Eine kleine Einführung in R

Andreas Handl

Torben Kuhlenkasper

8. Januar 2016

Grundlage des vorliegenden Skripts sind Aufzeichnungen von Andreas Handl, die er bis zum Jahr 2007 an der Universität Bielefeld verfasst und für seine Lehrveranstaltungen verwendet hat. Seit 2012 werden die Skripten von Torben Kuhlenkasper weitergeführt sowie fortlaufend aktualisiert und erweitert.

Anmerkungen und Vorschläge zur Verbesserung und Ergänzung sind jederzeit willkommen und können an `statistik@kuhlenkasper.de` *gesendet werden. Weitere Skripten sind unter* `www.skripten.kuhlenkasper.de` *zu finden.*

Inhaltsverzeichnis

1 R als mächtiger Taschenrechner **4**

2 Datenstrukturen **6**

3 Einlesen von Daten aus externen Dateien **16**

4 Selektion unter Bedingungen **19**

5 Grafiken in R **24**

6 Wie schreibt man eine Funktion? **33**

7 Pakete **38**

Da die Datensätze in diesem Skript klein sind, kann man alle Beispiele mit Papier, Bleistift und Taschenrechner in vertretbarer Zeit nachvollziehen. Bei größeren Datensätzen sollte man auf die Hilfe von Computern zurückgreifen. Hier kann der Anwender statistischer Verfahren unter einer Vielzahl von Statistikpaketen wählen. Unter diesen werden SAS, SPSS und STATA bei der Mehrzahl der professionellen Datenanalysen verwendet. Die drei genannten Pakete sind aber sehr teuer und es ist nicht einfach, neue Verfahren zu implementieren. Das Statistikpaket R erfreut sich sowohl an Hochschulen als auch in beruflichen Anwendungen immer größerer Beliebtheit. In R sind sehr viele statistische Verfahren vorhanden und es ist im Internet frei erhältlich. Es steht für die gängigen Betriebssysteme Microsoft Windows, Mac OS X und verschiedene Linux-Distributionen zur Verfügung, aktuell in der Version 3.0.1. Unter der Adresse

`http://cran.r-project.org`

kann R heruntergeladen werden.

1 R als mächtiger Taschenrechner

Befehlsmodus

R bietet eine **interaktive Umgebung**, den **Befehlsmodus**, in dem man die Daten direkt eingeben und analysieren kann. Nach dem Start des Programmes wird durch das **Bereitschaftszeichen** `>` angezeigt, dass eine Eingabe erwartet wird. Der Befehlsmodus ist ein mächtiger Taschenrechner. Wir können hier die Grundrechenarten Addition, Subtraktion, Multiplikation und Division mit den Operatoren `+`, `-`, `*` und `/` durchführen. Bei Dezimalzahlen verwendet man einen Dezimalpunkt und nicht das in Deutschland oft verwendete Dezimalkomma. Nachdem wir einen Befehl mit der Taste `return` abgeschickt haben, gibt R das Ergebnis in der nächsten Zeile aus. Hier sind einige einfache Beispiele:

```
> 2.1+2
[1] 4.1

> 2.1-2
[1] 0.1

> 2.1*2
[1] 4.2

> 2.1/2
```

```
[1] 1.05
```

Zum Potenzieren benutzen wir ^ :

```
> 2.1^2
[1] 4.41
```

Die Quadratwurzel von 2 erhalten wir also durch

```
> 2^0.5
[1] 1.414214
```

Man kann aber auch die **Funktion** sqrt verwenden. Dabei ist sqrt eine Abkürzung für square root, also Quadratwurzel. Namen von Funktionen sind in R unter mnemotechnischen Gesichtspunkten gewählt. Funktionen bieten die Möglichkeit, einen oder mehrere Befehle unter einem Namen abzuspeichern. Funktionen besitzen in der Regel **Argumente**. So muss man der Funktion sqrt mitteilen, von welcher Zahl sie die Quadratwurzel bestimmen soll. Diese Zahl ist Argument der Funktion sqrt. Die Argumente einer Funktion stehen in runden Klammern hinter dem Funktionsnamen und sind durch Kommata voneinander getrennt. Wir rufen die Funktion sqrt also mit dem Argument 2 auf:

Funktion
sqrt

Argument

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

R führt die Berechnung auf sehr viele Stellen genau nach dem Dezimalpunkt aus, zeigt jedoch weniger Stellen an. Soll das ausgebebene Ergebnis noch übersichtlicher werden, sollten wir runden und verwenden hierzu die Funktion round. Dabei können wir der Funktion round den Aufruf der Funktion der Funktion sqrt als Argument übergeben, was bei allen Funktionen möglich ist.

round

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

Jetzt ist das Ergebnis übersichtlich aber ungenau. Wir müssen der Funktion round also noch mitteilen, auf wie viele Stellen nach dem Dezimalpunkt wir runden wollen. Wie wir dies erreichen können, erfahren wir, indem wir die Funktion help mit dem Argument round aufrufen. Alternativ können die jeweilige Hilfeseite zu einer Funktion aufrufen, indem wir dem Namen der Funktion ein ? voranstellen: ?round oder help(round) öffnet die Hilfeseite für die Funktion round. Wir sehen, dass die Funktion folgendermaßen aufgerufen wird

help

```
round(x, digits = 0)
```

Neben dem ersten Argument, bei dem es sich um die zu rundende Zahl handelt, gibt es noch das Argument `digits`. Dieses gibt die Anzahl der Stellen nach dem Dezimalpunkt an, auf die gerundet werden soll, und nimmt standardmäßig den Wert 0 an.

`digits`

Funktionen in R besitzen zwei Typen von Argumenten. Es gibt Argumente, die beim Aufruf der Funktion angegeben werden müssen. Bei der Funktion `round` ist dies das Argument `x`. Es gibt aber auch optionale Argumente, die nicht angegeben werden müssen. In diesem Fall wird ihnen der Wert zugewiesen, der in der Kopfzeile zu finden ist. Das Argument `digits` nimmt also standardmäßig den Wert 0 an.

Wie übergibt man einer Funktion, die mindestens zwei Argumente besitzt, diese? Hierzu gibt es eine Reihe von Möglichkeiten, die wir an Hand der Funktion `round` illustrieren wollen. Kennt man die Reihenfolge der Argumente im Kopf der Funktion, so kann man sie ohne zusätzliche Angaben eingeben.

```
> round(sqrt(2),2)
[1] 1.41
```

Man kann aber auch die Namen der Argumente verwenden, wie sie im Kopf der Funktion stehen.

```
> round(x=sqrt(2),digits=2)
[1] 1.41
```

Verwendet man die Namen, so kann man die Argumente in beliebiger Reihenfolge eingeben.

```
> round(digits=2,x=sqrt(2))
[1] 1.41
```

Man kann die Namen der Argumente abkürzen, wenn sie dadurch eindeutig bleiben. Beginnen zwei Namen zum Beispiel mit `di`, so darf man `di` nicht als Abkürzung verwenden.

```
> round(x=sqrt(2),d=2)
[1] 1.41
```

2 Datenstrukturen

Bei statistischen Erhebungen werden bei jedem von n Merkmalsträgern jeweils p Merkmale erhoben. In diesem Kapitel werden wir lernen, wie man

die Daten eingibt und unter einem Namen abspeichert, mit dem man auf sie zurückgreifen kann.
Wir gehen zunächst davon aus, dass nur ein Merkmal erhoben wurde. Schauen wir uns ein Beispiel an.
Ein Schallplattensammler hat im letzten halben Jahr fünf Langspielplatten bei einem amerikanischen Händler gekauft und dafür folgende Preise in US Dollar bezahlt:

```
 22  30  16  25  27
```

Vektor
Komponente
`c`

Wir geben die Daten als **Vektor** ein. Ein Vektor ist eine Zusammenfassung von Objekten zu einer endlichen Folge und besteht aus **Komponenten**. Einen Vektor erzeugt man in R mit der Funktion `c` . Diese macht aus einer Folge von Zahlen, die durch Kommata getrennt sind, einen Vektor, dessen Komponenten die einzelnen Zahlen sind. Die Zahlen sind die Argumente der Funktion `c`. Wir geben die Daten ein.

```
> c(22,30,16,25,27)
```

Am Bildschirm erhalten wir folgendes Ergebnis:

```
[1] 22 30 16 25 27
```

Die Elemente des Vektors werden ausgegeben. Am Anfang steht `[1]`. Dies zeigt, dass die erste ausgebebene Zahl gleich der ersten Komponente des Vektors ist.

Variable
`<-`

Um mit den Werten weiterhin arbeiten zu können, müssen wir sie in einer **Variablen** speichern. Dies geschieht mit dem **Zuweisungsoperator** `<-` , den man durch die Zeichen `<` und `-` erhält. Auf der linken Seite steht der Name der Variablen, der die Werte zugewiesen werden sollen, auf der rechten Seite steht der Aufruf der Funktion `c`.
Die Namen von Variablen dürfen beliebig lang sein, dürfen aber nur aus Buchstaben, Ziffern und dem Punkt bestehen, wobei das erste Zeichen ein Buchstabe oder der Punkt sein muss. Beginnt ein Name mit einem Punkt, so dürfen nicht alle folgenden Zeichen Ziffern sein. Hierdurch erzeugt man nämlich eine Zahl.
Wir nennen die Variable `lp`. und geben ein

```
> lp<-c(22,30,16,25,27)
```

Workspace
`rm`

Eine Variable bleibt während der gesamten Sitzung im **Workspace** erhalten, wenn sie nicht mit dem Befehl `rm` gelöscht wird. Beim Verlassen von `R` durch

q() wird man gefragt, ob man den Workspace sichern will. Antwortet man mit ja, so sind auch bei der nächsten Sitzung alle Variablen vorhanden. Mit der Funktion ls kann man durch den Aufruf ls() alle Objekte im Workspace auflisten.

q

ls

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

Den Inhalt einer Variablen kann man sich durch Eingabe des Namens anschauen. Der Aufruf

```
> lp
```

liefert das Ergebnis

```
[1] 22 30 16 25 27
```

R unterscheidet Groß- und Kleinschreibung. Die Variablennamen lp und Lp beziehen sich also auf unterschiedliche Objekte.

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

Die Preise der Langspielplatten sind in US Dollar. Am 15.5.2006 kostete ein US Dollar 0.774 EURO. Um die Preise in EURO umzurechnen, muss man jeden Preis mit 0.774 multiplizieren. Um alle Preise umzurechnen, multiplizieren wir den Vektor lp mit 0.774

```
> 0.774*lp
[1] 17.028 23.220 12.384 19.350 20.898
```

Um das Ergebnis auf zwei Stellen zu runden, benutzen wir die Funktion round:

```
> round(0.774*lp,2)
[1] 17.03 23.22 12.38 19.35 20.90
```

Die Portokosten betragen jeweils 12 US Dollar. Wir addieren zu jeder Komponente von lp die Zahl 12

```
> lp+12
[1] 34 42 28 37 39
```

Auf Komponenten eines Vektors greift man durch **Indizierung** zu. Hierzu gibt man den Namen des Vektors gefolgt von eckigen Klammern ein, zwischen denen die Nummer der Komponente oder der Vektor mit den Nummern der Komponenten steht, auf die man zugreifen will. Diese Nummern in den eckigen Klammern entsprechen also den jeweiligen Positionen der Komponenten innerhalb des Vektors. Um den Preis der ersten Platte zu erfahren, gibt man ein:

Indizierung
`[ ]`

```
> lp[1]
[1] 22
```

Um den Preis der Platte zu erhalten, die man zuletzt gekauft hatte, benötigt man die Länge des Vektors `lp`. Diesen liefert die Funktion `length`.

`length`

```
> length(lp)
[1] 5
> lp[length(lp)]
[1] 27
```

Wir können auch gleichzeitig auf mehrere Komponenten zugreifen:

```
> lp[c(1,2,3)]
[1] 22 30 16
```

Einen Vektor mit aufeinander folgenden natürlichen Zahlen erhält man mit dem Operator `:` . Schauen wir uns einige Beispiele an.

`:`

```
> 1:3
[1] 1 2 3
> 4:10
[1]  4  5  6  7  8  9 10
> 3:1
[1] 3 2 1
```

Wir können also auch

```
> lp[1:3]
[1] 22 30 16
```

eingeben um die ersten drei Elemente des Vektors zu erhalten.
Schauen wir uns noch einige Funktionen an, mit denen man Informationen aus einem Vektor extrahieren kann. Die Summe aller Werte liefert die Funktion `sum`:

`sum`

```
> sum(lp)
[1] 120
```

Das Minimum erhalten wir mit der Funktion `min` — `min`

```
> min(lp)
[1] 16
```

und das Maximum mit der Funktion `max` — `max`

```
> max(lp)
[1] 30
```

Die Funktion `sort` sortiert einen Vektor aufsteigend. — `sort`

```
> sort(lp)
[1] 16 22 25 27 30
```

Setzt man das Argument `decreasing` auf den Wert `TRUE`, so wird absteigend sortiert.

```
> sort(lp,decreasing=TRUE)
[1] 30 27 25 22 16
```

Die bisherigen Beispiele haben reelle Zahlen, wie sie z.B. bei quantitativen Merkmalen auftreten, verwendet. Wie gibt man die Daten bei einem qualitativen Merkmal ein? Beginnen wir auch hier mit einem Beispiel. Hier ist die Urliste des Geschlechts von 10 Teilnehmern eines Seminars:

```
w m w m w m m m w m
```

Wir geben die Urliste als Vektor ein, dessen Komponenten **Zeichenketten** sind. Eine Zeichenkette ist eine Folge von Zeichen, die in Hochkomma stehen. So sind `"Berlin"` und `"Bielefeld"` Zeichenketten. — **Zeichenkette**

Wir nennen den Vektor `Geschlecht`:

```
> Geschlecht<-c("w","m","w","m","w","m","m","m","w","m")
> Geschlecht
[1] "w" "m" "w" "m" "w" "m" "m" "m" "w" "m"
```

Mit der Funktion `factor` transformieren wir den Vektor `Geschlecht`, dessen Komponenten Zeichenketten sind, in einen Vektor, dessen Komponenten die Ausprägungen eines **Faktors**, also eines qualitativen Merkmals, sind — `factor` — **Faktor**

```
> Geschlecht<-factor(Geschlecht)
> Geschlecht
[1] w m w m w m m m w m
Levels: m w
```

Wir werden bald sehen, mit welchen Funktionen man Informationen aus Vektoren vom Typ `factor` extrahieren kann. Hier wollen wir nur zeigen, dass man diese wie auch Vektoren, deren Komponenten numerisch sind, indizieren kann.

```
> Geschlecht[2]
[1] m
Levels: m w
> Geschlecht[5:length(Geschlecht)]
[1] w m m m w m
Levels: m w
```

Bisher haben wir nur ein Merkmal betrachtet. Wir wollen nun zeigen, wie man vorgeht, wenn mehrere Merkmale eingegeben werden sollen. Hierbei gehen wir zunächst davon aus, dass alle Merkmale den gleichen Typ besitzen, also entweder alle quantitativ oder alle qualitativ sind. Wir illustrieren die Vorgehensweise an einem Beispiel.
Bei einer Befragung gaben zwei Personen ihr Alter, das Alter ihrer Mutter und das Alter ihres Vaters an. Die Daten sind in Tabelle 1 zu finden.

Tabelle 1: Alter

Alter	Alter der Mutter	Alter des Vaters
29	58	61
26	53	54

Matrix

Liegen die Daten wie in Tabelle 1 vor, so sollte man sie als **Matrix** eingeben. Eine Matrix ist ein rechteckiges Zahlenschema, das aus r Zeilen und s Spalten besteht.

`matrix`

In R erzeugt man eine Matrix mit der Funktion `matrix`. Der Aufruf der Funktion `matrix` ist

```
matrix(data,nrow=1,ncol=1,byrow=F)
```

Dabei ist `data` der Vektor mit den Elementen der Matrix. Das Argument `nrow` gibt die Anzahl der Zeilen und das Argument `ncol` die Anzahl der Spalten

der Matrix an. Standardmäßig wird eine Matrix spaltenweise eingegeben. Wir geben also ein:

```
> alter<-matrix(c(29,26,58,53,61,54),2,3)
```

und erhalten

```
> alter
     [,1] [,2] [,3]
[1,]   29   58   61
[2,]   26   53   54
```

Sollen die Zeilen aufgefüllt werden, so muss das Argument `byrow` auf den Wert `TRUE` gesetzt werden:

```
> alter<-matrix(c(29,58,61,26,53,54),2,3,TRUE)
> alter
     [,1] [,2] [,3]
[1,]   29   58   61
[2,]   26   53   54
```

Auf Elemente einer Matrix greifen wir wie auf Komponenten eines Vektors durch Indizierung zu, wobei wir die Informationen, die sich auf Zeilen beziehen, von den Informationen, die sich auf Spalten beziehen, durch Komma trennen. Um auf das Element in der ersten Zeile und zweiten Spalte zuzugreifen, geben wir also ein:

```
> alter[1,2]
[1] 58
```

Alle Elemente der ersten Zeile erhalten wir durch

```
> alter[1,]
[1] 29 58 61
```

und alle Elemente der zweiten Spalte durch

```
> alter[,2]
[1] 58 53
```

Die Summe aller Werte erhält man mit der Funktion `sum`:

```
> sum(alter)
[1] 281
```

Oft ist man an der Summe der Werte innerhalb der Zeilen oder Spalten interessiert. Diese liefern die Funktionen `colSums` und `rowSums`.

colSums
rowSums

```
> rowSums(alter)
[1] 148 133

> colSums(alter)
[1]  55 111 115
```

Man kann aber auch die Funktion `apply` anwenden. Diese wird aufgerufen durch

apply

```
apply(x,margin,fun)
```

und wendet auf die Dimension `margin` der Matrix `x` die Funktion `fun` an. Dabei entspricht die erste Dimension den Zeilen und die zweite Dimension den Spalten. Die Summe der Werte in den Zeilen erhalten wir also durch

```
> apply(alter,1,sum)
[1] 148 133
```

und die Summe der Werte in den Spalten durch

```
> apply(alter,2,sum)
[1]  55 111 115
```

Wir können für `fun` natürlich auch andere Funktionen wie `min` oder `max` verwenden.
Einen Vektor mit den Zeilenminima liefert der Aufruf

```
> apply(alter,1,min)
[1] 29 26
```

und einen Vektor mit den Spaltenmaxima der Aufruf

```
> apply(alter,2,max)
[1] 29 58 61
```

Jetzt schauen wir uns an, wie man Datensätze abspeichert, die sowohl qualitative als auch quantitative Merkmale enthalten. Wir betrachten wieder ein Beispiel.
Bei einer Befragung wurden das Geschlecht und das Alter von drei Personen erhoben. Die Daten sind in Tabelle 2 zu finden.

Tabelle 2: Alter

Geschlecht	Alter
m	29
w	26
m	24

In R bieten **Datentabellen** die Möglichkeit, die Werte von Merkmalen unterschiedlichen Typs in einer Variablen abzuspeichern. Dabei muss bei jedem Merkmal die gleiche Anzahl von Beobachtungen vorliegen. Eine Datentabelle wird mit dem Befehl `data.frame` erzeugt. Das Beispiel illustriert die Vorgehensweise.

Datentabelle

`data.frame`

```
> sexage<-data.frame(sex=c("m","w","m"),age=c(29,26,24))
> sexage
  sex age
1   m  29
2   w  26
3   m  24
```

Auf eine Datentabelle kann man wie auf eine Matrix zugreifen.

```
> sexage[2,2]
[1] 26

> sexage[2,]
 sex age
2   w  26

> sexage[,1]
[1] m w m
Levels: m w
```

Der letzte Aufruf zeigt, dass ein Vektor, der aus Zeichenketten besteht, bei der Erzeugung einer Datentabelle automatisch zu einem Faktor wird.

Datentabellen sind **Listen**, die wie Matrizen behandelt werden können. Wir wollen uns hier nicht detailliert mit Listen beschäftigen, sondern nur darauf hinweisen, dass Listen aus Komponenten bestehen, von denen jede einen anderen Typ aufweisen kann. So kann die erste Komponente einer Liste eine Zeichenkette, die zweite ein Vektor und die dritte eine Matrix sein. Auf die

Liste

Komponenten einer Liste greift man entweder mit einer doppelten eckigen Klammer oder mit Name der Liste Name der Komponente zu.

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

> sexage$sex
[1] m w m
Levels: m w

> sexage[[2]]
[1] 29 26 24

> sexage$age
[1] 29 26 24
```

Mit der Funktion `attach` kann man auf die in einer Datentabelle enthaltenen Variablen unter ihrem Namen zugreifen, ohne den Namen der Datentabelle zu verwenden. Mit der Funktion `detach` hebt man diese Zugriffsmöglichkeit auf.

`attach`

`detach`

```
> attach(sexage)
> sex
[1] m w m
Levels: m w

> age
[1] 29 26 24

> detach(sexage)

> sex
Fehler: objekt "sex" nicht gefunden
> age
Fehler: objekt "age" nicht gefunden
```

Eine Datentabelle kann dabei folgendermaßen aufgebaut sein:

```
 Geschlecht Alter Film Bewertung Geld Satz
          m    30    n      <NA>  1.8    n
          w    23    j         g  1.8    n
```

```
w    26   j        g  1.8    j
m    33   n     <NA>  2.8    n
m    37   n     <NA>  1.8    n
m    28   j        g  2.8    j
w    31   j       sg  2.8    n
m    23   n     <NA>  0.8    n
w    24   j       sg  1.8    j
m    26   n     <NA>  1.8    n
w    23   j       sg  1.8    j
m    32   j        g  1.8    n
m    29   j       sg  1.8    j
w    25   j        g  1.8    j
w    31   j        g  0.8    n
m    26   j        g  2.8    n
m    37   n     <NA>  3.8    n
m    38   j        g   NA    n
w    29   n     <NA>  3.8    n
w    28   j       sg  1.8    n
w    28   j        m  2.8    j
w    28   j       sg  1.8    j
w    38   j        g  2.8    n
w    27   j        m  1.8    j
m    27   n     <NA>  2.8    j
```

Fehlt für eine Variable eine Ausprägung, so wird dies in R mit `NA` ("Not Available") gekennzeichnet.

3 Einlesen von Daten aus externen Dateien

Datei

Oft liegen die Daten außerhalb von R in einer **Datei** vor. In diesem Fall müssen sie nicht noch einmal eingeben werden, sondern können eingelesen werden. Wir gehen im Folgenden davon aus, dass die Daten aus Tabelle **??** auf Seite **??** in einer **ASCII-Datei** gespeichert wurden. Sie sieht folgendermaßen aus

```
Geschlecht Alter Mutter Vater Geschwister
m          29    58     61    1
w          26    53     54    2
m          24    49     55    1
w          25    56     63    3
w          25    49     53    0
```

```
w           23    55    55    2
m           23    48    54    2
m           27    56    58    1
m           25    57    59    1
m           24    50    54    1
w           26    61    65    1
m           24    50    52    1
m           29    54    56    1
m           28    48    51    2
w           23    52    52    1
m           24    45    57    1
w           24    59    63    0
w           23    52    55    1
m           24    54    61    2
w           23    54    55    1
```

Die Daten mögen auf dem Laufwerk `d:` im Verzeichnis (Ordner) `daten` in der Datei `bidaten.txt` stehen. Wir lesen sie mit der Funktion `read.table` ein. Diese besitzt eine Vielzahl von Argumenten, von denen nur der Dateiname obligatorisch ist. Zu diesem gehört die vollständige Pfadangabe. Dabei müssen für jeden Backslash zwei Backslash eingegeben werden, da in R der Backslash in einer Zeichenkette ein Steuerzeichen ist.

`read.table`

Stehen in der Kopfzeile der Datei die Namen der Variablen, so muss das Argument `header` auf den Wert `TRUE` gesetzt werden. Ansonsten wird unterstellt, dass keine Kopfzeile existiert.

Wird bei Dezimalzahlen das Dezimalkomma verwendet, so setzt man das Argument `dec` auf den Wert `","`. Standardmäßig wird der Dezimalpunkt verwendet.

Mit dem Argument `sep` kann man festlegen, durch welches Zeichen die Spalten in der ursprünglichen Datei getrennt sind, wobei unterstellt wird, dass das Leerzeichen verwendet wird.

Wir lesen die Daten ein und weisen sie der Variablen `bidaten` zu.

```
> bidaten<-read.table("d:\\daten\\bidaten.txt",header=TRUE)
> bidaten
    Geschlecht Alter Mutter Vater Geschwister
 1           m    29     58    61           1
 2           w    26     53    54           2
 3           m    24     49    55           1
 4           w    25     56    63           3
 5           w    25     49    53           0
```

```
  6          w   23   55   55         2
  7          m   23   48   54         2
  8          m   27   56   58         1
  9          m   25   57   59         1
 10          m   24   50   54         1
 11          w   26   61   65         1
 12          m   24   50   52         1
 13          m   29   54   56         1
 14          m   28   48   51         2
 15          w   23   52   52         1
 16          m   24   45   57         1
 17          w   24   59   63         0
 18          w   23   52   55         1
 19          m   24   54   61         2
 20          w   23   54   55         1
```

Es wird eine Datentabelle erzeugt, auf die wir auf die im letzten Kapitel beschriebene Art und Weise zugreifen können.

```
> attach(bidaten)

     The following object(s) are masked _by_ .GlobalEnv :

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

Wir sehen, dass wir vorsichtig sein müssen. Auf Seite 10 haben wir eine Variable `Geschlecht` erzeugt. Die Datentabelle `bidaten` enthält eine Variable mit dem gleichen Namen. Nach Eingabe des Befehls `attach(bidaten)` stehen uns unter dem Namen `Geschlecht` die Daten der zuerst erzeugten Variablen zur Verfügung. Wir nennen diese `Ges`. Wenn wir danach noch die Variable `Geschlecht` mit dem Befehl `rm` löschen, können wir auf die Variable `Geschlecht` aus der Datentabelle `bidaten` zugreifen.

```
> Ges<-Geschlecht
> rm(Geschlecht)
> Geschlecht
[1] m w m w w w m m m m w m m m w m w w m w
Levels: m w
```

Man kann Daten auch aus der **Zwischenablage** einlesen. Hierzu wählt man als Dateinamen `"clipboard"`. Dies ist vor allem dann sinnvoll, wenn man Datensätze aus dem Internet einliest. Man markiert die Daten und kopiert sie in die Zwischenablage. Mit `read.table("clipboard")` werden sie in R eingelesen.

4 Selektion unter Bedingungen

Bei der Datenanalyse werden oft Gruppen hinsichtlich eines oder mehrerer Merkmale verglichen. So könnte bei den Daten aus Tabelle **??** auf Seite **??** interessieren, ob sich das Alter der Studenten vom Alter der Studentinnen unterscheidet. Um diese Frage beantworten zu können, müssen wir zum einen die Werte des Alters selektieren, bei denen das Merkmal `Geschlecht` den Wert `w` aufweist, und zum anderen die Werte des Merkmals `Alter` selektieren, bei denen das Merkmal Geschlecht den Wert `m` aufweist. Wir müssen also überprüfen, welche Komponenten eines Vektors eine **Bedingung** erfüllen.
Um Bedingungen zu überprüfen, kann man in R die **Operatoren**

`==`	gleich
`!=`	ungleich
`<`	kleiner
`<=`	kleiner oder gleich
`>`	größer
`>=`	größer oder gleich

verwenden. Mit diesen Operatoren vergleicht man zwei Objekte. Schauen wir uns die Wirkung der Operatoren beim Vergleich von zwei Zahlen an.

```
> 3<4
[1] TRUE
> 3>4
[1] FALSE
```

Wir sehen, dass der Vergleich den Wert `TRUE` liefert, wenn die Bedingung wahr ist, ansonsten liefert er den Wert `FALSE`. Man kann auch Vektoren mit Skalaren vergleichen. Das Ergebnis ist in diesem Fall ein Vektor, dessen Komponenten `TRUE` sind, bei denen die Bedingung erfüllt ist. Ansonsten sind die Komponenten `FALSE`.
Wir betrachten die Variable `lp` von Seite 7.

```
> lp
[1] 22 30 16 25 27
> lp >= 25
[1] FALSE  TRUE FALSE  TRUE  TRUE
```

Man spricht auch von einem **logischen Vektor**. Wenn wir einen gleichlangen Vektor `x` mit einem logischen Vektor `l` durch `x[l]` indizieren, so werden aus `x` alle Komponenten ausgewählt, die in `l` den Wert `TRUE` annehmen. Der Aufruf

```
> lp[lp >= 25]
[1] 30 25 27
```

liefert also die Preise der Langspielplatten, die mindestens 25 US Dollar gekostet haben. Wenn wir wissen wollen, welche dies sind, so geben wir ein

```
> (1:length(lp))[lp >= 25]
[1] 2 4 5
```

Dieses Ergebnis hätten wir auch mit der Funktion `which` erhalten. `which`

```
> which(lp>=25)
[1] 2 4 5
```

Mit den Funktionen `any` und `all` kann man überprüfen, ob mindestens eine Komponente oder alle Komponenten eines Vektors eine Bedingung erfüllen. `any` `all`

```
> any(lp > 30)
[1] FALSE
> all(lp <= 30)
[1] TRUE
```

Zur ‹berprüfung von mindestens zwei Bedingungen dienen die Operatoren `&` und `|`. Der Operator `&` liefert genau dann das Ergebnis `TRUE`, wenn beide Bedingungen wahr sind, während dies beim Operator `|` der Fall ist, wenn mindestens eine Bedingung wahr ist.

```
> lp[lp < 30 & lp > 25]
[1] 27
> lp[lp < 30 | lp > 25]
[1] 22 30 16 25 27
```

Versuchen wir nun die Aufgabe von Seite 19 zu lösen. Wir wollen aus der Datentabelle `bidaten` auf Seite 17 das Alter der Studentinnen und das Alter der Studenten auswählen. Mit dem bisher gelernten erreichen wir das folgendermaßen:

```
> alter.w<-Alter[Geschlecht=="w"]
> alter.w
[1] 26 25 25 23 26 23 24 23 23
> alter.m<-Alter[Geschlecht=="m"]
> alter.m
[1] 29 24 23 27 25 24 24 29 28 24 24
```

Mit der Funktion `split` gelangen wir auch zum Ziel. `split`

```
> split(Alter,Geschlecht)
$m [1] 29 24 23 27 25 24 24 29 28 24 24

$w [1] 26 25 25 23 26 23 24 23 23
```

Die Funktion `split` erstellt eine Liste, deren erste Komponente das Alter der Studenten und deren zweite Komponente das Alter der Studentinnen enthält.

```
> alter.wm<-split(Alter,Geschlecht)
> alter.wm[[1]]
[1] 29 24 23 27 25 24 24 29 28 24 24
> alter.wm[[2]]
[1] 26 25 25 23 26 23 24 23 23
```

Auf die Komponenten dieser Liste können wir mit Hilfe der Funktionen `lapply` und `sapply` Funktionen anwenden. `lapply`
Beide Funktionen werden folgendermaßen aufgerufen: `lapply`

```
lapply(X,FUN)
sapply(X,FUN)
```

Dabei ist `X` eine Liste und `FUN` eine Funktion wie `min`, `max` oder `sort`.
Das Ergebnis von `lapply` ist eine Liste, deren i-te Komponente das Ergebnis enthält, das man erhält, wenn man die Funktion `FUN` auf die i-te Komponente der Liste `X` anwendet.
Das Ergebnis von `sapply` ist ein Vektor, falls das Ergebnis der Funktion `FUN` ein Skalar ist. Die i-te Komponente dieses Vektors enthält das Ergebnis, das man erhält, wenn man die Funktion `FUN` auf die i-te Komponente der Liste `X` anwendet.
Ist das Ergebnis der Funktion `FUN` ein Vektor mit einer festen Länge, so ist das Ergebnis von `sapply` ist eine Matrix, deren i-te Zeile das Ergebnis enthält, das man erhält, wenn man die Funktion `FUN` auf die i-te Komponente der Liste `X` anwendet.

Ansonsten sind die Ergebnisse der Funktionen `lapply` und `sapply` identisch. Wollen wir das Minimum des Alters der männlichen und der weiblichen Teilnehmer bestimmen, so geben wir ein

```
> lapply(split(Alter,Geschlecht),min)
$m [1] 23

$w [1] 23

> sapply(split(Alter,Geschlecht),min)
 m  w
23 23
```

Bei den geordneten Datensätzen des Alters der Frauen und Männer liefern `lapply` und `apply` identische Ergebnisse.

```
> lapply(split(Alter,Geschlecht),sort)
$m [1] 23 26 26 27 28 29 30 32 33 37 37 38

$w [1] 23 23 24 25 26 27 28 28 28 29 31 31 38

> sapply(split(Alter,Geschlecht),sort)
$m [1] 23 26 26 27 28 29 30 32 33 37 37 38

$w [1] 23 23 24 25 26 27 28 28 28 29 31 31 38
```

Eine weitere Möglichkeit zur Auswahl von Teilmengen einer Datentabelle bietet der Befehl `subset`. Der Aufruf

`subset`

```
subset(x,condition)
```

wählt aus der Datentabelle `x` die Zeilen aus, die die Bedingung `condition` erfüllen. Die Daten aller Studentinnen aus der Datei `bidaten` erhalten wir durch

```
> subset(bidaten,Geschlecht=="w")
   Geschlecht Alter Mutter Vater Geschwister
  2         w    26     53    54           2
  4         w    25     56    63           3
  5         w    25     49    53           0
  6         w    23     55    55           2
 11         w    26     61    65           1
```

```
15            w    23     52    52             1
17            w    24     59    63             0
18            w    23     52    55             1
20            w    23     54    55             1
```

und das Alter der Mütter der Studentinnen durch

```
> subset(bidaten,Geschlecht=="w",select=Mutter)
   Mutter
 2      53
 4      56
 5      49
 6      55
11      61
15      52
17      59
18      52
20      54
```

Man kann natürlich auch mehr als eine Bedingung angeben. Alle Studentinnen, die keine Geschwister haben, erhält man durch

```
> subset(bidaten,Geschlecht=="w" & Geschwister==0)
    Geschlecht Alter Mutter Vater Geschwister
 5           w    25     49    53           0
17           w    24     59    63           0
```

5 Grafiken in R

R bietet eine Reihe von Möglichkeiten, eine Grafik zu erstellen, von denen wir in diesem Skript eine Vielzahl kennen lernen werden. Wir wollen hier zunächst eine relativ einfache Grafik erstellen und betrachten folgende Funktion

$$F_n^*(x) = \begin{cases} 0 & \text{für } x < 20 \\ -0.8 + 0.04 \cdot x & \text{für } 20 \leq x \leq 25 \\ -2.2 + 0.096 \cdot x & \text{für } 25 < x \leq 30 \\ -0.28 + 0.032 \cdot x & \text{für } 30 < x \leq 35 \\ -0.28 + 0.032 \cdot x & \text{für } 35 < x \leq 40 \\ 1 & \text{für } x > 40 \end{cases} \tag{1}$$

Diese Funktion ist stückweise linear. Auf jedem Teilintervall müssen wir also eine Strecke zeichnen. Wir betrachten zunächst das Intervall $[20, 25]$. Hier lautet die Gleichung

$$F_n^*(x) = -0.8 + 0.04 \cdot x$$

Um eine Strecke zeichnen zu können, benötigen wir beide Endpunkte. Wir bestimmen $F_n^*(x)$ für $x = 20$ und $x = 25$. Es gilt

$$F_n^*(20) = -0.8 + 0.04 \cdot 20 = 0$$

und

$$F_n^*(25) = -0.8 + 0.04 \cdot 25 = 0.2$$

Wir zeichnen also eine Strecke durch die Punkte $(20, 0)$ und $(25, 0.2)$. Hierzu benutzen wir die Funktion `plot`. Diese benötigt als Argumente die gleich langen Vektoren `x` und `y`. Der Aufruf

`plot`

```
plot(x,y)
```

zeichnet die Punkte `(x[1],y[1])` und `(x[2],y[2])` in einem kartesischen Koordinatensystem. Wir geben also ein

```
> plot(c(20,25),c(0,0.2))
```

In Abbildung 1 auf der nächsten Seite links oben ist diese Grafik zu finden.

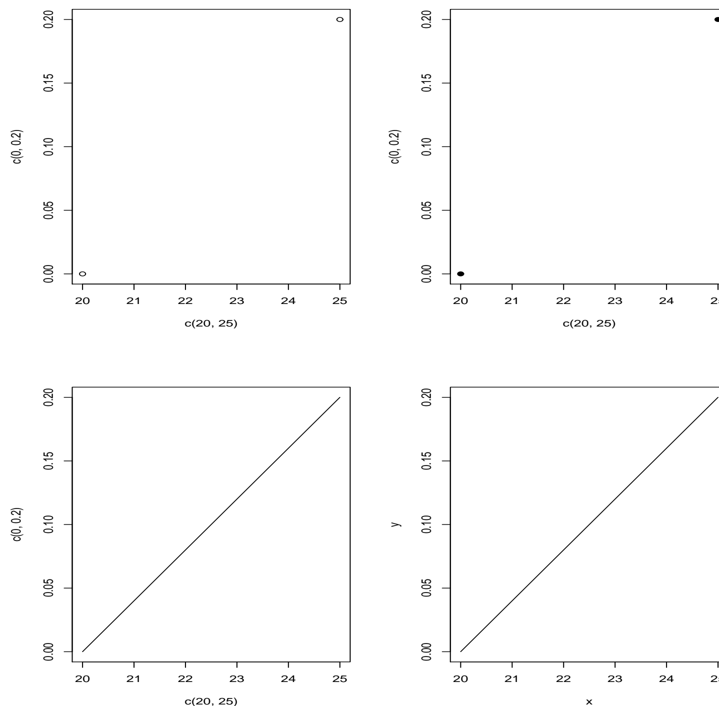

Abbildung 1: 4 Grafiken

Vier Bilder in einer Grafik erhält man durch

```
> par(mfrow=c(2,2))
```

Die Punkte in der Grafik in Abbildung 1 auf dieser Seite links oben sind offen. Sollen sie ausgemalt sein, so muss man das Argument `pch` auf den Wert 16 setzen. Dabei steht `pch` für plot character.

`pch`

`pch`

```
> plot(c(20,25),c(0,0.2),pch=16)
```

Das Ergebnis ist in der Grafik rechts oben in Abbildung 1 auf dieser Seite zu finden.
Die Größe der Achsenbeschriftung legt man mit dem Argument `cex.axis` fest. Dieses nimmt standardmäßig den Wert 1 an.

`cex.axis`

Sollen nicht die Punkte sondern die Strecke gezeichnet werden, so müssen wir das Argument `type` auf den Wert `"l"` setzen.

`type`

```
> plot(c(20,25),c(0,0.2),type="l")
```

Diese Grafik ist in Abbildung 1 auf der vorherigen Seite links unten zu finden. Der Standardwert von `type` ist `"p"`. Setzt man diesen auf den Wert `"o"`, so werden sowohl die Strecke als auch die Punkte gezeichnet.
Die Beschriftung der Abszisse und Ordinate ist unschön. Mit den Argumenten `xlab` und `ylab` legen wir die gewünschte Beschriftung als Zeichenketten fest.

`xlab`
`ylab`

```
> plot(c(20,25),c(0,0.2),type="l",xlab="x",ylab="y")
```

Diese Grafik ist in Abbildung 1 auf der vorherigen Seite rechts unten zu finden. Das gleiche Ergebnis können wir auch folgendermaßen erreichen:

```
> x<-c(20,25)
> y<-c(0,0.2)
> plot(x,y,type="l")
```

Die Größe der Buchstaben legt man mit dem Argument `cex.lab` fest. Dieses nimmt standardmäßig den Wert 1 an.

`cex.lab`

In den USA ist es üblich, dass die Beschriftung der Achsen parallel zu den Achsen gewählt wird. Dies ist auch Standard in R . Soll die Beschriftung der Ordinate orthogonal zu dieser Achse sein, so muss man eingeben

`las`

```
> par(las=1)
```

Diese Einstellung bleibt während der Sitzung mit R erhalten. Nach Eingabe dieses Befehls erhält man durch

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

die Grafik in Abbildung 2 auf der nächsten Seite links oben.

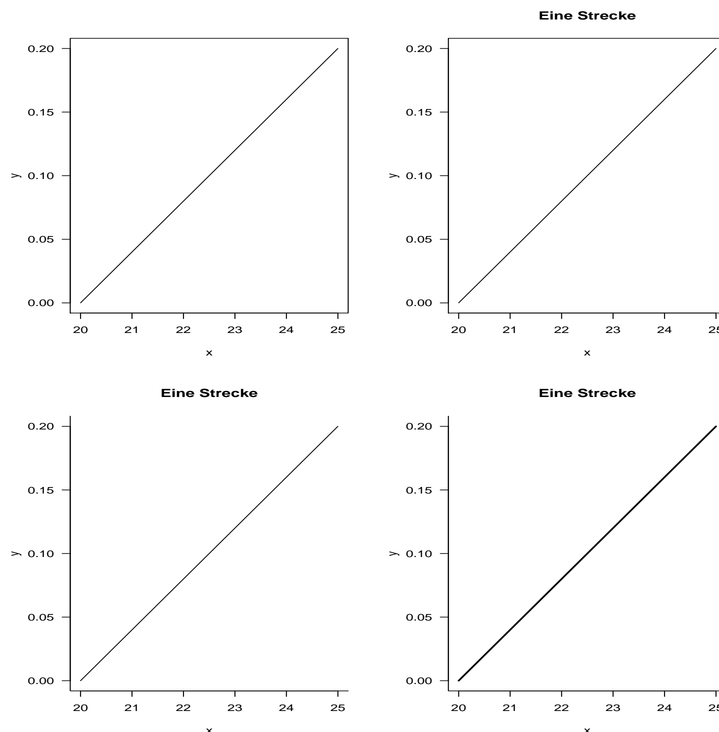

Abbildung 2: 4 Grafiken

Wir können über das Argument `main` eine ‹berschrift als Zeichenkette hinzufügen:

main

```
> plot(x,y,type="l",main="Eine Strecke")
```

In Abbildung 2 auf dieser Seite rechts oben ist diese Grafik zu finden.

Standardmäßig wird um die Grafik eine Box gezeichnet. Soll diese nur auf Höhe der der Abszisse und Ordinate erstellt werden, so muss man das Argument `bty` auf den Wert `"l"` setzen.

bty

```
> plot(x,y,type="l",main="Eine Strecke",bty="l")
```

Diese Grafik ist in Abbildung 2 auf dieser Seite links unten zu finden.

Standardmäßig nimmt `bty` den Wert `"o"` an.

Die Dicke der Linien legt man über das Argument `lwd` fest, das standardmäßig den Wert 1 annimmt. Doppelt so breite Linien erhält man durch:

lwd

```
> plot(x,y,type="l",main="Eine Strecke",bty="l",lwd=2)
```

In Abbildung 2 auf der vorherigen Seite ist diese Grafik rechts unten zu finden.
Nun wollen wir die Funktion aus Gleichung (1) auf Seite 24 im Intervall $[20, 40]$ zeichnen. Die ersten Koordinaten der Punkte sind

$$x_1 = 20 \quad x_2 = 25 \quad x_3 = 30 \quad x_4 = 35 \quad x_5 = 40$$

und die zugehörigen zweiten Koordinaten sind

$$y_1 = 0 \quad y_2 = 0.2 \quad y_3 = 0.68 \quad y_4 = 0.84 \quad y_5 = 1$$

‹bergibt man der Funktion `plot` die Vektoren `x` und `y`, die beide n Komponenten besitzen, so werden die Punkte `(x[1],y[1])` und `(x[2],y[2])`, `(x[2],y[2])` und `(x[3],y[3])`... `(x[n-],y[n-1])` und `(x[n],y[n])` durch Geraden verbunden.
Einen Vektor mit den Zahlen $20, 25, 30, 35, 40$ erhalten wir am einfachsten mit der Funktion `seq`. Diese wird folgendermaßen aufgerufen

`seq`

```
seq(from, to, by)
```

Es wird eine Zahlenfolge von `from` bis `to` im Abstand `by` erzeugt. Wir geben also ein

```
> x<-seq(20,40,5)
> x
[1] 20 25 30 35 40
```

Wir erstellen noch den Vektor `y`

```
> y<-c(0,0.2,0.68,0.84,1)
```

und zeichnen die Funktion

```
> plot(x,y,type="l",bty="l")
```

In Abbildung 3 auf der nächsten Seite ist diese Grafik links oben zu finden.

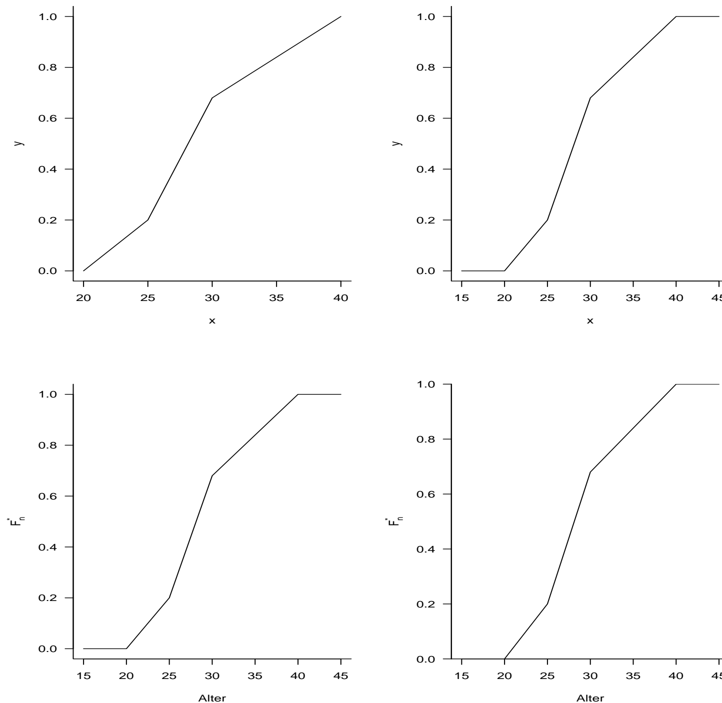

Abbildung 3: 4 Grafiken

Nun müssen wir noch den Bereich $x < 20$ und $y > 40$ berücksichtigen. Wir beginnen bei $x = 15$ und enden bei $x = 45$. Wir müssen also noch die Punkte $(15, 0)$ und $(45, 1)$ hinzufügen.

```
> x<-c(15,x,45)
> x
[1] 15 20 25 30 35 40 45
> y<-c(0,y,1)
> y
[1] 0.00 0.00 0.20 0.68 0.84 1.00 1.00
> plot(x,y,type="l",bty="l")
```

Diese Grafik ist in Abbildung 3 auf dieser Seite rechts oben zu finden.
Nun beschriften wir noch die Abszisse und die Ordinate mit den Argumenten `xlab` und `ylab`. An die Ordinate schreiben wir F_n^*. Dies ist eine Formel, die wir mit der Funktion `expression` erstellen. Ein tiefer gestelltes Zeichen gewinnt man, indem man es in eckige Klammern setzt und ein höher gestelltes

`expression`

durch "^". Beispiele für die Erstellung von Formeln erhält man durch den Aufruf von help(text).

```
> plot(x,y,type="l",bty="l",xlab="Alter",
      ylab=expression(F[n]^"*"))
```

In Abbildung 3 auf der vorherigen Seite ist diese Grafik links unten zu finden. Standardmäßig wird zwischen der Ordinate und dem Beginn der Kurve ein Zwischenraum gelassen. Diesen entfernen wir, indem wir den Parameter xaxs auf den Wert "i" setzen. Entsprechend gibt es den Parameter yaxs.

xaxs
yaxs

```
> plot(x,y,type="l",bty="l",xlab="Alter",
    ylab=expression(F[n]^"*"),yaxs="i")
```

In Abbildung 3 auf der vorherigen Seite ist diese Grafik rechts unten zu finden.
Wir wollen die Abbildung noch um die Gerade durch die Punkte $(20, 0)$ und $(40, 1)$ ergänzen. Hierzu benutzen wir die Funktion lines. Setzen wir das Argument lty auf den Wert 2, so wird eine gestrichelte Strecke gezeichnet.

lines
lty

```
> lines(c(20,40),c(0,1),lty=2,lwd=2)
```

Diese Gerade ist die Verteilungsfunktion der Gleichverteilung auf dem Intervall $[20, 40]$. Mit der Funktion legend fügen wir noch eine Legende hinzu.

legend

```
> legend(15,1,c("Daten","Gleichverteilung"),lty=1:2)
```

In Abbildung 4 auf der nächsten Seite ist diese Grafik links oben zu finden. Schauen wir uns noch einmal die Argumente xaxs und yaxs an. Die Grafik in Abbildung 1 links oben auf Seite 25 zeigt, warum eine Grafik in R nicht bei den Minima der Beobachtungen beginnt und bei den Maxima endet, wenn man eine Punktewolke zeichnet. Diese Punkte werden dann nämlich an den Rand gedrängt. Dies ist in der Grafik rechts oben in Abbildung 4 auf der nächsten Seite der Fall, in der wir xaxs und yaxs auf den Wert "i" gesetzt haben:

```
> plot(c(20,25),c(0,0.2),xlab="x",ylab="y",xaxs="i",yaxs="i")
```

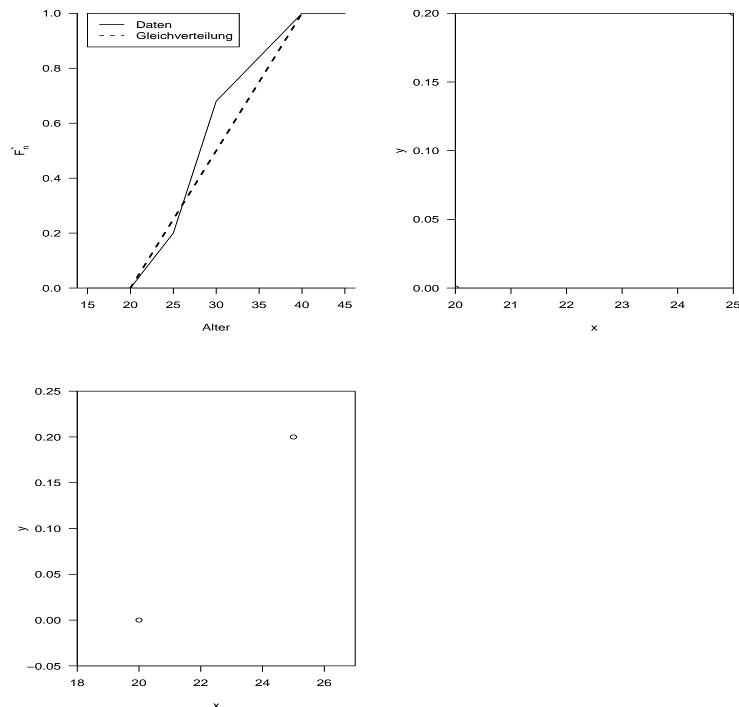

Abbildung 4: 3 Grafiken

Wir können den Bereich der Grafik durch die Argumente `xlim` und `ylim` festlegen. Die Grafik links unten in Abbildung 4 auf dieser Seite erhalten wir durch

xlim
ylim

```
> plot(c(20,25),c(0,0.2),xlab="x",ylab="y",
      xaxs="i",yaxs="i",xlim=c(18,27),ylim=c(-0.05,0.25))
```

Will man eine Funktion zeichnen, so kann man durch die Argumente `xaxs` und `yaxs` die Grafik verschönern. Schauen wir uns dies an einem Beispiel an. Wir wollen die Dichtefunktion der Standardnormalverteilung

$$\phi(x) = \frac{1}{\sqrt{2 \cdot \pi}} e^{-0.5 \cdot x^2}$$

im Intervall $[-4, 4]$ zeichnen. Die Zahl π erhält man in R durch

```
> pi
[1] 3.141593
```

und die Exponentialfunktion mit der `exp`

```
> exp(1)
[1] 2.718282
```

Die Dichtefunktion der Standardnormalverteilung in $x = -2, -1, 0, 1, 2$ erhalten wir also durch

```
> 1/sqrt(2*pi)*exp(-0.5*(-2:2)^2)
[1] 0.05399097 0.24197072 0.39894228 0.24197072 0.05399097
```

Mit der Funktion `curve` können wir die Dichtefunktion der Standardnormalverteilung folgendermaßen zeichnen

`curve`

```
> curve(1/sqrt(2*pi)*exp(-0.5*x^2),from=-4,to=4)
```

Die obere Grafik in Abbildung 5 auf dieser Seite zeigt das Bild.

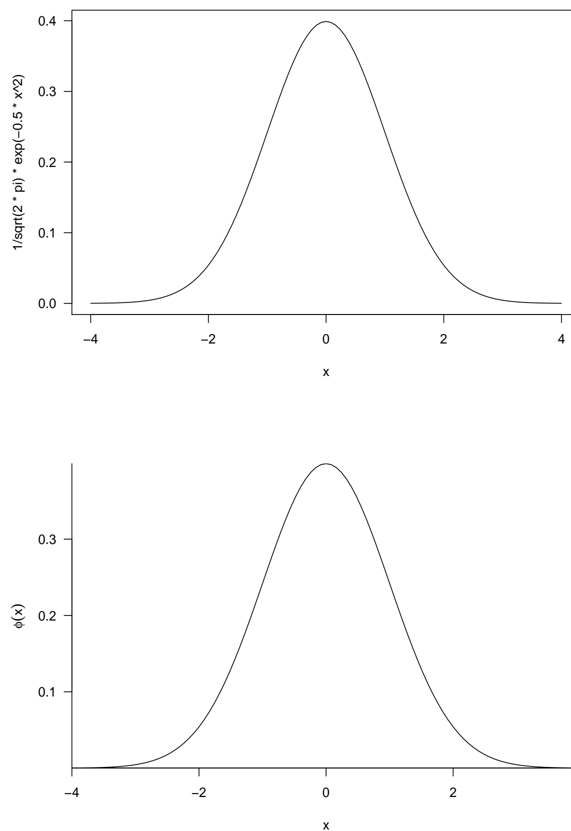

Abbildung 5: Dichtefunktion der Standardnormalverteilungf

Hier ist es sinnvoll, xaxs und yaxs auf den Wert "i" zu setzen. Außerdem beschriften wir noch die Ordinate und ändern die Box um die Grafik.

```
> curve(1/sqrt(2*pi)*exp(-0.5*x^2),from=-4,to=4,
       xaxs="i",yaxs="i",bty="l",ylab=expression(phi(x)))
```

Das Ergebnis ist in Abbildung 5 auf der vorherigen Seite unten zu finden.
Die Dichtefunktion der Standardnormalverteilung ist in R in der Funktion dnorm implementiert. Wir können also auch

dnorm

```
> curve(dnorm,from=-4,to=4,xaxs="i",yaxs="i",
       bty="l",ylab=expression(phi(x)))
```

eingeben und erhalten das gleiche Ergebnis.

6 Wie schreibt man eine Funktion?

Im letzten Kapitel haben wir gesehen, mit welcher Befehlsfolge wir die Funktion aus Gleichung (1) auf Seite 24 grafisch darstellen können. Es handelt sich um die approximierende empirische Verteilungsfunktion. Um diese zeichnen zu können, benötigen wir die Klassengrenzen $x_0^*, x_1^*, \ldots, x_k^*$ und die kumulierten relativen Häufigkeiten

$$F_n(x_0^*) = 0 \quad (2)$$

$$F_n(x_i^*) = \sum_{j=1}^{i} h_j \quad (3)$$

Durch Eingabe der in Kapitel 5 beschriebenen Befehlsfolge können wir für jeden Datensatz eine Grafik der approximierenden empirischen Verteilungsfunktion erstellen. Dies ist aber sehr mühselig. Wir können uns die Eingabe der Befehlsfolge ersparen, wenn wir eine Funktion schreiben, die diese Befehlsfolge ausführt. Funktionen bieten die Möglichkeit, eine Befehlsfolge unter einem Namen abzuspeichern und durch Aufruf des Namens der Funktion die für unterschiedliche Werte der Argumente auszuführen.
Eine Funktion wird in R durch folgende Befehlsfolge deklariert:

function

```
fname<-function(Argumente} {
 Koerper der Funktion
 return(Ergebnis)
}
```

Eine Funktion benötigt einen Namen und Argumente. Im Körper der Funktion steht die Befehlsfolge. Der Befehl `return` bewirkt, dass die Funktion verlassen wird und das Argument von `return` als Ergebnis der Funktion zurückgegeben wird. `return`

Wir nennen die Funktion `plot.ecdfapprox`. Es liegt nahe, die Koordinaten der Punkte, die durch Geraden verbunden werden sollen, als Argumente der Funktion `plot.ecdfapprox` zu wählen. Das sind die Klassengrenzen $x_0^*, x_1^*, \ldots, x_k^*$ und die kumulierten relativen Häufigkeiten

$$h_1 \quad h_1 + h_2 \quad \ldots \quad h_1 + h_2 + \ldots + h_k$$

Dem Anwender sind aber in der Regel eher die relativen Häufigkeiten

$$h_1, h_2, \ldots, h_k$$

der Klassen bekannt, aus denen er die kumulierten relativen Häufigkeiten bestimmt. Aus diesen bestimmt er dann die kumulierten relativen Häufigkeiten Diese Aufgabe soll die Funktion `plot.ecdfapprox` übernehmen. Sie erhält also als Argumente die Klassengrenzen und die relativen Häufigkeiten der Klassen. Wir nennen diese Argumente `grenzen` und `haeuf`. Wir schauen uns nun die Befehle an und beginnen mit dem Bereich $[x_0^*, x_k^*]$. Wir müssen zuerst die kumulierten relativen Häufigkeiten mit der Funktion `cumsum` bestimmen. `cumsum`

```
chaeuf<-cumsum(haeuf)
```

An der Stelle `grenzen[1]` ist die approximierende empirische Verteilungsfunktion gleich 0. Wir ergänzen den Vektor `chaeuf` vorne um den Wert 0.

```
chaeuf<-c(0,chaeuf)
```

Nun müssen wir noch den Bereich vor x_0^* und hinter x_k^* berücksichtigen. Die approximierende empirische Verteilungsfunktion nimmt vor x_0^* den Wert 0 und hinter x_k^* den Wert 1 an.

```
chaeuf<-c(0,chaeuf,1)
```

Bei welchem Werten auf der Abszisse soll die Zeichnung beginnen und enden? Im Beispiel hatten wir die Werte 15 und 45 gewählt. Wir wollen dem Benutzer aber nicht zumuten, dass er diese Werte vor dem Aufruf der Funktion `plot.ecdfapprox` festlegt und der Funktion als Argument übergibt, sondern bestimmen sie innerhalb der Funktion. Wir berechnen die Breite des Intervalls

```
b<-grenzen[length(grenzen)]-grenzen[1]
```

und setzen

```
ug<-grenzen[1]-0.25*b
```

und

```
og<-grenzen[length(grenzen)]+0.25*b
```

Wir erweitern den Vektor `grenzen` um diese Größen

```
grenzen<-c(ug,grenzen,og)
```

und zeichnen mit `bty="l"` die Box nur auf Höhe der Abszisse und Ordinate. Außerdem starten wir die Kurve mit `xaxs="i"` bei der Ordinate und beschriften die Ordinate mit F_n^*. Wir rufen die Funktion `plot` also folgendermaßen auf:

```
plot(grenzen,chaeuf,type="l",bty="l",xaxs="i",
   ylab=expression(F^"*"[n]))
```

Für die Beschriftung der Abszisse wählen wir den Namen des Merkmals. Diesen geben wir der Funktion als Argument.
Schauen wir uns das alles noch für ein Beispiel an.

```
> grenzen
[1] 20 25 30 35 40
> haeuf
[1] 0.20 0.48 0.16 0.16
> chaeuf<-cumsum(haeuf)
> chaeuf
[1] 0.20 0.68 0.84 1.00
> chaeuf<-c(0,chaeuf)
> chaeuf
[1] 0.00 0.20 0.68 0.84 1.00
> chaeuf<-c(0,chaeuf,1)
> chaeuf
[1] 0.00 0.00 0.20 0.68 0.84 1.00 1.00
> b<-grenzen[length(grenzen)]-grenzen[1]
> b
[1] 20
> ug<-grenzen[1]-0.25*b
> ug
[1] 15
```

```
> og<-grenzen[length(grenzen)]+0.25*b
> og
[1] 45
> grenzen<-c(ug,grenzen,og)
> grenzen
[1] 15 20 25 30 35 40 45
> plot(grenzen,chaeuf,type="l",bty="l",xaxs="i",
     ylab=expression(F^"*"[n]))
```

Jetzt können wir die Funktion erstellen.

```
 plot.ecdfapprox<-function(grenzen,haeuf,xname) {
 chaeuf<-c(0,0,cumsum(haeuf),1)
 b<-grenzen[length(grenzen)]-grenzen[1]
 ug<-grenzen[1]-0.25*b
 og<-grenzen[length(grenzen)]+0.25*b
 grenzen<-c(ug,grenzen,og)
 plot(grenzen,chaeuf,type="l",bty="l",xaxs="i",xlab=xname,
     ylab=expression(F^"*"[n]))
}
```

Jede Funktion sollte einen Kommentar enthalten, in dem die Argumente und der Output beschrieben werden. Der Kommentar steht hinter dem Zeichen #. Wir ergänzen die Funktion um Kommentare.

```
 plot.ecdfapprox<-function(grenzen,haeuf,xname)
 { # grafische Darstellung der
   # approximierenden empirischen Verteilungsfunktion
   # Grenzen: Vektor mit den Klassengrenzen
   # haeuf: Vektor mit relativen Häufigkeiten der Klassen
   # xname: Name des Merkmals
   chaeuf<-c(0,0,cumsum(haeuf),1)
   b<-grenzen[length(grenzen)]-grenzen[1]
   ug<-grenzen[1]-0.25*b
   og<-grenzen[length(grenzen)]+0.25*b
   grenzen<-c(ug,grenzen,og)
   plot(grenzen,chaeuf,type="l",bty="l",xaxs="i",xlab=xname,
     ylab=expression(F^"*"[n]))
}
```

In der Funktion `plot.ecdfapprox` wird nicht überprüft, ob die Argumente der Funktion richtig gewählt wurden. So muss der Vektor `grenzen` eine Komponente mehr als der Vektor `haeuf` enthalten. Im Kapitel 4 auf Seite 19 haben

wir gelernt, wie man in `R` Bedingungen überprüft. In unserem Fall geben wir eine Fehlermeldung aus, wenn `length(grenzen)` ungleich `1+length(haeuf)` ist; ansonsten erstellen wir die Grafik. Man spricht auch von einer bedingten Anweisung.
In R setzt man bedingte Anweisungen mit dem Konstrukt

```
if(Bedingung){Befehlsfolge 1} else {Befehlsfolge 2}
```

um.
In unserem Fall enthält die Befehlsfolge 1 die Fehlermeldung, die am Bildschirm erscheinen soll. Wir ergänzen die Funktion um die Befehlsfolge

```
 if(length(grenzen)!=(1+length(haeuf)))
 {return("Fehler: Der Vektor grenzen muss um eine
  Komponente laenger sein als der Vektor haeuf"}
 else
```

Somit sieht die Funktion folgendermaßen aus.

```
plot.ecdfapprox<-function(grenzen,haeuf,xname) { # grafische
Darstellung der
 # approximierenden empirischen Verteilungsfunktion
 # grenzen: Vektor mit den Klassengrenzen
 # haeuf: Vektor mit relativen Häufigkeiten der Klassen
 # xname: Name des Merkmals
if(length(grenzen)!=(1+length(haeuf))) {return(cat("Fehler: Der
Vektor grenzen muss um eine Komponente
            laenger sein als der Vektor haeuf"))}
else { chaeuf<-c(0,0,cumsum(haeuf),1)
  b<-grenzen[length(grenzen)]-grenzen[1]
  ug<-grenzen[1]-0.25*b
  og<-grenzen[length(grenzen)]+0.25*b
  grenzen<-c(ug,grenzen,og)
  plot(grenzen,chaeuf,type="l",bty="l",xaxs="i",xlab=xname,
       ylab=expression(F^"*"[n]))
} }
```

Die Eingabe einer Funktionsdefinition wird in R durch die Funktion `fix` unterstützt. Nach dem Aufruf

`fix`

```
fix(Name der Funktion)
```

landet man in einem Editor, der die Eingabe erleichtert.

7 Pakete

R ist ein offenes Programm, sodass es durch Funktionen, die von Benutzern erstellt wurden, erweitert werden kann. Diese Funktionen sind in Paketen (packages) enthalten. Um eine Funktion aus einem Paket benutzen zu können, muss man das Paket installieren und laden. Man installiert ein Paket, indem man auf den Schalter
Pakete
und danach auf den Schalter
Installiere Paket(e)
klickt. Es öffnet sich ein Fenster mit einer Liste, in der man auf den Namen des Paketes klickt. Hierauf wird das Paket installiert. Dazu muss natürlich eine Verbindung zum Internet vorhanden sein. Alternativ kann ein Paket auch über den Befehlsmodus heruntergeladen und installiert werden. Der Befehl

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

installiert in diesem Fall das Paket MASS. Eine Liste aller inzwischen verfügbaren Pakete für R erhält man unter

`http://cran.r-project.org/web/packages/`

Nachdem man

```
> library(Name des Paketes)
```

eingegeben hat, kann man die Funktionen des Paketes verwenden. Man muss ein Paket nur einmal installieren, muss es aber während jeder Sitzung einmal laden, wenn man es verwenden will.