3        EF    +   11
4  4        GH    +   14

# anderer Name für ID, initials gleich, zusätzlich: IV2, DV2
> (dfB <- data.frame(ID_mod=1:4, initials=c("AB", "CD", "EF", "GH"),
+                    IV2=c("A", "B", "A", "B"), DV2=c(91, 89, 92, 79)))
  ID_mod  initials  IV2  DV2
1      1        AB    A   91
2      2        CD    B   89
3      3        EF    A   92
4      4        GH    B   79

# initials nicht an by.x bzw. by.y übergeben -> doppelt übernommen
> merge(dfA, dfB, by.x="ID", by.y="ID_mod")
  ID initials.x IV1 DV1 initials.y IV2 DV2
1  1         AB   -  10         AB   A  91
2  2         CD   -  10         CD   B  89
3  3         EF   +  11         EF   A  92
4  4         GH   +  14         GH   B  79
```

Berücksichtigt werden bei dieser Art des Zusammenfügens nur Zeilen, bei denen die in beiden Datensätzen vorkommenden Variablen identische Werte aufweisen – im Kontext von Datenbanken wird dieses Verhalten als *inner join* bezeichnet.[9] Dabei werden alle Zeilen entfernt, deren Werte für die gemeinsamen Variablen zwischen den Datensätzen abweichen. Werden mit `by` Spalten als übereinstimmend gekennzeichnet, für die tatsächlich aber keine Zeile identische Werte aufweist, ist das Ergebnis deshalb ein leerer Datensatz.

```
> (dfC <- data.frame(ID=3:6, initials=c("EF", "GH", "IJ", "KL"),
+                    IV2=c("A", "B", "A", "B"), DV2=c(92,79,101,81)))
  ID  initials  IV2  DV2
1  3        EF    A   92
2  4        GH    B   79
3  5        IJ    A  101
4  6        KL    B   81
```

[9] Details zu verschiedenen join Operationen inkl. anschaulicher Visualisierungen geben Wickham und Grolemund (2017, Kap. 10): `http://r4ds.had.co.nz/relational-data.html`

```
> merge(dfA, dfC)
  ID  initials  IV1  DV1  IV2  DV2
1  3        EF    +   11    A   92
2  4        GH    +   14    B   79
```

Um das Weglassen solcher Zeilen zu verhindern, können die Argumente `all.x` bzw. `all.y` auf `TRUE` gesetzt werden. `all.x` bewirkt dann, dass alle Zeilen in `x`, die auf den übereinstimmenden Variablen andere Werte als in `y` haben, ins Ergebnis aufgenommen werden (im Kontext von Datenbanken ein *left outer join*). Die in `y` (aber nicht in `x`) enthaltenen Variablen werden für diese Zeilen auf `NA` gesetzt. Für das Argument `all.y` gilt dies analog (bei Datenbanken ein *right outer join*). Das Argument `all=TRUE` steht kurz für `all.x=TRUE` in Kombination mit `all.y=TRUE` (bei Datenbanken ein *full outer join*).

Im Beispiel sind die Werte bzgl. der übereinstimmenden Variablen in den ersten beiden Zeilen von `dfC` identisch mit jenen in `dfA`. Darüber hinaus enthält `dfC` jedoch auch zwei Zeilen mit Werten, die sich auf den übereinstimmenden Variablen von jenen in `dfA` unterscheiden. Um diese Zeilen im Ergebnis von `merge()` einzuschließen, muss deshalb `all.y=TRUE` gesetzt werden.

```
> merge(dfA, dfC, all.y=TRUE)
  ID  initials   IV1  DV1  IV2  DV2
1  3        EF     +   11    A   92
2  4        GH     +   14    B   79
3  5        IJ  <NA>   NA    A  101
4  6        KL  <NA>   NA    B   81
```

Analoges gilt für das Einschließen der ersten beiden Zeilen von `dfA`. Diese beiden Zeilen haben andere Werte auf den übereinstimmenden Variablen als die Zeilen in `dfC`. Damit sie im Ergebnis auftauchen, muss `all.x=TRUE` gesetzt werden.

```
> merge(dfA, dfC, all.x=TRUE, all.y=TRUE)
  ID  initials   IV1  DV1   IV2  DV2
1  1        AB     -   10  <NA>   NA
2  2        CD     -   19  <NA>   NA
3  3        EF     +   11     A   92
4  4        GH     +   14     B   79
5  5        IJ  <NA>   NA     A  101
6  6        KL  <NA>   NA     B   81
```

4.9 Organisationsform einfacher Datensätze ändern

Bisweilen weicht der Aufbau eines Datensatzes von dem beschriebenen ab, etwa wenn sich Daten derselben, in verschiedenen Bedingungen erhobenen Zielgrößen in unterschiedlichen Spalten befinden, wobei eine Spalte mit einer Bedingung korrespondiert. `stack()` ändert die Organisationsform der Daten so, dass der Datensatz eine Zeile pro Beobachtung, eine Spalte für die

Werte der Zielgröße in allen Bedingungen und eine Spalte mit dem Faktor enthält, dessen Stufen die zu jedem Wert gehörende Bedingung definieren.

`stack(x=⟨Datensatz / Liste⟩, select=⟨Spalten⟩)`

Unter `x` ist ein Datensatz oder eine Liste mit benannten Komponenten anzugeben. Bei einem Datensatz können die zu reorganisierenden Variablen über das Argument `select` ausgewählt werden. Es erwartet dafür einen Vektor mit Spaltenindizes oder Variablennamen. Das Ergebnis ist ein Datensatz, in dessen erster Variable `values` sich alle Werte der Zielgröße befinden. Diese Variable entsteht, indem die ursprünglichen Variablen von `x` durch `c(⟨Variable1⟩, ⟨Variable2⟩, ...)` aneinander gehängt werden. Die zweite Variable `ind` des erzeugten Datensatzes ist ein Faktor, dessen Stufen codieren, aus welcher Variable von `x` ein einzelner Wert der Variable `values` stammt. Hierzu werden die Variablennamen von `x` herangezogen.

```
> vec1  <- sample(1:10, 3, replace=TRUE)
> vec2  <- sample(1:10, 2, replace=TRUE)
> vec3  <- sample(1:10, 1, replace=TRUE)
> (lTmp <- list(cond1=vec1, cond2=vec2, cond3=vec3))
$cond1
[1] 7 9 1

$cond2
[1] 6 5

$cond3
[1] 2

> (res <- stack(lTmp))
  values   ind
1      7 cond1
2      9 cond1
3      1 cond1
4      6 cond2
5      5 cond2
6      2 cond3

> str(res)
'data.frame':   6 obs. of  2 variables:
 $ values: int  7 9 1 6 5 2
 $ ind   : Factor w/ 3 levels "cond1","cond2",..: 1 1 1 2 2 3
```

Die Funktion `unstack(⟨Datensatz⟩, form=⟨Modellformel⟩)` kehrt das Ergebnis von `stack()` um. Sie transformiert also einen Datensatz, der aus einer Variable mit den Werten der Zielgröße und einer Variable mit den zugehörigen Faktorstufen besteht, in einen Datensatz mit so vielen Spalten wie Faktorstufen. Dabei beinhaltet jede Spalte die zu einer Stufe gehörenden Werte der Zielgröße. Das Ergebnis von `unstack()` ist nur dann ein Datensatz, wenn alle Faktorstufen gleich häufig vorkommen, die resultierenden Variablen also dieselbe Länge aufweisen. Andernfalls ist das Ergebnis eine Liste.

```
> unstack(res)
$cond1
[1] 7 9 1

$cond2
[1] 6 5

$cond3
[1] 2
```

Kommen im Datensatz `x` mehrere Zielgrößen und Faktoren vor, kann in `unstack()` über eine an das Argument `form` zu übergebende *Modellformel* ⟨`AV`⟩ `~` ⟨`UV`⟩ festgelegt werden, welche Zielgröße (AV) ausgewählt und nach welchem Faktor (UV) die Trennung der Werte der Zielgröße vorgenommen werden soll (Abschn. 6.1).

```
# füge zwei neue Variablen zum Datensatz res hinzu
> Nj        <- 3
> res$IVnew <- factor(sample(rep(c("A", "B"), Nj), 2*Nj, replace=FALSE))
> res$DVnew <- sample(100:200, 2*Nj)
> unstack(res, form=DVnew ~ IVnew)
    A    B
1 183  193
2 129  115
3 142  140
```

Die Organisationsformen eines Datensatzes, zwischen denen `stack()` und `unstack()` wechseln, werden im Kontext von Daten aus Messwiederholungen als *Long*-Format und *Wide*-Format bezeichnet. Häufig sind die Datensätze dann jedoch zu komplex, um noch mit diesen Funktionen bearbeitet werden zu können. Die im folgenden Abschnitt beschriebene Funktion `reshape()` ist dann besser geeignet.

4.10 Organisationsform komplexer Datensätze ändern

Wurden an denselben Beobachtungsobjekten zu verschiedenen Messzeitpunkten Daten derselben Zielgröße erhoben, können die Werte jeweils eines Messzeitpunkts als zu einer eigenen Variable gehörend betrachtet werden. In einem Datensatz findet sich jede dieser Variablen dann in einer separaten Spalte. Diese Organisationsform folgt dem Prinzip, dass pro Zeile die Daten jeweils eines Beobachtungsobjekts aus verschiedenen Variablen stehen. Eine solche Struktur wird als Wide-Format bezeichnet, weil der Datensatz durch mehr Messzeitpunkte mehr Spalten hinzugewinnt, also breiter wird. Das Wide-Format entspricht einer multivariaten Betrachtungsweise von Daten aus Messwiederholungen (Abschn. 8.5.2, **??**). Es setzt voraus, dass die Objekte zu denselben Zeitpunkten beobachtet wurden und damit der Messzeitpunkt pro Spalte konstant ist.

Für die univariat formulierte Analyse von abhängigen Daten (Abschn. 8.5) ist jedoch oft die Organisation im Long-Format notwendig. Die zu den verschiedenen Messzeitpunkten gehörenden

Werte eines Beobachtungsobjekts stehen hier in separaten Zeilen. Auf diese Weise beinhalten mehrere Zeilen Daten desselben Beobachtungsobjekts. Der Name des Long-Formats leitet sich daraus ab, dass mehr Messzeitpunkte zu mehr Zeilen führen, der Datensatz also länger wird. Wichtig bei Verwendung dieses Formats ist zum einen das Vorhandensein eines Faktors, der codiert, von welchem Objekt eine Beobachtung stammt. Diese Variable ist dann jeweils über so viele Zeilen konstant, wie es Messzeitpunkte gibt. Zum anderen muss ein Faktor existieren, der den Messzeitpunkt codiert. Das Long-Format eignet sich auch für Situationen, in denen mehrere Objekte zu verschiedenen Zeitpunkten unterschiedlich häufig beobachtet wurden.

4.10.1 Vorgehen bei einem Messwiederholungsfaktor

Im Beispiel sei an vier Personen eine Zielgröße zu drei Messzeitpunkten erhoben worden. Bei zwei der Personen sei dies in Bedingung A, bei den anderen beiden in Bedingung B einer Intervention geschehen. Damit liegen zwei Gruppierungsfaktoren vor, zum einen als Intra-Gruppen Faktor der Messzeitpunkt, zum anderen ein Zwischen-Gruppen Faktor (*Split-Plot* Design, Abschn. 8.8). Zunächst soll demonstriert werden, wie sich das Long-Format manuell aus gegebenen Vektoren herstellen lässt. Soll mit einem solchen Datensatz etwa eine univariate Varianzanalyse mit Messwiederholung gerechnet werden, muss sowohl die Personen- bzw. Blockzugehörigkeit eines Messwertes als auch der Messzeitpunkt jeweils in einem Faktor gespeichert sein.

```
> Nj    <- 2                              # Gruppengröße
> P     <- 2                              # Anzahl Gruppen
> Q     <- 3                              # Anzahl Messzeitpunkte
> id    <- 1:(P*Nj)                       # Blockzugehörigkeit
> DV_t1 <- round(rnorm(P*Nj, -1, 1), 2)  # Zielgröße zu t1
> DV_t2 <- round(rnorm(P*Nj,  0, 1), 2)  # Zielgröße zu t2
> DV_t3 <- round(rnorm(P*Nj,  1, 1), 2)  # Zielgröße zu t3
> IVbtw <- factor(rep(c("A", "B"), Nj))  # Gruppe: between

# Datensatz im Wide-Format
> (dfW1 <- data.frame(id, IVbtw, DV_t1, DV_t2, DV_t3))
  id  IVbtw  DV_t1  DV_t2  DV_t3
1  1      A  -1.64   0.01   1.31
2  2      B  -0.81  -1.23   1.59
3  3      A  -1.43  -0.80   0.68
4  4      B  -1.79  -0.13  -0.14

# Variablen für Long-Format
> idL    <- factor(rep(id, Q))           # Faktor ID-Code
> DVl    <- c(DV_t1, DV_t2, DV_t3)       # gemeinsamer Vektor Zielgröße
> IVwth  <- factor(rep(1:3, each=P*Nj))  # Zeitpunkt: within
> IVbtwL <- rep(IVbtw, times=Q)          # Gruppe: between

# Datensatz im Long-Format
> dfL1a <- data.frame(id=idL, IVbtw=IVbtwL, IVwth=IVwth, DV=DVl)
> dfL1a[order(dfL1a$id), ]               # sortierte Ausgabe
```

```
   id  IVbtw  IVwth     DV
1   1      A      1  -1.64
5   1      A      2   0.01
9   1      A      3   1.31
2   2      B      1  -0.81
6   2      B      2  -1.23
10  2      B      3   1.59
3   3      A      1  -1.43
7   3      A      2  -0.80
11  3      A      3   0.68
4   4      B      1  -1.79
8   4      B      2  -0.13
12  4      B      3  -0.14
```

`reshape()` bietet die Möglichkeit, einen Datensatz ohne manuelle Zwischenschritte zwischen Wide- und Long-Format zu transformieren.

```
reshape(data=⟨Datensatz⟩, varying, timevar="time",
        idvar="id", direction=c("wide", "long"), v.names="⟨Name⟩")
```

- Zunächst wird der Datensatz unter `data` eingefügt. Um ihn vom Wide- ins Long-Format zu transformieren, muss das Argument `direction="long"` gesetzt werden.
- Daneben ist unter `varying` anzugeben, welche Variablen im Wide-Format dieselbe Zielgröße zu unterschiedlichen Messzeitpunkten repräsentieren. Die Variablen werden im Long-Format über unterschiedliche Ausprägungen der neu gebildeten Variable `time` identifiziert, deren Name über das Argument `timevar` auch selbst festgelegt werden kann. `varying` benötigt eine Liste, deren Komponenten Vektoren mit Variablennamen oder Spaltenindizes sind. Jeder Vektor gibt dabei eine Gruppe von Variablen an, die jeweils zu einer Zielgröße gehören.[10]
- Besitzt der Datensatz eine Variable, die codiert, von welcher Person ein Wert stammt, kann sie im Argument `idvar` genannt werden. Andernfalls wird eine solche Variable auf Basis der Zeilenindizes gebildet und trägt den Namen `id`. Auch andere Variablen von `data`, die pro Messzeitpunkt zwischen den Personen variieren, gelten als `idvar`, dies trifft etwa auf die Ausprägung von Zwischen-Gruppen Faktoren zu. Im Fall mehrerer solcher Variablen sind diese als Vektor von Variablennamen anzugeben.
- Der Name der Variable im Long-Format mit den Werten der Zielgröße kann über das Argument `v.names` bestimmt werden.

```
> dfL1b <- reshape(dfWide, varying=c("DV_t1", "DV_t2", "DV_t3"),
+                  direction="long", idvar=c("id", "IVbtw"),
+                  v.names="DV")

# erste Zeilen der sortierten Ausgabe, identisch zu dfL1a
> head(dfL1b[order(dfL1b$id), ])
```

[10] Im Fall zweier Zielgrößen, für die jeweils eine Gruppe von zwei Spalten im Wide-Format vorhanden ist, könnte das Argument also `varying=list(c("DV1_t1", "DV1_t2"), c("DV2_t1", "DV2_t2"))` lauten.

```
      id IVbtw time    DV
1.A.1  1     A    1 -1.68
1.A.2  1     A    2  0.78
1.A.3  1     A    3  2.73
2.B.1  2     B    1 -2.52
2.B.2  2     B    2  0.69
2.B.3  2     B    3  0.80
```

Die Variablen in der Rolle von `idvar` und `time` sollten Objekte der Klasse `factor` sein. Da `reshape()` die Variablen aber als numerische Vektoren generiert, müssen sie ggf. manuell umgewandelt werden mit:

- ⟨Datensatz⟩$⟨Variable⟩ <- factor(⟨Datensatz⟩$⟨Variable⟩)

Ist der Datensatz vom Long- ins Wide-Format zu transformieren, muss `direction="wide"` gesetzt werden. Für das Argument `v.names` wird jene Variable genannt, die die Werte der Zielgröße im Long-Format über alle Messzeitpunkte hinweg speichert. Diese Variable wird im Wide-Format auf mehrere Spalten aufgeteilt, die den Messzeitpunkten entsprechen. Dafür ist unter `timevar` anzugeben, welche Variable den Messzeitpunkt codiert. Unter `idvar` sind jene Variablen zu nennen, deren Werte die Daten der Zielgröße eines Objekts zuordnen bzw. pro Messzeitpunkt über die Objekte variieren, etwa weil sie die Ausprägung von Zwischen-Gruppen Faktoren darstellen.

```
> reshape(dfL1b, v.names="DV", timevar="IVwth",
+         idvar=c("id", "IVbtw"), direction="wide")
  id  IVbtw   DV.1   DV.2   DV.3
1  1      A  -1.64   0.01   1.31
2  2      B  -0.81  -1.23   1.59
3  3      A  -1.43  -0.80   0.68
4  4      B  -1.79  -0.13  -0.14
```

4.10.2 Vorgehen bei mehreren Messwiederholungsfaktoren

Ist eine Zielgröße an denselben Objekten mehrfach in den kombinierten Bedingungen zweier Intra-Gruppen Faktoren erhoben worden (Abschn. 8.7), ist das einfachste Vorgehen zum Wechsel vom Wide- ins Long-Format, zunächst beide Faktoren mit `interaction()` in einen einzigen Faktor umzuwandeln, der alle möglichen Stufenkombinationen enthält. Dann lässt sich wie oben beschrieben fortfahren. Alternativ ist `reshape()` zweimal anzuwenden. Im ersten Schritt werden die zu unterschiedlichen Bedingungen des ersten Faktors gehörenden Spalten zusammengefasst, im zweiten Schritt diejenigen des zweiten.

Im Beispiel seien an vier Personen in jeder Stufenkombination eines Faktors mit drei und eines Faktors mit zwei Messzeitpunkten Werte einer Zielgröße erhoben worden.

```
> N    <- 4                                    # Anzahl Personen
> id   <- 1:N                                  # ID-Code
> t_11 <- round(rnorm(N,  8,  2), 2)           # Zielgröße zu t1-1
> t_12 <- round(rnorm(N, 10,  2), 2)           # Zielgröße zu t1-2
```

```
> t_21 <- round(rnorm(N, 13,  2), 2)              # Zielgröße zu t2-1
> t_22 <- round(rnorm(N, 15,  2), 2)              # Zielgröße zu t2-2
> t_31 <- round(rnorm(N, 13,  2), 2)              # Zielgröße zu t3-1
> t_32 <- round(rnorm(N, 15,  2), 2)              # Zielgröße zu t3-2

# Datensatz im Wide-Format
> (dfW2 <- data.frame(id, t_11, t_12, t_21, t_22, t_31, t_32))
  id t_11  t_12  t_21  t_22  t_31  t_32
1  1 9.24 11.89 11.03 13.54 13.60 17.02
2  2 7.17  9.13 12.36 15.40 13.02 12.90
3  3 6.07  8.41  8.34 16.20 14.49 14.19
4  4 6.21  6.99 10.40 14.94 15.35 17.60

# Transformation ins Long-Format bzgl. IV1
> (dfL2_IV1 <- reshape(dfW2, varying=list(c("t_11", "t_21", "t_31"),
+                                         c("t_12", "t_22", "t_32")),
+                      direction="long", timevar="IV1", idvar="id",
+                      v.names=c("IV2-1", "IV2-2")))
    id IV1 IV2-1 IV2-2
1.1  1   1  9.24 11.89
2.1  2   1  7.17  9.13
3.1  3   1  6.07  8.41
4.1  4   1  6.21  6.99
1.2  1   2 11.03 13.54
2.2  2   2 12.36 15.40
3.2  3   2  8.34 16.20
4.2  4   2 10.40 14.94
1.3  1   3 13.60 17.02
2.3  2   3 13.02 12.90
3.3  3   3 14.49 14.19
4.3  4   3 15.35 17.60
```

Da `IV1` nun wie `id` pro Messzeitpunkt von `IV2` über die Personen variiert, muss die Variable im zweiten Schritt ebenfalls unter `idvar` genannt werden.

```
> dfL2_IV1_IV2 <- reshape(dfL2_IV1, varying=c("IV2-1", "IV2-2"),
+                         direction="long", timevar="IV2",
+                         idvar=c("id", "IV1"),  v.names="DV")

> head(dfL2_IV1_IV2)
      id IV1 IV2    DV
1.1.1  1   1   1  9.24
2.1.1  2   1   1  7.17
3.1.1  3   1   1  6.07
4.1.1  4   1   1  6.21
1.2.1  1   2   1 11.03
2.2.1  2   2   1 12.36
```

Auch die umgekehrte Transformation vom Long- ins Wide-Format benötigt zwei Schritte, wenn zwei Intra-Gruppen Faktoren vorhanden sind.

```
# Schritt 1: Stufen der IV2 in Spalten aufteilen
> dfW2_IV2 <- reshape(dfL2_IV1_IV2, v.names="DV", timevar="IV2",
+                     idvar=c("id", "IV1"), direction="wide")

# Schritt 2: Stufen der IV1 in Spalten aufteilen
> (dfW2_IV1_IV2 <- reshape(dfW2_IV2, v.names=c("DV.1", "DV.2"),
+                          timevar="IV1", idvar="id", direction="wide"))
      id DV.1.1 DV.2.1 DV.1.2 DV.2.2 DV.1.3 DV.2.3
1.1.1  1   9.24  11.89  11.03  13.54  13.60  17.02
2.1.1  2   7.17   9.13  12.36  15.40  13.02  12.90
3.1.1  3   6.07   8.41   8.34  16.20  14.49  14.19
4.1.1  4   6.21   6.99  10.40  14.94  15.35  17.60
```

4.11 Daten getrennt nach Gruppen auswerten und aggregieren

Nachdem Datensätze mit `split()` geteilt wurden (Abschn. 4.6), liegen die Daten aus den Bedingungen separat vor und können getrennt verarbeitet werden, etwa zur Berechnung von Kennwerten pro Gruppe. Bei numerischen Kennwerten eignet sich `sapply()`, um dieselbe Funktion auf jede Komponente einer Liste anzuwenden. Da jede Komponente der Liste ein Datensatz ist, muss selbst eine geeignete Funktion erstellt werden, die als Argument einen Datensatz akzeptiert und daraus den gewünschten Kennwert berechnet (Abschn. 12.2).

```
# berechne Mittelwert von IQ getrennt nach Gruppen
> dat_spl <- split(myDf1, myDf1$group)     # teile Datensatz in Gruppen
> sapply(dat_spl, function(x) { c(M=mean(x$IQ)) })
  CG.M     T.M   WL.M
105.50  101.00  85.25
```

Mit diesem Ansatz können auch gleichzeitig mehrere numerische Kennwerte pro Gruppe berechnet werden. Die einzelnen Kennwerte stehen in der ausgegebenen Matrix dann in den Zeilen, die zugehörige Gruppe ist durch die Spaltennamen gekennzeichnet. Da Ergebnistabellen oft so aufgebaut sind, dass eine Zeile pro Gruppe und eine Spalte pro Kennwert steht, ist das Ergebnis noch zu transponieren.

```
# berechne Mittelwert und Streuung von IQ getrennt nach Gruppen
> (m_sd <- sapply(dat_spl, function(x) {
+                 c(M=mean(x$IQ), SD=sd(x$IQ)) }))
          CG          T        WL
M  105.50000 101.000000 85.250000
SD  14.38749   8.406347  4.425306

> t(m_sd)                                          # transponiert
        M        SD
CG 105.50 14.387495
```

```
T  101.00  8.406347
WL  85.25  4.425306
```

Sollen die Gruppen durch Kombination der Stufen aus mehreren Faktoren gebildet werden, ist der Aufruf von `split()` entsprechend zu erweitern.

```
# teile Datensatz nach zwei Faktoren auf
> dat_spl2 <- split(myDf1, list(myDf1$sex, myDf1$group), drop=TRUE)
> sapply(dat_spl2, function(x) { c(M=mean(x$IQ), SD=sd(x$IQ)) }) # ...
```

Wenn getrennt nach Gruppenzugehörigkeit mehrere Kennwerte berechnet werden sollen, die nicht alle numerisch sind, kann auf `lapply()` ausgewichen und die erzeugte Liste manuell mit `do.call("rbind", ⟨Liste⟩)` zu einem Datensatz verbunden werden.

```
# Funktion: Datensatz Gruppenzugehörigkeiten & gruppenweise Kennwerte
> aggr_fun <- function(x) {
+     data.frame(Sex=unique(x$sex),
+                Group=unique(x$group),
+                M=mean(x$IQ),
+                SD=sd(x$IQ))
+ }

> m_sdL <- lapply(dat_spl2, aggr_fun)               # wende Funktion an
> do.call("rbind", m_sdL)        # verbinde erzeugte Liste zu Datensatz
     Sex Group      M        SD
f.CG   f    CG 113.00        NA
m.CG   m    CG 103.00 16.522712
f.T    f     T 105.00  9.899495
m.T    m     T  97.00  7.071068
m.WL   m    WL  85.25  4.425306
```

Um für Variablen eines Datensatzes Kennwerte nicht nur über alle Beobachtungen hinweg, sondern getrennt nach Gruppen zu berechnen, ist auch die Funktion `aggregate()` geeignet.

`aggregate(⟨Modellformel⟩, FUN=⟨Funktion⟩, data=⟨Datensatz⟩)`

Sie berechnet mit der Funktion `FUN` einen Kennwert für eine Variable ⟨`AV`⟩ getrennt nach Gruppen, die durch einen Faktor ⟨`UV`⟩ definiert werden. Beide Variablen müssen in einer Modellformel ⟨`AV`⟩ `~` ⟨`UV`⟩ aufgeführt sein (Abschn. 6.1). Stammen die Variablen aus einem Datensatz, ist dieser unter `data` zu nennen.

Die Modellformel erweitert sich zu ⟨`AV`⟩ `~` ⟨`UV1`⟩ `+` ⟨`UV2`⟩ `+ ...`, wenn die Gruppen durch Kombination mehrerer Faktoren gebildet werden sollen. Um denselben Kennwert jeweils von mehreren Variablen gleichzeitig zu berechnen, ist die Modellformel multivariat zu formulieren, d. h. links der `~` mit `cbind(⟨Variable1⟩, ⟨Variable2⟩, ...)`.

```
# Mitelwert jeweils von age und IQ getrennt nach sex und group
> aggregate(cbind(age, IQ) ~ sex + group, FUN=mean, data=myDf1)
  sex group   age     IQ
1   f    CG 24.00 113.00
```

```
2   m    CG 25.00 103.00
3   f     T 23.50 105.00
4   m     T 31.00  97.00
5   m    WL 27.25  85.25
```

Den Kennwert über alle Beobachtungen hinweg erhält man, wenn rechts der Tilde nur `~ 1` steht.

```
# Gesamtmittelwert jeweils von age und IQ
> aggregate(cbind(age, IQ) ~ 1, FUN=mean, data=myDf1)
       age    IQ
1 26.41667 97.25
```

Um gleichzeitig mehrere für jede Gruppe getrennt berechnete Kennwerte in einem Datensatz zusammenzustellen ist ein zweistufiges Vorgehen möglich. Zunächst werden verschiedene Kennwerte pro Gruppe wie gezeigt einzeln berechnet. Anschließend lassen sich die erzeugten Datensätze mit `merge()` zusammenzufügen. Inhaltlich passende Spaltennamen sind dabei mit dem Argument `suffixes` festzulegen.

```
# separate Datensätze für Mittelwert und Streuung je Gruppe
> d_mean <- aggregate(cbind(age, IQ) ~ sex+group, FUN=mean, data=myDf1)
> d_sd   <- aggregate(cbind(age, IQ) ~ sex+group, FUN=sd,   data=myDf1)

# verbinde Datensätze und benenne dabei Spalten passend
> merge(d_mean, d_sd, by=c("sex", "group"), suffixes = c(".M",".SD"))
  sex group age.M   IQ.M   age.SD     IQ.SD
1   f    CG 24.00 113.00       NA        NA
2   f     T 23.50 105.00 3.535534  9.899495
3   m    CG 25.00 103.00 5.000000 16.522712
4   m     T 31.00  97.00 4.242641  7.071068
5   m    WL 27.25  85.25 6.020797  4.425306
```

4.12 Funktionen auf Variablen anwenden

Der folgende Abschnitt stellt dar, wie allgemein Funktionen auf einzelne Variablen aus Datensätzen oder auf Gruppen von solchen Variablen angewendet werden können.

`lapply(X=⟨Liste⟩, FUN=⟨Funktion⟩, ...)` (*list apply*) verallgemeinert die Funktionsweise von `apply()` (Abschn. 3.7.5) auf Listen und Datensätze. `X` kann ein Vektor, eine Liste oder ein Datensatz sein. Bei einem Vektor wird die an `FUN` übergebene Funktion auf jedes Element angewendet, bei Listen dagegen auf jede Komponente bzw. im Fall eines Datensatzes auf jede Variable. Das Ergebnis ist eine Liste mit ebenso vielen Komponenten wie `X` Elemente bzw. Komponenten enthält. Ist `FUN` nur sinnvoll auf numerische Variablen anwendbar, können diese aus einem Datensatz zunächst mit `subset(..., select=⟨Indizes⟩)` extrahiert werden.

```
# Mittelwerte der numerischen Variablen
> numDf   <- subset(myDf1, select=c(age, IQ, rating))
```

```
> (myList <- lapply(numDf, mean))
$age
[1] 26.41667

$IQ
[1] 97.25

$rating
[1] 2.583333
```

`sapply(⟨Vektor⟩, FUN=⟨Funktion⟩)` (*simplified apply*) arbeitet wie `lapply()`, gibt aber nach Möglichkeit keine Liste, sondern einen einfacher zu verarbeitenden Vektor mit benannten Elementen aus. Gibt `FUN` pro Aufruf mehr als einen Wert zurück, ist das Ergebnis eine Matrix, deren Zeilen aus diesen Werten gebildet sind. Die Rückgabewerte sollten dann alle denselben Datentyp besitzen.[11]

```
# range der numerischen Variablen
> sapply(numDf, range)
     age   IQ  rating
[1,]  20   82       0
[2,]  35  122       5
```

Durch die Ausgabe eines Vektors eignet sich `sapply()` z. B. dazu, aus einem Datensatz jene Variablen zu extrahieren, die eine bestimmte Bedingung erfüllen – etwa einen numerischen Datentyp besitzen. Das Ergebnis kann anschließend als Indexvektor für die Spalten verwendet werden, um eine Gruppe von Variablen mit einer bestimmten Eigenschaft auszuwählen.[12]

```
> (numIdx <- sapply(myDf1, is.numeric))     # numerische Variable?
  id    sex  group   age    IQ  rating
TRUE  FALSE  FALSE  TRUE  TRUE    TRUE

> dataNum <- subset(myDf1, select=numIdx)   # nur numerische Variablen
> head(dataNum, n=3)
  id  age   IQ  rating
1  1   26  112       1
2  2   30  122       3
3  3   25   95       5

> data.matrix(dataNum)   # wandle numerische Variablen in Matrix um ...
```

Mit `do.call()` ist eine etwas andere automatisierte Anwendung einer Funktion auf die Komponenten einer Liste bzw. auf die Variablen eines Datensatzes möglich. Während `lapply()`

[11] Die verwandte Funktion `vapply()` unterscheidet sich nur dadurch, dass sie als letztes Argument zusätzlich den Prototypen eines Rückgabewerts von `FUN` erwartet, etwa `numeric(1)`. Dies macht `vapply()` im Kontext selbst geschriebener Funktionen (Abschn. 12.2) weniger fehleranfällig.

[12] `sapply()` ist auch für jene Fälle nützlich, in denen auf jedes Element eines Vektors eine Funktion angewendet werden soll, diese Funktion aber nicht vektorisiert ist – d. h. als Argument nur einen einzelnen Wert, nicht aber Vektoren akzeptiert. In diesem Fall betrachtet `sapply()` jedes Element des Vektors als eigene Variable, die nur einen Wert beinhaltet.

eine Funktion so häufig aufruft, wie Variablen vorhanden sind und dabei jeweils eine Variable als Argument übergibt, geschieht dies bei `do.call()` nur einmal, dafür aber mit mehreren Argumenten.

`do.call(what="⟨Funktionsname⟩", args=⟨Liste⟩)`

Unter `what` ist die aufzurufende Funktion zu nennen, unter `args` deren Argumente in Form einer Liste, wobei jede Komponente von `args` ein Funktionsargument liefert. Ist von vornherein bekannt, welche und wie viele Argumente `what` erhalten soll, könnte `do.call()` auch durch einen einfachen Aufruf von `⟨Funktion⟩(⟨Liste⟩[[1]], ⟨Liste⟩[[2]], ...)` ersetzt werden, nachdem die Argumente als Liste zusammengestellt wurden. Der Vorteil der Konstruktion eines Funktionsaufrufs aus dem Funktionsnamen einerseits und den Argumenten andererseits tritt jedoch dann zutage, wenn sich Art oder Anzahl der Argumente erst zur Laufzeit der Befehle herausstellen – etwa weil die Liste selbst erst mit vorangehenden Befehlen dynamisch erzeugt wurde.

Aus einer von `lapply()` zurückgegebenen Liste ließe sich damit wie folgt ein Vektor machen, wie ihn auch `sapply()` zurückgibt:

```
# äquivalent zu
# c(id=myList[[1]], age=myList[[2]], IQ=myList[[3]], rating=myList[[4]])
> do.call("c", myList)
      id        age         IQ    rating
6.500000  26.416667  97.250000  2.583333
```

Sind die Komponenten von `args` benannt, behalten sie ihren Namen bei der Verwendung als Argument für die unter `what` genannte Funktion bei. Damit lassen sich beliebige Funktionsaufrufe samt zu übergebender Daten und weiterer Optionen konstruieren: Alle späteren Argumente werden dafür als Komponenten in einer Liste gespeichert, wobei die Komponenten die Namen erhalten, die die Argumente der Funktion `what` tragen.

```
> work  <- factor(sample(c("home", "office"), 20, replace=TRUE))
> hiLo  <- factor(sample(c("hi", "lo"),       20, replace=TRUE))
> group <- factor(sample(c("A", "B"),         20, replace=TRUE))
> tab   <- table(work, hiLo, group)    # 3D-Kreuztabelle der Faktoren

# wandle 3D-Kreuztabelle mit ftable um -> lege fest, welche Faktoren
# in Zeilen (row.vars), welche in Spalten (col.vars) stehen sollen
> argLst <- list(tab, row.vars="work", col.vars=c("hiLo", "group"))
> do.call("ftable", argLst)
     hiLo    hi     lo
     group  A  B   A  B
work
home        1  3   1  3
office      2  3   2  5
```

4.13 Funktionen für mehrere Variablen anwenden

`lapply()` und `sapply()` wenden eine Funktion nacheinander auf jeweils eine Variable eines Datensatzes bzw. einer Liste an. `mapply()` verallgemeinert dieses Prinzip auf Funktionen, die aus mehr als einer einzelnen Variable Kennwerte berechnen. Dies ist insbesondere für viele inferenzstatistische Tests der Fall, die etwa in zwei Variablen vorliegende Daten aus zwei Stichproben hinsichtlich verschiedener Kriterien vergleichen.

```
mapply(FUN=⟨Funktion⟩, ⟨Datensatz 1⟩, ⟨Datensatz 2⟩, ...,
       MoreArgs=⟨Liste mit Optionen für FUN⟩)
```

Die anzuwendende Funktion ist als erstes Argument `FUN` zu nennen. Es folgen so viele Datensätze oder Listen, wie `FUN` Eingangsgrößen benötigt. Im Beispiel einer Funktion für zwei Variablen verrechnet die Funktion schrittweise zunächst die erste Variable des ersten zusammen mit der ersten Variable des zweiten Datensatzes, dann die zweite Variable des ersten zusammen mit der zweiten Variable des zweiten Datensatzes, etc. Sollen an `FUN` weitere Argumente übergeben werden, kann dies mit dem Argument `MoreArgs` in Form einer Liste geschehen.

Im Beispiel soll ein t-Test für zwei unabhängige Stichproben für jeweils alle Variablen-Paare zweier Datensätze berechnet werden (Abschn. 8.3). Dabei soll im t-Test eine gerichtete Hypothese getestet (`alternative="less"`) und von Varianzhomogenität ausgegangen werden (`var.equal=TRUE`). Die Ausgabe wird hier verkürzt dargestellt.

```
> N     <- 100
> x1    <- rnorm(N, 10, 10)         # Variablen für ersten Datensatz
> y1    <- rnorm(N, 10, 10)
> x2    <- x1 + rnorm(N, 5, 4)      # Variablen für zweiten Datensatz
> y2    <- y1 + rnorm(N, 10, 4)
> myDf2 <- data.frame(x1, y1)
> myDf3 <- data.frame(x2, y2)
> mapply(t.test, myDf2, myDf3,
+        MoreArgs=list(alternative="less", var.equal=TRUE))
            x1
statistic   -1.925841
parameter   198
p.value     0.02777827
alternative "less"
method      "Two Sample t-test"     # Ausgabe gekürzt ...

            y1
statistic   -33.75330
parameter   198
p.value     2.291449e-84
alternative "less"
method      "Two Sample t-test"     # Ausgabe gekürzt ...
```

Kapitel 11

Diagramme mit dem Basisumfang von R erstellen

Dieses Kapitel zeigt alternativ zum gedruckten Kapitel 11 (*Diagramme mit `ggplot2` erstellen*), wie dieselben Aufgaben mit dem Basisumfang von R bewältigt werden können.

In R werden zwei Arten von Grafikfunktionen unterschieden: *High-Level*-Funktionen erstellen eigenständig ein komplettes Diagramm inkl. Achsen, während *Low-Level*-Funktionen lediglich ein bestimmtes Element einem bestehenden Diagramm hinzufügen. Einen kurzen Überblick über die Gestaltungsmöglichkeiten vermittelt `demo(graphics)`.

11.1 Grafik-Devices

11.1.1 Aufbau und Verwaltung von Grafik-Devices

Die Ausgabe von Befehlen zum Erstellen einer Grafik kann in verschiedenen Ausgabekanälen, sog. *Devices* erfolgen. Ein Device verhält sich wie ein leeres Blatt Papier, auf dem mit Grafikfunktionen einzelne Inhalte eingezeichnet werden. In der Voreinstellung ist ein Device ein separates Grafikfenster, es können aber etwa auch Dateien in verschiedenen Grafikformaten als Device dienen.

Sofern noch kein Grafikfenster existiert, öffnet es sich mit Eingabe des ersten High-Level-Grafikbefehls automatisch – in RStudio im *Plots*-Tab. Hier werden dann alle weiteren Ausgaben grafischer Funktionen hinein gezeichnet, wobei im Fall von High-Level-Funktionen ein ggf. bereits vorhandener Inhalt gelöscht wird. Soll für die Ausgabe einer Grafikfunktion zusätzlich zu bereits bestehenden ein neues, zunächst leeres Fenster geöffnet werden, geschieht dies unter Windows mit

```
> windows(width=⟨Breite⟩, height=⟨Höhe⟩)
```

Unter MacOS ist `quartz()` und unter Linux `x11()` der äquivalente Befehl. Unabhängig vom Betriebssystem erzielt `dev.new()` in der Voreinstellung denselben Effekt. Breite und Höhe des Fensters können über die Argumente `width` und `height` in der Einheit inch bestimmt werden. Bei mehreren geöffneten Devices sind alle bis auf eines inaktiv, das im Fenstertitel die Bezeichnung *(ACTIVE)* trägt. Die Bezeichnung bedeutet, dass in dieses Device die Ausgabe der folgenden Grafikfunktion gezeichnet wird, während die Inhalte der anderen Devices unverändert bleiben.

Das aktive Device wird mit `dev.off()` geschlossen. Handelt es sich um ein Grafikfenster, hat dies denselben Effekt, wie das Fenster per Mausklick zu schließen. Die Ausgabe von `dev.off()`

gibt an, welches fortan das aktive Device ist. Alle offenen Devices lassen sich gleichzeitig mit `graphics.off()` schließen.

```
> dev.new(); dev.new()                    # zwei Devices öffnen
> dev.off()                               # aktuelles Device schließen
windows
      2

> graphics.off()                          # alle Devices schließen
```

11.1.2 Grafiken speichern

Alles, was sich in einem Grafikfenster anzeigen lässt, kann auch als Datei in unterschiedlichen Formaten gespeichert werden – in RStudio im *Plots*-Tab mit dem Eintrag *Export*. Grafiken lassen sich auch ohne den Umweg eines Grafikfensters in Dateien speichern. Unabhängig davon, in welchem Format dies geschehen soll, sind dafür drei Arbeitsschritte notwendig:

- Zunächst muss die Datei als Ausgabekanal (also als aktives Device) festgelegt werden. Dazu dient etwa die `pdf()` Funktion, wenn die Grafik im PDF-Format zu speichern ist.
- Es folgen Befehle zum Erstellen von Diagrammen oder Einfügen von Grafikelementen, deren Ausgabe dann nicht auf dem Bildschirm erscheint, sondern direkt in die Datei umgeleitet wird.
- Schließlich ist der Befehl `dev.off()` oder `graphics.off()` notwendig, um die Ausgabe in die Datei zu beenden und das Device zu schließen.

Als Dateiformate stehen viele der üblichen bereit, vgl. `?device` für eine Aufstellung. Als Beispiel seien hier die Funktionen `pdf()` und `jpeg()` betrachtet.

```
> pdf(file="⟨Dateiname⟩", width=⟨Breite⟩, height=⟨Höhe⟩)
> jpeg(filename="⟨Dateiname⟩", width=⟨Breite⟩, height=⟨Höhe⟩,
+      units="px", quality=⟨Bildqualität⟩)
```

Unter `file` bzw. `filename` ist der Name der Ausgabedatei einzutragen – ggf. inkl. einer Pfadangabe (vgl. Kapitel 1.2.3). Mit `width` und `height` wird die Größe der Grafik kontrolliert. Beide Angaben sind bei `pdf()` in der Einheit inch, während bei `jpeg()` über das Argument `units` festgelegt werden kann, auf welche Maßeinheit sie sich beziehen. Voreinstellung ist die Anzahl der pixel, als Alternativen stehen `in` (inch), `cm` und `mm` zur Auswahl. Schließlich kann bei Bildern im JPEG-Format festgelegt werden, wie stark die Daten komprimiert werden sollen, wobei die Kompression mit einem Verlust an Bildinformationen verbunden ist. Das Argument `quality` erwartet einen sich auf die höchstmögliche Bildqualität beziehenden Prozentwert.

```
> pdf("pdf_test.pdf")                     # Device öffnen (mit Dateinamen)
> plot(1:10, rnorm(10))                   # Grafik einzeichnen
> dev.off()                               # Device schließen
```

11.2 Streu- und Liniendiagramme

In zweidimensionalen Streudiagrammen werden mit `plot()` Wertepaare in Form von Punkten in einem kartesischen Koordinatensystem dargestellt, wobei ein Wert die Position des Punkts entlang der x-Achse und der andere Wert die Position des Punkts entlang der y-Achse bestimmt. Die Punkte können dabei für ein Liniendiagramm durch Linien verbunden oder für ein Streudiagramm als Punktwolke belassen werden.

11.2.1 Streudiagramme mit `plot()`

```
> plot(x=⟨Vektor⟩, y=⟨Vektor⟩, type="⟨Option⟩", main="⟨Diagrammtitel⟩",
+      sub="⟨Untertitel⟩")
```

Unter `x` und `y` sind die x- bzw. y-Koordinaten der Punkte jeweils als Vektor einzutragen. Wird nur ein Vektor angegeben, werden seine Werte als y-Koordinaten interpretiert und die x-Koordinaten durch die Indizes des Vektors gebildet. Das Argument `type` hat mehrere mögliche Ausprägungen, die das Aussehen der Datenmarkierungen im Diagramm bestimmen (Tab. 11.1, Abb. 11.1). Der Diagrammtitel kann als Zeichenkette für `main` angegeben werden, der Untertitel für `sub`. Für weitere Argumente vgl. `?plot.default`. Die Koordinaten der Punkte können auch als Modellformel angegeben werden, womit der Aufruf `plot(⟨y-Koord.⟩ ~ ⟨x-Koord.⟩, data=⟨Datensatz⟩)` lautet. Stammen die in der Modellformel verwendeten Variablen aus einem Datensatz, ist dieser unter `data` zu nennen.

Tabelle 11.1: Mögliche Werte für das Argument `type` von `plot()`

Wert für `type`	Bedeutung
`"p"`	Punkte
`"l"`	durchgehende Linien. Durch eng gesetzte Stützstellen können Funktionskurven beliebiger Form approximiert werden
`"b"`	Punkte und Linien, getrennt
`"o"`	Punkte und Linien, überlappend
`"h"`	senkrechte Linien zu jedem Datenpunkt (Spike Plot)
`"n"`	fügt dem Diagramm keine Datenpunkte hinzu (No Plotting)

```
> x <- sort(rnorm(20))                    # sortierte x-Koordinaten
> y <- 0.4*x + rnorm(20, mean=0, sd=4)    # y-Koordinaten

# das Argument xlab=NA entfernt die Bezeichnung der x-Achse
> plot(x, y, type="p", xlab=NA, main="type p")
> plot(x, y, type="l", xlab=NA, main="type l")
> plot(x, y, type="b", xlab=NA, main="type b")
> plot(x, y, type="o", xlab=NA, main="type o")
> plot(x, y, type="s", xlab=NA, main="type h")
> plot(x, y, type="h", xlab=NA, main="type h")
```

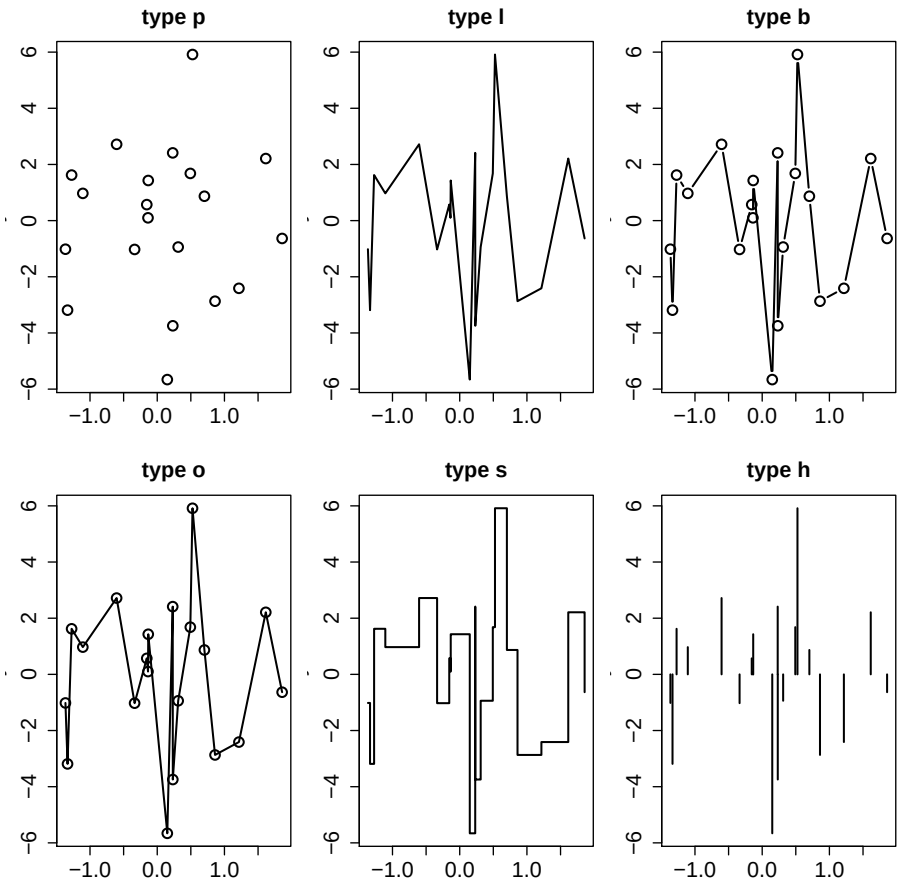

Abbildung 11.1: Mit `plot(type="⟨Option⟩")` erzeugbare Diagrammarten

11.2.2 Streudiagramme mit `matplot()`

So wie durch `plot()` ein Streudiagramm einer einzelnen Datenreihe erstellt wird, erzeugt `matplot()` ein Streudiagramm für mehrere Datenreihen gleichzeitig (Abb. 11.2).

```
> matplot(x=⟨Matrix⟩, y=⟨Matrix⟩, type="⟨Option⟩", pch="⟨Symbol⟩")
```

Die Argumente sind dieselben wie für `plot()`, lediglich x- und y-Koordinaten können nun als Matrizen an `x` und `y` übergeben werden, wobei jede ihrer Spalten als eine separate Datenreihe interpretiert wird. Haben dabei alle Datenreihen dieselben x-Koordinaten, kann `x` auch ein Vektor sein. Wird nur eine Matrix mit Koordinaten angegeben, werden diese als y-Koordinaten gedeutet und die x-Koordinaten durch die Zeilenindizes der Werte gebildet. In der Voreinstellung werden die Datenreihen in unterschiedlichen Farben dargestellt. Als Symbol für jeden Datenpunkt dienen die Ziffern 1–9, die mit der zur Datenreihe gehörenden Spaltennummer korrespondieren. Mit dem Argument `pch` können auch andere Symbole Verwendung finden (vgl. Abschn. 11.3.1, Abb. 11.3).

```
> vec <- seq(from=-2*pi, to=2*pi, length.out=50)
> mat <- cbind(2*sin(vec), sin(vec-(pi/4)), 0.5*sin(vec-(pi/2)))
> matplot(vec, mat, type="b", xlab=NA, ylab=NA, pch=1:3,
```

```
+         main="Sinuskurven")
```

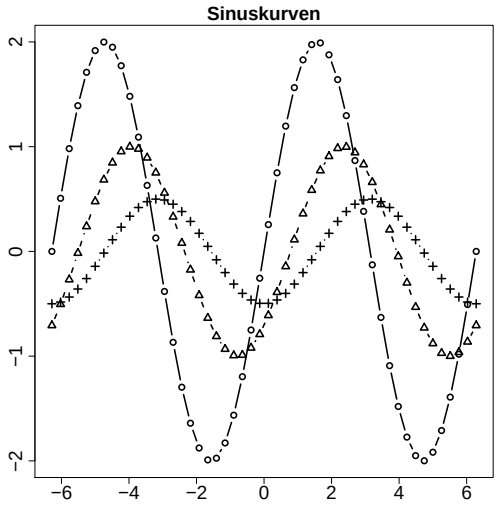

Abbildung 11.2: Mit `matplot()` erzeugtes Streudiagramm

11.3 Diagramme formatieren

`plot()` akzeptiert wie auch andere High-Level-Grafikfunktionen eine Vielzahl weiterer Argumente, um ein Diagramm flexibel anzupassen. Einige der wichtigsten Möglichkeiten zur individuellen Gestaltung werden im Folgenden vorgestellt.

11.3.1 Grafikelemente formatieren

Die Formatierung von Grafikelementen ist in vielen Aspekten variabel, etwa hinsichtlich der Art, Größe und Farbe der verwendeten Symbole oder Linien. Dafür akzeptieren die meisten High-Level-Funktionen einen gemeinsamen Satz zusätzlicher Argumente. Darunter befinden sich einige der in Tab. 11.2 für die `par()` Funktion beschriebenen, mit der sich die Einstellungen auch separat für ein Device bestimmen lassen. Die aktuell für das aktive Device gültigen Einstellungen für diese Argumente lassen sich durch `par("⟨Arg1⟩", "⟨Arg2⟩", ...)` als Liste ausgeben, d. h. durch Nennung der relevanten Argumente ohne Zuweisung von Werten. Ohne weitere Argumente gibt `par()` die aktuellen Werte für alle Parameter aus.

Tabelle 11.2: Grafikoptionen steuern mit Argumenten von `par()`

Argument	Wert	Bedeutung
`cex`	⟨`Zahl`⟩	Vergrößerungsfaktor für die Datenpunkt-Symbole. Voreinstellung ist der Wert 1
`col`	"⟨`Farbe`⟩"	Farbe der Datenpunkt-Symbole sowie bei `par()` zusätzlich des Rahmens um die Plot-Region, vgl. Abschn. 11.3.2

Tabelle 11.2: (Forts.)

`lty`	`1`, `2`, `3`, `4`, `5`, `6` bzw. ⟨`Schlüsselwort`⟩	Linientyp: Schlüsselwörter sind `"solid"`, `"dashed"`, `"dotted"`, `"dotdash"`, `"longdash"`, `"twodash"` (Abb. 11.3)
`lwd`	⟨`Zahl`⟩	Linienstärke, auch bei Datenpunktsymbolen. Voreinstellung ist der Wert 1.
`pch`	⟨`Zahl 1-25`⟩ bzw. `"`⟨`Buchstabe`⟩`"`	Art der Datenpunkt-Symbole, vgl. `?points` und Abb. 11.3. Wird ein Buchstabe angegeben, dient dieser als Symbol der Datenpunkte

Abbildung 11.3 veranschaulicht die mit `lty` und `pch` einstellbaren Linientypen und Datenpunkt-Symbole. Symbole 21–25 sind ausgefüllte Datenpunkte, deren Füllfarbe in Zeichenfunktionen über das Argument `bg="`⟨`Farbe`⟩`"` definiert wird, während `col="`⟨`Farbe`⟩`"` die Farbe des Randes bezeichnet (vgl. Abschn. 11.3.2).

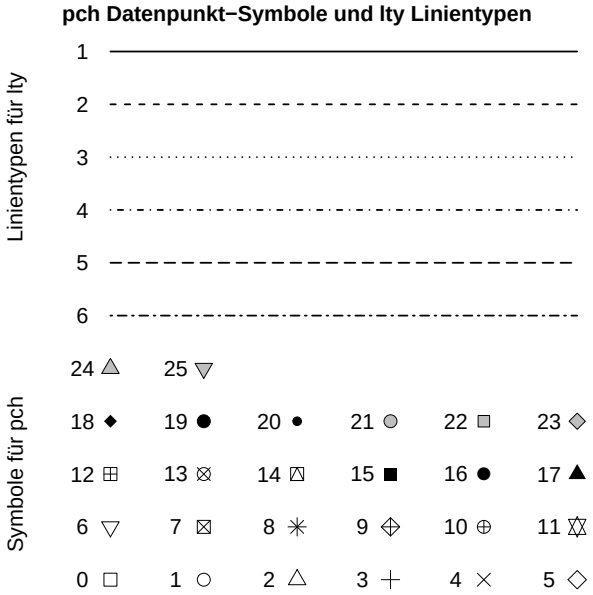

Abbildung 11.3: Datenpunkt-Symbole und Linientypen zur Verwendung für die Argumente `pch` und `lty` von Grafikfunktionen

Anstatt die in Tab. 11.2 genannten Argumente direkt beim Aufruf von Grafikfunktionen mit anzugeben, können sie durch `par(`⟨`Option`⟩`=`⟨`Wert`⟩`)` festgelegt werden. Die so geänderten Parameter sind Einstellungen für das aktive Device. Sie gelten für alle folgenden Ausgaben in dieses Device bis zur nächsten expliziten Änderung, oder bis ein neues Device aktiviert wird. Der auf der Konsole nicht sichtbare Rückgabewert von `par()` enthält die alten Einstellungen der geänderten Optionen in Form einer Liste, die auch direkt wieder an `par()` übergeben werden kann.

11.3.2 Farben spezifizieren

Häufig ist es sinnvoll, Diagrammelemente farblich hervorzuheben, etwa um die Zusammengehörigkeit von Punkten innerhalb von Datenreihen zu kennzeichnen und verschiedene Datenreihen leichter voneinander unterscheidbar zu machen. Auch Text- und Hintergrundfarben können in Diagrammen frei gewählt werden. Zu diesem Zweck lassen sich Farben in unterschiedlicher Form an die entsprechenden Funktionsargumente (meist `col`) übergeben:

- Als Farbname, z. B. `"green"` oder `"blue"`, vgl. `colors()` sowie Glynn (2005).
- Als natürliche Zahl, die als Index für die derzeit aktive Farbpalette interpretiert wird. Eine Farbpalette ist dabei ein vordefinierter Vektor von Farben, der mit `palette()` ausgegeben werden kann (vgl. `?rainbow` für weitere Paletten). Die voreingestellte Palette beginnt mit den Farben `"black"`, `"red"`, `"green3"` – der Index `2` entspräche also der Farbe Rot.
- Im Hexadezimalformat, wobei die Intensitäten der Monitor-Grundfarben Rot, Grün und Blau in der Form `"#RRGGBB"` mit Werten für `RR`, `GG` und `BB` im Bereich von `00` bis `FF` angegeben werden. `"#FF0000"` entspräche Rot, `"#00FF00"` Grün.
- In der Funktion `rgb(red=`⟨`Rot`⟩`, green=`⟨`Grün`⟩`, blue=`⟨`Blau`⟩`)` können die Intensitäten der Monitor-Grundfarben Rot, Grün und Blau mit Zahlen im Wertebereich von 0–1 angegeben werden. So gibt etwa `rgb(0, 1, 1)` die Farbe `"#00FFFF"` (Cyan) aus.
- Ein vierter Wert für Farben im Hexadezimalformat oder im Aufruf von `rgb()` kann den Grad des *alpha-blendings* für simulierte Transparenz definieren. Niedrige Werte stehen für sehr durchlässige, hohe Werte für opaque Farben (Abb. 7.1). Alpha-blending wird nur von manchen Devices unterstützt, etwa von `pdf()` oder `png()`.

11.3.3 Achsen formatieren

Ob durch die Darstellung von Datenpunkten in einem Diagramm automatisch auch Achsen generiert werden, kontrollieren in High-Level-Funktionen die Argumente `xaxt` für die x-Achse, `yaxt` für die y-Achse und `axes` für beide Achsen gleichzeitig. Während für `xaxt` und `yaxt` der Wert `"n"` übergeben werden muss, um die Ausgabe der entsprechenden Achse zu unterdrücken, akzeptiert das Argument `axes` dafür den Wert `FALSE`.

Achsen können mit `axis()` auch separat einem Diagramm hinzugefügt werden, wobei sich Lage, Beschriftung und Formatierung der Achsenmarkierungen festlegen lassen (vgl. Abschn. 11.5.6). Die Argumente `xlim=`⟨`Vektor`⟩ und `ylim=`⟨`Vektor`⟩ von High-Level-Funktionen legen den durch die Achsen abgedeckten Wertebereich in Form eines Vektors mit dem kleinsten und größten Achsenwert fest. Fehlen diese Argumente, wird jeder Bereich automatisch anhand der darzustellenden Daten bestimmt. Die Achsenbezeichnungen können in High-Level-Funktionen über die Argumente `xlab="`⟨`Name`⟩`"` und `ylab="`⟨`Name`⟩`"` gewählt oder mit Setzen auf `NA` unterdrückt werden.

11.4 Säulen- und Punktdiagramme

Mit `barplot()` erstellte Säulendiagramme eignen sich zur Darstellung von Kennwerten von Variablen, die getrennt für verschiedene Gruppen berechnet wurden. Dazu zählen etwa absolute oder relative Häufigkeiten von Gruppenzugehörigkeiten oder der jeweilige Mittelwert einer Variable in verschiedenen Stichproben. Der Kennwert jeder Gruppe wird dabei durch eine Säule repräsentiert, deren Höhe seine Größe widerspiegelt.

11.4.1 Einfache Säulendiagramme

Sollen Kennwerte einer Variable getrennt für Gruppen dargestellt werden, die sich aus den Stufen eines einzelnen Faktors ergeben, lautet die Grundform von `barplot()`:

```
> barplot(height=⟨Vektor⟩, ⟨Argumente⟩)
```

- Unter `height` ist ein Vektor einzutragen, wobei jedes seiner Elemente den Kennwert für jeweils eine Bedingung repräsentiert und damit die Höhe einer Säule festlegt.
- `horiz` bestimmt, ob vertikale Säulen (Voreinstellung `FALSE`) oder horizontale Balken gezeichnet werden. Der (auf der Konsole nicht sichtbare) Rückgabewert von `barplot()` enthält die x-Koordinaten der eingezeichneten Säulen bzw. die y-Koordinaten der Balken.
- `names.arg` nimmt einen Vektor von Zeichenketten an, der die Beschriftung der Säulen definiert.
- Ist das Minimum des mit `ylim` definierten Wertebereichs der y-Achse größer als 0, muss `xpd=FALSE` gesetzt werden, um Säulen nicht unterhalb der x-Achse zeichnen zu lassen. Andernfalls erscheinen in der Voreinstellung `xpd=TRUE` die Säulen unterhalb der x-Achse, auch wenn diese nicht bei 0 beginnt.

Als Beispiel diene die Verteilung von Haarfarben (Abb. 11.4).

```
> (tab <- xtabs(~ hair, data=datW))              # absolute Häufigkeiten
hair
black  brown  red  blond
   12     27    8     17

> barplot(tab, ylim=c(0, 30), xlab="Hair color", ylab="N", col="black",
+         main="Absolute Häufigkeiten")
```

11.4.2 Gruppierte und gestapelte Säulendiagramme

Gruppierte Säulendiagramme stellen Kennwerte von Variablen getrennt für Gruppen dar, die sich aus der Kombination zweier Faktoren ergeben. Dafür kann die Zusammengehörigkeit einer aus mehreren Säulen bestehenden Gruppe grafisch durch ihre räumliche Nähe innerhalb der Gruppe und die gleichzeitig größere Distanz zu anderen Säulengruppen kenntlich gemacht werden. Eine weitere Möglichkeit besteht darin, jede Einzelsäule nicht homogen, sondern als Stapel mehrerer Segmente darzustellen (Abb. 11.5).

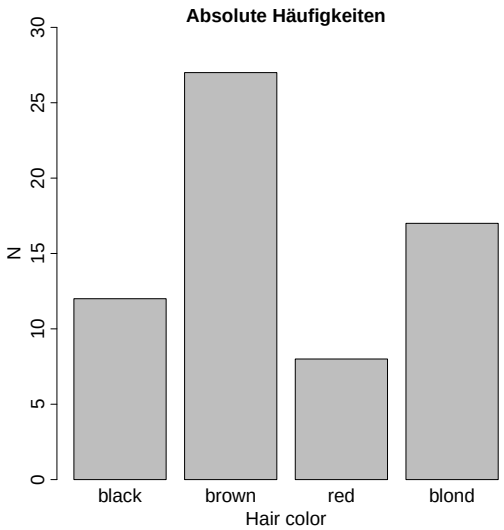

Abbildung 11.4: Säulendiagramm absoluter Häufigkeiten

- Für gruppierte oder gestapelte Säulendiagramme werden die Daten an `height` in Form einer Matrix übergeben, deren Werte die Säulenhöhen bzw. Balkenlängen festlegen.
- Das Argument `beside` kontrolliert, welche Darstellungsart gewählt wird: `TRUE` bewirkt Säulengruppen, die Voreinstellung `FALSE` gestapelte Säulen.
- Ist `beside=FALSE`, definiert jede Spalte der Datenmatrix die innere Zusammensetzung einer Säule, indem die einzelnen Werte einer Spalte die Höhe der Segmente bestimmen, aus denen die Säule besteht. Bei `beside=TRUE` definiert eine Spalte der Datenmatrix eine Säulengruppe, deren jeweilige Höhen durch die Einzelwerte in der Spalte festgelegt sind.
- `names.arg` nimmt einen Vektor von Zeichenketten an, der die Beschriftung der Säulengruppen definiert.
- `legend.text` kontrolliert, ob eine Legende eingefügt wird, Voreinstellung ist `FALSE`. Auf `TRUE` gesetzt erscheint eine Legende, die auf den Zeilennamen der Datenmatrix basiert und sich auf die Bedeutung der Säulen innerhalb einer Gruppe bzw. auf die Segmente einer Säule bezieht. Alternativ können die Einträge der Legende als Vektor von Zeichenketten angegeben werden.

```
# gemeinsame Häufigkeiten von Haar- und Augenfarbe
> (cTab <- xtabs(~ hair + eye, data=datW))
                eye
hair    brown blue hazel green
  black     7    1     4     0
  brown     9   10     6     2
  red       2    3     2     1
  blond     2   12     1     2
```

Die in jeder der vier Gruppen vorhandenen Säulen sollten farblich getrennt werden, um die Zugehörigkeit zur Substichprobe deutlich zu machen. Dazu wird an `col` ein Vektor mit pas-

send vielen Farben übergeben, den R intern so häufig recycled (vgl. Abschn. 3.4.4), wie es Säulengruppen gibt.

```
> barplot(cTab, beside=FALSE, legend.text=TRUE, xlab="Eye color",
+         ylab="N", main="Absolute Häufigkeiten einer Kreuztabelle")

> barplot(cTab, beside=TRUE, ylim=c(0, 15), legend.text=TRUE, ylab="N",
+         xlab="Eye color", col=c("red", "green", "blue", "gray"),
+         main="Absolute Häufigkeiten einer Kreuztabelle")
```

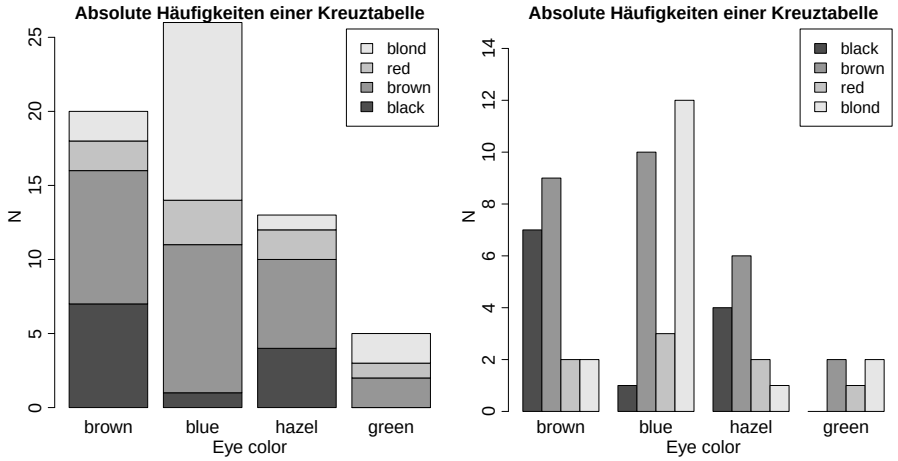

Abbildung 11.5: Gestapeltes und gruppiertes Säulendiagramm absoluter Häufigkeiten

11.4.3 Dotchart

Die Funktion `dotchart()` dient der Darstellung von Rohdaten einzelner Beobachtungsobjekte, aber auch von aggregierten Kennwerten von Variablen – etwa des Mittelwerts pro Gruppe. Jeder Wert wird dabei durch einen Punkt repräsentiert, dessen x-Koordinate seine Größe widerspiegelt und dessen y-Koordinate das Beobachtungsobjekt codiert. Das Ergebnis von `dotchart()` ist analog zu einem horizontalen Balkendiagramm, wobei statt der Balken lediglich Punkte eingezeichnet werden.

```
> dotchart(x=⟨Daten⟩, labels="⟨Namen⟩", groups=⟨Faktor⟩)
```

Für `x` ist der Datenvektor anzugeben. Über das Argument `labels` lassen sich mittels eines Vektors aus Zeichenketten derselben Länge wie `x` die Bezeichnungen der Datenpunkte nennen. Sollen Daten aus verschiedenen, durch die Kombination zweier Faktoren gebildeten Gruppen dargestellt werden, sind die Daten in Form einer Matrix zu übergeben, deren Werte die x-Koordinaten der Punkte festlegen. Dabei definiert jede Spalte der Datenmatrix eine Punktgruppe, deren Punkte vertikal nahe beieinander gezeichnet werden, während die durch verschiedene Spalten definierten Punkte stärker räumlich getrennt sind.

Stellen die Daten Kennwerte verschiedener Gruppen dar, können sie auch als Vektor `x` unter gleichzeitiger Angabe von `groups` übergeben werden. Für `groups` ist dann ein Faktor derselben Länge wie `x` zu nennen, der die Gruppenzugehörigkeit jedes Wertes definiert – auf diese Weise

lassen sich auch Daten ungleich großer Gruppen darstellen (Abb. 11.6). Für die Einfärbung der Datenpunkte entsprechend der Gruppenzugehörigkeit lässt sich die Eigenschaft von Faktoren ausnutzen, dass ihre Stufen intern über natürliche Zahlen repräsentiert sind. Diese Zahlen lassen sich mit `unclass()` ausgeben und als Indizes eines Farbvektors verwenden.

```
> DV <- datW$DV                                    # AV
> IV <- datW$group                                 # Gruppenzugehörigkeit
> Mj <- tapply(DV, IV, mean)                       # Gruppenmittel
> dotchart(DV, gdata=Mj, pch=20, color=(1:3)[unclass(IV)],
+          gcolor="blue", gpch=16, groups=IV, xlab="AV", ylab="Gruppen",
+          main="Individuelle Ergebnisse + Mittelwerte aus 3 Gruppen")
```

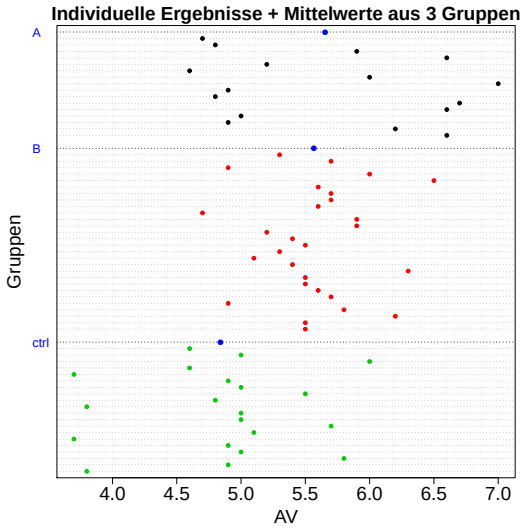

Abbildung 11.6: Dotchart von individuellen Messwerten in drei Gruppen

11.5 Elemente einem bestehenden Diagramm hinzufügen

Ein bereits erstelltes Diagramm lässt sich nachträglich durch zusätzliche Elemente erweitern, die durch Low-Level-Funktionen erzeugt werden (Tab. 11.3). Auch einige High-Level-Funktionen besitzen das Argument `add=TRUE`, durch das ihre Ausgabe dem derzeit aktiven Device hinzugefügt wird, ohne dessen Inhalte zuvor zu löschen. In diesem Fall kann meist durch das Argument `at=`⟨`Position`⟩ bestimmt werden, an welcher Stelle der Plot-Region die zusätzlichen Daten erscheinen sollen. Ist kein `add` Argument vorhanden, bewirkt die vorhergehende Ausführung von `par(new=TRUE)`, dass das Ergebnis des folgenden High-Level-Befehls im schon geöffneten Device erscheint. Später eingefügte Elemente werden immer über bereits bestehende gezeichnet – neue Inhalte übermalen also ältere Inhalte, die sich an derselben Stelle befinden.

11.5.1 Punkte

Tabelle 11.3: Mögliche Diagrammelemente hinzufügende Funktionen

Diagrammelement	hinzufügende Low-Level-Grafikfunktion
Punkte	`points()`, `matpoints()`
Geradenabschnitte	`lines()`, `matlines()`, `segments()`, `abline()`
Polygone	`rect()`, `polygon()`
Funktionsgraphen	`curve()`
Text	`title()`, `legend()`, `text()`
Achsen	`axis()`

```
> points(x=⟨x-Koordinaten⟩, y=⟨y-Koordinaten⟩, type="⟨Option⟩")
> matpoints(x=⟨Matrix⟩, y=⟨Matrix⟩, type="⟨Option⟩")
```

Ähnlich wie `plot()` fügt `points()` einem geöffneten Diagramm Punkte hinzu, die Argumente `x`, `y` und `type` stimmen in ihrer Bedeutung mit jenen von `plot()` überein. Analog zu `matplot()` können mit `matpoints()` für x- und y-Koordinaten separate Matrizen angegeben und so gleichzeitig mehrere Datenreihen spezifiziert werden.

```
> xA <- seq(-15, 15, length.out=200)
> yA <- sin(xA) / xA                              # Sinc-Funktion
> plot(xA, yA, type="l", xlab="x", ylab="sinc(x)",
+      main="Punkte und Linien einfügen", lwd=1.6)

> xB <- seq(-15, 15, length.out=30)
> yB <- sin(xB) / xB
> points(xB, yB, col="red", pch=20)
```

11.5.2 Linien

```
> lines(x=⟨x-Koordinaten⟩, y=⟨y-Koordinaten⟩, type="⟨Option⟩")
> matlines(x=⟨Matrix⟩, y=⟨Matrix⟩, type="⟨Option⟩")
```

Datenpunkte verbindende Linien werden mit `lines()` analog zu Punkten erstellt. Dabei geben die Vektoren `x` und `y` die Koordinaten der zu verbindenden Punkte an, das Argument `type` bestimmt wie in `plot()` den Linientyp. `matlines()` arbeitet wie `matpoints()`, die (x, y)-Koordinaten können also für mehrere Datenreihen gleichzeitig in Form jeweils einer Matrix an die Argumente `x` und `y` übergeben werden (Abb. 11.7).

```
> yC <- sin(pi * xA) / (pi * xA)                  # normierte Sinc-Funktion
> lines(xA, yC, col="blue", type="l", lwd=1.6)

> abline(a=⟨y-Achsenabschnitt⟩, b=⟨Steigung⟩,
+        h=⟨y-Koordinate⟩, v=⟨x-Koordinate⟩, coef=⟨Vektor⟩)
```

Mit `abline()` werden in ein Diagramm Geradenabschnitte gezeichnet, die auf unterschiedliche Art spezifizierbar sind. Geraden können über die Gleichung $Y = bX + a$ mit den Parametern `a` für den Schnittpunkt mit der y-Achse und `b` für die Steigung beschrieben werden. Diese beiden Parameter erwartet alternativ auch das Argument `coef` in Form eines Vektors mit zwei Elementen. Statt dieser Parameter akzeptiert `abline()` auch ein mit `lm()` erstelltes Objekt, um die angepasste Vorhersagegerade einer einfachen linearen Regression darzustellen (vgl. Abschn. 7.2).

Ein über die gesamte Breite der Plot-Region gehender horizontaler Geradenabschnitt kann über das Argument `h` in seiner y-Koordinate bezeichnet werden, ein vertikaler entsprechend über das Argument `v` in seiner x-Koordinate. Es lassen sich mehrere horizontale oder vertikale Geradenabschnitte mit nur einem Aufruf von `abline()` erzeugen, etwa um ein Gitter zu zeichnen. Dafür werden sowohl `h` als auch `v` gleichzeitig verwendet und für sie Vektoren von Koordinaten angegeben (Abb. 11.7).

```
> abline(h=0, v=0, col="green", lwd=1.6)
```

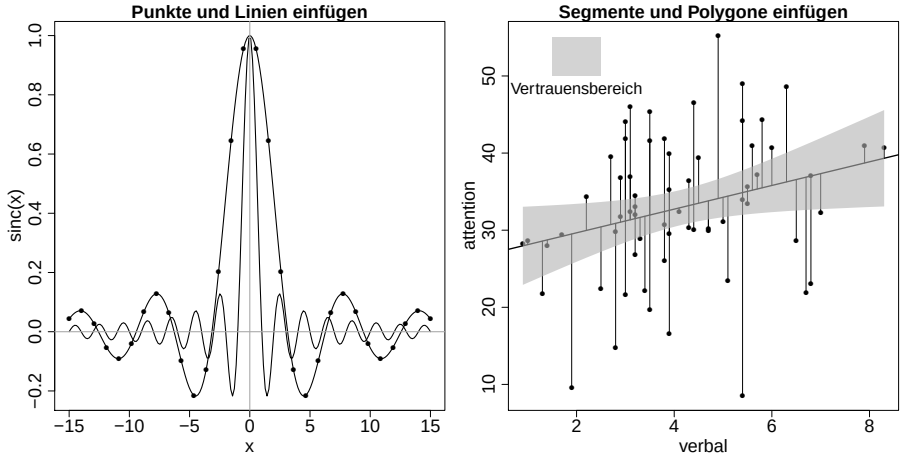

Abbildung 11.7: Einfügen von Diagrammelementen: Punkte, Linien und Polygone

Einzelne Liniensegmente können mit `segments()` eingezeichnet werden. Dazu sind die (x, y)-Koordinaten der Start- und Endpunkte der Segmente anzugeben.

```
> segments(x0=⟨x Start⟩, y0=⟨y Start⟩, x1=⟨x Ziel⟩, y1=⟨y Ziel⟩)
```

Unter `x0` und `y0` sind die Koordinaten des Startpunkts zu nennen, von dem ausgehend das Segment gezeichnet wird. Unter `x1` und `y1` wird der Zielpunkt des Segments mit seinen Koordinaten angegeben. Wenn mehrere Linien zu zeichnen sind, lassen sich die Koordinaten auch als Vektoren eingeben (Abb. 11.7).

```
# Lineare Regression: Differenz Vorhersage-Kriterium (Residuen)
> fit  <- lm(attention ~ verbal, data=datW)     # Regression
> Yhat <- fitted(fit)                           # Vorhersage
> plot(attention ~ verbal, data=datW, pch=16,   # Streudiagramm
+      main="Segmente und Polygone einfügen")

# Residuen einzeichnen
```

```
> with(datW, segments(x0=verbal, y0=attention, x1=verbal, y1=Yhat))
> abline(fit, col="blue", lwd=2)                 # Regressionsgerade
```

11.5.3 Polygone

Die Funktion `polygon()` erzeugt Vielecke beliebiger Gestalt, deren Eckpunkte über ihre jeweiligen (x, y)-Koordinaten zu definieren sind und in Form von Vektoren an `x` und `y` übergeben werden können. Das Polygon wird geschlossen, indem R den ersten und letzten Punkt miteinander verbindet. Soll kein Rahmen gezogen werden, ist `border=NA` zu setzen. Mittels eng gesetzter Eckpunkte lassen sich auch runde Formen approximieren (Abb. 11.7).

```
> polygon(x=⟨x-Koordinaten⟩, y=⟨y-Koordinaten⟩, border=NULL)
```

Dem Streudiagramm mit Regressionsgerade und Residuen soll nun der Vertrauensbereich hinzugefügt werden (vgl. Abschn. 7.5).

```
> pred <- predict(fit, interval="confidence", level=0.95)
> fOrd <- with(datW, order(verbal))
> with(datW, polygon(c(verbal[fOrd],      verbal[rev(fOrd)]),
>                    c(  pred[fOrd, "lwr"], pred[rev(fOrd), "upr"]),
>                    border=NA, col=rgb(0.7, 0.7, 0.7, 0.6)))
```

Die Funktion `rect()` erzeugt beliebig dimensionierte Rechtecke, die sich in der Plot-Region frei plazieren lassen.

```
> rect(xleft=⟨x-Koord. links⟩, ybottom=⟨y-Koord. unten⟩,
+      xright=⟨x-Koord. rechts⟩, ytop=⟨y-Koord. oben⟩, border=NULL)
```

Die Argumente `xleft`, `ybottom`, `xright` und `ytop` akzeptieren Vektoren, die jeweils die Koordinaten der linken (x), unteren (y), rechten (x) und oberen Seiten (y) der zu zeichnenden Rechtecke enthalten. Soll kein Rahmen um ein Rechteck gezogen werden, ist `border=NA` zu setzen (Abb. 11.7).

```
> rect(1.5, 50, 2.5, 55, col="lightgray", border=NA)
> text(2, 48, labels="Vertrauensbereich")
```

11.5.4 Funktionsgraphen

```
> curve(expr=⟨Funktion⟩, from=⟨Zahl⟩, to=⟨Zahl⟩, n=101, add=FALSE)
```

Die High-Level-Funktion `curve()` dient dazu, Graphen von beliebigen Funktionen mit einer Veränderlichen zu erstellen. Dabei kann die darzustellende Funktion als Funktionsgleichung mit `x` als Variable spezifiziert werden (z. B. `dnorm(x, mean=1, sd=1)` oder `x^2 + 10`). Hier wird also dieselbe Notation benutzt, wie um aus einem bestehenden Vektor `x` einen neuen Vektor mit den Funktionswerten zu generieren. Als Kurzform lässt sich auch nur der Name einer Funktion angeben (z. B. `dnorm`), die dann mit den Voreinstellungen für ihre Argumente aufgerufen wird. In welchem Wertebereich die Funktion ausgewertet werden soll, bestimmen

die Argumente `from` und `to`. Dabei legt das Argument `n` die Anzahl der Stützstellen fest, an wie vielen gleichabständigen Stellen in diesem Bereich also Funktionswerte bestimmt und eingezeichnet werden. Um den Funktionsgraphen dem aktuell aktiven Device hinzuzufügen, ohne dessen Inhalte zu löschen, ist `add=TRUE` zu setzen (Abb. 11.8).

```
> curve(dnorm(x, mean=1, sd=1), from=-7, to=7, col="blue", lwd=2,
+       xlab=NA, axes=FALSE)

# Normalverteilung mit mu=0, sigma=2
> curve((1/(2*sqrt(2*pi))) * exp(-0.5*(((x-0)/2)^2)),
+       add=TRUE, lwd=2, lty=2)
```

11.5.5 Text

```
> title(main="⟨Titel⟩", sub="⟨Untertitel⟩")
```

Der Diagrammtitel lässt sich in High-Level-Grafikfunktionen über das Argument `main="⟨Name⟩"`, ein Untertitel über `sub="⟨Name⟩"` hinzufügen. Soll der Diagrammtitel nachträglich festgelegt werden, kann die separat aufzurufende Low-Level-Funktion `title()` Verwendung finden, die ihrerseits `main` und `sub` als Argumente besitzt (Abb. 11.8).

```
> title(main="zwei Normalverteilungskurven", sub="Untertitel")
```

Eine Legende zur Erläuterung der verwendeten Symbole erstellt die `legend()` Funktion.

```
> legend(x=⟨x-Koordinate⟩, y=⟨y-Koordinate⟩, legend="⟨Text⟩",
+        col="⟨Farben⟩", lty=⟨Linientypen⟩, lwd=⟨Linienstärken⟩,
+        pch=⟨Symbole⟩)
```

Die Legende wird entweder über die Angabe von (x, y)-Koordinaten für die Argumente `x` und `y` positioniert, oder durch Nennung eines der Schlüsselwörter `"bottomright"`, `"bottom"`, `"bottomleft"`, `"left"`, `"topleft"`, `"top"`, `"topright"`, `"right"` oder `"center"` für `x`. Der Legendentext selbst ist als Vektor von Zeichenketten an `legend` zu übergeben, wobei jedes Element dieses Vektors einen Legendeneintrag definiert. Welches Aussehen dem Symbol neben einem Eintrag verliehen wird, kontrollieren die Argumente `col` (Farbe), `lty` (Linientyp), `lwd` (Linienstärke) und `pch` (Symbol), vgl. Abschn. 11.3.1. Für diese Argumente ist jeweils ein Vektor derselben Länge wie `legend` einzugeben, wobei `NA` Einträge bedeuten, dass die definierte Eigenschaft nicht auf das zugehörige Legendensymbol zutrifft. Sind in einem Diagramm etwa sowohl zwei Punkt-Reihen als auch zwei Linien enthalten, würde die Kombination von `pch=c(19, 20, NA, NA)` und `lty=c(NA, NA, 1, 2)` bewirken, dass die ersten beiden Legendeneinträge mit den Symbolen 19 und 20 dargestellt werden, die letzten beiden Einträge mit den Linientypen 1 und 2 (Abb. 11.8).

```
> legend(x="topleft", legend=c("N(1, 1)", "N(0, 2)"),
+        col=c("blue", "black"), lty=c(1, 2))
```

Mit `text()` lässt sich allgemein Text in ein Diagramm einfügen.

```
> text(x=⟨x-Koordinaten⟩, y=⟨y-Koordinaten⟩, labels="⟨Name⟩")
```

Zunächst sind mit den Argumenten `x` und `y` die Koordinaten des Texts festzulegen. Sollen mehrere Textelemente gleichzeitig eingefügt werden, sind hier Vektoren zu übergeben. In der Voreinstellung beziehen sich die Koordinaten auf den Mittelpunkt des Texts (Abb. 11.8).

```
> text(x=3.6,  y=0.35, labels="Normalverteilung\nN(1, 1)")
> text(x=-3.5, y=0.1,  labels="N(0, 2)")
```

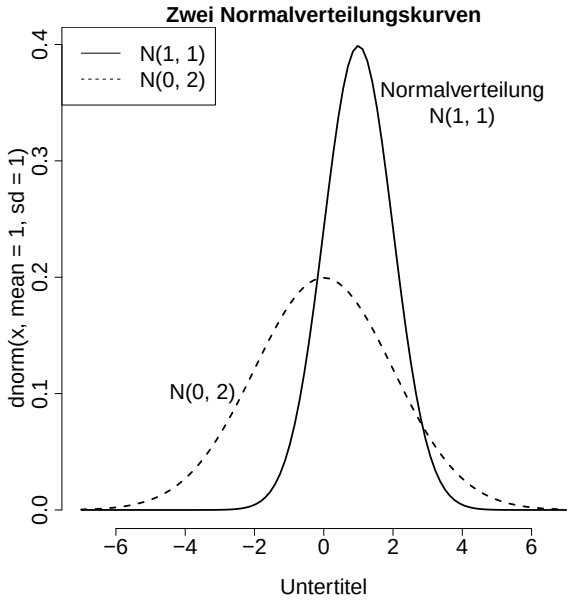

Abbildung 11.8: Diagrammelemente einfügen: Funktionsgraphen, Text und Legende

11.5.6 Achsen

Ob durch die Darstellung von Datenpunkten in einem Diagramm automatisch auch Achsen generiert werden, kontrollieren in High-Level-Grafikfunktionen die Argumente `xaxt` für die x-Achse, `yaxt` für die y-Achse und `axes` für beide Achsen gleichzeitig. Während für `xaxt` und `yaxt` der Wert `"n"` übergeben werden muss, um die Ausgabe der entsprechenden Achse zu unterdrücken, akzeptiert `axes` dafür den Wert `FALSE`.

Für eine feinere Kontrolle über den Wertebereich sowie über Aussehen und Lage der Wertemarkierungen der Achsen empfiehlt es sich, ihre automatische Generierung zunächst mit `axes=FALSE` zu unterdrücken. Nachdem das Diagramm erstellt wurde, können dann mit dem Befehl `axis()` samt seiner Argumente zur Formatierung und Positionierung Achsen hinzugefügt werden (Abb. 11.8).

```
> axis(side=⟨Nummer⟩, at=⟨Markierungen⟩, labels="⟨Wertebeschriftungen⟩",
+      pos=⟨Position⟩)
```

Achsen lassen sich an allen Diagrammseiten darstellen, was über das Argument `side` kontrolliert wird. Mögliche Werte sind `1` (unten, x-Achse), `2` (links, y-Achse), `3` (oben, alternative x-Achse) und `4` (rechts, alternative y-Achse). Die Wertemarkierungen der Achse lassen sich über `at` in Form eines Vektors festlegen. Das Argument `labels` bestimmt die Beschriftung

dieser Markierungen und erwartet einen numerischen Vektor oder einen Vektor von Zeichenketten.

```
> axis(side=1, at=seq(-6, 6, by=2))
> axis(side=2, at=seq(0, 0.4, by=0.1))
```

11.5.7 Fehlerbalken

Fehlerbalken werden zusätzlich zu Kennwerten von Variablen vor allem in Säulen- und Punktdiagrammen eingezeichnet, um die Variabilität der Daten auszudrücken. Als Maß der Variabilität kann dabei u. a. die Breite eines statistischen Konfidenzintervalls für einen Parameter (z. B. den Erwartungswert) oder ein deskriptives Maß wie die Streuung verwendet werden.

Für die Darstellung von Fehlerbalken enthält das Paket `DescTools` die Funktion `ErrBars()`. Das vertikale Zentrum der Fehlerbalken wird als Punkt gezeichnet, so dass es nicht notwendig ist, zusätzlich Säulen oder Punkte zur Veranschaulichung des Parameters zu zeichnen, dessen Variabilität über einen Fehlerbalken dargestellt wird (Abb. 11.9).

```
ErrBars(from=⟨y-Koordinate unten⟩, to=⟨y-Koordinate oben⟩,
        pos=⟨x-Koordinate⟩, horiz=FALSE)
```

Unter `from` und `to` werden die y-Koordinaten der unteren bzw. oberen Grenzen der Fehlerbalken definiert. Für `pos` sind die x-Koordinaten der Fehlerbalken anzugeben. Mit `horiz` wird definiert, ob die Balken vertikal (Voreinstellung `FALSE`) oder horizontal (`TRUE`) zu zeichnen sind. Die Funktion verfügt über weitere Optionen zur Formatierung der Fehlerbalken bzgl. ihrer Farbe, Linienstärke, etc.

```
> Mj <- with(datW, tapply(DV, group, FUN=mean))    # Gruppenmittel
> Sj <- with(datW, tapply(DV, group, FUN=sd))      # Gruppenstreuungen
> Nj <- xtabs(~ group, data=datW)                  # Zellbesetzungen

# halbe Breite des 95% t-Konfidenzintervalls für mu in 1 Stichprobe
> CIwidths <- qt(0.975, df=Nj-1) * Sj / sqrt(Nj)

# Punktdiagramm getrennt nach Gruppen - vgl. 10.6.3 für stripchart()
> stripchart(DV ~ group, data=datW, method="jitter", xaxt="n", pch=16,
+            xlab="Gruppe", ylim=c(0, 40), col="darkgray", vert=TRUE,
+            main="Rohdaten und Konfidenzintervalle")

> library(DescTools)                               # für ErrBars()
> ErrBars(from=Mj-CIwidths, to=Mj+CIwidths, pos=1:3, length=0.1,
+         col="blue", col.pch="blue", lwd=2, pch=19)

> axis(side=1, at=1:3, labels=LETTERS[1:3])        # Gruppenbezeichnungen
```

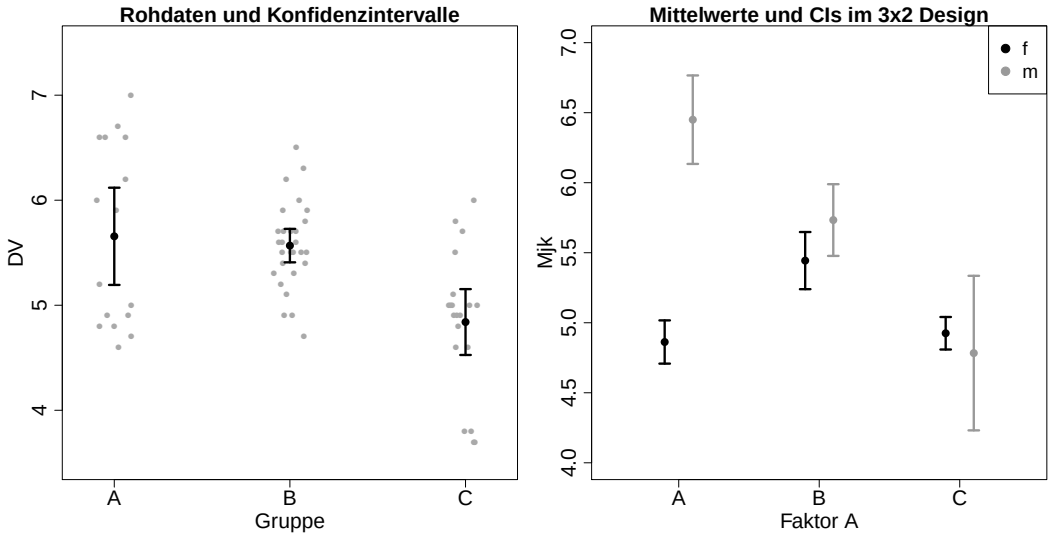

Abbildung 11.9: Fehlerbalken mit `plotCI()` einfügen

11.6 Verteilungsdiagramme

Verteilungsdiagramme dienen dazu, sich einen Überblick über die Lage und Verteilungsform der in einer Stichprobe erhobenen Daten zu verschaffen.

11.6.1 Histogramm und Schätzung der Dichtefunktion

Für Stichproben stetiger Variablen, die eine Vielzahl unterschiedlicher Werte enthalten, kann ein Histogramm für die Darstellung der empirischen Häufigkeitsverteilung verwendet werden. Histogramme stellen nicht die Häufigkeit einzelner Werte, sondern die von disjunkten Wertebereichen anhand von Säulen dar (Abb. 11.10).

```
> hist(x=⟨Vektor⟩, breaks=⟨Grenzen⟩, freq=NULL)
```

Die Daten sind in Form eines Vektors `x` zu übergeben. Die Intervallgrenzen werden über das Argument `breaks` festgelegt, wobei mit einer einzelnen Zahl deren Anzahl und mit einem Vektor deren genaue Lage vorgegeben werden kann. In der Voreinstellung wird beides nach einem in der Hilfe beschriebenen Algorithmus entsprechend den Daten in `x` gewählt. Bei gleichabständigen Klassengrenzen werden in der Voreinstellung `freq=NULL` absolute Häufigkeiten angezeigt. Mit `freq=FALSE` entspricht der Flächeninhalt jeder Säule der relativen Häufigkeit des Intervalls, `TRUE` erzwingt absolute Häufigkeiten auch bei ungleichen Klassenbreiten.

Für die individuelle Wahl der Klassengrenzen empfiehlt es sich, zunächst die Spannweite der Daten auszuwerten. Intervallgrenzen in regelmäßigen Abständen können dann mit `seq()` generiert werden. Werte, die genau auf einer Grenze liegen, werden immer der unteren Klasse zugeordnet, die Klassen sind also nach unten offene und nach oben geschlossene Intervalle.

```
# darzustellender Wertebereich
> fromTo <- round(range(datW$iq), -1) + c(-10, 10)
> limits <- seq(from=fromTo[1], to=fromTo[2], 5)
```

```
> hist(datW$iq, freq=FALSE, xlim=fromTo, xlab="IQ",
+      ylab="relative Häufigkeit",
+      breaks=limits, main="Histogramm und Normalverteilung")
```

Soll über das Histogramm zum Vergleich eine theoretisch vermutete Dichtefunktion gelegt werden, kann diese etwa mit `curve(..., add=TRUE)` hinzugefügt werden. Dafür muss das Histogramm relative Häufigkeiten darstellen.

```
# füge Dichtefunktion einer Normalverteilung hinzu
> curve(dnorm(x, mean(datW$iq), sd(datW$iq)), lwd=2, col="blue", add=TRUE)
```

Als Alternative lässt sich mit `density(⟨Vektor⟩)` die Dichtefunktion der Variable schätzen und grafisch darstellen. Die Schätzmethode kann über eine Reihe zusätzlicher Argumente kontrolliert werden, vgl. `?density`. Über `plot(density(...))` wird die Schätzung der Dichtefunktion in einem separaten Diagramm veranschaulicht, während `lines(density(...))` die geschätzte Dichtefunktion einem bereits geöffneten Diagramm hinzufügt. Beim Histogramm ist dabei darauf zu achten, relative Häufigkeiten darzustellen.

```
> hist(datW$iq, freq=FALSE, xlim=fromTo, xlab="IQ",
+      main="Histogramm und Schätzung der Dichte")

> lines(density(datW$iq), lwd=2, col="blue")
```

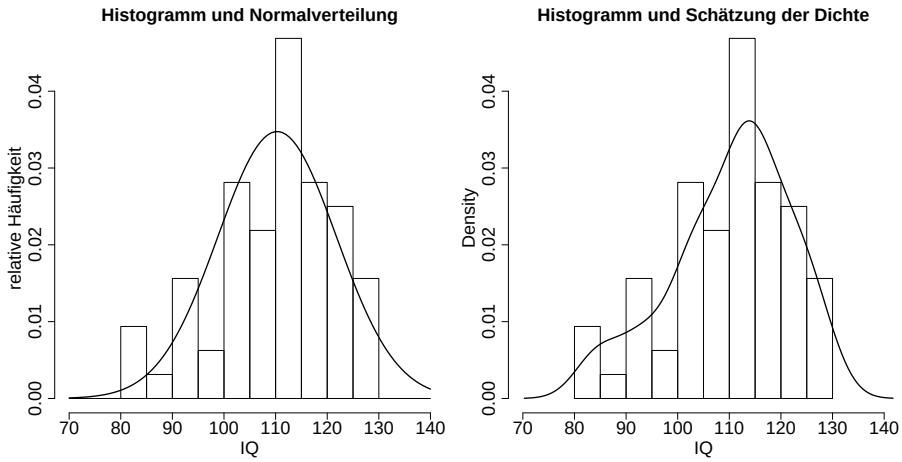

Abbildung 11.10: Histogramm mit Normalverteilung bzw. mit Schätzung der Dichtefunktion

11.6.2 Boxplot

Ein Boxplot stellt die Lage und Verteilung empirischer Daten durch die gleichzeitige Visualisierung verschiedener Kennwerte dar. Der Median wird dabei durch eine schwarze horizontale Linie innerhalb einer Box gekennzeichnet, deren untere Grenze sich auf Höhe des ersten und deren obere Grenze sich auf Höhe des dritten Quartils befindet. Innerhalb des so gebildeten Rechtecks liegen damit die mittleren 50% der Werte, seine Länge ist gleich dem Interquartilabstand. Jenseits der Box erstrecken sich nach oben und unten dünne Striche, deren Enden

jeweils den extremsten Wert angeben, der noch keinen Ausreißer darstellt. Als Ausreißer werden dabei in der Voreinstellung solche Werte betrachtet, die um mehr als das Anderthalbfache des Interquartilabstands unter oder über der Box liegen. Solche Ausreißer werden schließlich durch Kreise gekennzeichnet (Abb. 11.11).

```
> boxplot(x=⟨Vektor⟩, horizontal=FALSE)
```

Bei einem einzelnen Boxplot wird unter `x` der Datenvektor eingegeben. Stattdessen kann für `x` auch eine Modellformel der Form ⟨`AV`⟩ `~` ⟨`Faktor`⟩ übergeben werden, wobei ⟨`Faktor`⟩ dieselbe Länge wie der Vektor ⟨`AV`⟩ besitzt und für jeden von dessen Werten codiert, zu welcher Bedingung er gehört. Stammen die in der Modellformel verwendeten Variablen aus einem Datensatz, ist dieser unter `data` zu nennen. Hiermit werden getrennt für die von ⟨`Faktor`⟩ definierten Gruppen Boxplots nebeneinander in einem Diagramm dargestellt. Für horizontal verlaufende Boxen ist `horizontal=TRUE` zu setzen. Der auf der Konsole nicht sichtbare Rückgabewert enthält in Form einer Liste Angaben zu den dargestellten statistischen Kennwerten.

```
> boxplot(DV ~ group, data=datW, ylab="Score", col="lightgray",
+         main="Boxplots der Scores in 3 Gruppen")

> Mj <- with(datW, tapply(DV, group, mean))       # Gruppenmittel
> points(1:P, Mj, pch=16, cex=2)                  # zeige Gruppenmittel
```

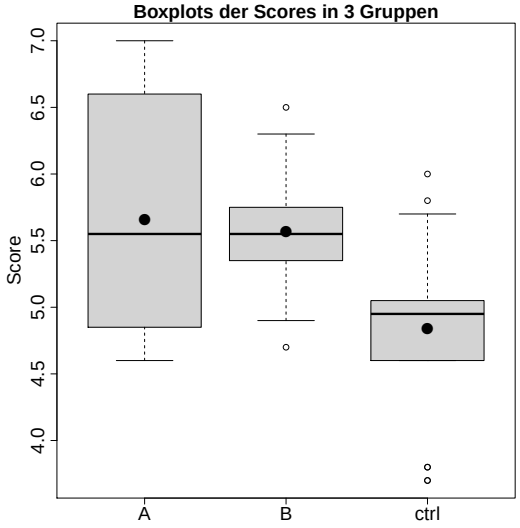

Abbildung 11.11: Boxplots getrennt nach Gruppen mit Mittelwerten

11.6.3 Stripchart

Ein mit `stripchart()` erstelltes eindimensionales Streudiagramm eignet sich zur Veranschaulichung der empirischen Verteilung quantitativer Variablen, wenn der Stichprobenumfang gering ist. Statt wie ein Boxplot Daten summarisch anhand ihrer wichtigsten Verteilungsparameter zu illustrieren, stellt ein Stripchart alle vorkommenden Werte selbst dar. Zu diesem Zweck wird jeder Einzelwert als Punkt auf einer horizontalen Achse repräsentiert, wobei der Wert die x-Koordinate des Punkts bestimmt (Abb. 11.12).

```
> stripchart(x=⟨Vektor⟩, method="overplot", vertical=FALSE)
```

Die im Diagramm einzutragenden Daten werden als Vektor für `x` übergeben. In der Voreinstellung `"overplot"` bewirkt das Argument `method`, dass Bindungen durch dasselbe Symbol repräsentiert werden. Die so erstellte Grafik liefert damit keinen Aufschluss darüber, wie oft ein bestimmter Wert vorkommt. Um dies zu erreichen, gibt es zwei Methoden. Mit `method="jitter"` werden die Werte durch Symbole mit derselben x-Koordinate, aber einem zufälligen vertikalen Versatz dargestellt. Durch `method="stack"` werden die Symbole eines mehrfach vorkommenden Wertes vertikal gestapelt. `vertical=TRUE` vertauscht im Diagramm die Rolle von x- und y-Achse.

Ein Stripchart kann auch die Verteilung einer quantitativen Variable für mehrere Gruppen gleichzeitig veranschaulichen. Dafür können die Daten als Modellformel ⟨`AV`⟩ `~` ⟨`Faktor`⟩ übergeben werden, wobei ⟨`Faktor`⟩ dieselbe Länge wie der Vektor ⟨`AV`⟩ besitzt und für jeden von dessen Werten codiert, zu welcher Bedingung er gehört. In diesem Fall wird für jede Stufe von ⟨`Faktor`⟩ jeweils ein Stripchart der zugehörigen Daten im Diagramm eingezeichnet, wobei unterschiedliche Faktorstufen durch vertikal getrennte Achsen kenntlich gemacht werden (Abb. 11.12). Stammen die in der Modellformel verwendeten Variablen aus einem Datensatz, ist dieser unter `data` zu nennen.

```
> stripchart(DV ~ group, data=datW, xlab="Score", ylab="Gruppe", pch=1,
+            col="blue", main="Scores in 3 Gruppen",
+            sub="jitter-Methode", method="jitter")

> stripchart(DV ~ group, data=datW, xlab="Score", ylab="Gruppe", pch=16,
+            col="red", main="Scores in 3 Gruppen",
+            sub="stack-Methode", method="stack")
```

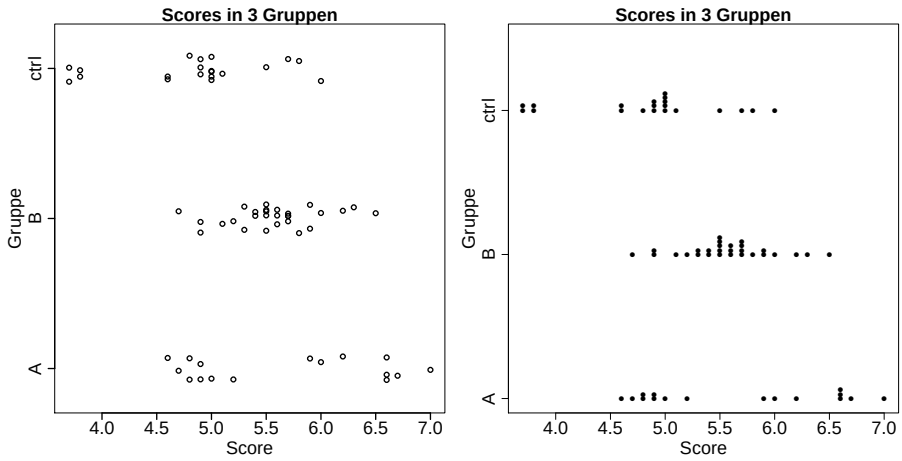

Abbildung 11.12: Stripcharts mit verschiedenen Methoden zur Darstellung einzelner Werte

11.6.4 Quantil-Quantil-Diagramm

Inwieweit die empirischen Werte einer Variable mit der Annahme einer bestimmten Verteilung verträglich sind, kann mit einem Q-Q-Diagramm (Quantil-Quantil Darstellung) heuristisch

abgeschätzt werden.

Vergleich zweier Stichproben

Wenn die Verteilung eines Merkmals in zwei Gruppen identisch ist, stimmen auch die Quantile überein (vgl. Abschn. 6.2.3). Dafür lassen sich die beobachteten Quantile von zwei Variablen zu denselben Wahrscheinlichkeiten in einem Diagramm gegeneinander auftragen. Sind ihre Quantile identisch, fallen die Punkte bei gleicher Achsenskalierung auf die Winkelhalbierende. Unterscheiden sich die Verteilungen lediglich durch ihre Skalierung, fallen die Punkte auf eine Gerade. Ein solches Diagramm erstellt `qqplot(x=⟨Vektor⟩, y=⟨Vektor⟩)` (Abb. 11.13). Für `x` und `y` sind dafür die zu vergleichenden Variablen einzutragen, die auch unterschiedliche Länge haben können.

```
> DV1 <- rnorm(200)               # normalverteilte Messwerte simulieren
> DV2 <- rf(100, df1=3, df2=15)   # F-verteilte Werte simulieren, anderes N
> qqplot(DV1, DV2, xlab="Quantile N(0, 1)", ylab="Quantile F(3, 15)",
+        main="Quantile F(3, 15) ~ N(0, 1)-Verteilung")
```

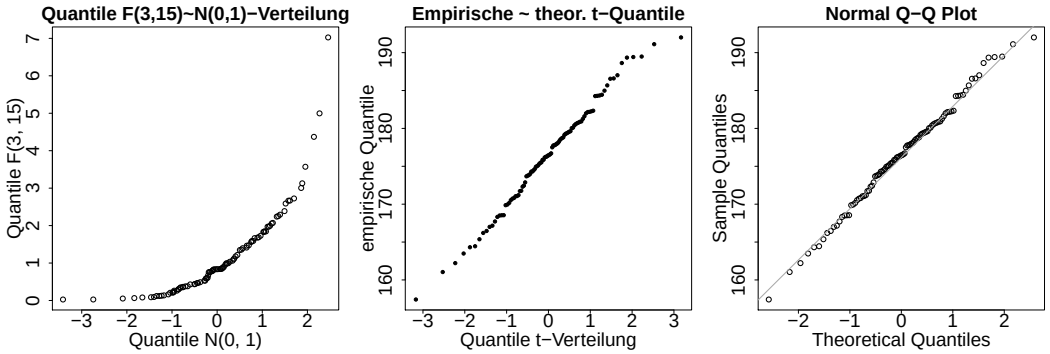

Abbildung 11.13: Quantil-Quantil Darstellung zum Vergleich von zwei Stichproben, zur Überprüfung von Normalverteiltheit und zum Vergleich mit t-Verteilung

Vergleich mit theoretischer Verteilung

Die Quantile von Daten einer Stichprobe können auch mit Quantilen verglichen werden, die sich aus der Annahme einer bestimmten Verteilung ergeben. Hierfür werden die erwarteten Quantile als x- und die tatsächlichen Quantile als y-Koordinaten von Punkten in einem Streudiagramm verwendet. Die empirischen Quantile sind die sortierten Werte eines Vektors. Die Funktion `ppoints(⟨Vektor⟩)` bestimmt, welches der zu einem Quantil gehörende Wert der empirischen kumulierten Häufigkeitsverteilung als Schätzung der Verteilungsfunktion ist. Ihre Ausgabe kann als Argument für die Quantilfunktion einer infrage kommenden Verteilung verwendet werden, um die theoretischen Quantile zu erhalten (vgl. Abschn. 6.2.3). Quantile zu denselben Wahrscheinlichkeiten lassen sich dann mit `plot()` gegeneinander auftragen.

```
> height <- rnorm(100, mean=175, sd=7)
> cProb  <- ppoints(height)       # kumulierte Wkt. für t-Quantile
> qTheo  <- qt(cProb, df=10)      # t-Quantile
```

```
> qEmp   <- sort(height)          # empirische Quantile

# Q-Q-Plot Vergleich empirische Quantile mit t-Quantilen
> plot(qTheo, qEmp, main="Empirische ~ theoretische t-Quantile",
+      xlab="Quantile t-Verteilung", ylab="empirische Quantile", pch=20)
```

Für den Spezialfall eines Vergleichs mit der Standardnormalverteilung existiert `qqnorm()`. Eine Referenzgerade wird durch `qqline()` in das Diagramm eingetragen. Im Vergleich zu `qqplot()` entfällt hier die Angabe des ersten Datenvektors, da die x-Koordinaten von den theoretisch erwarteten Quantilen gebildet werden (Abb. 11.13).

```
> qqnorm(height)
> qqline(height, col="red", lwd=2)
```

11.6.5 Kreisdiagramm

Das Kreis- oder auch Tortendiagramm ist eine Möglichkeit, die Werte einer diskreten Variable grob zu veranschaulichen. Hier repräsentiert die Größe eines farblich hervorgehobenen Kreissektors den Anteil des zugehörigen Wertes an der Summe aller Werte (Abb. 11.14).

```
> pie(x=⟨Vektor⟩, labels="⟨Namen⟩", col="⟨Farben⟩")
```

Für `x` ist ein Datenvektor mit nicht negativen Werten anzugeben. Sollen die einzelnen Sektoren mit einer Bezeichnung versehen werden, kann ein Vektor aus entsprechenden Zeichenketten für das Argument `labels` übergeben werden. Besitzt `x` benannte Elemente, dienen die Namen automatisch als Sektoren-Labels. Das Argument `col` kontrolliert die Farbe der Sektoren.

```
> tab <- xtabs(~ eye, data=datW)    # absolute Häufigkeiten Augenfarbe
> pie(tab, col=1:4, main="Häufigkeiten von Augenfarben")
```

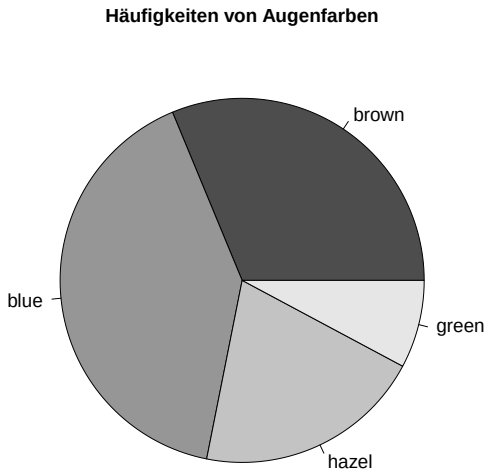

Abbildung 11.14: Kreisdiagramm als Mittel zur Darstellung von Kategorienhäufigkeiten

11.6.6 Gemeinsame Verteilung zweier Variablen

Die in Abschn. 11.2 vorgestellte Funktion `plot()` eignet sich dafür, die gemeinsame Verteilung von zwei Variablen in Form eines Streudiagramms zu untersuchen. Stammen die Daten aus verschiedenen Gruppen, kann die Gruppenzugehörigkeit über die Farbe oder den Typ der Datenpunktsymbole gekennzeichnet werden. Hierfür lässt sich die Eigenschaft von Faktoren ausnutzen, dass ihre Stufen intern über natürliche Zahlen repräsentiert sind, die sich mit `unclass()` ausgeben lassen und damit als Indizes dienen können (Abb. 11.15).

```
# Datensatz der Gruppen-Zentroide
> ctrJ <- aggregate(cbind(verbal.pre, verbal.post) ~ sex,
+                   FUN=mean, data=datW)

# gemeinsame Verteilung mit 2 Datenpunktsymbolen und Legende
> plot(verbal.post ~ verbal.pre, data=datW, pch=c(4, 16)[unclass(sex)],
+      lwd=2, col=c("black", "darkgray")[unclass(sex)],
+      main="Gemeinsame Verteilung getrennt nach Geschlecht")

# Zentroide einzeichnen
> points(verbal.post ~ verbal.pre, data=ctrJ, pch=c(4, 16),
+        col=c("black", "darkgray"), cex=3, lwd=5)

> legend(x="topleft", legend=c("female", "male"), pch=c(4, 16),
+        col=c("black", "darkgray"))
```

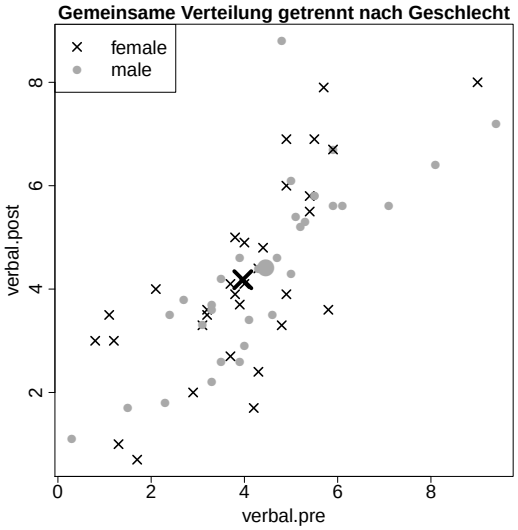

Abbildung 11.15: Gemeinsame Verteilung zweier Variablen getrennt nach Gruppen mit Zentroiden

Sind nur wenige Wertepaare möglich und deswegen Bindungen in den Daten vorhanden, würden mehrere gleiche Wertepaare durch dasselbe Symbol im Diagramm dargestellt. Sollen dagegen unter Verzicht auf die präzise Positionierung ebenso viele Symbole wie Wertepaare angezeigt werden, kann dies mit `jitter(⟨Variable⟩)` erreicht werden. Diese Funktion ändert die Werte

der Variable um einen kleinen zufälligen Betrag und bewirkt dadurch beim Zeichnen jedes Datenpunkts einen zufälligen Versatz entlang der zugehörigen Achse (Abb. 11.16).

```
> vec1 <- sample(1:10, 100, replace=TRUE)
> vec2 <- sample(1:10, 100, replace=TRUE)
> plot(vec2 ~ vec1, main="Punktwolke")      # Datenpunkte ohne jitter()

# Datenpunkte + zufälliger Versatz x- und y-Richtung durch jitter()
> plot(jitter(vec2) ~ jitter(vec1), main="Punktwolke mit jitter")
```

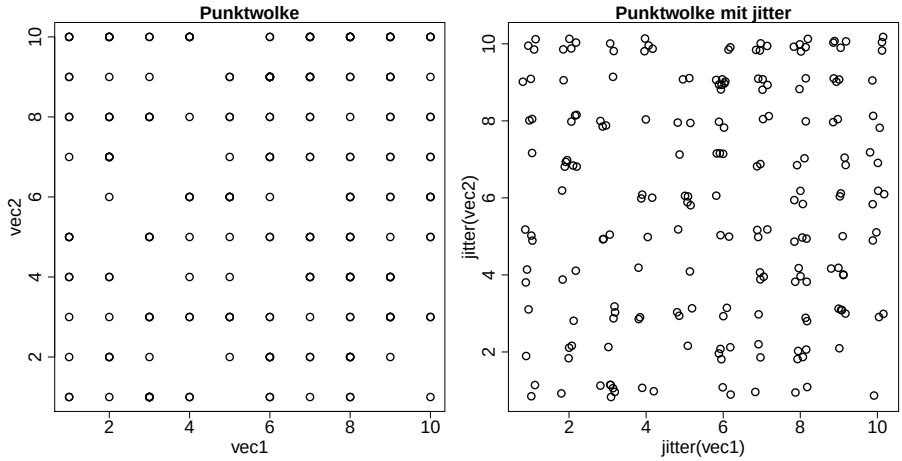

Abbildung 11.16: Streudiagramm von Daten mit Bindungen ohne und mit Anwendung von `jitter()`

Alternativ lässt sich mittels alpha-blending (vgl. Abschn. 11.3.2) jeder Punkt transparent machen, wodurch viele übereinander liegende Punkte an gesättigteren Farben zu erkennen sind. Mit der Funktion `hexbin()` aus dem gleichnamigen Paket (Carr, Lewin-Koh & Maechler, 2020) lässt sich ein Diagramm erstellen, das die Diagrammfläche in hexagonale Regionen einteilt und die Dichte der Datenpunkte in jeder Region ähnlich einem Höhenlinien-Diagramm (vgl. Abschn. 11.7.1) farblich repräsentiert. Auch `smoothScatter()` codiert die Dichte von Datenpunkten über farblich abgesetzte Diagrammregionen, verwendet dabei jedoch einen 2D-Kerndichteschätzer zur Glättung der Regionengrenzen.

11.7 Multivariate Daten visualisieren

Da Diagramme nur zweidimensional sein können, ergibt sich bei der Visualisierung der gemeinsamen Verteilung von drei oder mehr Variablen das Problem, dass die räumliche Lage eines Datenpunkts in der Plot-Region nicht mehr als zwei Komponenten repräsentieren kann. Für drei Variablen existieren als Ausweg verschiedene Diagrammtypen, die sich darin unterscheiden, auf welche Weise die dritte Komponente grafisch codiert wird: So wird versucht, einen räumlichen Tiefeneindruck zu erzeugen und die dritte Komponente als Höhe eines Datenpunkts über einer Ebene zu repräsentieren (dreidimensionale Streudiagramme und Gitterflächen). Oder die dritte Komponente wird nicht räumlich, sondern farblich bzw. durch andere grafische Unterscheidungsmerkmale symbolisiert, etwa durch Höhenlinien.

Bei mehr als drei Variablen kann auf die direkte Veranschaulichung aller Komponenten zugunsten einer aufgeteilten Grafik verzichtet werden, in der eine Serie uni- oder bivariater Diagramme nebeneinander für alle Stufen bzw. Stufenkombinationen weiterer Variablen angeordnet ist.

11.7.1 Höhenlinien

Den Diagrammtypen, die Höhenlinien oder Gitter zur Visualisierung der z-Koordinate verwenden, ist die Art der Angabe von Koordinaten gemein. Sie benötigen zum einen zwei Vektoren, die die Werte auf der x- und y-Achse festlegen. Als drittes Argument erwarten die Funktionen eine Matrix, die Werte für jede Kombination der übergebenen (x, y)-Koordinaten enthält und deswegen so viele Zeilen wie x- und so viele Spalten wie y-Koordinaten besitzt. Die Werte dieser Matrix definieren die z-Koordinate als dritte Komponente für jedes (x, y)-Koordinatenpaar.

Höhenlinien symbolisieren die z-Koordinate wie topografische Karten durch die Zugehörigkeit eines Punkts zu einer Region der Diagrammfläche, die durch eine geschlossene Höhenlinie definiert wird. Dieses Vorgehen ist mit einer Vergröberung der Daten verbunden, da Wertebereiche in Kategorien zusammengefasst werden.

```
> contour(x=⟨x-Koordinaten⟩, y=⟨y-Koordinaten⟩, z=⟨z-Koordinaten⟩,
+         nlevels=⟨Anzahl Höhenlinien⟩, levels=⟨Kategorien⟩,
+         labels="⟨Namen⟩", drawlabels=TRUE)
```

Für `x` und `y` muss jeweils ein Vektor mit den x- bzw. y-Koordinaten der Datenpunkte übergeben werden. Die Matrix `z` definiert die z-Koordinaten in der o. g. Form. Welche Kategorien Verwendung finden, kann über die Argumente `levels` und `nlevels` kontrolliert werden. Für `levels` ist ein Vektor aus Kategorienbezeichnungen zu übergeben. In diesem Fall erübrigt sich die Verwendung von `nlevels`, da die Zahl der Kategorien aus der Länge von `levels` folgt. Andernfalls ist für `nlevels` die gewünschte Anzahl an Kategorien zu nennen. Das Argument `drawlabels` bestimmt, ob die Kategorienbezeichnungen im Diagramm eingetragen werden.

Im folgenden Beispiel soll die zweidimensionale Sinc-Funktion dargestellt werden (Abb. 11.17).

```
> X <- Y <- seq(-10, 10, length.out=50)
> f <- function(x, y) { r <- sqrt(x^2+y^2); 10 * sin(r)/r }
> Z <- outer(X, Y, FUN=f)
> contour(X, Y, Z, main="Höhenlinien für 2D Sinc-Funktion")
```

Die durch die Höhenlinien definierten Regionen können mit `filled.contour()` auch eingefärbt werden. Dabei stammen die Farben aus einem Farbverlauf, dessen Zuordnung zu Kategorien auf der rechten Seite des Diagramms in einer Legende erläutert wird (Abb. 11.17). Der Aufruf gleicht dem für `contour()` stark, jedoch kann über das zusätzliche Argument `color.palette` eine eigene Farbpalette spezifiziert werden.

```
> filled.contour(X, Y, Z, main="Farbige Höhenlinien für 2D Sinc-Funktion")
```

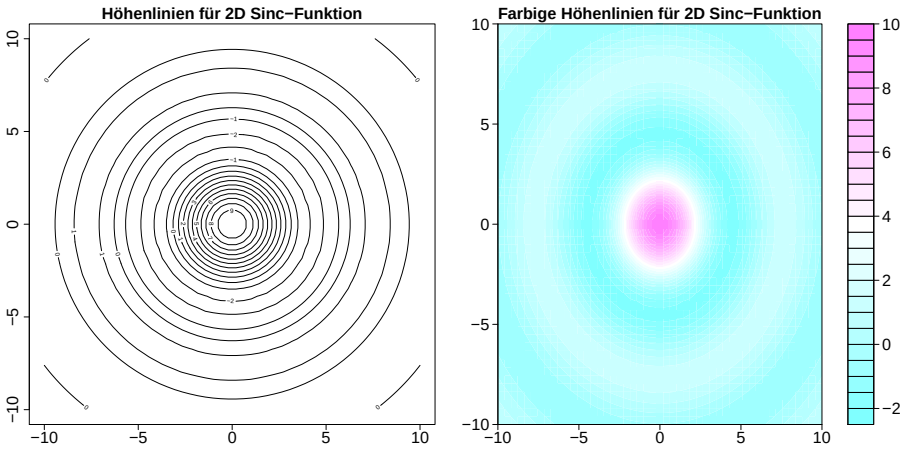

Abbildung 11.17: Höhenlinien für dreidimensionale Daten

11.7.2 Dreidimensionale Gitter und Streudiagrammme

Die Funktion `persp()` erzeugt Diagramme mit Tiefeneindruck, in der die Datenpunkte zu einer Gitterfläche verbunden sind. Der Augpunkt, d. h. die Perspektive, aus der die simulierte dreidimensionale Szene gezeigt wird, ist frei wählbar (Abb. 11.18).

```
> persp(x=⟨x-Koordinaten⟩, y=⟨y-Koordinaten⟩, z=⟨z-Koordinaten⟩,
+       theta=0, phi=15, r=sqrt(3))
```

Die Argumente `x`, `y` und `z` haben dieselbe Bedeutung wie in Höhenlinien-Diagrammen. Die Argumente `theta`, `phi` und `r` bestimmen die Blickrichtung auf das Diagramm in Form von Polarkoordinaten, die innerhalb einer gedachten Kugel mit dem Ursprung des Koordinatensystems im Zentrum definiert sind. Dabei bezeichnet `theta` den Azimuth (Längengrad) und `phi` die Höhe über dem Äquator (Breitengrad, Elevation). Die Entfernung zum Diagramm als Kugelradius kontrolliert `r`. Die Funktion verfügt über weitere Argumente, u. a. zur Kontrolle der perspektivischen Verzerrung und zur räumlichen Kompression der z-Achse.

```
> persp(X, Y, Z, xlab="x", ylab="y", zlab=NA, theta=5, phi=35,
+       main="2D Sinc-Funktion")
```

Mit Funktionen aus dem Paket `rgl` (Adler & Murdoch, 2021) erstellte Diagramme – etwa `plot3d()`, `hist3d()` oder `persp3d()` als Pendants zu den konventionellen Funktionen `plot()`, `hist()` und `persp()`, erlauben eine interaktive Manipulation: Durch Anklicken der Diagrammfläche lässt sich die Perspektive auf das Diagramm beliebig ändern, indem die linke Maustaste gedrückt gehalten und die Maus bewegt wird (Abb. 11.18).

```
> plot3d(x=⟨x-Koordinaten⟩, y=⟨x-Koordinaten⟩, z=⟨x-Koordinaten⟩)
```

Im Unterschied zu `contour()` und `persp()` müssen die (x, y, z)-Koordinaten der Punkte hier in Form dreier Vektoren gleicher Länge an die Argumente `x`, `y` und `z` übergeben werden. Das `rgl` Paket bringt eigene Funktionen zum Einfügen aller in Abschn. 11.5 für zweidimensionale Diagramme beschriebenen Grafikelemente mit – vgl. `help(package="rgl")` sowie allgemein `demo(rgl)` für die Gestaltungsmöglichkeiten des Paketes.

```
> vecX <- rep(seq(-10, 10, length.out=10), times=10)
> vecY <- rep(seq(-10, 10, length.out=10),  each=10)
> vecZ <- vecX*vecY
> library(rgl)                                    # für plot3d()
> plot3d(vecX, vecY, vecZ, main="3D Scatterplot",
+        col="blue", type="h", aspect=TRUE)

> spheres3d(vecX, vecY, vecZ, col="red", radius=2)
> grid3d(c("x", "y+", "z"))
```

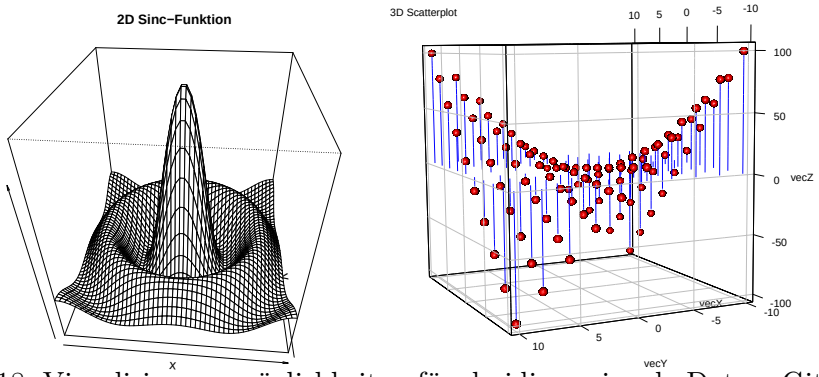

Abbildung 11.18: Visualisierungsmöglichkeiten für dreidimensionale Daten: Gitter und Streudiagramm

11.7.3 Matrix aus Streudiagrammen

Enthält ein Datensatz viele Variablen, können ihre paarweise gebildeten gemeinsamen Verteilungen in einer Matrix aus Streudiagrammen simultan veranschaulicht werden, wie sie `pairs()` erzeugt. Die Matrix enthält jeweils so viele Zeilen und Spalten, wie Variablen vorhanden sind, wobei sich in einer Zelle (i, j) das Streudiagramm der gemeinsamen Verteilung der i-ten und j-ten Variable befindet. Auf der Hauptdiagonale werden in der Voreinstellung die Variablennamen ausgegeben. An der Hauptdiagonale gespiegelte Zellen ((i, j) und (j, i)) stellen dieselbe gemeinsame Verteilung dar, allerdings mit vertauschten Achsen (Abb. 11.19).

```
> pairs(x=⟨Matrix⟩, labels=⟨Variablennamen⟩, panel=⟨Zeichenfunktion⟩,
+       lower.panel=⟨Zeichenfunktion⟩, upper.panel=⟨Zeichenfunktion⟩,
+       diag.panel=⟨Zeichenfunktion⟩)
```

Für `x` ist eine spaltenweise aus Variablen gebildete Matrix (oder ein Datensatz) anzugeben, deren paarweise gemeinsame Verteilungen dargestellt werden sollen. Ergeben sich die Variablennamen nicht aus den Spaltennamen von `x`, können sie als Vektor für `labels` genannt werden. `panel` erwartet den Namen einer Zeichenfunktion, deren Ergebnis in den Zellen außerhalb der Diagonale der erzeugten Grafik-Matrix abgebildet wird. Die Funktion muss als Argumente zwei Vektoren (Spalten von `x`) verarbeiten können und darf kein neues Diagramm öffnen, muss also eine Low-Level-Funktion sein. Voreinstellung ist `points` für ein Streudiagramm. Sollen die Zellen oberhalb und unterhalb der Hauptdiagonale der Grafik-Matrix unterschiedliche

Diagrammtypen zeigen, können für `lower.panel` und `upper.panel` separate Zeichenfunktionen angegeben werden, die dieselben Bedingungen wie jene für `panel` erfüllen. Durch Angabe einer Zeichenfunktion für `diag.panel`, die als Argument die Daten der Variable i erhält, lassen sich auch auf der Diagonale Diagramme einfügen (Voreinstellung ist `NULL` für die Ausgabe der Variablennamen).

Im Beispiel seien VPn in zwei Gruppen aufgeteilt und an ihnen vier AVn erhoben worden. Die Streudiagramme sollen die Gruppenzugehörigkeit über Art und Farbe der Datenpunktsymbole kenntlich machen (vgl. Abschn. 11.6.6).

```
> pairs(subset(datW, select=c(education, iq, attention, verbal)),
+       main="Streudiagramm-Matrix", pch=c(4, 16)[unclass(datW$sex)],
+       col=c("red", "blue")[unclass(datW$sex)])
```

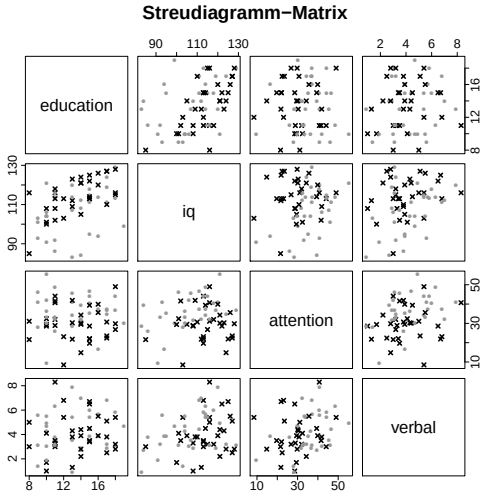

Abbildung 11.19: Matrix von paarweisen Streudiagrammen mehrerer Variablen mit `pairs()`

11.8 Mehrere Diagramme in einem Grafik-Device darstellen

In ein Device können auch mehrere Diagramme gezeichnet werden. Sollen nicht nur Diagramme gleichen Typs, sondern auch unterschiedliche Grafiken in einem Device dargestellt werden, lässt sich dessen Fläche manuell in mehrere rechteckige Bereiche aufteilen. Diese können dann mit jeweils einem Diagramm gefüllt werden. Auf diese Weise simultan dargestellte Diagramme erleichtern den Vergleich von Ergebnissen in verschiedenen Teilstichproben, lassen sich aber auch nutzen, um dieselben Daten parallel auf verschiedene Arten zu visualisieren.

Ein Weg Device-Flächen zu unterteilen, ist das Argument `mfrow` von `par()`.[1]

```
> par(mfrow=c(⟨Anzahl Zeilen⟩, ⟨Anzahl Spalten⟩))
```

Als Argument ist ein Vektor mit zwei Elementen zu übergeben, die die Anzahl der Zeilen und Spalten definieren, aus denen sich dann die einzelnen Regionen der Device-Fläche ergeben.

[1] Für flexiblere Alternativen vgl. `?layout` und `?split.screen`.

Diese Regionen besitzen dieselbe Größe. Mit `mfrow` werden durch sich anschließende High-Level-Funktionen nacheinander die Zeilen mit Diagrammen gefüllt (Abb. 11.20).

```
> dev.new(width=14, height=7)                    # Diagrammgröße ändern
> par(mfrow=c(1, 2))                             # Diagrammfenster aufteilen
> boxplot(rt(100, 5), xlab=NA, notch=TRUE, main="Boxplot")
> plot(rnorm(10), pch=16, xlab=NA, ylab=NA, main="Streudiagramm")
```

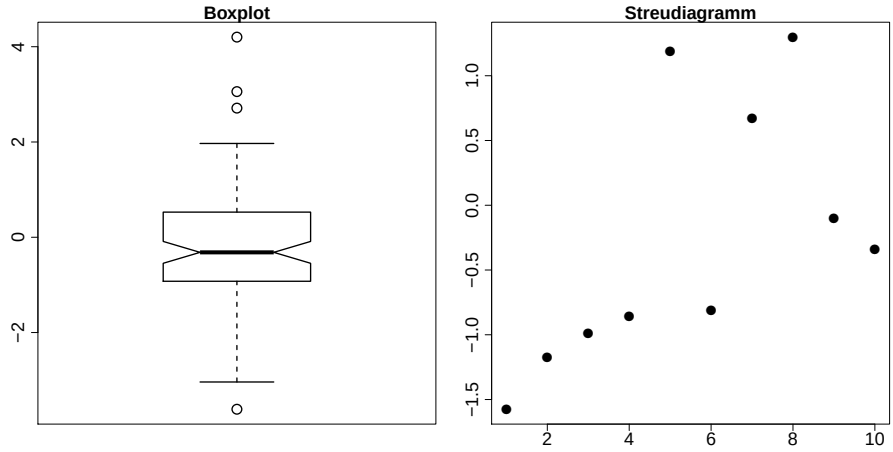

Abbildung 11.20: Mehrere Diagramme mit `par(mfrow)` in ein Device zeichnen