R-Anleitung

Experimentalpsychologische Übungen

Author: Siegfried Macho

University of Fribourg: 2019

Inhaltsverzeichnis

Die folgende Anleitung soll eine kleine Übersicht über die Anwendung von R im Rahmen der Experimentalpsychologischen Übungen geben. Es wird gezeigt, wie mit Hilfe einer Abfolge von Befehlen bzw. Funktionen Probleme, welche im Rahmen der Experimentellen Übungen auftauchen, gelöst werden können. In der folgenden Darstellung werden R-Befehle und Ergebnisse von R durch den Zeichensatz `Courier New` repräsentiert.

1. Grundlagen

Das Programm R verfügt über eine vollständige Programmiersprache mit der gleichen Mächtigkeit wie andere Programmiersprachen (z.B. C++ oder Python). Zentral ist die Tatsache, dass das Programm viele eingebaute Funktionen besitzt, welche komplexe Operationen durchführen, wie z.B. statistische Analysen oder Datenmanipulationen. Manche dieser Funktionen befinden sich in speziellen Paketen, welche mit Hilfe der Funktion `install.packages()` installiert und mit Hilfe der Funktion `library()` oder `require()` in die aktuelle Anwendung eingebunden werden können und somit der Benutzerin zur Verfügung stehen.
Die spezielle Anwendung R-Studio ermöglicht eine menügesteuerte Verwendung des Programms. Alle menügesteuerten Aktionen werden jedoch in Befehle übersetzt, welche sodann vom Programm ausgeführt werden. Man kann daher, die Befehle auch in ein Skript schreiben und dieses aufrufen, was zum gleichen Ergebnis führt. Der Vorteil dieser letzteren Methode besteht darin, dass ein vorhandenes Skript – mit leichten Abänderung – wiederverwendet werden kann, falls ein ähnliches Problem zu bearbeiten ist.
In diesem Kapitel werden einige elementare Datenstrukturen von R, sowie einfache Befehlsstrukturen beschrieben.

1.1 Elementare Datenstrukturen

Die wichtigsten Datenstrukturen in R sind:

1. Vektoren
2. Matrizen
3. Listen
4. Datenframes

Hier eine kurze Beschreibung mit Beispielen.

1.1.1 Vektoren

R ist eine Sprache, die vorwiegend mit Vektoren arbeitet. Vektoren sind hierbei als eine Abfolge von Elementen zu betrachten. Hier einige Beispiele:
Der Befehl:

```
>x <- c(1, 3, 7, 9)
```

ergibt einen Vektor. Konkret erzeugt die Funktion `c()` aus den einzelnen Elemente 1, 3, 7, und 9 einen Vektor, der mit Hilfe der Zuordnungsoperation `<-` der Variable `x` zugeordnet wird.
Das Symbol `>` am Beginn ist kein Teil des Befehls. Es symbolisiert den Befehlsinterpreter von R. Im Folgenden bedeutet dies, dass ein Befehl eingegeben wird und es sich nicht um einen Output des Programms handelt.
Eintippen des Namens der Variable:

```
>x
```

zeigt ihren Inhalt an:

```
[1] 1 3 7 9
```

Jedes Element kann durch Verwendung eckiger Klammern angesprochen werden. Der Befehl:

```
>e3 <- x[3]
```

ordnet der Variable `e3` das dritte Element von `x` zu. Eingabe des Namens der Variable:

```
>e3
```

ergibt als Ergebnis:

```
[1] 7
```

Es handelt sich hierbei um einen einzelnen numerischen Wert, der jedoch vom Programm als Vektor der Länge 1 betrachtet wird. Diesem Vektor kann ein zweites Element hinzugefügt werden:

```
>e3[2] <- 100
>e3
[1] 7 100
```

Es können auch nur Teile eines Vektors angesprochen werden, indem als Index für die Adressierung des Inhalts ein Vektor verwendet wird. Der Befehl:

```
>e34 <- x[c(4,3)]
```

ordnet der Variable `e34` den Vektor mit dem 4. und 3. Element von `x` zu.

```
>e34
```

ergibt:

```
[1] 9 7
```

Werden zur Adressierung von Elementen negative Indizes übergeben, so werden die zugehörigen Elemente entfernt:

```
>e12 <- x[c(-3,-4)]
```

ergibt den Vektor `e12`, welcher nur die ersten beiden Elemente von `x` enthält.

```
e12
```

ergibt:

```
[1] 1 3
```

Vektoren können auch andere Typen als Zahlen als Elemente enthalten, z.B.

```
>v <- c("a", "x", "s")
```

ist ein Vektor bestehend aus 3 Buchstaben:

```
>v
[1] "a" "x" "s"
```

1.1.2 Matrizen

Eine Matrix ist eine zweidimensionale Anordnung von Objekten des gleichen Typs z. B. von Zahlen oder Buchstaben. Der Befehl:

```
>X <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), nr = 3)
```

Erzeugt aus dem Vektor `c(1, 2, 3, 4, 5, 6, 7, 8, 9)` eine 3×3-Matrix:

```
>X
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

Das Argument `nr = 3` («number of rows») gibt an, dass die Matrix 3 Zeilen hat. Analog gibt `nc = 3` an, dass die Matrix 3 Spalten besitzt Letzteres muss jedoch nicht angegeben werden, da das Programm aufgrund Länge des übergebenen Vektors und der Anzahl spezifizierte Zeilen die Anzahl Spalten selbst berechnet.
Man beachte, dass die Matrix spaltenweise gefüllt wird. Ist ein zeilenweises Füllen gewünscht, so muss das Argument `byrow = T` übergeben werden:

```
> Y <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), nr = 3,
byrow = T)
> Y
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

1.1.2.1 Adressierung von Elementen

Der Zugriff auf ein Element erfolgt durch Angabe der Zeilen- und Spaltennummer, z.B.

```
>X[2, 3]
[1] 8
```

Durch Weglassen der Zeilen- oder Spaltennummer kann auf die gesamte Spalte bzw. Zeile zugegriffen werden, z.B.

```
>X[2,]
[1] 2 5 8
```

Es wird der Inhalt der 2. Zeile zurückgegeben, analog:

```
>y <- X[,3]
```

```
>y
[1] 7 8 9
```

Spalte 3 wird der Variable `y` zugeordnet. Es ist auch möglich, auf den Inhalt mehrere Zeilen oder Spalten zuzugreifen, z.B.

```
> Z <- X[,c(3,1)]
> Z
     [,1] [,2]
[1,]    7    1
[2,]    8    2
[3,]    9    3
```

Die Variable `Z` repräsentiert eine Matrix, welche aus der 3. und 1. Spalte von `X` besteht.

1.1.2.2 WICHTIGE FUNKTIONEN IM ZUSAMMENHANG MIT MATRIZEN

Die folgenden Funktionen (bzw. Operationen) sind im Zusammenhang mit Matrizen interessant:

Funktion / Operation	Beschreibung

`colSums()`	Berechnet die Summen der einzelnen Spalten:
	`>colSums(X)`
	`[1]  6 15 24`

`rowSums()`	Berechnet die Summen der einzelnen Zeilen:
	`>rowSums(X)`
	`[1] 12 15 18`

`t()`	Transponiert die Matrix: Zeilen werden zu Spalten:
	`> t(X)`
	`     [,1] [,2] [,3]`
	`[1,]    1    2    3`
	`[2,]    4    5    6`
	`[3,]    7    8    9`

`%*%`	Führt eine Matrixmultiplikation durch.
`solve()`	Bildet die Inverse einer Matrix.

`apply()`	Dient zur Anwendung einer Funktion auf die Zeilen oder Spalten:
	`> apply(X, 2, sum)`
	`[1]  6 15 24`
	Die Funktion `sum` wird auf die Spalten der Matrix angewendet. Die `2` im 2. Argument sagt der Funktion, dass die Funktion `sum` auf die Spalten und nicht auf die Zeilen anzuwenden ist. Im letzteren Fall ist eine `1` zu übergeben.
	Das Ergebnis ist identisch zu jenem von `colSums()` [siehe oben].

Bemerkung: Weitere Beispiele folgen.

1.1.3 Listen

Eine *Liste* ist – wie der Name schon sagt – eine Liste von Elementen. Im Unterschied zu Vektoren und Matrizen können die Elemente einer Liste von unterschiedlichem Typ sein. Eine Liste wird mit Hilfe der Funktion `list()` erzeugt, z.B.

```
L <- list(1, "a", matrix(c(1, 2, 3, 4), nr = 2))
> L
[[1]]
[1] 1

[[2]]
[1] "a"

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

Die Liste `L` enthält 3 Elemente, eine Zahl (Typ `numeric`), einen Buchstaben (Typ `character`) und eine Matrix (Typ `matrix`) [siehe Abschnitt 1.1.3.1].
Für die Adressierung der einzelnen Elemente werden nun doppelte eckige Klammern verwendet, z.B.

```
> L[[3]]
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

1.1.3.1 Wichtige Funktionen im Zusammenhang mit Listen

Zwei Funktionen sind im Zusammenhang mit Listen interessant:

1. Die Funktion `unlist()` wandelt – falls möglich – eine Liste in einen Vektor um, z.B.:

   ```
   unlist(L)
   [1] "1" "a" "1" "2" "3" "4"
   ```

   Hierbei wurden alle Elemente in den Typ `character` umgewandelt.

2. Die Funktion `lapply()` wendet eine Funktion auf alle Elemente der Liste an, z.B.

   ```
   lapply(L, class)
   [[1]]
   [1] "numeric"

   [[2]]
   [1] "character"

   [[3]]
   [1] "matrix"
   ```

Es wurde die Funktion `class()`, welche den Typ eines Objekts zurückgibt, auf jedes Element der Liste `L` angewendet. Das Ergebnis ist eine Liste, welche als einzelne Elemente die Bezeichnung der Typen der Listenelemente von `L` enthält. Man beachte den Unterschied:

```
> class(L)
[1] "list"
```

Hier wird der Befehl auf die Liste angewendet und nicht auf die Elemente.

1.1.4 Daten-Frames

Daten werden gewöhnlich als Daten-Frames repräsentiert: Typ: `data.frame`. Hierbei handelt es sich um eine Menge von Listen, wobei jede Liste einen Datenvektor enthält.

```
> head(Daten)
  Person  Bedingung Itemtyp Altnennung
1      1 ungeblockt     Alt         11
2      1 ungeblockt   Köder          4
3      1 ungeblockt     Neu          0
4      2 ungeblockt     Alt          8
5      2 ungeblockt   Köder          5
6      2 ungeblockt     Neu          0
```

`Daten` ist ein Daten-Frame und die Funktion `head()` zeigt die ersten Elemente des Daten-Frames, sowie die Namen der einzelnen Spalten. Analog zeigt die Funktion `tail()` die letzten Elemente des Daten-Frames.

```
> tail(Daten)
   Person Bedingung Itemtyp Altnennung
31     11  geblockt     Alt          6
32     11  geblockt   Köder          2
33     11  geblockt     Neu          0
34     12  geblockt     Alt          8
35     12  geblockt   Köder          3
36     12  geblockt     Neu          0
```

Die gleiche Methode wie für Matrizen funktioniert auch für die Adressierung von Elementen eines Daten-Frames, z.B.

```
> Daten[2, 3]
[1] Köder
Levels: Alt Köder Neu
```

Es wird das Element `Köder` in der 2. Zeile und 3. Spalte zurückgegeben. Die Information in der zweiten Zeile zeigt an, dass es sich um ein Element des Typs `factor` handelt, welcher die Stufen `Alt Köder Neu` umfasst (siehe unten).
Wird der Zeilenindex weggelassen, so wird die gesamte Spalte ausgegeben und analog bei Weglassung des Spaltenindex, z.B.:

```
> Daten[,3]
```

```
 [1] Alt   Köder Neu   Alt   Köder Neu   Alt   Köder
Neu   Alt   Köder Neu
[13] Alt   Köder Neu   Alt   Köder Neu   Alt   Köder
Neu   Alt   Köder Neu
[25] Alt   Köder Neu   Alt   Köder Neu   Alt   Köder
Neu   Alt   Köder Neu
Levels: Alt Köder Neu
```

Eine alternative Möglichkeit für den Zugriff auf einer Spalte besteht in der Verwendung des Dollarzeichens mit dem Namen der Spalte, z.B.

```
> class(Daten$Itemtyp)
[1] "factor"
```

Die Funktion `class()` wird auf die Spalte mit dem Namen `Itemtyp` angewendet und liefert den Typ der Spalte zurück. Im aktuellen Fall ist die Spalte – wie schon oben ausgeführt – vom Typ `factor`.
Die folgenden Funktionen (bzw. Operationen) sind im Zusammenhang mit Daten-Frames interessant:

Funktion / Operation	Beschreibung

`read.table()`
`read.csv()`
`read_excel()`
	Einlesen von Daten in einen Daten-Frame von:
	❒ Textdatei
	❒ Komma-separierter Datei
	❒ Excel-Datei (diese Funktion befindet sich im Paket `readxl`)

`write.table()` `write.csv()`	Speichern eines Daten-Frames in eine: ❒ Textdatei ❒ Komma-separierte Datei

1.2 Elementare Befehlsstrukturen

1.2.1 Definition von Funktionen

Die Spezifikation einer selbstgeschriebene R-Funktion umfasst vier Komponenten:

1. Das reservierte Wort `function`,
2. Einer Liste von Argumenten, welche der Funktion übergeben werden können. Dies werden unmittelbar nach `function` in runden Klammern aufgelistet.
   Jedes Argument hat einen Namen und es kann ein default-Wert spezifiziert werden, welcher verwendet wird, falls der Funktion beim Aufruf kein Argument übergeben wird.
3. In geschwungen Klammern eine Liste von Befehlen.
4. Die Zuordnung zu einem Namen, unter welchem die Funktion aufgerufen werden kann.

Hier als Beispiel eine Funktion, welche den Standardfehler eines Datenvektors berechnet.

```
standard_error <- function(x, n = 1) {
    sd(x) / sqrt(n)
}
```

Die Funktion besitzt zwei Argumente: `x` und `n`. Hierbei bezeichnet `x` einen Datenvektor, welcher der Funktion als erstes Argument übergeben wird und `n` bezeichnet die Stichprobengrösse. Die Funktion berechnet den Standardfehler des Mittelwerts (*SE*):

$$SE = \frac{s}{\sqrt{n}}$$

s ist hierbei die Standardabweichung der Werte in `x`, welche mit Hilfe der vorgegebenen Funktion `sd()` berechnet wird (`sqrt()` berechnet die Wurzel):

$$s = \sqrt{\frac{1}{n-1} \cdot \sum_{i=1}^{n} \left(x_i - \bar{x}\right)^2}$$

Eine Funktion gibt immer den zuletzt berechneten Wert zurück. Mit Hilfe der Funktion `return()` kann eine Funktion sofort verlassen werden, mit dem in dem in der Klammer befindlichen Wert als Ergebnis.
Nehmen als Beispiel eine Funktion, welche den Vektor, bestehend aus *SE* und den Mittelwert $\bar{x}$ zurückgibt:

```
se_mean <- function(x) {
    n <- length(x)
    se <- sd(x) / sqrt(n)
    return(c(se, mean(x)))
}
```

In der ersten Zeilen wird die Anzahl Elemente mit Hilfe der Funktion `length()` berechnet und der Variable `n` zugeordnet. Sodann wird der Standardfehler berechnet und der Variable se zugeordnet. In der nächsten Zeile wird der Vektor mit dem Standardfehler und der Mittelwert, welcher mit der Funktion `mean()` berechnet wird, mit Hilfe der Funktion `c()` generiert und mittels der Funktion `return()` zurückgegeben.

Bemerkung:
Die Funktion `return()` ist an dieser Stelle überflüssig, da das Ergebnis des letzten Ausdrucks `c(se, mean(x))` der gewünschte Vektor ist und der letzte Ausdruck immer zurückgegeben wird.

Werden mehrere Befehle in einer Zeile geschrieben, so sind diese durch einen Strichpunkt zu trennen, z.B.

```
se_mean <- function(x) {
    n <- length(x); c(sd(x)/ sqrt(n), mean(x))
}
```

Hier wurden die einzelnen Schritte zu 2 Befehlen kombiniert, welche durch Strichpunkt getrennt in einer Zeile geschrieben wurden.

Die Funktion erhält einem Namen – im aktuellen Fall `se_mean` – unter welchem sie aufgerufen werden kann, z.B.

```
> x <- c(3, 1, 5, 7, 9)
> se_mean(x)
[1] 1.414214 5.000000
```

Die im Datenvektor `x` übergebenen Werte besitzen einen Standardfehler von *SE* = 1.4142 (= $\sqrt{2}$) und einen Mittelwert von $\bar{x} = 5$.

2. Randomisieren von Zeilen und Spalten einer Matrix oder eines Daten-Frames

Aufgabe: *Randomisieren der Wortlisten*

Die 24 Wortlisten mit je 6 Worten pro Liste sollen wie folgt randomisiert werden:

1. Eine zufällige Reihenfolge der 24 Liste ist zu erstellen. Aus den ersten 12 der zufällig gereihten Wortlisten wird dann die Lernliste 1 und aus den zweiten 12 die Lernliste 2 erstellt.
2. Die Worte innerhalb jeder Wortliste sind in eine zufällige Reihung zu bringen, wobei für jede Wortliste eine andere Zufallsreihung gebildet werden soll.

R-Lösung: *Randomisieren der Wortlisten*

Wir nehmen an , dass die Listen als 6×24 Daten-Frame `Listen` mit den 24 Listen als Spalten vorliegt, oder analog als 6×24 Matrix. Die folgenden beiden Befehle erledigen die beiden Teilaufgaben.

```
L1 <- Listen[, sample(24)]
L2 <- apply(L1, 2, sample)
```

Der Befehl in der erste Zeile erzeugt eine Zufallsreihung der 24 Listen und der zweite Befehl führt eine Zufallsreihung der 6 Worte innerhalb jeder Liste durch (für jede Liste eine andere Zufallsreihenfolge).

Die fertigen Listen befinden sich in `L2`.

3. Zusammenfassung der Rohdaten

Aufgabe: *Zusammenfassen der Rohdaten*

Wir nehmen an, die Daten befinden sich in dem Daten-Frame `Data`, der wie folgt aussieht:

```
> head(Data)
  Vp     T2      T1    Typ Antwort n key_resp_2.keys_raw
1  1   Berg   Ärger Koeder       n 1                  'n'
2  1  Musik schwarz Koeder       n 1                  'a'
3  1  Nadel    Brot Koeder       n 1                  'n'
4  1  Fluss   Stuhl Koeder       n 1                  'n'
5  1    rau    kalt Koeder       n 1                  'n'
```

```
6  1 Schlaf  Doktor Koeder       n 1                 'a'
```

Hierbei gilt:

`Vp` bezeichnet die Versuchspersonennummer.

`T1` und `T2` bezeichnen die Testlisten (irrelevant).

`Typ` bezeichnet den Worttyp.

`key_resp_2.keys_raw` bezeichnet die Antwort der Person.

Die Aufgabe besteht darin, für die verschiedenen Worttypen die Anzahl Alt-Antworten zu sammeln, d.h. die Antworten `'a'` (= Vp hat die Taste "a" gedrückt).

R-Lösung: *Zusammenfassen der Rohdaten*

Der folgende Befehl erledigt die Aufgabe:

```
X <- aggregate(key_resp_2.keys_raw ~ Typ + Vp, data
= Data, FUN = function(x) sum(x == "'a'"))
```

- Das erste Argument der Funktion bildet die Formel:

  `key_resp_2.keys_raw ~ Typ + Vp`

  Diese teilt der Funktion mit, dass die nachfolgend übergebene Funktion auf die Variable `key_resp_2.keys_raw` angewendet wird, für jede `Typ` × `Vp`-Kombination.
- Das zweite Argument der Funktion gibt an, welcher Daten-Frame als Input für die Berechnung verwendet wird.
- Das dritte Argument ist die Funktion:

  `function(x) sum(x == "'a'")`

  Diese bildet zuerst ein Vektor bestehend aus den Werten `TRUE` und `FALSE`, je nachdem, ob ein Eintrag im übergebenen Vektor `x` den Wert `'a'` aufweist oder nicht.
  Die Werte des resultierenden Vektors werden aufsummiert, mittels der Funktion `sum()`. Hierbei wird automatisch `TRUE` in 1 und `FALSE` in 0 umgewandelt.

Beachte:

Die Funktion `aggregate()` bewirkt, dass die Berechnungen, welche von der übergebenen Funktion ausgeführt werden, separat für jede `Typ` × `Vp`-Kombination durchgeführt werden.

Das Ergebnis der Berechnung befindet sich im Daten-Frame `X`, der wie folgt aussieht:

```
> X
         Typ Vp key_resp_2.keys_raw
1        Alt  1                   6
2     Koeder  1                   5
3 Unverwandt  1                   1
4        Alt  2                   5
5     Koeder  2                   4
6 Unverwandt  2                   3
7        Alt  3                   4
8     Koeder  3                   5
9 Unverwandt  3                   9
```

Zu diesem Daten-Frame muss noch ein Vektor hinzugefügt werden, welcher die Information über die Bedingung enthält, z.B.

```
X$Bedingung <- factor(rep(c(1, 2, 1), each = 3),
labels = c("ungeblockt", "geblockt"))
```

Die Funktion `factor()` bildet einen Faktor. Im aktuellen Fall hat dieser 2 Stufen und die Bezeichnungen *ungeblockt* und *geblockt*.
Die Funktion `rep()` [Abkürzung für `replicate()`] repliziert die Elemente im übergebenen Vektor so oft wie gewünscht. Im aktuellen Fall wird der Vektor `1, 2, 1` übergeben und die Funktion macht daraus den Vektor: `1, 1, 1, 2, 2, 2, 1, 1, 1`.

Bemerkung:
Das Argument `each = 3` wurde verwendet, weil andernfalls der Vektor `1, 2, 1, 1, 2, 1, 1, 2, 1` erzeugt würde, d.h. der gesamte Vektor wird 3× hintereinander gesetzt.

Es ergibt sich:

```
> X
          Typ Vp key_resp_2.keys_raw  Bedingung
1        Alt  1                   6 ungeblockt
2     Koeder  1                   6 ungeblockt
3 Unverwandt  1                   6 ungeblockt
4        Alt  2                   6   geblockt
5     Koeder  2                   6   geblockt
6 Unverwandt  2                   6   geblockt
7        Alt  3                   4 ungeblockt
8     Koeder  3                   5 ungeblockt
9 Unverwandt  3                   9 ungeblockt
```

Die zusammengefassten Daten können für die Erstellung der Graphik und der statistischen Analyse verwendet werden.

4. Erstellen eines Balken-Diagramms

Im Folgenden werden Methoden der graphischen Darstellung der Daten mittels Balkendiagramm besprochen. Wir gehen davon aus, dass sich die Daten im Daten-Frame `Daten` befinden, welcher wie folgt aussieht:

```
> head(Daten)
  Vp  Bedingung Itemtyp  N
1  1 ungeblockt     Alt 11
2  1 ungeblockt   Köder  4
3  1 ungeblockt     Neu  0
4  2 ungeblockt     Alt  8
5  2 ungeblockt   Köder  5
6  2 ungeblockt     Neu  0
```

Auf der Basis von `Daten` werden die Graphiken erstellt. Für die Erstellung der Graphik verwenden wir das R-Paket `ggplot2`.

Aufgabe: *Erstellen eines Balkendiagramms*

Es soll ein Balkendiagramm erstellt, welches die Prozentwerte der Alt-Antworten als Funktion der *Bedingung* (geblockt vs. ungeblockt) und des *Itemtyps* (Alt, Köder, Neu) darstellt. Zusätzlich sollen Fehlerbalken, welche 95% Konfidenzintervalle für die Mittelwerte (Prozente) darstellen, im Diagramm enthalten sein.
Abb. 4-1 zeigt, wie das mittel `ggplot` erstellte Balkendiagramm aussehen soll.

Die Erstellung der Graphik umfasst drei Schritte:

1. Eine Berechnung der relevanten Daten: Die Prozentwerte, sowie die oberen und unteren Grenzen der Fehlerbalken.
2. Die Zusammenfassung der Daten zu einer Datenstruktur, auf deren Basis das Diagramm erstellt wird.
3. Die Spezifikation der einzelnen Komponenten der Graphik.

Im Folgenden werden die drei Schritte im Detail beschrieben.

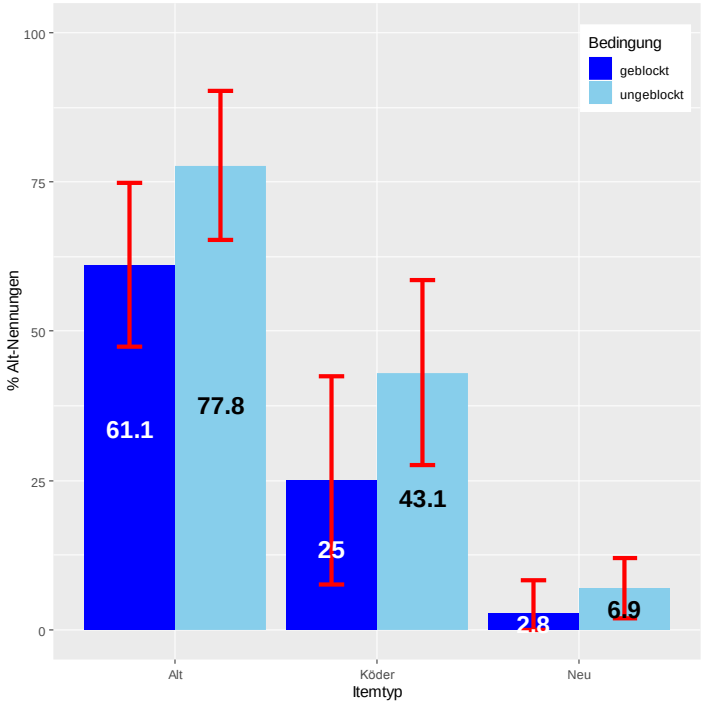

Abb. 4-1: *Prozentsatz der Alt-Antworten als Funktion der Variablen Itemtyp und Bedingung (Balkendiagramm erstellt mit Hilfe des R.Pakets ggplot2).*

Schritt 1: *Berechnung der relevanten Daten:*

Wir nehmen an, dass die Daten in dem Daten-Frame `Daten` befindet, der die Eingangs des Abschnitts dargestellte Struktur besitzt. Hieraus sollen nun die Prozentwerte für die einzelnen Kombinationen von *Bedingung* und *Itemtyp*, sowie die oberen und unteren Grenzen des 95%-Konfidenzinterfalls errechnet werden.

Für diese Berechnung wird die Funktion `aggregate()` verwendet. Diese ermöglicht es, eine Berechnung separat für alle Kombinationen der Stufen verschiedener Faktoren auszuführen.

Für die Berechnung der Prozentwerte wird folgender Befehl ausgeführt:

```
M <- aggregate(N ~ Bedingung + Itemtyp, data =
Daten, function(x) { y = x/12 * 100; mean(y) })
```

- Das erste Argument der Funktion bildet die Formel:

  `N ~ Bedingung + Itemtyp`

  Diese teilt der Funktion mit, dass die nachfolgend übergebene Funktion auf die Variable `N` angewendet wird, für jede `Bedingung` × `Itemtyp`-Kombination.
- Das zweite Argument der Funktion gibt an, welcher Daten-Frame als Input für die Berechnung verwendet wird.
- Das dritte Argument ist die Funktion:

  `function(x) { y = x/12 * 100; mean(y) }`

  Diese berechnet zuerst aus den Rohhäufigkeiten die Prozentwerte, indem die Häufigkeiten durch die maximal mögliche Anzahl von Antworten dividiert und mit 100 multipliziert wird. Aus den einzelnen Prozentwerten wird dann mit Hilfe der R-Funktion `mean()` der Mittelwert errechnet.

  Bemerkung:

  Die beiden Befehle innerhalb der Funktion können in einen einzigen zusammengefasst werden:

  `mean(x/12 * 100)`

Für die Berechnung der unteren Grenze des 95%-Konfidenzintervalls wird folgender Befehl verwendet:

```
LO <- aggregate(N ~ Bedingung + Itemtyp, data =
Daten, function(x) { y = x/12 * 100; max(mean(y) -
qnorm(.975) * sd(y)/sqrt(length(y)), 0) } )
```

Der Befehl unterscheidet sich von jenem zur Berechnung der Prozentwerte nur in der übergebenen Funktion:

```
function(x) { y = x/12 * 100; max(mean(y) -
qnorm(.975) * sd(y)/sqrt(length(y)), 0) }
```

Diese Funktion führt folgende Schritte durch:

1. Der Befehl `y = x/12 * 100` wandelt wiederum die Roh-

häufigkeiten in Prozentwerte um.

2. Nun wird die untere Grenze des Konfidenzintervalls berechnet. Der Ausdrick:

   ```
   mean(y) + qnorm(.025) * sd(y)/sqrt(length(y))
   ```

   berechnet die Formel:

   $$\bar{y} + z_{.025} \cdot \frac{s}{\sqrt{n}}$$

   s ist hierbei die Standardabweichung, welche durch die Funktion `sd(y)` berechnet wird (Vgl. oben, Abschnitt 1.2.1).

   Die Funktion `qnorm(.025)` berechnet das .025 Quantil, d.h. jenen Wert auf der *x*-Achse, links von welchem sich 2.5% der Fläche unter der Standardnormalverteilung befinden (vgl. die linke hellblaue Fläche unter der Normalverteilung in Abb. 4-2). Rechts davon befindet sich daher 97.5% der Fläche.

   Damit die Berechnung keine negativen Werte ergibt (was bei %-Werte unsinnig ist), wird die Funktion `max` verwendet, welche den grösseren von 2 Werte (berechneter Wert oder 0) zurückgibt.

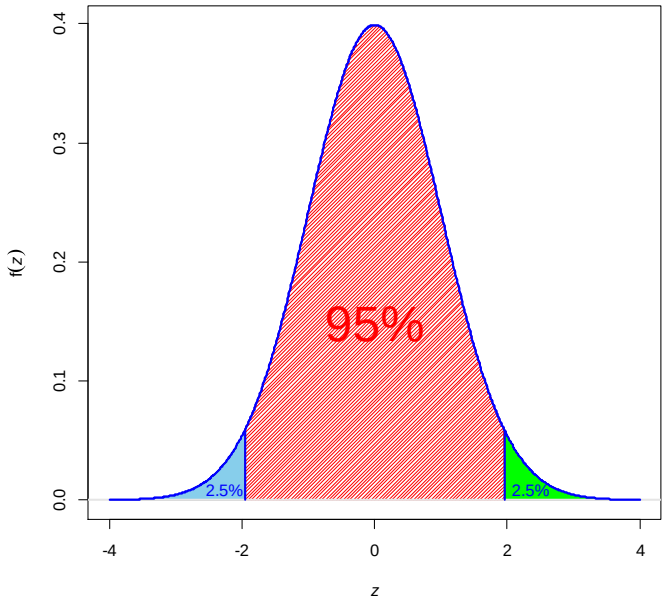

Abb. 4-2: *Flächen unter der Standardnormalverteilung.*

Analog wird für die Berechnung der oberen Grenze des 95%-Konfidenzintervalls folgender Befehl verwendet:

```
UP <- aggregate(N ~ Bedingung + Itemtyp, data =
Daten, function(x) { y = x/12 * 100; min(mean(y) +
qnorm(.975) * sd(y)/sqrt(length(y)), 100) } )
```

Dieser unterscheidet sich nur von jener für die untere Grenze, indem nun der Ausdruck:

$$\bar{y} + z_{.975} \cdot \frac{s}{\sqrt{n}}$$

berechnet wird.
Somit schneidet die untere Grenze 2.5% und die obere Grenze 97.5% der Fläche der Standardnormalverteilung ab (vgl. die hellgrüne Fläche auf der rechten Seite von Abb. 4-2). Es verbleiben daher in der Mitte genau 95% (die rot schraffierte Fläche in Abb. 4-2).

Das Ergebnis der Berechnung sind die 3 Daten-Frames: `M`, `LO` und `UP`, welche wie folgt aussehen:

```
> M
   Bedingung Itemtyp         N
1   geblockt     Alt 61.111111
2 ungeblockt     Alt 77.777778
3   geblockt   Köder 25.000000
4 ungeblockt   Köder 43.055556
5   geblockt     Neu  2.777778
6 ungeblockt     Neu  6.944444

> LO
   Bedingung Itemtyp         N
1   geblockt     Alt 47.337888
2 ungeblockt     Alt 65.362762
3   geblockt   Köder  7.612158
4 ungeblockt   Köder 27.608577
5   geblockt     Neu  0.000000
6 ungeblockt     Neu  1.925007

> UP
   Bedingung Itemtyp         N
1   geblockt     Alt 74.884334
2 ungeblockt     Alt 90.192793
3   geblockt   Köder 42.387842
4 ungeblockt   Köder 58.502534
5   geblockt     Neu  8.222122
6 ungeblockt     Neu 11.963882
```

Schritt 2: *Zusammenfassung der berechneten Daten in einen Daten-Frame*

Wir stellen nun die Ergebnisse in dem Daten-Frame `Graph` zusammen:

```
Graph <- cbind(M, Lower = LO$N, Upper = UP$N,
Labels = round(M$N, 1))
```

Die Funktion `cbind()` reiht die Spalten von `M`, `LO$N`, `UP$N` und

`M$N` nebeneinander. Das Resultat ist der folgende Daten-Frame:

```
   Bedingung Itemtyp         N     Lower     Upper Labels
1   geblockt     Alt 61.111111 47.337888 74.884334   61.1
2 ungeblockt     Alt 77.777778 65.362762 90.192793   77.8
3   geblockt   Köder 25.000000  7.612158 42.387842   25.0
4 ungeblockt   Köder 43.055556 27.608577 58.502534   43.1
5   geblockt     Neu  2.777778  0.000000  8.222122    2.8
6 ungeblockt     Neu  6.944444  1.925007 11.963882    6.9
```

Spalte `N` enthält die Prozentwerte, `Lower` die untere und `Upper` die obere Grenze des 95%-Konfidenzintervalls. Die Spalte `Labels` enhält die gerundeten %-Werte, welche als Text den Balken eingeschrieben werden. Auf der Basis dieser Daten wird nun das Balkendiagramm erstellt.

Schritt 3: *Erstellen des Balkendiagramms mit ggplot*

Die Erstellung des Diagramms erfolgt in einzelnen Schritten, welche zur Konfiguration der verschiedenen Komponenten des Diagramms dienen. Hierbei wird zuerst das Graphikobjekt `Balken` erstellt, welchem schrittweise die gesamte notwendige Konfigruationsinformation hinzugefügt wird.

1. Bildung des Objekts `Balken`, welche die grundlegende Informationen enthält:

   ```
   Balken <- ggplot(Graph, aes(Itemtyp, N, fill =
   Bedingung))
   ```

   - ❒ Der Funktion `ggplot()` wird zuerst der Daten-Frame mit den zu verwendenden Daten übergeben. Auf diese wird in allen folgenden Funktionen zugegriffen.
   - ❒ Mit Hilfe der Funktion `aes()` im zweiten Argument wird festgelegt, welche Variable mit der *x*- und *y*-Achse verbunden wird. Im aktuellen Fall bildet `Itemtyp` die *x*- und `N` die *y*-Achse (*Beachte:* Beide Variablen befinden sich im Datenframe `Graph`).
     Das Argument `fill` innerhalb der Funktion `aes()` repräsentiert jene Variable, deren Stufen die Balken bilden. Im aktuellen Fall gibt es zwei Balken, einen für die geblockte und einen für die ungeblockte Bedingung (vgl. Abb. 4-1)

2. *Hinzufügen der Information zur Konfigurierung der Balken:* Nun wird zur bestehenden Information die Information zur Konfigurierung der Balken hinzugefügt:

   ```
   Balken <- Balken + geom_bar(stat = "identity",
   position = "dodge")
   ```

   - ❒ der Funktion `geom_bar()` wird im ersten Argument `stat` mitgeteilt, dass die Werte für `N` – wie im Daten-Frame vorhanden – zu übernehmen sind (Es wäre auch möglich, diese berechnen zu lassen).

- ❒ Im zweiten Argument wird der Funktion mitgeteilt, dass die Balken nebeneinander zu positionieren sind (und nicht z.B. aufeinander gestaffelt).

Diese Konfiguration würde ausreichen. Es sollen jedoch auch die Balkenfarben nach eigenem Wunsch festgelegt werden. Hierzu eignet sich folgender Befehl:

```
Balken <- Balken + scale_fill_manual(values =
c("blue", "skyblue"))
```

Beachte: Wiederum wird diese Konfigurationsinformation zum bestehenden Balkenobjekt hinzugefügt.

Hinweis:
Mit Hilfe der Funktion `colors()` können alle in R definierten Farbnamen angezeigt werden.

3. Hinzufügen der Information zur Konfiguration der Fehlerbalken:

```
dodge <- position_dodge(width = 0.9)
Balken <- Balken + geom_errorbar(aes(ymin =
Lower, ymax = Upper), position = dodge, width =
0.25, col = "red", lwd = 1.5)
```

   - ❒ Der Befehl in der ersten Zeile erzeugt eine Variable `dodge`, welche der Funktion `geom_errorbar()` übergeben wird.
   - ❒ Die obere und untere Grenze werden mit Hilfe der Funktion `aes()` spezifiert, wobei die Werte aus dem Daten-Frame genommen werden.
   - ❒ Die horizontale Position der Balken wird mit Hilfe der Variable `dodge`, gesteuert, welche zuvor mit dem oben gezeigten Befehl erzeugt wurde. Sie bewirkt, dass die Balken horizontal korrekt positioniert werden.
   - ❒ Die folgenden Argumente spezifizieren:
     (a) Die Breite der begrenzenden horizontalen Linien der Fehlerbalken (Argument `width`).
     (b) Die Farbe (Argument `col`) und
     (c) Die Dicke der Linien (Argument `lwd`).

4. *Datenwerte für Balken:* Hinzufügen der durch die Balken angezeigten Werte:

```
Balken <- Balken + geom_text(aes(label =
Labels), position = position_dodge(width=0.9),
col = rep(c("black", "white"), 3), size = 6,
vjust = c(13, 9, 7, 4, 1.5, 1), fontface =
"bold")
```

   - ❒ Im Argument: `aes(label = Labels)` werden die anzuzeigenden Datenlabels übergeben, welche in die Balken eingeschrieben werden.
   - ❒ Die vertikale Position wird festgelegt (Im Argument

`position`).

- ❒ Die Farben der Datenlabels werden spezifiziert (Argument `col`).
- ❒ Die Grösse der Datenlabels wird spezifiziert (Argument: `size`).
- ❒ Die vertikale Abweichung von der Balkenhöhe wird festgelegt (Argument: `vjust`)
- ❒ Es wird festgelegt, dass die Zahlen fett angezeigt werden. (Argument `boldface`).

5. *Fixieren des Bereichs der y-Skala:* Die *y*-Skala soll den Bereich von 0-100 umfassen. Dies wird mit Hilfe des folgenden Befehls erreicht:

```
Balken <- Balken + scale_y_continuous(limits =
c(0, 100))
```

6. Hinzufügen der Information zur Position der Legende und zu den Bezeichnungen der Achsen:

```
Balken <- Balken + theme(legend.position =
c(0.9, 0.9))
Balken <- Balken + labs(x = "Itemtyp", y = "%
Alt-Nennungen")
```

- ❒ Der erste Befehl legt die Legendenposition fest, wobei für die *x*- und *y*-Position der Legende Zahlen zwischen 0 und 0.95 verwendet werden müssen. Im aktuellen Fall wird die Legende links oben angezeigt.
  Falls die Legende nicht konfiguriert wird, so wird sie ausserhalb der Graphik positioniert.
- ❒ Der zweite Befehl dient zur Spezifikation der Achsenbeschriftungen.

Die Konfiguration des Diagramms ist nun beendet und die Graphik kann entweder durch Eingabe des Namens des Objekts `Balken` oder durch den Befehl `print(Balken)` angezeigt werden.

Übung 4-1:
Erstellen Sie mit Hilfe Ihrer Daten ein Balken-Diagramm, bei welchem die Variable *Bedingung* die *x*-Achse bildet.

Hinweis:
Das Diagramm sollte ungefähr wie in Abb. 4-3 aussehen. Die einseitigen Fehlerbalken ergeben sich, indem man den Ausdruck zur Konfiguration der Fehlerbalken vor den Befehl zur Konfiguration der Balken setzt. Dies bewirkt dass die Fehlerbalken zuerst geplottet und damit von den anschliessend gezeichneten Balken überdeckt werden.

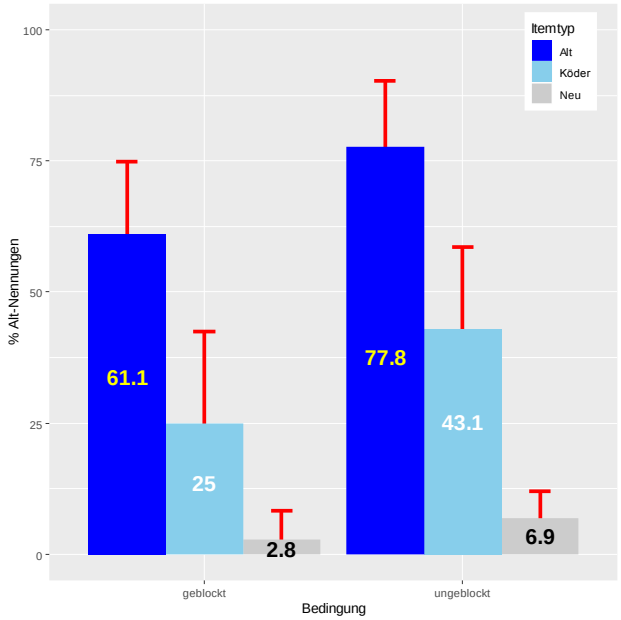

Abb. 4-3: *Prozentsatz der Alt-Antworten als Funktion der Variablen Bedingung und Itemtyp (Balkendiagramm erstellt mit Hilfe des R.Pakets ggplot2).*

Falls die untere Grenze des Konfidenzintervalls 0 ist, so wird diese im Graphen sichtbar sein. Dies kann verhindert werden, indem man vor Konfiguration des Diagramms in der Datenstruktur `Graph`, welche die Daten für das Diagramm enthält, den Wert 0 durch einen kleinen positiven Wert (z.B. 2) ersetzt. (Nach Zeichnen des Graphen kann der Wert zurückgesetzt werden).

5. Statistische Analyse

5.1 Durchführung einer Varianzanalyse

Wir verwenden den gleichen Daten-Frame `Daten`, welcher in Kapitel 4 für die Erstellung des Balkendiagramms verwendet wurde:

```
> head(Daten)
  Vp  Bedingung Itemtyp  N
1  1 ungeblockt     Alt 11
2  1 ungeblockt   Köder  4
3  1 ungeblockt     Neu  0
4  2 ungeblockt     Alt  8
5  2 ungeblockt   Köder  5
6  2 ungeblockt     Neu  0
```

Aufgabe: *Durchführen einer Varianzanalyse mit einem between-subjects und einem within-subjects Faktor*

Es ist eine zwei-faktorielle Varianzanalyse mit der Variable *Bedingung* als between-subjects Faktor (Gruppierungsfaktor) und der Variable *Itemtyp* als within-subjects Faktor (Wiederhol-

ungsfaktor) durchzuführen.

R-Lösung: *Durchführung einer Varianzanalyse mit ezANOVA*

Die Varianzanalyse wird mit Hilfe der Funktion `ezANOVA()` durchgeführt, welche sich im Paket `ez` befindet.
Gegeben: Der Datenframe `Daten`:
Die Varianzanalyse erfolgt mit Hilfe des Befehls:

```
ezA.obj <- ezANOVA(data = Daten, wid = Vp, dv = N,
within = Itemtyp, between = Bedingung, detailed =
T)
```

- Im Argument `data` wird der Funktion der Daten-Frame mit den Daten übergeben.
- Im Argument `wid` wird die Versuchspersonen-Variable, in welche die within-Variablen eingebettet sind, übergeben.
- In den Argumenten `within` und `between` werden die Namen der within und between Variablen übergeben.

  Hinweis:
  Falls mehrere Variablen zu übergeben sind, so erfolgt dies auf folgende Weise: `within = .(Var1, Var2)` [Vgl. die Hilfeseite der Funktion].

Die Ergebnisse werden im Objekt `ezA.obj` gespeichert. Bei Eingabe des Namens wird der Inhalt (und damit das Ergebnis der Varianzanalyse) angezeigt.

Bemerkung:
Falls die unabhängigen Variablen nicht als Faktoren (Typ `factor`) definiert sind, so gibt das Programm eine Warnung aus. Die Berechnung ist aber dennoch korrekt.

Interpretation der Ausgabe von ezANOVA

Die Funktion `ezANOVA()` gibt eine Liste zurück, die in der Variable `ezA.obj` gespeichert wird (siehe obigen Befehl). Durch Eingabe der Variable wird die Liste angezeigt. Im aktuellen Fall hat die Liste folgenden Inhalt:

```
$ANOVA
             Effect DFn DFd     SSn    SSd       F     p p<.05   ges
1       (Intercept)   1  10 676.000 45.556 148.390 0.000     * 0.865
2         Bedingung   1  10  21.778 45.556   4.780 0.054       0.171
3           Itemtyp   2  20 361.500 60.111  60.139 0.000     * 0.774
4 Bedingung:Itemtyp   2  20   5.056 60.111   0.841 0.446       0.046

$`Mauchly's Test for Sphericity`
             Effect    W     p p<.05
3           Itemtyp 0.84 0.456
4 Bedingung:Itemtyp 0.84 0.456

$`Sphericity Corrections`
             Effect   GGe p[GG] p[GG]<.05   HFe p[HF] p[HF]<.05
```

```
3           Itemtyp 0.862 0.000         * 1.025 0.000         *
4 Bedingung:Itemtyp 0.862 0.432           1.025 0.446
```

Die Liste umfasst die folgenden 3 Einträge:

```
$ANOVA
$`Mauchly's Test for Sphericity`
$`Sphericity Corrections`
```

Der Eintrag `$ANOVA` enthält die Tabelle der Varianzanalyse mit den folgenden Spalten:

`Effect`: Name des Effekts (Name der UV).
`DFn`: Nenner-Freiheitsgrade.
`DFd`: Zähler-Freiheitsgrade.
`SSn`: Sum of Squares des Nenners.
`SSd`: Sum of Squares des Zählers.
`F`: *F*-Statistik: $\texttt{F} = \frac{\texttt{SSn}/\texttt{dfn}}{\texttt{SSd}/\texttt{dfd}}$.
`p`: Der zur *F*-Statistik gehörige *p*-Wert.
`p<.05`: Ein Indikator, ob signifikant auf 5% Niveau.
`ges`: Generalisiertes Eta-Squared (Mass der Effektstärke).

Der Listeneintrag `$`Mauchly's Test for Sphericity`` enthält die Ergebnisse von Mauchlys Sphärizitätstest. Folgende Spalten sind relevant:

`W`: Mauchlys W.
`p`: Der zu gehörige *p*-Wert.
`p<.05`: Ein Indikator, ob signifikant auf 5% Niveau.

Bemerkung:
Die Zeile mit der Bezeichnung `(Intercept)` kann ignoriert werden. Hier wird nur getestet, ob der Gesamtmittelwert von 0 verschieden ist (was für unser Experiment trivialerweise der Fall ist).

Der Listeneintrag `$`Sphericity Corrections`` enthält die Koeffizienten zur Korrektur der Freiheitsgrade, um Verletzungen der *Sphärizität* bzw. *Compound Symmetry* auszugleichen. Die Spalten haben die folgende Bedeutung:

`GGe`: Greenhouse-Geisser epsilon (ε_{GH}): Mit diesem Wert werden die Zähler- und Nenner-Freiheitsgrade multipliziert, welche für die Berechnung der Wahrscheinlichkeit aufgrund des *F*-Wertes (und der Freiheitsgrade) verwendet werden (siehe die Spalte `p[GG]`).
`p[GG]`: Die berechneten *p*-Werte, wobei die Freiheitsgrade für den *F*-Test mit ε_{GH} multipliziert wurden.
`p[GG]<.05`: Ein Indikator, ob signifikant auf 5% Niveau.
`HFe`: Huynh-Feldt epsilon (ε_{HF}): Mit diesem Wert werden

die Zähler- und Nenner-Freiheitsgrade multipliziert, welche für die Berechnung der Wahrscheinlichkeit aufgrund des *F*-Wertes (und der Freiheitsgrade) verwendet werden (siehe die Spalte `p[HF]`).

`p[HF]`: Die berechneten *p*-Werte, wobei die Freiheitsgrade für den *F*-Test mit ε_{HF} multipliziert wurden.

`p[HF]<.05`: Ein Indikator, ob signifikant auf 5% Niveau.

Prinzip 5-1: *Verwendung der Greenhouse-Geisser und Huynh-Feld-Anpassungen*

Es gelten folgende Regeln:

1. Falls $\varepsilon_{GH} > .75$, verwende Huynh-Feldt, sonst Greenhouse-Geisser Anpassung.
2. Falls $\varepsilon_{HF} \geq 1$ ist keine Anpassung notwendig. Man verwendet in diesem Fall die Werte in der Tabelle der Varianzanalyse (`$ANOVA`).
3. Bei kleinen Stichproben hat Mauchlys Test eine geringe Teststärke. Daher sollte man schon bei *p*-Werten von $p < .2$ davon ausgehen, dass die Sphärizität verletzt ist.
   Bei grossen Stichproben hat Mauchlys Test eine grosse Teststärke und daher kann ein strikteres Kriterium für die Verletzung der Sphärizität verwendet werden, z.B.: $p < .01$.

5.2 Mittelwertsvergleiche

Falls die Varianzanalyse ein signifikantes Ergebnis für die UV *Itemtyp* ergibt, so ist festzustellen, zwischen welchen Stufen statistisch signifikante Unterschiede vorhanden sind. Dies kann mit Hilfe von Mittelwertsvergleichen geschehen.

R-Lösung: *Post-hoc Einzelvergleiche mit Korrektur der alpha-Fehler-Kumulation*

Die Funktion `pairwise.t.test()` kann zum Vergleich von Mittelwertsunterschieden verwendet werden:

```
PT.obj <- pairwise.t.test(x = Daten$N, g =
Daten$Itemtyp, p.adjust.method = "bonferroni",
paired = T)
```

- Im Argument `x` wird der Funktion die abhängige Variable *N* übergeben.
- Im Argument `g` wird der Funktion die unabhängige Variable *Itemtyp* übergeben.
- Für die Kontrolle der α-Fehler-Kumulation wird die Bonferroni-Prozedur verwendet (Für andere Anpassungsoptionen, siehe Hilfeseite).
- `paired = T` zeigt an, dass es sich um abhängige Stichproben (Wiederholungsfaktor) handelt.

Das Ergebnis wird in der Variable `PT.obj` gespeichert. Hierbei handelt es sich um eine Liste. Der Eintrag `p.value` enthält die berechneten *p*-Werte:

```
>PT.obj$p.value
                Alt         Köder
Köder 7.346533e-04            NA
Neu   3.234605e-07 0.0007274284
```

Die Unterschiede zwischen allen Stufen sind signifikant.

Übung 5-1:

Prüfen Sie, ob die von der Funktion `pairwise.t.test()` ermittelten *p*-Werte korrekt sind, indem Sie 3 *t*-Tests *für abhängige Stichproben* durchführen und die erhaltenen *p*-Werte mit Hilfe der Bonferroni-Methode korrigieren.

Hinweise:

1. Setzen Sie für die Funktion `t.test()` das Argument `var.equal = TRUE`.
2. Die Bonferroni-Korrektur multipliziert die erhaltenen *p*-Werte mit der Anzahl durchgeführter Tests.