Datenanalyse: Eye-tracking 2020/2021

Katja Suckow

2021-01-21

Contents

1 Willkommen **5**

1.1 Was braucht man dafür? 5

1.2 Einleitung: Datentypen in der Linguistik 9

1.3 Population und Stichprobe 9

1.4 Deskriptive Statistik und Inferenzstatistik 10

1.5 R als Taschenrechner: Beispiele 10

1.6 Einige vordefinierte Beispielfunktionen 11

1.7 Vergleichsoperatoren 13

1.8 Logische Vergleichsoperatoren 15

1.9 Variablen in R 15

1.10 Hilfefunktionen 16

1.11 Pakete Laden 16

2 Datenbeschreibungen in R **17**

2.1 R-Vektoren 17

2.2 R-Datentypen 18

2.3 R-Data frames 20

2.4 R-Matritzen 21

2.5 R - nützliche Funktionen 22

3 Beschreibung Funktionen: Datenauswertung **25**

3.1 Grafische Darstellung der Daten 30

Chapter 1

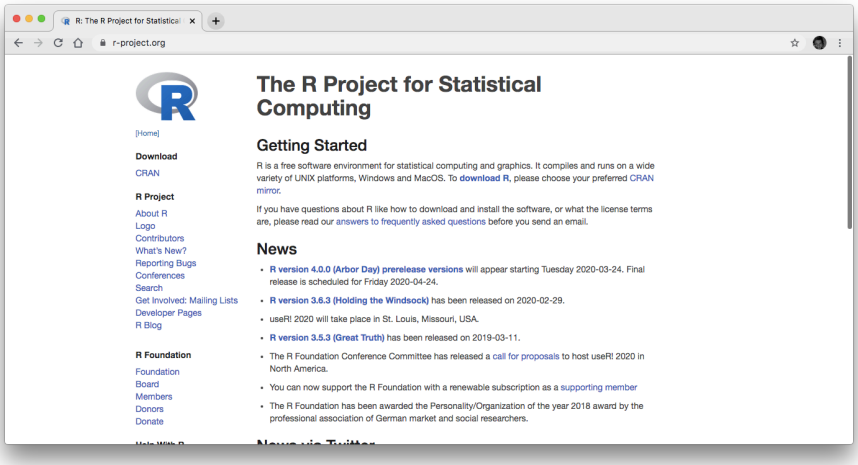
Willkommen

Da wir uns leider in diesem Jahr erstmal nicht persönlich treffen können, habe ich hier eine kleine Übersicht zu den verwendeten R-Befehlen erstellt.

1.1 Was braucht man dafür?

Dafür muss man auf seinem Rechner **R** und **Rstudio** installieren (beide sollten auf Windows, Mac und Linux laufen).

1. **R** frei verfügbare Software https://www.r-project.org/



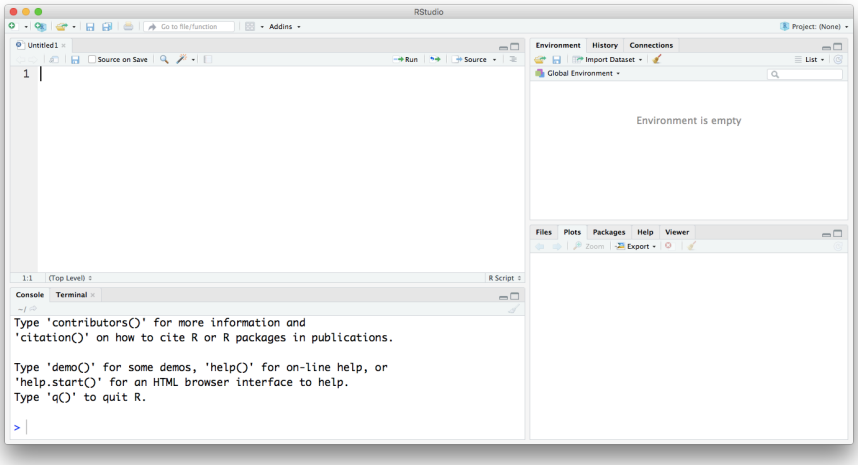
R ist eine freie Programmiersprache zur statistischen Datenanalyse und

Erstellung von Grafiken. Sie kann auch als Skriptsprache benutzt werden, um einfache Skripte und Programme zu schreiben. **R** ist vor allem in der Wissenschaft weit verbreitet und löst hier zunehmend SPSS ab. **R** bietet viele Methoden und Pakete zu statistischen Auswertung und Datendarstellung, die ständig in open-source weiterentwickelt werden, es steht kostenlos zur Verfügung und kann auf jeder Plattform (Windows, Mac, Linux) laufen. R besteht aus 3 Hauptfenstern: (1) **Konsole**, um direkt Befehle einzugeben, (2) **Editor**, um eine Abfolge an Befehlen zu speichern und auszuführen, (3) **Grafikfenster**

2. **RStudio** https://www.rstudio.com/products/rstudio/download Es gibt verschiedene Möglichkeiten mit R zu arbeiten. Wir werden die grafische Oberfläche, die **RStudio** bietet, nutzen. **RStudio** bietet eine gut handhabe Oberfläche, viel Unterstützung und viele integrierte Apps (Shiny, Markdown, Bookdown …), die auf R zurückgreifen. Unterschiedliche Editoren zur Bearbeitung von R Dateien sind:

- RStudio (für die gemeinsame Datenauswertung im Praktikum empfohlen)
- Notepad++ (Windows)
- Textwrangler
- . . .

Figure 1.1: RStudio Logo



So sieht RStudio ohne Inhalt oder Daten aus.

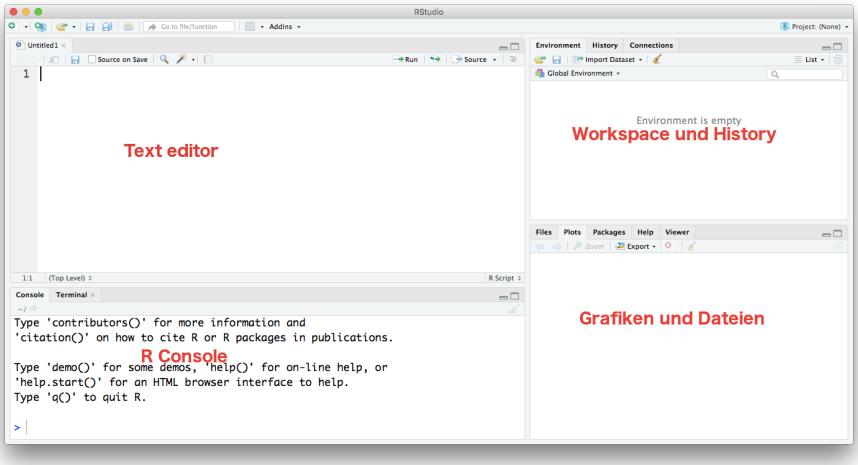


Figure 1.2: Die 4 Hauptfenster von RStudio

In der Standarddarstellung befindet sich oben links der **Code Editor**, oben rechts können sie **Workspace und History** ansehen, unten links befindet sich die **R Console**, in der Sie den Code direkt eingeben können und unten rechts lassen sich unter anderem **Plots und Files** anzeigen. Schauen Sie sich die verschiedenen Möglichkeiten an, die die verschiedenen Tabs in den Ecken bieten.

Skript

Das Skript enthält eine Sammlung von Befehlen. Diese können auch direkt über Konsole ausgeführt werden. Es empfiehlt sich jedoch sehr, diese im **Code**

Editor zu speichern, um später noch einmal darauf zugreifen zu können.

Workspace

Der Workspace enthält Sammlung von Objekten, die in einer Session erstellt wurden. Diese kann explizit gespeichert werden, um die einzelnen Objekte noch einmal neu zu laden.

So sieht Rstudio mit ein bischen mehr Inhalt aus.

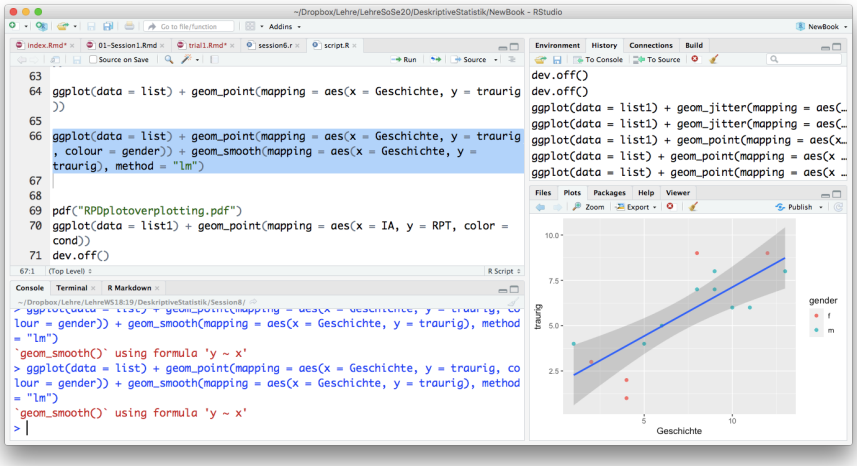


Figure 1.3: RStudio mit Datensatz und grafischer Darstellung eines Datensatzes

- **oben links** steht der Code, der gespeichert und auch aus dem Editor ausgeführt werden kann. (Mac: Markierung des auszuführenden Codes und dann **CMD + Enter**)
- **unten links** man kannn den auszuführenden Code auch direkt unten in das Konsolenfenster eingeben. Dann wird dieser allerdings nicht gespeichert. Es wird empfohlen, Code der erstmal nur ausprobiert wird, direkt in das Konsolenfenster einzutragen. Wenn der Code das korrekte Ergebnis liefert, sollte er im Editor in einem Skript gespeichert werden.
- **oben rechts** lassen sich History der eingegebenen Befehle und auch die Inhalte des Workspace anzeigen
- **unten rechts** kann man sich die geladenen Pakete für **R**, sowie die verschiedenen erstellten Grafiken anzeigen lassen

Die Anordnung der Fenster lassen sich über **RStudio** > **Preferences** und dann **Pane Layout** ändern. Über **Preferences** können Sie sich generell anzeigen lassen, welche Änderungen Sie u.a. bei der Anzeige vornehmen können bzw.~wollen.

3. **Shiny App**

Das ist eine App, die in RStudio integriert ist und die deskriptive Daten mit **R** bildlich und interaktiv veranschaulichen kann und auch veröffentlicht. Wenn man sich einmal ein bisschen mit R auseinandergesetzt hat, kann man sich mit Shiny weiter "austoben".

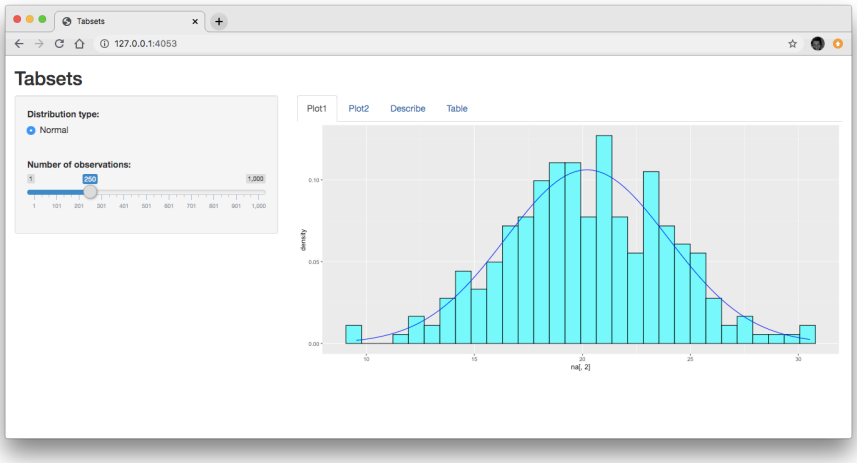


Figure 1.4: Beispiel des Screenshots einer Shiny App, die interaktiv die Normalverteilung einer übergebenen Anzahl beobachteter Werte anzeigt.

1.2 Einleitung: Datentypen in der Linguistik

Qualitative Daten Nicht-numerische, oft verschriftlichte oder in audiovisueller Form vorliegende Daten. Verwendung häufig explorativ und hypothesengenerierend. (e.g., Interviews)

Quantitative Daten Numerische (bzw. in numerischer Form überführte) Daten, die mit dem Ziel der Überprüfung von Hypothesen und Theorien erhoben werden

1.3 Population und Stichprobe

In der psycholinguistischen Forschung werden Daten in einer kontrollierten Umgebung (Experiment) erhoben, die Aussagen für die Gesamtpopulation zu einer bestimmten linguistischen Fragestellung machen zu können.

Dabei wird eine kleine Gruppe der Population (Stichprobe) in einem Experiment getestet. Bestimmte Daten dieser Stichprobe werden erhoben und anschliessend die Ergebnisse dieser Daten auf die Gesamtpopulation projiziert. Die erhobenen Daten der Stichprobe müssen erst in deskriptive statistische Kennwerte umgewandelt werden und anschliessend mittels statistischer Verfahren Hypothesen geprüft werden, um Aussagen über die Gesamtpopulation machen zu können.

1.4 Deskriptive Statistik und Inferenzstatistik

Deskriptive Statistik übersichtliche Darstellung der erhobenen Daten einer Stichprobe in Form statistischer Kennwerte (Mittelwerte, Streuung, Verteilung, Grafiken)

Inferenzstatistik Statistische Verfahren, die es ermöglichen Rückschlüsse aus den Daten auf die Gesamtpopulation zu ziehen, um Schätzungen für Populationswerte zu berechnen oder experimentelle Hypothesen zu testen.

1.5 R als Taschenrechner: Beispiele

Um sich erstmal mit **R** vertraut zu machen, kann man R erstmal als Taschenrechner einsetzen. Dabei ist die Anwendung recht assoziativ und kann wie folgt eingesetzt werden.

Probieren Sie diese Beispiele erstmal selbst zuerst in Ihrem Terminal und dann aus dem Editor heraus aus!

Addition +

```
5 + 7
```

```
## [1] 12
```

Subtraktion -

```
325 - 18
```

```
## [1] 307
```

Multiplikation *

```
43 * 21
```

```
## [1] 903
```

Division /

```
442 / 13
```

```
## [1] 34
```

Exponent ^

```
32^2
```

```
## [1] 1024
```

1.6 Einige vordefinierte Beispielfunktionen

Funktionen können auch sehr einfach in **R** selbst geschrieben werden. Dazu kommenw wir später auch noch. Man muss allerdings nicht alles neu erfinden, besonders weil **R** auch viele Funktionen vordefiniert mitbringt.

Die Syntax ist relativ simpel. Die Funktion hat einen eigenen Namen und die übergebenen Argumente, mit denen die Funktion Berechnungen durchführt werden in der Klammer nach dem Funktionsnamen übergeben.

Unten sind ein paar Beispiele für frequente leicht nachvollziehbare Funktionen aufgelistet.

Summe sum() - Bildung der Summe, der übergebenen Argumente:

```
sum(3,45,12,34)
```

```
## [1] 94
```

Wurzel sqrt() - Bildung der Wurzel von einem übergebenen Argument:

```
sqrt(1024)
```

```
## [1] 32
```

Minimum min() - Ermittlung der kleinsten Zahl aus einer Liste von übergebenen Werten:

```
min(43,24,11,23,76,14,56,99,12)
```

```
## [1] 11
```

Maximum max() - Ermittlung der höchsten Zahl aus einer Liste von übergebenen Werten:

```
max(43,24,11,23,76,14,56,99,12)
```

```
## [1] 99
```

Absoluter Wert (Betrag) abs() - Ermittlung des absoluten Betrags aus einem Wert:

```
abs(-25)
```

```
## [1] 25
```

Aneinanderreihung concatenate c() verbindet in Klammern stehende Zahlen zu einer Liste

```
c(2,6,7)
```

List ls()

- listet alle Elemente im aktuellen Workspace auf

```
ls()
```

Gib Pfad des aktuellen working directory zurück getwd()

- Gibt Verzeichnis des aktuellen Workspace aus

```
getwd()
```

Setze working directory auf folgenden Pfad setwd()

- Setzt den aktuellen Workspace in angegebenes Verzeichnis

```
setwd(<Pfad>)
```

Lies Datensatz ein read.table()

- liest einen Datensatz in aktuellen Workspace ein

```
read.table(<file>, header = T)
```

write.table()

- exportiert eine Variable aus aktuellem working directory in die angegebene Dateinamen

```
write.table(<Variablenname>,file = "")
```

1.7 Vergleichsoperatoren

Hier sind die Beispiele logischer Vergleichsoperatoren. Deren Interpretation gibt entweder **TRUE** oder **FALSE** zurück.

gleich ==

- Vergleich ob die Werte beider Ausdrücke identisch sind
- ja: **TRUE**
- nein: **FALSE**

```
4 == 6
```

```
## [1] FALSE
```

ungleich: **!=**

- Vergleich ob die Werte beider Ausdrücke ungleich sind
- ja sie sind ungleich: **TRUE**
- nein sie sind gleich: **FALSE**

```
4 != 6
```

```
## [1] TRUE
```

größer: >

- ist der Wert links größer als der Wert rechts?

```
4 > 6
```

```
## [1] FALSE
```

kleiner: $<$

- ist der Wert links kleiner als der Wert rechts?

```
4 < 6
```

```
## [1] TRUE
```

größer gleich: $>=$

- ist der Wert links größer oder gleich gleich der Wert rechts?

```
4 >= 6
```

```
## [1] FALSE
```

```
4 >= 4
```

```
## [1] TRUE
```

kleiner gleich: $<=$

- ist der Wert links kleiner als der Wert rechts?

```
4 <= 6
```

```
## [1] TRUE
```

```
4 <= 4
```

```
## [1] TRUE
```

1.8 Logische Vergleichsoperatoren

Konjunktion, logisches "Und": **&&** oder **&**

- beim logischen **Und** werden die Werte der beiden Ausdrücke verglichen - nur wenn beide **TRUE** ergeben, wird der Gesamtausdruck **TRUE** ergeben
- im unteren Beispiele sind beide Ausdrück **TRUE**

```
4 <= 4 & 4 >= 4
```

```
## [1] TRUE
```

Disjunktion, logisches "oder": || oder |

- beim logischen **Oder** reicht es, wenn ein zu vergleichender Ausdruck **TRUE** ist, um den gesamten Ausdruck **TRUE** zu machen.

```
4 < 6 || 5 > 8
```

```
## [1] TRUE
```

Negation: **!**

- bei der Negation wird der Wahrheitsgehalt eines Ausdruckes umgekehrt.

```
!(4 < 6 || 5 > 8)
```

```
## [1] FALSE
```

1.9 Variablen in R

Verschiedene Werte oder Ergebnisse einer Berechnung können leicht in Variablen (Platzhalter) gespeichert werden. Diese Variablen müssen in **R** mit einem Buchstaben beginnen. Die Zuweisung von Werten in eine Variable erfolgt entweder mit einem Pfeil **<-** oder mit einem simplen = (beides ohne Unterschied möglich):

```
a <- 15
b = 23
ab = c(a,b)
ab
```

```
## [1] 15 23
```

1.10 Hilfefunktionen

Mit der Hilfefunktion **?** lässt sich zu jeder R-Funktion eine hilfreiche Beschreibung anzeigen. Schreiben Sie einfach das **?** direkt vor die Funktion zu der Sie mehr erfahren wollen und **R** öffnet ein Hilfsfenster. Probieren Sie es einmal aus und schreiben:

?sum

Mit **?sum** können Sie sich die Informationen zum **sum** Befehl anzeigen lassen. Meist werden am Ende der Beschreibung auch Beispiele zur Benutzung angezeigt.

Alternativ können Sie auch den **help()** benutzen:

```
?sum
help(sum)
```

1.11 Pakete Laden

Pakete sind eine Sammlung verschiedener Funktionen zu einem Thema.

```
# installiert das Paket
install.packages("tidyverse")

# lädt das Paket
library(tidyverse)
```

Chapter 2

Datenbeschreibungen in R

Kurze Zusammenfassung einiger zentraler Funktionen für das Praktikum.

```
# Daten einlesen
getwd()
# Der Pfad zu Eurem Workspace ist auf jedem Rechner anders
# Deshalb läßt sich das nicht vorgeben und Ihr könnt
# diesen nicht einfach von jemand kopieren.
setwd("<Setzt bitte hier den Pfad zum Workspace auf Eurem Rechner ein>")
# Der Datensatz sollte gugVP11.dat
# sollte im Pfad Eures Workspace gespeichert sein
list = read.table("gugVP11.txt",header = T)
```

```
# Inhalt des Workspace
# welche Variablen wurden eingelesen oder selbst erstellt
ls()

# listet die ersten 6 Zeilen auf
head(list)

# Berechne die Summe aller Lesezeiten
sum(list$time)
```

2.1 R-Vektoren

Vektoren sind eine Datenstruktur, die Elemente desselben Datentyps in Form einer Liste enhält. Diese Datentypen können *logisch*, *integer*, *double*, *character* oder *komplex* sein.

```
x = c(1:30)
#typeof() gibt den Datentyp einer Variable wieder
typeof(x)
```

```
## [1] "integer"
```

```
#str() listet den Datentyp und die
#verschiedenen Ausprägungen einer Variable auf
str(x)
```

```
##  int [1:30] 1 2 3 4 5 6 7 8 9 10 ...
```

```
#length() gibt die Länge einer Liste wieder
length(x)
```

```
## [1] 30
```

2.2 R-Datentypen

R weist die Datentypen automatisch zu - Variablentyp muss nicht explizit festgelegt werden. Manchmal kann man den Variablentyp ändern, wenn bestimmte Funktionen durchgeführt werden sollen.

integer sind ganzzahlige Werte im endlichen Bereich \ - standardisierter Bereich ist normalerweise von -32768 bis +32768 - wenn der Wert größer als 32768 ist, dann gibt es eine Art Überlauf

```
x = c(1:30)
typeof(x)
```

```
## [1] "integer"
```

```
str(x)
```

```
##  int [1:30] 1 2 3 4 5 6 7 8 9 10 ...
```

double bezeichnet Fliesskommazahlen, wird in R auch manchmal als **numeric** wiedergegeben.

```
x = 3.232
str(x)
```

```
##  num 3.23
```

```
typeof(x)
```

```
## [1] "double"
```

Character enthält Buchstabenfolgen. Diese müssen in ” ” stehen, um sie von den Variablen zu unterscheiden.

```
x = "Langeweile"
x
```

```
## [1] "Langeweile"
```

```
str(x)
```

```
##  chr "Langeweile"
```

```
typeof(x)
```

```
## [1] "character"
```

Factor sind Variablen, die bestimmte endliche Werte annehmen können. Sie bezeichnen kategorische Werte. **Factor** können als Zahlen oder Buchstaben dargestellt werden.

```
x = as.factor(c("apfel","birne","erdbeere","erdbeere"))
y = as.factor(c(1,2,2,1,1,1,8))
str(x)
```

```
##  Factor w/ 3 levels "apfel","birne",..: 1 2 3 3
```

```
str(y)
```

```
##  Factor w/ 3 levels "1","2","8": 1 2 2 1 1 1 3
```

logical Evaluation einer logischen Frage: Darstellung als **TRUE** oder **FALSE**; kann nur zwei Werte annehmen.

- Siehe hierzu die Inhalte von letzter Woche und die Folien und Übungen im

complex Daten, die nicht in traditioneller Weise dargestellt werden.

Die **str()** Funktion stellt die Datentypen der Werte in einer Tabelle dar:

```
# listet Datentypen aller Spalten auf
list = read.table("gugVP11.txt",header = T)
str(list)
```

str() listet die Tabelle nach den Spalten auf:

1. zuerst erscheint der Spaltennahme
2. dann der Datentyp: **int** = integer, ganzzahliger Datentyp in einem bestimmten Bereich **Factor**, Daten können nur einen bestimmten Wert annehmen
3. danach erscheinen die verschiedenen Beobachtungen der jeweiligen Spalte

2.3 R-Data frames

Data Frames ist die Datenstruktur in Form einer Tabelle (zweidimensional) mit **Zeilen** und **Spalten**. Deren Werte kann eine Mischung aus Zahlen und Buchstaben enthalten.

```
# ein Data Frame hat folgende Form
# diese Information wird später
# noch einmal wichtig
name[<zeile>, <spalte>]
```

```
# so kann man eine Liste generieren
# c() steht für "concatenate"", verbinden
# oder aneinanderreihen
y = c("Unsinn","Quatsch","rot")
y
```

```
## [1] "Unsinn"  "Quatsch" "rot"
```

```
#so kann man einen Data Frame generieren
a = data.frame(5:7,y,8:10, check.names = FALSE)
a
```

```
##   5:7       y 8:10
## 1   5  Unsinn    8
## 2   6 Quatsch    9
## 3   7     rot   10
```

So sieht der Inhalt des erstellten Data Frame schlußendlich aus; wie eine Tabelle ohne Überschrift und ohne Spalten und Zeilennamen

2.4 R-Matritzen

Matritzen in **R** haben ähnlich wie Dataframes die Form einer Tabelle, nur bestehen Matritzen nur aus Zahlen und enthalten keine Buchstaben.

```
# so kann man eine Matrix von 1 bis 30
# aufsteigend erstellen und diese auf
# 10 Spalten und 3 Zeilen verteilen
a = matrix(data = 1:30, ncol = 10, nrow = 3, byrow = TRUE)

# so sieht der Inhalt dann aus
a
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]    1    2    3    4    5    6    7    8    9    10
## [2,]   11   12   13   14   15   16   17   18   19    20
## [3,]   21   22   23   24   25   26   27   28   29    30
```

Mit der Funktion **dim()** lässt sich die Größe eines Dataframes oder einer Matritze anzeigen. Nehmen wir mal die selbsterstellte Matrix mit dem Namen **a**: **dim(a)** gibt die Anzahl der Zeilen und die Anzahl der Spalten der jeweiligen Tabelle wieder.

```
dim(a)
```

```
## [1]  3 10
```

Hier zeigt sich die schon angedeutete Anordnung in den eckigen Klammern: zuerst die links die **Zeile** dann rechts die **Spalte**.

```
# folgender Code zeigt, wie man auf einzelne
# Spalten zugreifen kann so kann man sich die
# 1. Spalte der Tabelle anzeigen lassen
a[,1]
```

```
## [1]  1 11 21
```

```
# hier zeigt sich, wie man sich die 1. Zeile
# der Tabelle anzeigen lassen kann
a[1,]
```

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

Die **mean()** Funktion gibt den Mittelwert von einer Reihe von Zahlen wieder

```
# hier wird der Mittelwert der 1. Zeile der
# Tabelle a ermittelt und ausgegeben
mean(a[1,])
```

```
## [1] 5.5
```

Man kann in der Tabelle den einzelnen Spalten und Zeilen Namen geben. Falls dies im originalen eingelesenen Datensatz nicht schon geschehen ist, gibt es hierfür die Funktionen **colnames()** und **rownames()**

```
# die einzelnen Spalten werden alphabetisch
# angeordnet
colnames(a) = c("a","b","c","d","e","f","g","h","i","j")

# die einzelnen Zeilen bekommen so einen eigenen
# Farbnamen wenn die Zuweisungen Buchstaben
# enthalten, müssen diese in Hochkommata geschrieben
# werden bei der Zuweisung von Zahlen, müssen
# diese nicht in Hochkommata stehen
rownames(a) = c("rot","blau","weiss")

# gib den Inhalt der Tabelle a aus
a
```

```
##        a  b  c  d  e  f  g  h  i  j
## rot    1  2  3  4  5  6  7  8  9 10
## blau  11 12 13 14 15 16 17 18 19 20
## weiss 21 22 23 24 25 26 27 28 29 30
```

2.5 R - nützliche Funktionen

as.factor() Mit dieser Funktion lässt sich ein Vector als Faktor kodieren. Ein als Faktor kodierter Vektor kann nur Werte aus bestimmten vordefiniertem Bereich enthalten.

Kopieren Sie jeweils den Code und lassen Sie sich die Ergebnisse anzeigen.

```
x = c(1:30)
str(x)
```

```
##  int [1:30] 1 2 3 4 5 6 7 8 9 10 ...
```

```
x = as.factor(x)
str(x)
```

```
##  Factor w/ 30 levels "1","2","3","4",..: 1 2 3 4 5 6 7 8 9 10 ...
```

levels() Zeigt die Levels eines Factors an.

```
x = c(1:30)
x = as.factor(x)
levels(x)
```

```
##  [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10" "11" "12" "13" "14" "15"
## [16] "16" "17" "18" "19" "20" "21" "22" "23" "24" "25" "26" "27" "28" "29" "30"
```

table() erstellt eine Übersicht über die Werte einer Spalte.

```
x = c(1:30)
x = as.factor(x)
table(x)
```

```
## x
##  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
##  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
## 27 28 29 30
##  1  1  1  1
```

```
# Hilfefunktionen in R
help("paketname")
?Paketname
```

```
install.packages("dplyr")
library(dplyr)

# zeige nur Zeilen, die bestimmte Bedingung erfüllen
filter()
```

```
filter(list,time < 150)

# ordne Zeilen nach einer bestimmten Spalte
arrange()
arrange(list,time)
```

```
# zusammenfassen der Daten
# min, max, mean, median, var, sd - anwendbar
summarise()
summarise(list,avg = mean(time))

# wähle Spalten nach Namen aus
select()
select(list, VP, time)

# mache Berechnungen und hänge Spalten an
mutate()
mutate(list, time2 = time/100)
```

Eine **Funktion** ist die Aneinanderreihung auszuführender Befehle. Man kann sie selbst definieren, schnell wiederholen und schnell anwenden.

```
<Name> = function(<Liste von übergebenen Argumenten>){
  <Befehle, die ausgeführt werden sollen>
}
```

Beispiel einer Funktion. Probiert diese doch einmal aus!

```
# "percent" berechnet Prozent a von b in Variable d
percent = function(a,b){
    d = 100/b * a
    d
}
percent(34,78)
```

```
## [1] 43.58974
```

Übergebt der Funktion andere Werte.

Chapter 3

Beschreibung Funktionen: Datenauswertung

Nach dem Setzen des Workspace und dem Kopieren des Datensatzes in Workspace Pfad: Einlesen der Daten mit **read.table()**. **head(list)** stellt die ersten 6 Zeilen des eingelesenen Datensatzes dar.

```
# Einlesen
read.table("gugVP11.txt", header = T)

# Speichern in Variable list
list = read.table("gugVP11.txt", header = T)

# Anzeigen der Zeilen 1:6 der Variablen
# list, die den eingelesenen Datensatz enthält.
head(list)
```

str() gibt die Eigenschaften und Datentypen des eingelesenen Datensatzes für alle Spalten wieder. Dies hilft bei der Überprüfung, ob der Datensatz korrekt eingelesen wurde.

```
str(list)
```

table() listet für eine gegebene Spalte die Häufigkeit der auftauchenden Werte auf. Dies gibt die Möglichkeit nach Versuchspersonen oder Items zu schauen, die eventuell nicht besonders funktioniert haben. Ein Beispiel:

```
table(list$code)
```

c() Aneinanderreihung verschiedener übergebener Werte in eine Liste

```
c(3,45,21,17)
```

```
## [1]  3 45 21 17
```

```
variable = c(3,45,21,17)
variable
```

```
## [1]  3 45 21 17
```

rbind(): Datensätze mit gleicher Spaltenzahl werden untereinander angeordnet (Zeilen zusammengefügt) **cbind()**: Datensätze mit gleicher Zeilenanzahl werden nebeneinander angeordnet (Spalten zusammengefügt)

```
# list[<zeile>,<spalte>]
# Ordnet die Zeilen 1 bis 3 und 10 bis 20
# des Datensatzes list untereinander an
rbind(list[1:3,],list[10:20,])

# Ordnet die Spalten des Datensatzes list
# nebeneinander an
cbind(list[,2:4],list[,6:8])
```

```
# code: Übungen 1-6
# code: kritische Items animat: 7 < 400
# code: Filler 401 - 460
# kritische Items inanimat:  > 500

#list[<zeile>,<spalte>]

list1 = rbind(list[(list$code > 7) & (list$code < 400),],list[list$code > 500,])
```

dim() gibt die Dimensionen des übergebenen Datensatzes zurück: Anzahl der Zeilen und Anzahl der Spalten.

```
dim(list)
```

%% Berechne Modulo zwischen zwei Zahlen. Die bisher dargestellten Berechnungen haben bekannte und gebräuchliche Formaten. Der Modulo gibt den Rest eine ganzzahligen Division wieder. 8 %% 2 = 0, weil bei der ganzzahligen Division nur 0 übrig bleibt. 8 %% 3 = 2, weil bei der ganzzahligen Division von 8 % 3, 2 übrig bleibt. (8 / 3 = 2.67; 2 * 3 = 6; 8 - 6 = 2)

```
# Item und Bedingungen werden zusammen in Code
# dargestellt mit diesen Schritten und der Hilfe des
# Modulo Operators werden die beiden getrennt
# im Datensatz dargestellt

# dieser Befehl schreibt die Bedingung in Spalte 10
list1[,10] = list1$code %% 10

# dieser Befehl schreibt das Item (numerierten kritischen Satz) in Spalte 11
list1[,11] = (list1$code - list1$code %% 10) / 10
```

Einfache Berechnung der Rereading Times: Total Reading Time (Spalte 8) - First Path Time (Spalte 6)

```
# Berechnung der Rereading Times, Ergebnis wird
# in Spalte 12 geschrieben
list1[,12] = list1[,8] - list1[,6]
```

colnames() schreibt Spaltennamen für einen Datensatz - ähnlich funktioniert **rownames()** für die Zeilen

```
# Spaltennamen zuordnen (kürzen der übegebenen Spaltennamen)
colnames(list1) = c("VP","list","code","answ","FFD","FPT","RPT","TRT","IA","cond","item","RER")
```

Das Experiment hatte 4 Listen (mixed Design mit within subjects Design Faktor and between subjects Design Faktor). Liste 1 + 2 animate Bedingungen, Liste 3 + 4 inanimate Bedingungen. Folgende Befehle, um den beiden Bedingungen in Liste 3 + 4, die Bedingungen 3 (grammatisch, inanimat) und Bedingung 4 (ungrammatisch, inanimat zuzuordnen. Bedingung 1 soll grammatisch/animat und Bedingung 2 ungrammatisch/animat sein.)

```
# ordne den inanimaten Bedingungen die Codierung 3 und 4 zu
list1[(list1$cond == 1)& (list1$code > 500),10] = 3

list1[(list1$cond == 2)& (list1$code > 500),10] = 4
```

matrix() Befehl erstellt eine Matrix mit übergebenen Zahlen, in gegebener Spaltenanzahl und Zeilenanzahl dargestellt wird. Dadurch entsteht neue Tabelle.

```
# Erstelle eine Matrix bestehend aus 6 Spalten
#und 4 Zeilen, die mit Nullen gefüllt sind
listffd = matrix(data = 0,ncol=6,nrow = 4)
```

```
# 6 Spalten stellen die 6 Interest Areas da
# Spaltennamen für die Matrix
colnames(listffd) = c("ia1","ia2","ia3","ia4","ia5","ia6")

# 4 Zeilen stellen die 4 Bedingungen da
# Zeilennamen für die Matrix
rownames(listffd) = c("cond1","cond2","cond3","cond4")
```

```
# Einzelne Beispielschritte für Mittelwertberechnung

# Zeige Datensatz für Bedingung 1 und Interest Area 1
list1[(list1$cond ==1)& (list1$IA == 1),]

# Zeige die Werte der First Fixation Duration,
#für Bedingung 1 und Interest Area 1
list1[(list1$cond == 1)&(list1$IA == 1),5]

# Berechne den Mittelwert der First Fixation Duration
#Werte für Bedingung 1 und Interest Area 1
mean(list1[(list1$cond == 1)&(list1$IA == 1),5])
```

Eine **Funktion function()** ist die Aneinanderreihung auszuführender Befehle. Man kann sie selbst definieren, schnell wiederholen und schnell anwenden.

```
<Name> = function(<Liste von übergebenen Argumenten>){
  <Befehle, die ausgeführt werden sollen>
}


# Funktion "percent" berechnet Prozent a von b in Variable d
# Name für erstellte Funktion kann selbst gewählt werden,
# "percent" hier entspricht einer Variable
percent = function(a,b){
    d = 100/b * a
    d
}
# Beipielaufführung dieser Funktion mit
#übergebenen Argumenten 34, 78
percent(34,78)
}
```

Probiert die Funktion mit anderen Zahlen aus.

Die folgende Funktion übergibt alle Mittelwerte (Bedingung 1 bis 4 auf allen Interest Areas 1 bis 6) in eine erstellte Matrix **matr**. (Die verwendeten **for()**

Schleifen in der **matall** Funktion kann man sich einmal in der R-Dokumentation durchlesen, sind aber kein relevanter Teil unseres Praktikums.)

```
# Beispielfunktion zur Übergabe aller Mittelwerte
# in die erstellte Matrix
matall = function(list,x){
  matr = matrix(data = 0, ncol = 6,nrow = 4)
  colnames(matr) = c("ia1","ia2","ia3","ia4","ia5","ia6")
  rownames(matr) = c("cond1","cond2","cond3","cond4")
  for (j in 1:4){
    for(i in 1:6){
      matr[j,i] = mean(list[(list$cond==j)&(list$IA ==i),x], na.rm=T)
    }
  }
  matr
}
```

Wenn dieser Code für die Funktion ausgeführt wird, passiert erstmal gar nichts. Die Funktion muss aufgerufen werden und in der Klammer beide Argumente übergeben werden. Das Argument **list** sollte mit der **list1** übergeben werden, **x** enthält den Namen der Spalte, für den die Funktion ausgeführt werden soll.

```
# Ausführung der Funktion, um Mittelwerte für
# die Lesezeit First Fixation Duration des Datensatzes
#list1 zu berechnen (First fixation Duration steht in Spalte 5)
listffd = matall(list1,5)

# Ausführung der Funktion, um Mittelwerte für
# die Lesezeit First Path Time des Datensatzes
# list1 zu berechnen (First Pass Time steht in Spalte 6)
listfpt = matall(list1,6)

# Ausführung der Funktion, um Mittelwerte
# für die Lesezeit Regression-Path Time des Datensatzes
# list1 zu berechnen (Regression Path Time steht in Spalte 7)
listrpt = matall(list1,7)

# Ausführung der Funktion, um Mittelwerte
#für die Lesezeit Total Reading Time des Datensatzes
#list1 zu berechnen (Total Reading Time steht in Spalte 8)
listtrt = matall(list1,8)

# Ausführung der Funktion, um Mittelwerte für die Lesezeit
# Rereading Time des Datensatzes list1 zu berechnen
# (Rereading Time steht in Spalte 12)
listrer = matall(list1,12)
```

3.1 Grafische Darstellung der Daten

Mit der Erstellung von **listffd**, **listfpt**, **listrpt**, **listtrt** und **listrer** haben wir Tabellen, die die jeweiligen Lesezeiten in ms für alle Interest Areas 1 bis 6 und alle 4 Bedingungen darstellt. Unterschiede in den Mittelwerten lassen sich auch schon in den Tabellen ablesen. Trotzdem möchte man die manchmal Daten gern grafisch darstellen. **R** bietet dafür sehr viele verschiedene Pakete wie z.B. *ggplot* an. Die Verwendung von ggplot würde hier aber zu weit führen, da wir wirklich nur 3 Treffen für die Datenauswertung geplant hatten. Wir werden hier die Basisfunktionen von R benutzen. Wer sich da noch weiter reinarbeiten will, kann immer gern die offiziellen Seiten von R, die Hilfefunktion oder Google benutzen.

Schaut Euch doch erstmal die erstellten Tabellen mit den Lesezeiten an

```
listffd
listfpt
listrpt
listtrt
listrer
```

Der folgende Befehl erstellt eine einfache Grafik der Lesezeiten in der übergebenen Tabelle da.

```
matplot(,t(listffd))
```

Die Grafik sieht ein bisschen langweilig und auch wenig informativ aus. Wir können verschiedene Informationen zufügen.

```
matplot(,t(listffd), main = c("First Fixation Durations auf den Interest Areas"),type="
```

Probiert doch mal den obigen Befehl aus. Wenn eine Grafik erstellt wird, ändert doch mal ein paar der Einstellungen (z.B., Farbe, Liniendicke *lwd*, den Titel oder die Achsenbeschriftungen.

Das folgende Beispiel zeigt, wie man die Grafik in ein .pdf Dokument exportieren kann.

```
# Erstmal das Verzeichnis und den Namen angeben,
# unter der die Grafik gespeichert werden soll

pdf("<Hier Ihr Verzeichnis + Name.pdf>")

# Erstellt die Grafik
matplot(,t(listffd), main = c("First Fixation Durations auf den Interest Areas"),type="
```

```
# Fügt die Achsenbeschriftungen hinzu
axis(1,1:6,c("IA1","IA2","IA3","IA4","IA5","IA6"),cex.axis = 1)
axis(2, cex.axis = 1.0)

# Fügt eine Legende hinzu
legend(x=c(3.5,5),
       y=c(300,320),
       legend = c("cond1","cond2","cond3","cond4"),
       lwd=4,
       lty=c(1,2,3,4),
       col=c("black","grey","darkblue","lightblue"),
       xjust=1,
       yjust=1,
       cex = 1)
#trace=TRUE)

# Ende
dev.off()
```

Dieser Befehl wird eine Grafik erstellen und diese als .pdf exportieren. Bitte ausprobieren, schauen, ob die Grafik erstellt wurde und sich öffnen lässt. Danach einzelne Änderungen an der Formatierung ausprobieren (andere Farben, andere Achsenbeschriftungen, andere Titel o.ä.)