

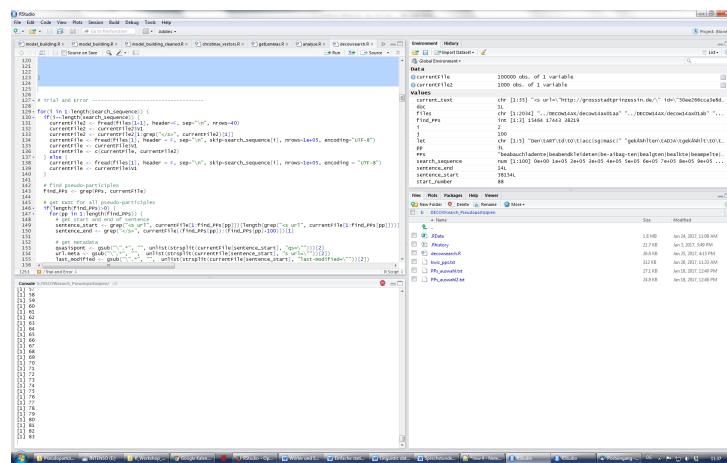
Schnelleinstieg in R

Dieses Tutorial gibt einen knappen¹ und kondensierten Überblick darüber, wie man mit R (R Core Team 2016) Daten auswerten und grundlegende monovariate statistische Tests durchführen kann. Es kann natürlich die einschlägigen Einführungen und insbesondere eigenständiges *learning-by-doing* nicht ersetzen. Auch erklärt es nicht die Hintergründe der statistischen Tests – also das „Was“ – sondern beschränkt sich auf das „Wie“. Es ist dabei jedoch so geschrieben, dass es auch für AnfängerInnen verständlich sein sollte, und eignet sich somit hoffentlich gut als erster Einstieg, wenngleich gegen Ende auch ein paar Themen für Fortgeschrittene angesprochen werden. Zu den ab Abschnitt 2 diskutierten Beispielen gibt es R-Skripts im Ordner R_Einstieg, die den Code und einige zusätzliche Kommentare enthalten.

0. Wie benutze ich R?

Dieser Abschnitt überschneidet sich mit z.T. mit dem, was bereits im allerersten Tutorial („Grundlegendes“) behandelt wurde; hier wird ein etwas ausführlicherer Einstieg in R geboten, den Sie aber, wenn Sie sich bereits mit R und RStudio vertraut gemacht haben, getrost überspringen können.

R gibt es unter www.r-project.org kostenlos zum Download; die Installation ist sehr einfach. Ich empfehle, zusätzlich die (ebenfalls kostenlose) graphische Benutzeroberfläche RStudio zu benutzen (<https://www.rstudio.com/>). Wenn man RStudio startet, sieht man ein viergeteiltes Fenster:



Unten links ist die **Konsole**, also quasi das, was man benutzt, wenn man R ohne RStudio verwendet. Hier kann man ganz einfach Befehle eingeben wie

```
> 1+1  
[1] 2
```

In Blau (nach dem >, das nicht mit eingegeben wird, sondern schon da steht und den Beginn der Befehlszeile markiert) ist der Befehl zu sehen, in Schwarz der Output von R.

¹ Zumindest war das so geplant, als ich mit dem Schreiben begonnen habe...

Oben links befindet sich der Skript-Editor (falls er nicht zu sehen ist, einfach mit File > New File > R Script ein neues Skript starten). Hier lassen sich Skripte, also einfache Textdokumente mit Code, erstellen und vor allem auch speichern – das ist gerade für „Trial and Error“ sehr nützlich. Mit Strg+Eingabetaste überträgt man den Code der Zeile, in der man sich gerade befindet, in die Konsole, wo er dann ausgeführt wird. Man kann auch beliebig viele Zeilen an Code markieren und sie mit Strg+Eingabetaste ausführen, oder mit Klick auf „Run“ am oberen rechten Rand des Editors.

Oben rechts sehen wir, welche **Objekte** sich gerade im sog. Environment befinden. R ist eine objektbasierte Programmiersprache – quasi alles, womit wir in R arbeiten, ist ein „Objekt“. Unten rechts lässt sich z.B. sehen, welche Daten im Arbeitsverzeichnis liegen (Reiter „Files“). Das Arbeitsverzeichnis können Sie mit `setwd` ändern, also z.B. `setwd("C:/Statistikuebungen")`. Der Pfad, den Sie dabei angeben, muss natürlich schon existieren, ansonsten erhalten Sie eine Fehlermeldung. Mit `getwd()` kann man das Arbeitsverzeichnis erfragen. Weiterhin kann man unten rechts auch die Hilfefunktion zu einzelnen Funktionen benutzen (Reiter „Help“) – dieser Reiter wird auch geöffnet, wenn man R um Hilfe bittet, indem man den Namen einer Funktion mit vorangestelltem Fragezeichen eingibt, z.B. `?sqrt`. Außerdem werden unten rechts Grafiken angezeigt, wenn man welche erstellt (Reiter „Plots“).

1. Grundlagenwissen zu R und erste Schritte

1.1 Datentypen in R

Wenn man R verwendet, ist es zunächst wichtig, sich mit den Datentypen in R vertraut zu machen, um häufige Fehler zu vermeiden. Schauen wir uns zunächst die wichtigsten „Bausteine“ an: Zahlen (*numeric*), Characters (*characters*) und Faktoren (*factors*).

1.1.1 Zahlen, Characters und Faktoren

Zahlen sind, wie der Name schon sagt, numerische Einheiten. Mit der Funktion `str`, die die Struktur eines Objekts ausgibt, können wir uns das für eine beliebige Zahl anzeigen lassen:

```
> str(1)
  num 1
```

Das `num` steht für *numeric*.

Characters, also „Zeichen“, sind zumeist Buchstaben, Symbole etc.:

```
str("bla")
chr "bla"
```

In R stehen sie in **Anführungszeichen**. Warum genau, werden wir gleich sehen. Auch Zahlen können in Zeichen überführt werden, indem man sie in Anführungszeichen schreibt oder den Befehl `as.character` verwendet.

```
> str("1")
  chr "1"
> str(as.character(1))
  chr "1"
```

Schließlich gibt es noch **Faktoren**, die bei Anfängern oft für Verwirrung sorgen. Faktoren kann man quasi als Kategorien sehen, die unterschiedliche Ausprägungen (*levels*) haben.

Wenn man beispielsweise „Geschlecht“ als binäre Kategorie operationalisiert, gibt es die Ausprägungen „männlich“ oder „weiblich“. Sowohl Zahlen als auch Zeichen können in Faktoren überführt werden.

```
> as.factor(1)
[1] 1
Levels: 1
> as.factor("bla")
[1] bla
Levels: bla
```

Es ist wichtig, den Unterschied zwischen Characters und Faktoren zu verstehen: Während Erstere einfach „nur“ Zeichenketten sind, sind Faktoren gewissermaßen Namen für Kategorien mit einer endlichen Anzahl an Ausprägungen. Dieser Unterschied wird noch sehr wichtig, wenn wir uns mit den nächstgrößeren Einheiten befassen, zu denen wir die soeben vorgestellten „Bausteine“ kombinieren können: Vektoren, Matrizen und Dataframes.

Exkurs: Objekte benennen

Bis jetzt haben wir einfach nur Datenstrukturen eingegeben und einmalig verwenden – häufig wollen sie wir aber mehrfach verwenden. Das ist möglich, indem wir unseren Daten Namen zuweisen – sie werden dann im Environment gespeichert (also das, was oben rechts in RStudio zu sehen ist). Zum Beispiel können wir sagen, dass die Zahl 2 als „x“ gespeichert werden soll, die Zahl 4 als „y“. Dann können wir mit x und y auch mathematische Operationen wie Multiplikation durchführen:

```
> x <- 2
> y <- 4
> x*y
[1] 8
```

Dass x*y auch tatsächlich das gleiche ist wie 2*4, können wir mit == überprüfen:

```
> x*y==2*4
[1] TRUE
```

Beachten Sie das doppelte Gleichheitszeichen: Das einfache = wird in R als Zuweisungsoperator benutzt. Seine Funktion ist also mit dem oben verwendeten <- vergleichbar:

```
> x = 5
> x
[1] 5
```

Allerdings wird häufig empfohlen, lieber den Pfeil als Zuweisungsoperator zu verwenden. Natürlich kann man auch größere Objekte benennen, wie Vektoren, Matrizen und Dataframes, denen wir uns jetzt zuwenden.

1.1.2 Vektoren, Matrizen und Dataframes

Einzelne Zahlen oder Characters lassen sich zu **Vektoren** kombinieren, also quasi einfachen Ketten. Dies geschieht mit dem Befehl *c()*:

```
c(1, 2, 3, 4)
```

```
[1] 1 2 3 4

> c("bla", "blubb")
[1] "bla"    "blubb"
```

Am konkreten Beispiel kann ein Vektor eine Reihe von Messwerten oder auch eine Reihe von Wörtern enthalten.

Während Vektoren eindimensional sind, kann man sich **Matrizen** und **Dataframes** wie eine Tabelle vorstellen. Ein Vektor mit numerischen Werten lässt sich einfach in eine Matrix überführen, indem wir die Anzahl der Zeilen (*nrow*) und/oder die Anzahl der Spalten (*ncol*) angeben.

```
> matrix(c(1,2,3,4), ncol=2)
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

In den eckigen Klammern, mit denen die Zeilen und Spalten hier nummeriert sind, zeigt sich auch eine wichtige Konvention in R: Erst Zeilen, dann Spalten. [1,] ist die erste Zeile, [1,1] die erste Spalte. [1,1] wäre also zu lesen als: erste Zeile, erste Spalte, [1,2] als: erste Zeile, zweite Spalte.

Matrizen können nur numerische Werte enthalten:

```
matrix(c("a", "b"), c("a", "b"))
Error in matrix(c("a", "b"), c("a", "b")) : non-numeric matrix extent
```

Dataframes hingegen können auch Characters und Faktoren enthalten – BLA und BLUBB sind hier die (optionalen) Spaltennamen:

```
> data.frame(BLA=c("a", "b"), BLUBB=c("a", "b"))
   BLA BLUBB
1   a     a
2   b     b
```

Tatsächlich werden nicht-numerische Werte in Dataframes standardmäßig als Faktoren interpretiert. Das können wir sehen, wenn wir den eben erzeugten data.frame wieder mit dem *str*-Befehl näher untersuchen:

```
> str(data.frame(BLA=c("a", "b"), BLUBB=c("a", "b")))
'data.frame': 2 obs. of 2 variables:
 $ BLA : Factor w/ 2 levels "a","b": 1 2
 $ BLUBB: Factor w/ 2 levels "a","b": 1 2
```

Das gilt es im Hinterkopf zu behalten, wenn man z.B. Tabellen in R einliest (s.u.) – denn hier werden Zeichenstränge ebenfalls als Faktoren behandelt, was gerade bei Korpusdaten oft keinen Sinn macht. Oft empfiehlt es sich daher, mit dem Befehl

```
options(stringsAsFactors = F)
```

die Default-Einstellung zu ändern. Wenn wir jetzt den Befehl von oben wiederholen, sehen wir, dass der Inhalt der Spalten BLA und BLUBB als Characters, nicht als Faktoren, interpretiert wird:

```
> str(data.frame(BLA=c("a", "b"), BLUBB=c("a", "b")))
'data.frame': 2 obs. of 2 variables:
```

```
$ BLA : chr "a" "b"
$ BLUBB: chr "a" "b"
```

Das „Klammer-Prinzip“ [Zeile,Spalte] haben wir oben bereits kennengelernt. Damit können wir auch auf einzelne Zeilen bzw. Spalten Bezug nehmen. Dafür benennen wir unseren Dataframe zuerst – beachten Sie, dass **Kommentare** mit # beginnen: Diese Teile gehören nicht zum Code, sondern kommentieren ihn lediglich. Das ist gerade bei längeren und komplizierteren Skripts nützlich, da man so z.B. Überschriften ins Skript einfügen kann.

```
> mydf <- data.frame(BLA=c("a", "b"), BLUBB=c("c", "d"))
> mydf # wir lassen uns den Dataframe anzeigen
      BLA BLUBB
1     a     c
2     b     d
> mydf[1,] # erste Zeile
      BLA BLUBB
1     a     c
> mydf[,1] # erste Spalte
[1] "a" "b"
> mydf[2,1] # zweite Zeile, erste Spalte
[1] "b"
```

1.1.3 Listen

Ein weiterer Datentyp, der bei Anfängern manchmal für Verwirrung sorgt, sind Listen. Listen kann man sich vorstellen wie einen Datenkatalog – in einer Liste kann man Daten ganz verschiedenen Typs kombinieren, z.B. einen Vektor, einen Dataframe, eine Matrix:

```
> list(c(1,2,3), matrix(c(1,2,3,4, ncol=2)), data.frame(BLA=1, BLUBB=2))
[[1]]
[1] 1 2 3

[[2]]
[,1]
[1,] 1
[2,] 2
[3,] 3
[4,] 4
[5,] 2

[[3]]
      BLA BLUBB
1     1     2
```

Auf Listenelemente kann man mit doppelten Klammern Bezug nehmen. Um das zu illustrieren, benennen wir die Liste zunächst wieder:

```
> mylist <- list(c(1,2,3), matrix(c(1,2,3,4, ncol=2)), data.frame(BLA=1, BLUBB=2))
> mylist[[1]] # erstes Listenelement: der Vektor
[1] 1 2 3
> mylist[[2]] # zweites Listenelement: die Matrix
[,1]
[1,] 1
[2,] 2
[3,] 3
```

```
[4,]    4
[5,]    2
> mylist[[3]][1,] # drittes Listenelement: der df, davon die erste Zeile.
BLA BLUBB
1   1
```

2. Auswertung von Korpus-Konkordanzen und anderen Tabellen

2.1 Einlesen von Daten

Die Datentypen, die wir in der empirischen Sprachwissenschaft untersuchen, sind teils recht unterschiedlich – zu den verbreitetsten gehören aber sicherlich Korpus-Konkordanzen und Tabellen, in denen z.B. Ergebnisse von Experimenten gespeichert sind. Diese Datentypen können relativ einfach in R eingelesen und dort ausgewertet werden. Man muss sich jedoch einiger „Stolperfallen“ bewusst sein:

- Encoding!** Wie bereits an anderer Stelle erwähnt, sollte man bei Daten mit Sonderzeichen immer darauf achten, dass sie richtig eingelesen werden (siehe auch Tutorial „Grundlegendes“).
- Fehlende Daten:** Leere Zellen können u.U. dazu führen, dass R beim Einlesen eine Fehlermeldung generiert. Nehmen wir als Beispiel die Datei *leerzellen.csv* im Ordner *R_Einstieg*. Wenn wir sie in Calc öffnen, sieht sie so aus:

Linker Kontext	Key	Rechter Kontext	Annotation
Das ist ein	Beispiel	für ein unvollständiges Spreadsheet	test
Dieses	Beispiel	zeigt, was passiert, wenn in einem Spreadsheet Zellen fehlen.	
In diesem	Beispiel	ist nämlich in der zweiten Zeile die Spalte "Annotation" leer.	test

Wir sehen, dass in der Spalte „Annotation“ ein Wert fehlt (in der dritten Zeile bzw. der zweiten Zeile nach der Header-Zeile). Hinter den Kulissen handelt es sich bei der Datei um ein tab-separiertes Spreadsheet; in einem Texteditor sieht es so aus (Tabstopps sind hervorgehoben):

Wir sehen, dass in Zeile 3 nur zwei Tabstopps vorhanden sind, in allen anderen drei. Wenn wir die Datei mit folgendem Code in R einlesen wollen:

```
read.table("leerezellen.csv", sep = "\t", quote = "",  
          fileEncoding = "UTF-8", head = T)
```

... dann bekommen wir eine Fehlermeldung: „line 2 did not have 4 elements“. Nach dem, was wir gerade gesehen haben, wissen wir, warum. R erwartet, dass jede Zeile

gleich viele Elemente hat.

Beachten Sie, dass die vierte Spalte in der betroffenen Zeile nicht leer ist, sondern in dieser Spalte schlichtweg nicht existiert – denn wie wir gesehen haben, fehlt das Element, das die Zellen voneinander trennt, also der Tabstop. Anders ist es im Dokument *leerezellen2.csv*, ebenfalls im Ordner *R_Einstieg*. Diese Datei unterscheidet sich nur in einem kleinen Detail von *leerezellen.csv*: In der kritischen Zeile gibt es hier einen Tabstop. Die Zelle ist zwar immer noch leer, aber sie „existiert“ immerhin, denn der Tabstop, der die dritte von der vierten Spalte abgrenzt, ist jetzt da.

Linker_Kontext	Key	Rechter_Kontext	Annotation
1			
2			
3			
4			

Wenn wir diese Datei – mit dem gleichen Code wie oben (nur die 2 in den Dateinamen einfügen) in R einlesen, sind wir erfolgreicher. Die leere Zelle wird automatisch als NA (= Not Available) eingelesen.

Manchmal kann es aber vorkommen, dass wir mit nicht ideal formatierten Daten arbeiten müssen, in denen die Zeilen unterschiedlich viele Elemente haben (das sollte nicht vorkommen und wenn man selbst Daten generiert, sollte man das tunlichst vermeiden, aber bei manchen Konkordanz-Exporten u.ä. hat man das nicht in der Hand). Für solche Fälle gibt es in den *read*-Funktionen von R die Option *fill = TRUE*:

```
read.table("leerezellen.csv", sep = "\t", quote = "",  
          fileEncoding = "UTF-8", head = T, fill = T)
```

Mit dieser Option wird die fehlende Zelle automatisch aufgefüllt. Dabei gilt es jedoch zu beachten, dass immer die letzte Spalte aufgefüllt wird: Sollten aus irgendwelchen Gründen andere Spalten fehlen, dann rutschen die verbleibenden Zellen nach links!

Es ist also wichtig, die Eigenschaften der Datei, die wir einlesen, zu kennen, um Fehler zu vermeiden: Handelt es sich um eine UTF-8- oder eine ASCII-kodierte Datei? Werden Semikola, Kommata oder Tabstopps als Trennzeichen benutzt? Gibt es Texttrenner (meist Anführungszeichen, was die *read*-Funktionen in R defaultmäßig annehmen) oder müssen wir, wie oben geschehen, ein *quote*-Argument setzen? Hat das Spreadsheet, das wir einlesen, eine Kopfzeile (Argument *head*) so wie die Beispieldatei, in der *Linker_Kontext*, *Key*, *Rechter_Kontext* und *Annotation* als Überschriften für die jeweilige Spalte fungieren, oder geht es gleich in der ersten Zeile mit den ersten Werten los?

Wenn wir die Antworten auf diese Fragen kennen, dann können wir die Daten in aller Regel fehlerfrei einlesen und können im Idealfall schon zur eigentlichen Auswertung übergehen. In einigen Fällen kann es allerdings auch nötig sein, zunächst noch kleine Veränderungen vorzunehmen.

2.2 Datenmanipulation

Manchmal müssen wir die Daten zunächst noch manipulieren – damit ist natürlich nicht gemeint, dass wir Daten fälschen oder sie anderweitig in einer Weise verändern, dass sie von den tatsächlich gewonnenen Werten abweichen. Vielmehr ist es z.B. möglich, dass wir Daten

ausschließen möchten oder dass wir mehrere Spreadsheets, die wir getrennt voneinander bearbeitet haben, zusammenfügen wollen. Nehmen wir beispielsweise an, wir untersuchen die nonagentive Konstruktion *gehören* + Partizip (*das gehört mal untersucht*): Hier könnte es sein, dass wir zunächst eine Konkordanz für *gehören* + Partizip bearbeiten und anschließend eine für die umgekehrte Reihenfolge, weil das Verb ja in Nebensätzen an letzter Stelle steht (*weil das mal untersucht gehört*). In diesem Fall hätten wir zwei Konkordanzen, die die Ergebnisse zweier unterschiedlicher Suchanfragen erfassen, aber ansonsten im Idealfall genau gleich aufgebaut sind.

Auch kann es vorkommen, dass wir z.B. eine Annotation, die wir an Types vorgenommen haben, weil sie für alle Tokens desselben Types gleich ist, auf Tokens übertragen wollen. Ein Beispiel für Letzteres wäre eine Konkordanz, in der wir Substantive, die von Verben abgeleitet sind (z.B. *Ritt* von *reiten*, *Schnitt* von *schneiden*) auf ihr Basisverb hin annotieren: Das Basisverb von *Ritt* ist immer *reiten*, egal in welchem Kontext die Nominalisierung vorkommt. Deshalb kann es zeitsparend sein, zunächst eine Liste der Types zu exportieren, diese zu annotieren und dann die annotierte Typeliste wieder mit der ursprünglichen Konkordanz, die alle Tokens enthält, zusammenzuführen.

Auf die drei genannten Beispiele der Datenmanipulation – Daten filtern, mehrere Spreadsheets desselben Typs zusammenführen (z.B. [*gehören* + PP] plus [PP + *gehören*]) sowie mehrere Spreadsheets unterschiedlichen Typs (z.B. Token-Konkordanz und Type-Liste) – wollen wir im Folgenden genauer eingehen. (Wenn Sie sich sicher sind, diese Abschnitte nicht zu brauchen, weil Sie ohnehin eine direkt einlesbare Tabelle in einem Tabellenkalkulationsprogramm kreieren, können Sie auch gleich zu Abschnitt 2.3 springen.)

2.2.1 Daten filtern

Sehr häufig kommt es vor, dass sich in Korpus-Konkordanzen Treffer finden, die zwar mit unserer Suchanfrage übereinstimmen, aber eben doch nicht dem entsprechen, was wir eigentlich finden wollen – sogenannte **Fehltreffer**. Ein einfaches Beispiel: Wenn wir Nominalisierungen mit dem Suffix *-ung* untersuchen, dann finden wir fast zwangsläufig auch Wörter wie *Ursprung*, die zwar auch mit der Zechenfolge *u-n-g* enden, aber nicht zur *ung*-Nominalisierung gehören (sonst würde es ja *die Ursprungung* heißen, nicht *der Ursprung*). Solche Fehltreffer müssen manuell ausgeschlossen werden. Eine Möglichkeit ist, sie direkt in Excel oder Calc zu löschen. Eine andere Möglichkeit besteht darin, in der Konkordanz eine eigene Spalte anzulegen, in der Fehltreffer als solche ausgezeichnet werden, um sie später ausschließen zu können. Das hat den Nachteil, dass man es im schlimmsten Fall vergisst, aber es hat mehrere große Vorteile. Erstens: Wenn man der *best-practice*-Strategie folgt, die Daten einschließlich aller Annotationen im Sinne größtmöglicher Replizierbarkeit öffentlich zugänglich zu machen, dann erhöht man somit zusätzlich die Transparenz, weil sofort erkennbar wird, welche Treffer man ausgeschlossen hat – somit kann jede/r selbst beurteilen, ob der Ausschluss im jeweiligen Einzelfall berechtigt war oder nicht, denn längst nicht jeder Fall ist so klar wie unser Beispiel *Ursprung*, das eindeutig nicht zur *ung*-Nominalisierung gehört. Das führt uns unmittelbar zum zweiten Punkt: Gerade weil wir es nicht immer mit klaren, eindeutig abgrenzbaren Kategorien zu tun haben, kann es manchmal vorkommen, dass wir unsere Ausschlusskriterien im Nachhinein noch anpassen möchten. Wenn wir, um beim Beispiel der *ung*-Nominalisierung zu bleiben, zunächst entscheiden, eindeutig lexikalisierte Treffer wie *Kleidung* auszuschließen, dann aber erkennen, dass sich keine klare Grenze zwischen lexikalisierten und nicht-lexikalisierten Derivaten ziehen lässt, dann wäre es fatal, wenn wir alle Belege, die wir jetzt doch behalten möchten, schon gelöscht hätten.

Der Datensatz *ung_bsp.csv* im Ordner *R_Einstieg* greift das Beispiel der *ung*-Nominalisierung mit einer Mini-Konkordanz auf, die bereits auf Fehltreffer annotiert ist (Fig. 1 zeigt die gesamte Konkordanz); das Skript *datenmanipulation.R* zeigt mehrere Wege auf, wie man die

in der Spalte *Anmerkungen* als *Fehltreffer* annotierten Belege entfernen kann. Im Folgenden diskutieren wir diese unterschiedlichen Wege kurz.

	A	B	C	D	E	F	G	H	I
1	Number	text_author							
2	133373037 Christian von Wolff	•ematischen Wissenschaften	1710 / noch dieses jenes beweigt Anmerkung				Right_Context		
3	133476551 Christian von Wolff	•ematischen Wissenschaften	1710 D den Brenn. Punct lich ein Anmerkung				23. Weil der Beweß einerley bleibt / wenn man f Anmerkung		
4	133442228 Christian von Wolff	•ematischen Wissenschaften	1710 der Sonne untergehet / inp Anmerkung				472. Man erweicht aber folche Observacionen / da Anmerkung		
5	73383516 Christian Ludwig Liscow	cher und Ernsthafer Schriften	1739 man es in den Anmerkunk Anmerkungen				173. Der Aufgang eines Sternes mit der Sonne un Anmerkung		
6	73235977 Christian Ludwig Liscow	cher und Ernsthafer Schriften	1739 lo haben lich auch Leute g Anmerkungen				mufften auch etwas enträglicher feyn , als die Kappe Anmerkung		
7	14038170 Johann Jacob Bodmer	•d ander geistvollen Schriften	1741 Der Autor ist dann beifßen Anmerkungen				über die Geschichte von der Zerföhrung der Stadt. Anmerkung		
8	26750640 Franz Ludwig von Cancrin	•n, und in den Saalfeldischen	1767 in den Schmelzöfen von de Anmerkung				zu erklären und zu bekräftigen . Wir wollen ihn feine Anmerkung		
9	68642847 Johann Heinrich Lambert	•l mathematischen Erkenntniß	1771 in jedem Falle das Produk Anmerkung				. Man glaubt , die thaliterischen Schiefern waren für Anmerkung		
10	79416696 Johann Tobias Mayer	•hrbegriff der höhern Analysis	1818 122. IV.) angeführten Int. Anmerkung				zum Befürder der Berechnung der Wahrscheinlichkeit Anmerkung		
11	26825472 Franz Ludwig von Cancrin	•n, und in den Saalfeldischen	1767 lehr großen Vorteil machen Anmerkung				1. Bekanntlich kann jede Potenz eines Sinus oder Anmerkung		
12	72364140 Henricus Lettus	ndischen Chronik Andre Theil	1753 einem hohen Alter der Ruh Anmerkungen				des 44. S. im 8. Stük. Die 2. Anmerkung . Man treib Anmerkung		
13	51234763 Albrecht von Haller	tie des menschlichen Körpers	1759 unterhalb derselben heraus Anmerkungen				über D. Laurent. Müllers leptontionalische Hiltoni Anmerkung		
14	9891872 Georg Beseler	• für die Preußischen Staaten	1851 dem Ausdruck , verführen Deutung				über des Walaeus Briefe , S. 621. fol. Eben das fah Anmerkung		
15	13527539 Johann Friedrich Blumenbach	•fangsprünze der Physiologie	1795 eigentliche Verbindung aus Ursprung				delfeben abflichlich dem Richter überlassen , we Deutung		
16	58628799 Christian Hofmann von Hoffmannswaldau	•ßher ungedruckter Gedichte	1703 fürcke SO fern der heide Ursprung				der Nerven. Monn. (Fil.) observations on the stritu Ursprung		Fehltreffer
17	52021929 Albrecht von Haller	tie des menschlichen Körpers	1772 von der Seele nicht hervor Ursprung				/ ich / der lorbeern keine gränzte / Da ihre tapferke Ursprung		Fehltreffer
18	10493085 Otto von Bismarck	Gedanken und Erinnerungen	1898 daß wir ihm nicht feindlic Ursprung				diefer Entelechia zu erklären . Kartelus schreibt di Ursprung		Fehltreffer
19	91514992 Peter Pomet	•rialist und Specerey-Händler	1717 Raths und oberften Leib-M-Ursprung				für gefährlich halten (er thut es ja auch) , und da Ursprung		Fehltreffer
20	93798770 Friedrich August Quesnstedt	Handbuch der Mineralogie	1855 sind ein Oel und Harz der Ursprungs				gewiß entdecket werden , bevoraus , welche einen Ursprung		Fehltreffer
21	41442340 Louise von François	Die letzte Reckenburgerin	1871 Auch das Haus , in welche Ursprungs				it es viel feltern . Doch findet man z. B. mitten in Ursprung		Fehltreffer
22	22						. Ein weiland Herzog hatte es für feinen Leibbader Ursprung		Fehltreffer

Fig. 1: Mini-Konkordanz mit ung-Nominalisierungen.

Es geht in diesen Beispielen darum, eine Teilmenge (*subset*) der Daten zu erhalten; deshalb nennt man die Operationen, denen wir uns nun zuwenden, auch *subsetting*. Man kann zum Beispiel einzelne Zeilen oder Spalten auswählen, indem man die entsprechenden Zeilen- oder Spaltennummern angibt. Dabei gilt, dass wir zum Subsetsen eckige Klammern benutzen, die dem Namen des Dataframes folgen, wobei die Zeilen *vor* dem Komma stehen, die Spalten danach, also

`dataframe[zeilen, Spalten]`

Das bedeutet: Wenn wir die erste Spalte des Dataframe *ung* ausgeben wollen, benutzen wir

`ung[,1]`

Wenn wir die erste und zweite Zeile ausgeben wollen, benutzen wir

`ung[1:2,]`

und so weiter.

Statt Zahlen können wir aber auch eine Funktion, z.B. ein *which*-Statement, einsetzen. Wenn wir z.B. eingeben `which(ung$Anmerkungen=="Fehltreffer")`, dann gibt R die Zeilen aus, in denen die Spalte *Anmerkungen* den Wert „Fehltreffer“ hat:

```
> which(ung$Anmerkungen=="Fehltreffer")
[1] 14 15 16 17 18 19 20
```

Oder umgekehrt, wenn wir wissen wollen, in welchen Zeilen die Spalte *Anmerkungen* nicht den Wert „Fehltreffer“ hat:

```
> which(ung$Anmerkungen!="Fehltreffer")
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13
```

Dieses *which*-Statement können wir zum Subsetsen verwenden:

```
ung2 <- ung[which(ung$Anmerkungen!="Fehltreffer"), ]
```

ist das gleiche wie `ung[c(1,2,3,4,5,6,7,8,9,10,11,12,13),]`.

Stolperfalle: Komma nicht vergessen!

Ein häufiger (Anfänger-)Fehler besteht darin, dass man beim Subsetten das Komma vergisst:

```
ung[which(ung$Anmerkungen!="Fehltreffer")]
Error in ` [.data.frame` (ung, which(ung$Anmerkungen !=
"Fehltreffer")) :
  undefined columns selected
```

Auch sollte man darauf achten, dass man das Komma an der richtigen Stelle setzt, um nicht versehentlich Spalten statt Zeilen auszuwählen (oder umgekehrt).

Eine alternative Variante besteht darin, die Daten mit einem Minuszeichen auszuschließen. Davon rate ich allerdings ab, weil es in manchen Fällen zu Problemen führen kann. Schauen wir uns noch einmal das Beispiel an:

```
ung2a <- ung2a[-which(ung2$Anmerkungen=="Fehltreffer"), ]
```

Dieser Code funktioniert einwandfrei und generiert das gleiche Ergebnis wie der obige. Problematisch wird es allerdings, wenn die Variablenausprägung, die uns als Ausschlusskriterium dient (hier: dass die Spalte *Anmerkungen* den Wert „Fehltreffer“ hat) in den Daten gar nicht existiert. Das ist z.B. im eben kreierten Dataframe *ung2a* der Fall, denn alle Zeilen, in denen das der Fall ist, haben wir ja da bereits ausgeschlossen. Sehen wir uns an, was passiert, wenn wir den Code darauf anwenden:

```
> ung2b <- ung2a[-which(ung2a$Anmerkungen=="Fehltreffer"), ]
> ung2b
[1] Number      text_author   text_title   text_year
Left_Context Keyword
[7] Right_Context Lemma      Anmerkungen
<0 rows> (or 0-length row.names)
```

Wir bekommen, wie Sie sehen, einen leeren Dataframe. Wie kommt das? Es kommt daher, dass `which(ung2a$Anmerkungen=="Fehltreffer")` eine leere Menge ist und somit 0. Der obige Code ist somit äquivalent zu `ung2a[-0,]`. Minus null ist das gleiche wie Null, und deshalb ist der Code auch äquivalent zu `ung2a[0,]`. Dieser Code sagt R, dass es die nullte Zeile von *ung2a* ausgeben soll. Und genau das tut es dann eben auch... Noch etwas effizienter geht das alles mit der Funktion *subset* bzw. der konzeptionell fast äquivalenten Funktion *filter* aus dem Paket *dplyr*:²

² In R gibt es sehr oft mehrere Wege zum gleichen Ziel. Wer sich über die Unterschiede zwischen *subset* und *filter* informieren möchte, kann das z.B. auf der Plattform Stackoverflow tun, die ohnehin eine Fundgrube ist, wenn beim Programmieren Fragen auftreten:
<https://stackoverflow.com/questions/39882463/difference-between-subset-and-filter-from-dplyr> (abgerufen am 15. Oktober 2017)

```

ung3 <- subset(ung, Anmerkungen!="Fehltreffer")
library(dplyr)
ung3 <- filter(ung, Anmerkungen!="Fehltreffer")

```

Diese Funktion ist vor allem dann hilfreich, wenn wir mit relativ komplexen Argumenten arbeiten wollen und viele verschiedene Bedingungen gleichzeitig angeben müssen, nach denen wir filtern.

2.2.2 Tabellen gleichen Typs kombinieren

Die Dateien *gehoert_verboten.csv* und *verboten_gehoert.csv* im Ordner *R_Einstieg* enthalten die KWIC-Konkordanzen, die man als Ergebnis erhält, wenn man im ZEIT-Korpus des DWDS nach der (genauen) Wortfolge *gehört verboten* sowie *verboten gehört* sucht. Diese Dateien können wir in R einlesen und ganz einfach mit der Funktion *rbind* (das *r* darin steht für *rows*) verbinden:

```

# die beiden Dataframes einlesen
gv <- read.csv("gehoert_verboten.csv")
vg <- read.csv("verboten_gehoert.csv")

# beide kombinieren
beide <- rbind(gv, vg)

```

Manchmal kann es hilfreich sein, noch eine Spalte hinzuzufügen, die angibt, aus welchem Dataframe die jeweilige Zeile ursprünglich stammt; dafür eignet sich die Funktion *mutate* aus dem Paket *dplyr* hervorragend, die den Daten einfach eine Spalte hinzufügt – z.B. gibt *mutate(gv, DF = "gv")* den Dataframe *gv* mit einer zusätzlichen Spalte aus, der in allen Zeilen den Wert *gv* hat. Kombiniert mit dem obigen Code erhalten wir also:

```

beide <- rbind(mutate(gv, DF="gv"),
                 mutate(vg, DF="vg"))

```

2.2.3 Tabellen unterschiedlichen Typs kombinieren

Etwas komplizierter wird es, wenn wir Tabellen kombinieren möchten, die wir nicht einfach „aneinanderkleben“ bzw. zusammenbinden können, wie wir es oben getan haben. Der Dataframe *farben.csv* im Ordner *R_Einstieg* enthält eine Mini-Konkordanz mit einer kleinen Auswahl an Belegen aus dem DWDS-Kernkorpus, die man erhält, wenn man nach Farbadjektiven sucht.

Ich habe die einzelnen Types in eine eigene Tabelle exportiert und mit einer Lemmaannotation versehen. Falls Sie wissen wollen, wie das geht: Wenn man mit

```
farben <- read.csv("farben.csv")
```

die Datei eingelesen hat, kann man sich z.B. mit *unique(farben\$Hit)* die unterschiedlichen Variablenausprägungen, die in der Spalte *Hit* vorkommen, ausgeben lassen – mit anderen Worten also: die Types. (Genauer: Die Wortform-Types, denn *grüner* und *grünes* zählen hier

natürlich als unterschiedliche Types, weil es schlichtweg um unterschiedliche Zeichenfolgen geht.)

Lesen wir zunächst den Lemma-Dataframe ein und schauen uns dann an, wie die beiden Dataframes aussehen. Dafür können wir die Funktion `str()` verwenden, die uns die Struktur eines Dataframes verrät.

```
> farben_lemmas <- read.delim("farben_lemmas.csv")
> str(farben)
'data.frame': 50 obs. of 7 variables:
 $ No.      : int 1 2 3 4 5 6 7 8 9 10 ...
 $ Date     : Factor w/ 11 levels "1900-01-02","1900-01-03",...: 1 1
2 2 3 4 4 5 6 7 ...
 $ Genre    : Factor w/ 4 levels "Belletristik",...: 4 4 4 4 4 2 4 4
3 4 ...
 $ Bibl     : Factor w/ 40 levels "Brief von Wilhelm Busch an
Johanna Keßler vom 18.12.1900. In: ders., Gesammelte Werke, Berlin:
Directmedia Publ. 2002 [1900]", ...| __truncated__,...: 22 38 23 24 2 33 2
2 37 34 ...
 $ ContextBefore: Factor w/ 50 levels "- Ohrringe mit",...: 23 11 30 9
38 16 3 31 21 20 ...
 $ Hit      : Factor w/ 14 levels "blau","blaue",...: 7 5 11 1 12 12
9 12 4 4 ...
 $ ContextAfter : Factor w/ 50 levels ", hochgewölbten Himmel
aufzusehen!",...: 36 22 2 25 45 11 21 32 31 15 ...
> str(farben_lemmas)
'data.frame': 14 obs. of 2 variables:
 $ Token   : Factor w/ 14 levels "blau","blaue",...: 7 5 11 1 12 9 4 2 10
13 ...
 $ Lemma   : Factor w/ 4 levels "blau","gelb",...: 2 2 3 1 3 3 1 1 3 3 ...
```

Wir sehen: `farben` hat insgesamt 7 Spalten, `farben_lemmas` nur 2. Um die Sache noch etwas komplizierter zu machen, habe ich die Spaltennamen in `farben_lemmas` auch so vergeben, dass es keine Überschneidung zwischen den beiden Dataframes gibt. In Fig. 2 und Fig. 3 sehen wir die beiden Dataframes im Überblick, so wie wir sie in R sehen, wenn wir die `View`-Funktion in R verwenden: `View(farben)` bzw. `View(farben_lemmas)`.

	No.	Date	Genre	Bibl	ContextBefore	Hit	ContextAfter
1	1	1900-01-02	Zeitung	Frankfurter Zeitung (Abend-Ausgabe), 02.01.1900	Die Vereinigten Staaten machen auf den Philippinen di...	gelben	Politiker sich dachten; Spanien wird immer kraftloser, i...
2	2	1900-01-02	Zeitung	National-Zeitung (Abend-Ausgabe), 02.01.1900	Auf den Zinnen der alten Hohenzollernburg wehte die	gelbe	Kaisermane stände neben der purpurnen Königsflagge un...
3	3	1900-01-03	Zeitung	Frankfurter Zeitung und Handelsblatt (Abend), 03.01.1...	Ganz Amerika weiß beispielweise heute schon, daß ge...	grünem	, mit Silberfäden durchwirkten Satin trug und daß ein ...
4	4	1900-01-03	Zeitung	Freisinnige Zeitung, 03.01.1900	An die Villa schließt sich ein großer Palmengarten mit ...	blau	leuchtenden Gardasee genießt.
5	5	1900-02-05	Zeitung	Die Fackel [Elektronische Ressource], 2002 [1900]	Nach jahrelangen Bemühungen, nachdem alle Facultat...	grünen	Tische ein Reformwerk geschaffen worden, das keinerl...
6	6	1900-02-25	Gebrauchsliteratur	Kerr, Alfred: Kein Raum für Brunn In: ders., Mein Berli...	Das starkgeistige Fräulein Dumont vom Deutschen The...	grünen	Blattgewächsen stand, im weißen Kleid (hinter ihr sah ...
7	7	1900-02-25	Zeitung	Die Fackel [Elektronische Ressource], 2002 [1900]	Aber	grün	Ist der Mann darum keineswegs.
8	8	1900-03-31	Zeitung	Die Fackel [Elektronische Ressource], 2002 [1900]	Geschlechter werden und vergehen, hoffnungsvoll ko...	grünen	Nebel unklarer Volksstimmungen, Wünsche, Herrschbe...
9	9	1900-05-02	Wissenschaft	Mitteilungen der deutschen Gesellschaft für Natur- un...	Die	blauen	Mongolenflecke der Neugeborenen.
10	10	1900-09-12	Zeitung	Kölnerische Zeitung (Abend), 12.09.1900	Der heutige zweite Manövertag brachte den Zuschauer...	blauen	Corps endete, werden die packenden Gefechtsmoment...
11	11	1900-09-12	Zeitung	Kölnerische Zeitung (Abend), 12.09.1900	Von dem	blauen	Corps (Garde) waren die Garde-Cavallerie-Division in ...
12	12	1900-09-12	Zeitung	Kölnerische Zeitung (Abend), 12.09.1900	Nach längerem Gefechte bei Karlshof und Frankenhorst,...	blau) wurde zurückgeworfen und gab ihre vorgeschobenen...
13	13	1900-09-12	Zeitung	Kölnerische Zeitung (Abend), 12.09.1900	Am Nachmittag gegen 4 Uhr nahm das	blaue	Corps (Garde)
14	14	1900-11-27	Zeitung	Kölnerische Zeitung (Abend), 27.11.1900	"Auf diesem Berufe lag früher der schöne	blaue	Duft ferner Berge."

Fig. 2: Struktur des Dataframes „farben“

	Hit	Lemma
1	gelben	gelb
2	gelbe	gelb
3	grünem	grün
4	blau	blau
5	grünen	grün
6	grün	grün
7	blauen	blau
8	blaue	blau
9	grüne	grün
10	grünes	grün
11	gelbem	gelb
12	violett	violett
13	blauem	blau
14	gelber	gelb

Showing 1 to 14 of 14 entries

Fig. 3: Struktur des Dataframes „farben_lemmas“.

Wie können wir diese beiden Dataframes nun so zusammenführen, dass jeder Beleg in der „Hit“-Spalte des Dataframes *farben* die entsprechende Lemma-Annotation aus dem Dataframe *farben_lemmas* bekommt? Eine Möglichkeit wäre, eine Funktion zu schreiben, die jedem Element in der „Hit“-Spalte das entsprechende Lemma zuweist. Einfacher geht es jedoch mit der Funktion *merge()*. Wie der Name schon sagt, erlaubt es diese Funktionen, zwei Dataframes miteinander zu vereinen, sie also zu mergen. Als Argumente nimmt *merge()* zunächst die Namen der beiden Dataframes. Wenn wir einfach nur angeben

```
merge(farben, farben_lemmas)
```

scheitern wir aber zunächst, denn wenn wir nur diese Argumente angeben, dann sucht R nach Spaltennamen, die in beiden Dataframes übereinstimmen – und findet nichts. Es gibt nun zwei Möglichkeiten, damit umzugehen. Die erste, einfachere besteht darin, den Spaltennamen in *farben_lemmas* einfach von „Tokens“ in „Hit“ umzubenennen, sodass diejenige Spalte, die die Keywords enthält und auf die hin gemerget werden soll, in beiden Dataframes den gleichen Namen hat:

```
colnames(farben_lemmas)[which(colnames(farben_lemmas)=="Token")] <-  
"Hit"  
merge(farben, farben_lemmas)
```

Alternativ können wir mit den Argumenten *by.x* und *by.y* zwei verschiedene Spaltennamen angeben. *x* in *by.x* heißt einfach, dass es sich auf das erste der beiden in der Funktion angegebenen Argumente bezieht (also den ersten Dataframe), und entsprechend heißt das *y* in *by.y* einfach, dass es hier um den Spaltennamen im zweiten angegebenen Dataframe geht. Im folgenden Code lesen wir zuerst die Datei noch einmal ein, damit wir wieder den ursprünglichen Spaltennamen haben:

```
farben_lemmas <- read.delim("farben_lemmas.csv")  
merge(farben, farben_lemmas, by.x = "Hit", by.y = "Token")
```

2.3. Auswertung von Daten

Nach diesen etwas langwierigen, aber in vielen Fällen wichtigen vorbereitenden Schritten kommen wir nun zur eigentlichen Datenauswertung. Natürlich können wir hier nur die wichtigsten Funktionen diskutieren. Zunächst sehen wir uns an, wie man einfache

tabellarische Auswertungen vornimmt. Dabei lernen wir auch, wie man mit Textelementen in R umgeht. Anschließend befassen wir uns mit der Visualisierung von Ergebnissen in Plots.

2.3.1 Einfache tabellarische Auswertung

Als Beispiel benutzen wir einen Datensatz, der je 5000 Belege für die Suchanfrage "`$p=/ADJA/g $l=/Frau/g`" und "`$p=/ADJA/g $l=/Mann/g`" umfasst, es geht also um die Lemmata *Frau* und *Mann*, wenn sie unmittelbar einem attributiven Adjektiv folgen. Da die beiden Konkordanzen in zwei unterschiedlichen Dateien gespeichert sind, können wir dabei auch gleich noch einmal *rbind* von oben üben.

Zu den Daten:

- `adja_mann.csv` enthält die Ergebnisse für Adjektiv + *Mann*,
- `adja_frau.csv` enthält die Ergebnisse für Adjektiv + *Frau*,
- `mann_frau_lemma.csv` enthält den von mir manuell nachkorrigierten Output des TreeTagger³, mit dessen Hilfe wir die Daten der beiden Konkordanzen mit Lemmainformationen versehen können.

Zunächst lesen wir die Daten ein:

```
frau <- read.csv("adja_frau.csv")
mann <- read.csv("adja_mann.csv")
lemmas <- read.delim("mann_frau_lemma.csv")
```

Als nächstes führen wir die Dataframes zusammen, indem wir zunächst *rbind* verwenden, um die beiden Konkordanzen quasi aneinanderzuhängen; dabei ergänzen wir wieder eine Spalte, aus der hervorgeht, aus welchem der beiden ursprünglichen Dataframes die jeweiligen Daten stammen:

```
mannfrau <- rbind(mutate(frau, df = "Frau"),
                     mutate(mann, df = "Mann"))
```

Als nächstes ergänzen wir das Tagging aus dem *lemmas*-Dataframe:

```
mannfrau <- merge(mannfrau, lemmas, by = "Hit", all.x = T)
```

Das Argument `by = "Hit"` könnten wir theoretisch weglassen, da R automatisch auf diejenige Spalte hin merkt, deren Name in beiden Dataframes vorkommt (und das ist hier nur bei *Hit* der Fall). Das Argument `all.x = T` ist jedoch wichtig, da es sicherstellt, dass keine Belege unter den Tisch fallen: Wäre es FALSE, dann würden alle Belege weggelassen, die im *lemmas*-Dataframe nicht vorkommen (das sind nicht viele, aber durchaus ein paar, die der TreeTagger aus unterschiedlichen Gründen nicht verarbeiten konnte).

Wir können uns nun an einer ersten, sehr einfachen Tabelle probieren, indem wir z.B. überprüfen, wie sich die Daten, die wir gewonnen haben, auf die einzelnen Textsorten des DWDS (in der Spalte „Genre“) verteilen:

```
> table(mannfrau$Genre)
```

³ <http://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/>

Das allein ist natürlich noch wenig aussagekräftig, da wir nicht wissen, wie sich die Gesamtfrequenzen zusammensetzen. Diese können wir direkt aus dem DWDS abfragen (siehe R-Skript *datenauswertung.R*, ebenfalls im Ordner *R_Einstieg*). So können wir die relativen Frequenzen berechnen und z.B. normalisierte Frequenzen pro 10.000 Wörter ausgeben. Diese können wir mit Hilfe der *barplot*-Funktion auch wie in Fig. 4 visualisieren. (Das dient aber nur der Veranschaulichung, und wir dürfen den enorm hohen Balken im gesprochenen Genre nicht überinterpretieren: Im Gegensatz zu den anderen Textsorten ist dieses Genre nämlich extremst unterrepräsentiert, deshalb ist zu erwarten, dass nicht nur die absolute, sondern auch die relative Frequenz hier höher ausfällt.)

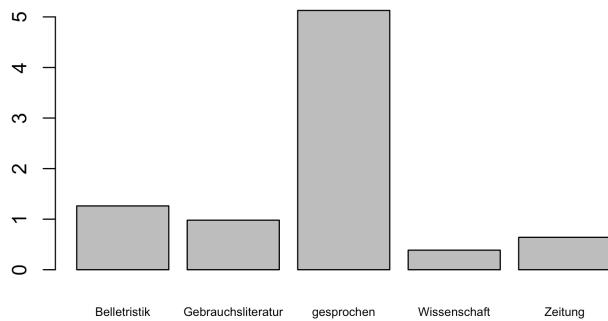


Fig. 4: Frequenzen von *ADJ+Mann* oder *ADJ+Frau* pro 10000 Wörter im DWDS-Kernkorpus.

Lässt man die gesprochenen Daten außer Acht und betrachtet nur die Textsorten, auf die hin das Korpus einigermaßen ausgewogen ist, so zeigen sich zwar keine wirklich nennenswerten Frequenzunterschiede, doch sofern die Rangfolge, die sich zeigt, halbwegs aussagekräftig ist, lässt sie sich doch recht gut erklären – in belletristischen Texten, wo *ADJ+Mann/Frau* am häufigsten vorkommt, ist zu erwarten, dass gerade bei der ersten Nennung einer Figur *Mann* oder *Frau* vorkommt und dass die Romanfigur durch attributive Adjektive näher beschrieben wird; in wissenschaftlichen Texten hingegen ist eine solche Bezugnahme selten nötig. Wir können auch komplexere Tabellen erstellen: Wenn wir z.B. die Verteilung auf Textsorten nach Geschlecht getrennt auswerten wollen, können wir eine entsprechende Kreuztabelle erstellen:

```
> table(mannfrau$df, mannfrau$Genre)
```

Mit dem Paket *dplyr* gibt es übrigens auch eine alternative Methode, zu diesem Ergebnis zu kommen:

```
mannfrau %>% select(Lemma, df) %>% table
```

Die Pipes (%>%) reichen quasi den Output des vorherigen Elements an die nächste Funktion weiter.⁴ Das, was links von der Pipe steht (man spricht hier von *left-hand side* oder kurz LHS), wird als erstes Argument der Funktion auf der rechten Seite (*right-hand side*, RHS)

⁴ Die Pipes gehören zum Paket *magrittr*, das automatisch geladen wird, sobald man *dplyr* lädt.

genommen. `2 %>% sqrt()` ist also z.B. äquivalent zu `sqrt(2)`. `select()`, das oben nach der ersten Pipe steht, wählt einzelne Spalten eines Dataframes aus, generiert also ein Subset:

```
> select(mannfrau, Lemma, df)
      Lemma    df
1      krank Mann
2      n-jährig Mann
3      n-jährig Frau
4      n-jährig Frau
5      n-jährig Frau
6      n-jährig Mann
7      n-jährig Frau
8      n-jährig Frau
9      n-jährig Frau
10     n-jährig Mann
```

Der obige Code `mannfrau %>% select(Lemma, df) %>% table` ist also äquivalent zur etwas verschachtelteren Variante `table(select(mannfrau, Lemma, df))`. Während in diesem Fall auch die verschachtelte Variante noch halbwegs gut lesbar ist, kann das „Piping“ in vielen Fällen die Syntax komplexer Datenanalysen ein wenig entschlacken und damit den Code auch besser lesbar machen. (Siehe auch **Infobox: Warum gut lesbarer Code wichtig ist** weiter unten.)

Das Schöne am „Piping“ ist, dass wir nach und nach verschiedene Subsets ausprobieren und daraus dann Tabellen oder auch Plots generieren können. So können wir zum Beispiel die obige Piping-Sequenz einfach fortsetzen, um die Tabelle nach der Häufigkeit zu sortieren, mit der ein Adjektiv-Lemma mit *Frau* oder *Mann* kollokiert.

Auch hier gibt es wie immer mehrere Möglichkeiten, ich zeige nur eine. Zunächst gilt es, wie so oft, eine Hürde zu überwinden. Der Output der `table()`-Funktion ist eine Matrix. Wie wir oben gesehen haben, nehmen Matrizen nur numerische Werte. Wenn die Tabelle ausgegeben wird, sehen wir zwar die Lemmas...

	df	
Lemma	Frau	Mann
abendländisch	1	0
abergläubisch	1	0
abgebildet	1	0
abgebraucht	0	1
abgeführt	1	0
abgehetzt	1	1
abgerissen	0	1
abgestorben	1	0
abgewandt	0	1
abhängig	2	0

... aber das ist deshalb so, weil sie als Zeilennamen fungieren (genau wie z.B. im *mannfrau*-Dataframe *Lemma* als Spaltenname fungiert und nicht zum „eigentlichen“ Dataframe gehört). Das sieht man auch, wenn man die Struktur der Tabelle abfragt:

```
> mannfrau %>% select(Lemma, df) %>% table %>% str
'table' int [1:1857, 1:2] 1 1 1 0 1 1 0 1 0 2 ...
- attr(*, "dimnames")=List of 2
..$ Lemma: chr [1:1857] "abendländisch" "abergläubisch" "abgebildet" "abgebraucht" ...
..$ df   : chr [1:2] "Frau" "Mann"
```

Die Lemma-Spalte ist lediglich ein „Attribut“. Wenn wir das Ganze nun mit der Funktion `as.data.frame.matrix()` in einen Dataframe überführen⁵, um die auf Matrizen nicht anwendbare nützliche Funktion `arrange` aus dem `dplyr`-Paket verwenden zu können, dann bleiben die Zeilennamen weiterhin als Zeilennamen erhalten:

```
> mannfrau %>% select(Lemma, df) %>% table %>%
+     as.data.frame.matrix
      Frau Mann
abendländisch      1   0
abergläubisch      1   0
abgebildet         1   0
abgebraucht        0   1
abgeführt          1   0
abgehetzt          1   1
aberissen           0   1
abgestorben         1   0
abgewandt          0   1
abhängig            2   0
abstinent           1   0
abwägend            0   1
```

Wenn wir nun aber gleich `arrange` anwenden, haben wir genau deshalb ein Problem:

```
> mannfrau %>% select(Lemma, df) %>% table %>%
+     as.data.frame.matrix %>% arrange(Frau)
      Frau Mann
1      0   1
2      0   1
3      0   1
4      0   1
5      0   1
6      0   1
7      0   1
8      0   1
9      0   1
10     0   1
11     0   1
```

Die `arrange`-Funktion ignoriert die Zeilennamen, sodass wir nun nicht wissen, zu welchem Lemma die angezeigten Werte gehören. Praktischerweise gibt es im Paket `tibble` eine Funktion `rownames_to_column`, auf die wir zurückgreifen können; alternativ können wir die Zeilennamen manuell als eigene Spalte hinzufügen:

Variante 1

```
mannfrau %>% select(Lemma, df) %>% table %>% as.data.frame.matrix %>%
  tibble::rownames_to_column()
```

Variante 2

```
mannfrau %>% select(Lemma, df) %>% table %>% as.data.frame.matrix %>%
  mutate(Lemma = row.names(.))
```

⁵ Diese etwas obskure Funktion hat den Vorteil, dass sie die Struktur der Tabelle beibehält, während die gängigere Funktion `as.data.frame()` sie in ein anderes, nämlich in das sog. lange Format überführt – wenn Sie den Unterschied kennenlernen wollen, probieren Sie es einfach aus! Das lange Format hat in vielen Fällen Vorteile und ist z.B. als Input für Grafiken, die mit dem Paket `ggplot` erstellt werden, quasi obligatorisch. Für unsere Zwecke ist aber das „breite“ Format in diesem Fall besser geeignet.

Der Unterschied zwischen den beiden Varianten ist, dass die neue Spalte in Variante 1 die erste Spalte ist und den Namen *rownames* trägt, während sie in der zweiten Variante an letzter Stelle steht und wir ihr den Namen *Lemma* zugewiesen haben. Die zweite Variante habe ich vor allem deshalb angegeben, um zu illustrieren, wie man den Punkt in *dplyr* verwendet, um das, was auf der linken Seite der pipe (%>%) steht, wieder aufzugreifen, wenn es nicht oder nicht nur als erstes Argument der Funktion nach der Pipe verwendet wird, z.B. 7 %>% *sum*(3, .) (auch wenn es bei einer Summe natürlich völlig egal ist, an welcher Stelle der einzelne Summand steht, bei vielen anderen Funktionen ist es das aber nicht).

Mit dem Dataframe, den wir über den obigen Befehl generiert haben, können wir uns nun eine nach Frequenzen sortierte Tabelle ausgeben lassen, indem wir den bereits erwähnten *arrange*-Befehl benutzen und ihn mit dem Befehl *desc*, einer weiteren Funktion aus dem Paket *dplyr*, kombinieren. *desc* sorgt dafür, dass die Sortierung in absteigender Reihenfolge erfolgt. Weil der Dataframe sehr viele Zeilen hat und wir uns nur für die, sagen wir, 20 frequentesten adjektivischen Kollokate von *Frau* interessieren, benutzen wir außerdem die Funktion *head*, die die ersten paar Zeilen (defaultmäßig 5, hier wählen wir jedoch 20) eines Datenobjekts ausgibt:

```
> mannfrau %>% select(Lemma, df) %>% table %>% as.data.frame.matrix %>%
+   mutate(Lemma = row.names(.)) %>% arrange(desc(Frau)) %>% head(20)
   Frau Mann      Lemma
1    474  771      jung
2    369  316       alt
3    312   0      gnädig
4    164  23       schön
5    147 103      andere
6    135  75      deutsch
7    135 116      klein
8    109  21       lieb
9     73  17 verheiratet
10   64  74       gut
11   62  43       arm
12   59  33      erst
13   53  18      eigen
14   37  29  n-jährig
15   36   0 schwanger
16   35  41      weiß
17   33  29      dick
18   33   2 geschieden
19   33  26      klug
20   29  20 geliebt
```

Das gleiche können wir für *Mann* wiederholen. Diese Ergebnisse sind schon weitaus aufschlussreicher als die Auswertung nach Textsorten, die wir weiter oben ausprobiert haben: Frauen sind, wie sich zeigt, in Texten des 20. Jahrhunderts oft *schön*, *lieb* oder *verheiratet*, Männer hingegen *föhrend*, *frei*, *berühmt* oder *stark*. Noch deutlicher wird dies, wenn wir eine distinktive Kollexemanalyse (Gries & Stefanowitsch 2004) vornehmen; wie das mit Hilfe des Pakets *collostructions* von Flach (2017) geht, ist im Skript *datenauswertung.R* in einem Exkurs dargestellt.

Exkurs: Warum gut lesbarer Code wichtig ist

Wer mit dem Programmieren beginnt, fängt in der Regel einfach an, um nach und nach immer komplexere Aufgaben bewältigen zu können. Mit den Aufgaben wird auch der Code komplexer. Manchmal hat man am Ende einen Wust an Code, der zwar einmal funktioniert hat, aber kaum noch entzifferbar ist und beim erneuten Ausführen Fehler generiert. Das ist vor allem dann ärgerlich, wenn man den Code später noch einmal benutzen möchte, ihn aber selbst nicht mehr versteht. Hier ein paar Empfehlungen, wie man in R lesbar und nachvollziehbar coden kann:

1. Kommentarfunktion nutzen! Alles, was einer Raute (#) folgt, wird in R als „Kommentar“ gesehen und nicht mit ausgewertet. Das kann man nutzen, um zu erklären, was z.B. die jeweils folgende Funktion macht.
2. Bei der Namenswahl angemessene Entscheidungen treffen. - Wenn man Objekte benennt (z.B. den Dataframe, den man aus einem Spreadsheet einliest), dann hat man im Grunde die Wahl zwischen zwei Optionen: Entweder man gibt den Objekten aussagekräftige Namen, die aber u.U. recht lang sein können, oder man benutzt sehr kurze Namen, die man jeweils im Kommentar erklärt, sobald man sie definiert. Welche Option man wählen sollte, hängt auch von der zu erwartenden Komplexität des Codes ab: Braucht man viele komplexe Operationen, können lange Namen schnell dazu führen, dass der Code umständlich und unleserlich wirkt – `df[which(df$Anzahl_Tokens>100),]` ist nun einmal einfacher und knackiger als `meine_daten_gefiltert_ohne_20_jahrhundert[which(meine_daten_gefiltert_ohne_20_jahrhundert$Anzahl_Tokens>100),]`. Wickham (2015: 67) empfiehlt, nicht mehr als 80 Zeichen pro Zeile zu verwenden (was bei sehr langen Namen schwierig ist...).
3. Den Workspace **nicht** speichern. Wenn man RStudio verlässt, bietet es an, den Workspace zu sichern, was bedeutet, dass das nächste Mal, wenn man das Skript (oder ein anderes Skript im gleichen Arbeitsverzeichnis) öffnet, alle Objekte, die man in der Sitzung erstellt hat, noch im Environment sind. Ein Beispiel: Ich habe ein Skript, das nur besteht aus `mann <- read.csv("mann.csv")`, und führe es aus. Nun wird die Datei *mann.csv* eingelesen und in R als Dataframe, also als ein R-Objekt, gespeichert. Wenn ich nun das Programm verlasse und den Workspace sichere, muss ich beim nächsten Öffnen diese Zeile nicht mehr ausführen: Das Objekt *mann* ist noch im Environment verfügbar. Das kann praktisch sein, hat aber immense Nachteile, wenn Skripts komplexer werden: So kann es vorkommen, dass ich ein Stück Code weiter nach oben im Skript copy&paste und vergesse, dass dieses Stück Code auf Objekte zugreift, die jetzt erst im weiteren Verlauf des Skripts definiert werden. Wieder ein Beispiel: Ich habe den Dataframe *mann* und definiere einen Dataframe, in dem nur Belletristik vorkommt:

```
mann_bell <- subset(mann, Genre == "Belletristik")
```

Dann will ich Fehltreffer herausfiltern:

```
mann_bell <- subset(mann_bell, Lemma != "DISCARD")
```

Dann beschließe ich, dass das Herausfiltern von Fehltrefern eigentlich konzeptionell an erster Stelle stehen sollte und stelle deshalb das Skript um:

```
mann <- read.csv("mann.csv")
mann_bell <- subset(mann_bell, Lemma != "DISCARD")
mann_bell <- subset(mann, Genre == "Belletristik")
```

Sie merken, dass dieses Skript nicht funktionieren kann, denn das Objekt *mann_bell* ist ja in der zweiten Zeile noch gar nicht definiert, und trotzdem will ich schon ein Subset davon nehmen! Solange ich aber mit meinem gespeicherten Workspace weiterarbeite, entgeht mir das womöglich.

2.3.2 Datenvisualisierung

Wie für eigentlich alle Abschnitte in diesem Tutorial gilt auch und gerade für den folgenden, dass man über das Thema, mit dem er sich befasst, ganze Bücher füllen könnte (und kann, z.B. Murrell 2006 und Chang 2013). Ich beschränke mich darauf, kurz einige wichtige Prinzipien der Datenvisualisierung zu erläutern, um dann zu zeigen, wie man die verbreitetsten Visualisierungstypen in R erstellen kann.

Zunächst müssen wir uns darüber klar werden, welchem Ziel die Datenvisualisierung überhaupt dient. Letzten Endes erfüllt sie den Zweck, die Daten zu erschließen, Muster darin zu erkennen und dadurch Antworten auf unsere Forschungsfragen zu erhalten. Gleichzeitig lassen sich die Daten in visualisierter Form leserfreundlicher darstellen, als es etwa bei einer Tabelle oder gar bei einer langen Auflistung in Textform möglich wäre. Deshalb gilt es, bei der Visualisierung das richtige Maß zu finden: Einerseits sollten die Plots, die wir erstellen, natürlich einigermaßen ansehnlich aussehen, andererseits sollten sie vor allem zweckmäßig sein und nicht durch übertriebene visuelle Spielereien von ihrem eigentlichen Inhalt ablenken. Ist Letzteres der Fall, dann spricht man auch von „Overplotting“; Fig. 5 zeigt ein Beispiel dafür: Weder die Dreidimensionalität noch die unterschiedlichen Formen der Objekte wären nötig, um die Informationen zu transportieren, die in den Daten stecken. Fig. 6 zeigt eine „schlankere“ Visualisierung derselben (fiktiven) Daten.

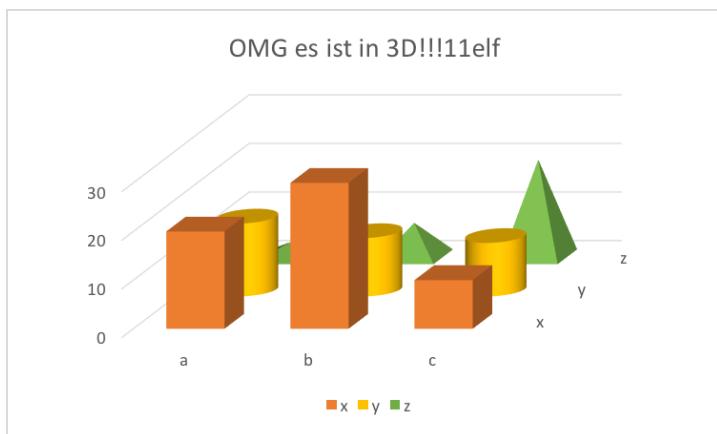


Fig. 5: Beispiel für „Overplotting“: Das 3D trägt ebensowenig zum Informationsgehalt des Plots bei wie die unterschiedlichen Formen, die benutzt werden.

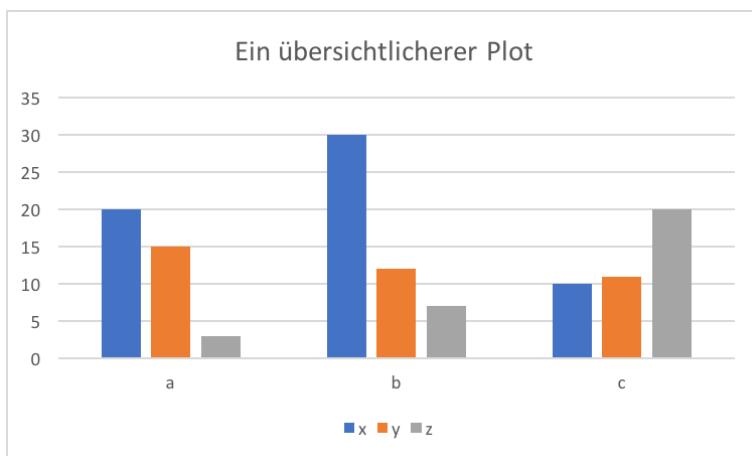


Fig. 6: Die Daten aus Fig. 5 in einer weniger verschönerten Darstellung.

Ein Plot sollte also seine Informationen auf möglichst effiziente und schnörkellose Weise präsentieren. Zugleich sollte er sie natürlich möglichst vollständig wiedergeben. Die Betonung liegt hier auf „möglichst“, denn selbstverständlich gehen immer Informationen verloren, wenn wir sie in einer grafischen Darstellung „verdichten“. Fig. 7 zeigt ein Beispiel dafür, wie eine suboptimale Darstellungsweise Fehlinterpretationen herbeiführen oder zumindest befördern kann: Beide Plots zeigen dieselben Daten, doch ist in der Variante links die y-Achse beschnitten. Der Unterschied zwischen den beiden Gruppen wirkt dadurch sehr viel größer, als es in der Darstellung rechts der Fall ist.

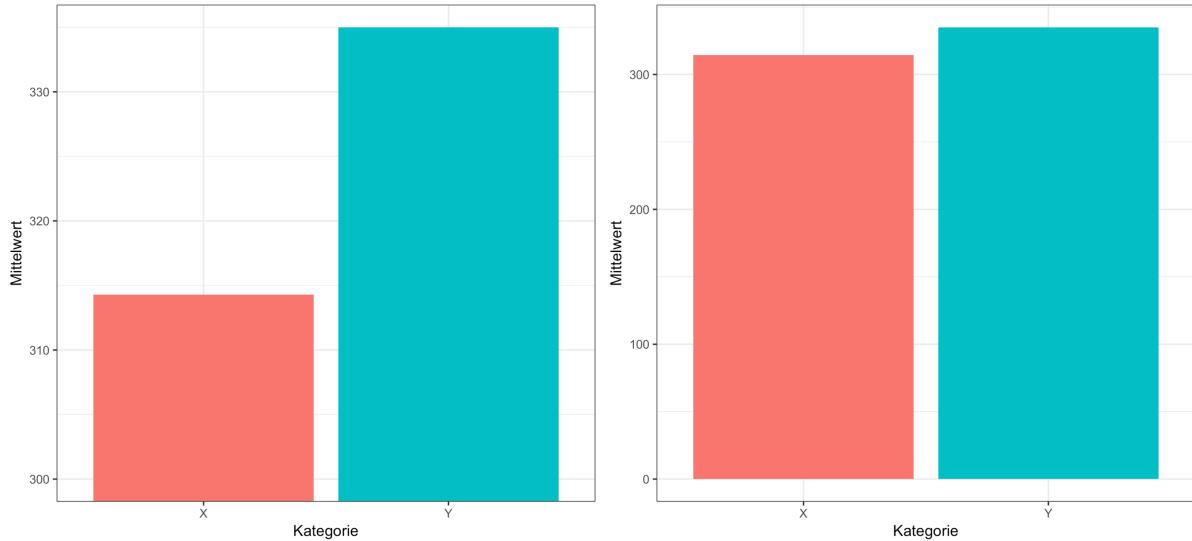


Fig. 7: Wenn die Achse beschnitten ist, wirkt der Unterschied viel größer, als er eigentlich ist: Ein eindrückliches Beispiel dafür, wie man mit Statistik lügen (oder zumindest übertreiben) kann.

Wenn es darum geht, wie man mit Statistik lügen kann, sind Fälle wie Fig. 7 ein immer wieder angeführtes Beispiel. In manchen Fällen kann es freilich für die Wahl der linken Darstellungsvariante gute konzeptionelle Gründe geben. Wenn man es beispielsweise mit einem Phänomen zu tun hat, bei dem schon minimale Unterschiede hochrelevant sind, kann es u.U. gerechtfertigt sein, die y-Achse zu beschneiden – man muss sich jedoch bewusst sein, dass in diesem Fall die Visualisierung mehr als sonst schon eine Interpretation darstellt. Die Grafiken in Fig. 7 zeigen beide den Mittelwert der fiktiven Daten, die ihnen zugrundeliegen (im Skript *visualisierung.R* im Ordner *R_Einstieg* können Sie die Daten und Plots selbst generieren). Dabei gehen freilich ebenfalls Informationen verloren: Angenommen, wir haben die Verteilungen in (1)...

- (1) a. 10, 100, 1000, 90
 b. 300, 300, 300, 300

... so haben sowohl die Verteilung in a) als auch die in b) jeweils den Mittelwert 300, obwohl sie unterschiedlicher kaum sein könnten! Deshalb arbeitet man häufig mit Fehlerbalken (*error bars*), um die Varianz darzustellen, die in den Daten besteht. Fig. 8 zeigt die oben unter (1)

genannten Verteilungen mit Fehlerbalken, die den Standardfehler des Mittelwerts zeigen.⁶ Fig. 9 zeigt die Daten aus Fig. 7 noch einmal mit Fehlerbalken.

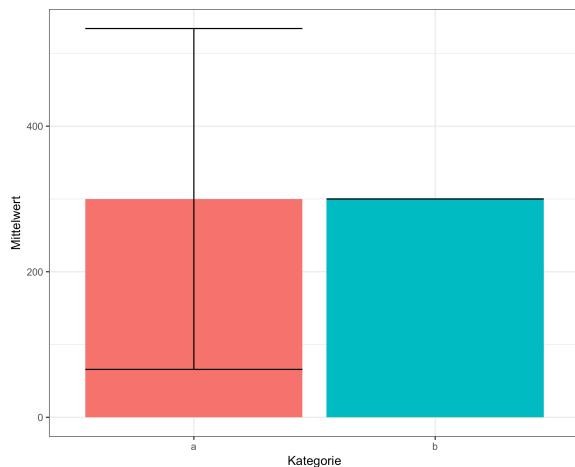


Fig. 8: Darstellung der unter (1)a und b genannten Verteilungen: Der Balken selbst zeigt den Mittelwert, die Fehlerbalken zeigen den Standardfehler.

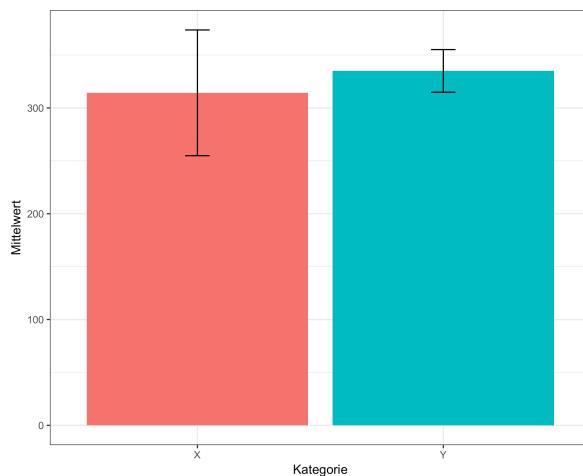


Fig. 9: Barplot aus Fig. 7 mit Fehlerbalken (error bars).

Wir sehen also, dass unterschiedliche Visualisierungen unterschiedlich informativ sein können. Das gilt natürlich auch für verschiedene Arten von Plots. Beispielsweise haben Tortendiagramme (*pie charts*) mittlerweile einen sehr schlechten Ruf, weil sie relativ schlecht

⁶ An dieser Stelle ist nur wichtig, dass der Fehlerbalken anzeigt, wie stark die Daten „streuen“. Falls Sie sich dafür interessieren, wie man den Standardfehler berechnet, hier eine kurze Erläuterung. Der Standardfehler errechnet sich aus der Standardabweichung geteilt durch die Wurzel der Anzahl an Elementen, aus denen der Mittelwert errechnet wird. Die Standardabweichung wiederum ist die Wurzel der Varianz, die ihrerseits die mittlere quadratische Abweichung der einzelnen Elemente vom Mittelwert ist, geteilt durch die Anzahl der Elemente minus 1. In Beispiel (1)a wäre die Varianz also: $((10-300)^2 + (100-300)^2 + (1000-300)^2 + (90-300)^2) / (4-1) = 219400$. Die Standardabweichung ist dann die Wurzel aus 219400, also 468,4. Den Standardfehler erhalten wir, wie gesagt, wenn wir die Standardabweichung durch die Wurzel der Anzahl an Elementen teilen, also 468,4 durch Wurzel aus 4 = 234,2008. Die Daten weichen in beide Richtungen ab, daher ist die Untergrenze des Fehlerbalkens der Mittelwert minus Standardfehler, die Obergrenze hingegen ist der Mittelwert plus Standardfehler.

interpretierbar sind, wenn mehrere der dargestellten Kategorien ähnliche Werte aufweisen. Auch Balkendiagramme geraten zusehends in Verruf; es gab sogar eine Kickstarter-Kampagne *#barbarplots*, die Geld sammelte, um Merchandising-Produkte wie T-Shirts oder Taschen zu produzieren, die verschiedene Visualisierungen desselben Datensatzes unter der Überschrift *Friends don't let friends make bar plots* zeigen.⁷ Da Torten- und Balkendiagramme gleichwohl nach wie vor sehr verbreitet sind und in manchen Fällen durchaus auch ihren Nutzen haben können, zeige ich im Folgenden, wie man sie mit R erstellt. Danach wenden wir uns alternativen Visualisierungsmöglichkeiten zu, insbesondere sog. Scatterplots und Lineplots.

Zwei Wege zum Plot: Base Graphics und *ggplot2*

Es gibt zwei verbreitete Möglichkeiten, Grafiken in R zu erstellen: Einerseits kann man die Plot-Funktionen verwenden, die bereits in R integriert sind, andererseits das Paket *ggplot2*. Für AnfängerInnen ist die erste Option meiner Meinung nach die wesentlich einfachere. Gleichwohl hat *ggplot2* enorme Vorteile, denn einiges, was man in den Base Graphics auf teils recht komplizierte Weise manuell machen muss, kann *ggplot2* (fast) automatisch. Daher gehe ich im Folgenden auf beide Möglichkeiten ein.

Einfache Tortendiagramme in Base Graphics

Beginnen wir mit dem ungeliebten, aber einfachen Tortendiagramm. Greifen wir wieder auf den Adjektiv + *Mann/Frau*-Datensatz zurück, den wir zunächst einlesen:

```
# Daten einlesen
mannfrau <- rbind(mutate(read.csv("adja_mann.csv", fileEncoding = "UTF-8"),
                           df = "Mann"),
                      mutate(read.csv("adja_frau.csv", fileEncoding = "UTF- 8"), df =
                           "Frau"))
```

Nun können wir ein Tortendiagramm der Textsortenverteilung erstellen. Nichts einfacher als das: Wir bringen zunächst die Daten in tabellarische Form, wie wir es oben bereits gelernt haben, und können diese Tabelle dann direkt als Input für die Funktion *pie()* verwenden, die ein Piechart erstellt:

```
mannfrau$Genre %>% table %>% pie
```

Auf Wunsch können wir die Grafik auch noch modifizieren, indem wir z.B. manuell Farben auswählen oder eine Überschrift hinzufügen; eine Übersicht der Argumente, die *pie()* nimmt, erhalten Sie in der Hilfe, wenn Sie *?pie* eingeben. Ein Beispiel:

```
mannfrau$Genre %>% table %>% pie(col = c("darkgreen", "orange", "red",
                                             "blue", "yellow"))
title("Tortendiagramm")
```

In *ggplot2* sind Tortendiagramme nicht ohne weiteres zu erstellen (wohl auch aufgrund der oben bereits genannten Vorbehalte). Deshalb gehen wir gleich zum nächsten Format über, nämlich zu Balkendiagrammen (*barplots*).

⁷ <https://www.kickstarter.com/projects/1474588473/barbarplots> (abgerufen am 17.10.2017)

Balkendiagramm mit Base Graphics

Ungeachtet der Probleme, die Balkendiagramme mit sich bringen, zählen sie gerade in der Sprachwissenschaft nach wie vor zu den wichtigsten Darstellungsformen. Genau wie das Tortendiagramm ist auch ein einfaches Balkendiagramm in R bestechend einfach zu erstellen:

```
mannfrau$Genre %>% table %>% barplot
```

Auch hier können wir noch einige Modifikationen vornehmen. Beispielsweise werden – je nachdem, in welcher Größe man den Plot darstellt – ggf. einige der sog. Argumentnamen unterschlagen, also der Kategorienbezeichnungen, die unterhalb der Balken stehen: In Fig. 10 werden nur „Belletristik“, „gesprochen“ und „Zeitung“ angezeigt, die beiden anderen Textsorten werden stillschweigend unterschlagen.

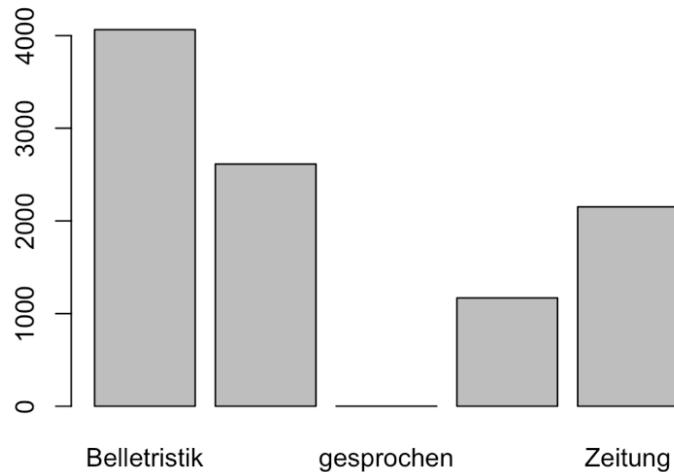


Fig. 10: Ein einfaches Balkendiagramm. Es werden nicht alle Argumentnamen angezeigt, weil der Plot zu klein und/oder die Schrift zu groß ist.

Wenn wir sicherstellen möchten, dass alle Textsorten angezeigt werden, haben wir mehrere Möglichkeiten. Zum einen können wir den Plot in ausreichender Größe speichern (zum Exportieren siehe **Infobox: Plots exportieren** weiter unten). Zum Vergleich: Der Code

```
png("plot_gross.png", width=6, height=5, un="in", res=300)
mannfrau$Genre %>% table %>% barplot
dev.off()
```

generiert den Plot in Fig. 10 oben,

```
png("plot_gross.png", width=12, height=10, un="in", res=300)
mannfrau$Genre %>% table %>% barplot
dev.off()
```

hingegen den in Fig. 11 unten.

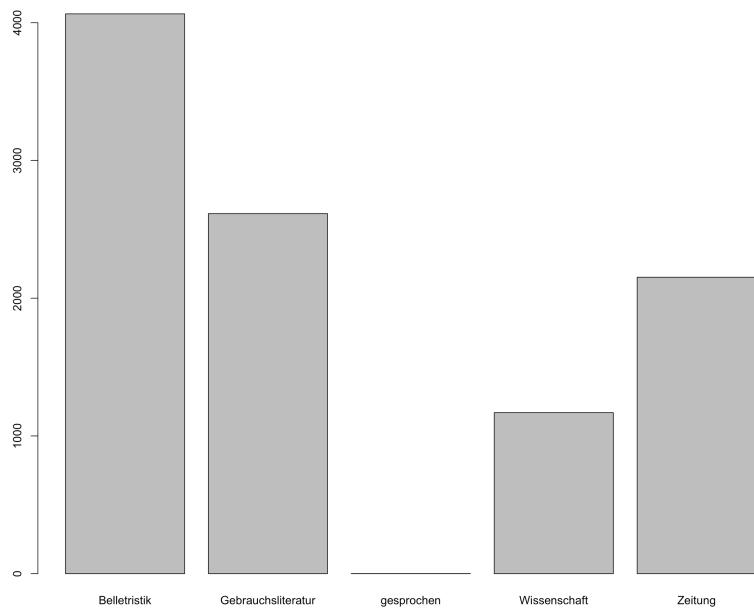


Fig. 11: Dieser Plot wurde mit den Maßen 12x10 Inch exportiert, deshalb sind alle Argumentnamen dargestellt (wenn auch nur sehr klein)..

Eine andere Möglichkeit mit ähnlichem Effekt besteht darin, die Schriftgröße zu reduzieren. Das ist bei R-Plots, die mit Base Graphics erstellt werden, mit Hilfe des *cex*-Arguments möglich, wobei es unterschiedliche *cex*-Argumente für unterschiedliche Elemente gibt. Relevant für uns ist *cex.names*, womit man die Schriftgröße der Argumentnamen reguliert; analog kann man z.B. mit *cex.axis* die Größe der Achsenbeschriftungen regulieren.

```
# Balkendiagramm mit verkleinerter Schriftgroesse der Argumentnamen und
# Achsenbeschriftungen
mannfrau$Genre %>% table %>% barplot(cex.names=0.6, cex.axis=0.8)
```

Balkendiagramm mit ggplot2

Während wir mit den Base Graphics einen Plot aus der Tabelle (also einer Matrix) kreiert haben, erfordert *ggplot2* einen Dataframe als Input. Hier können wir auf unseren Original-Dataframe *mannfrau* zurückgreifen⁸:

```
ggplot(mannfrau, aes(x = Genre)) + geom_bar()
```

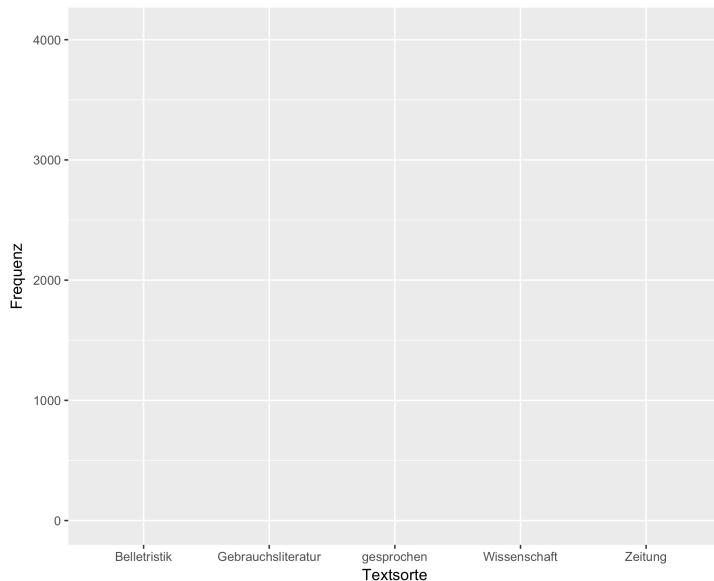
Wie bereits erwähnt, ist *ggplot* zunächst etwas schwer zu verstehen, aber nach kurzer Eingewöhnungszeit macht alles Sinn. Sehen wir uns daher die einzelnen Funktionen und Argumente näher an. Wichtig zu wissen ist zunächst, dass *ggplot* mit Schichten (*layers*) arbeitet, die quasi übereinandergestapelt werden. Diese Schichten werden einfach mit +-Zeichen verbunden, und komplexe Plots können enorm viele dieser Schichten enthalten. Mit den Argumenten, die man über das +-Zeichen dem bereits bestehenden Plot hinzufügt, kann

⁸ Dieser Abschnitt war ursprünglich anderthalb Seiten lang, bis ich unter *?geom_bar()* gesehen habe, wie unglaublich einfach es ist, einen Barplot direkt aus dem Dataframe zu erstellen. Das zeigt, wie sinnvoll es auch für Fortgeschrittene sein kann, gelegentlich einen Blick in die Hilfefunktion zu werfen!

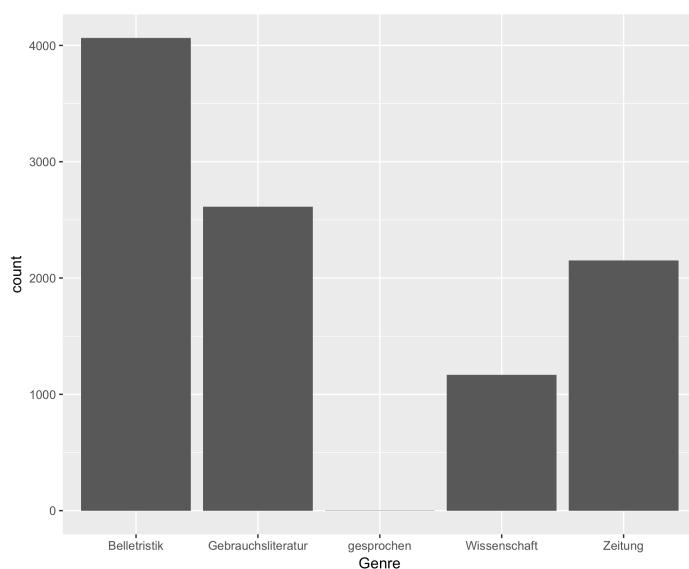
man z.B. das Aussehen des Plots verändern oder auch zusätzliche Elemente wie die oben bereits erwähnten *error bars* hinzufügen.

Die Kernfunktion ist `ggplot()` – diese steht immer ganz am Anfang und enthält als erstes Argument den Dataframe, der die Daten enthält. Mit dem *aes*-Argument werden die *aesthetics* festgelegt, d.h. diejenigen Elemente, die letztlich in den Plot Eingang finden sollen. Hier kann man i.d.R. mit den Argumenten *x* und *y* angeben, welche Elemente auf der x- und auf der y-Achse dargestellt werden sollen. Bei Barplots jedoch genügt eine die Angabe von *x*, denn `geom_bar()` zählt dann die Elemente, die in *x* (hier also: in der Spalte „Genre“) auftreten, und plottet diese dann auf der y-Achse.

Würden wir nur den ersten Teil des Plots vor dem Pluszeichen ausführen, dann erhielten wir einen leeren Plot:



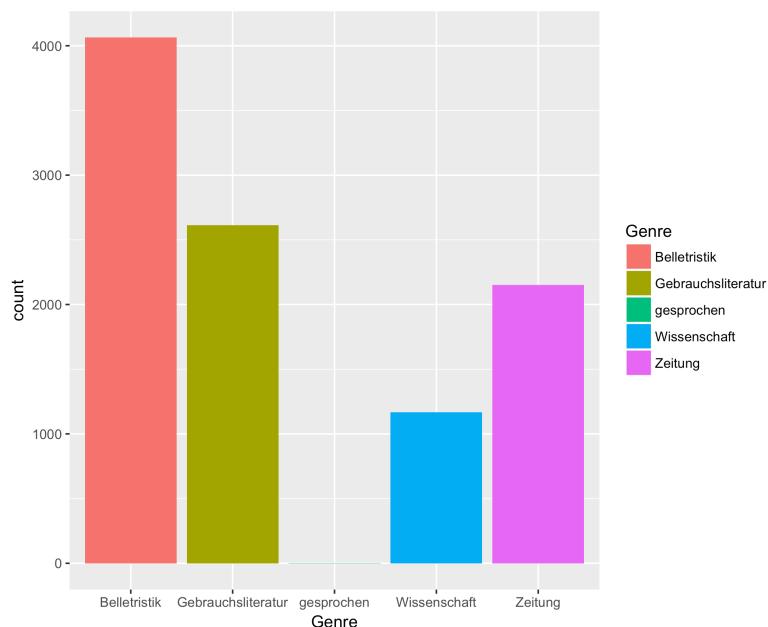
Das liegt daran, dass die entscheidende „Schicht“ noch fehlt, nämlich die, die spezifiziert, um was für einen Plot es sich denn nun handeln soll. Hier wählen wir `geom_bar()` (für Barplot). Nun sieht der Plot schon ganz ansehnlich aus:



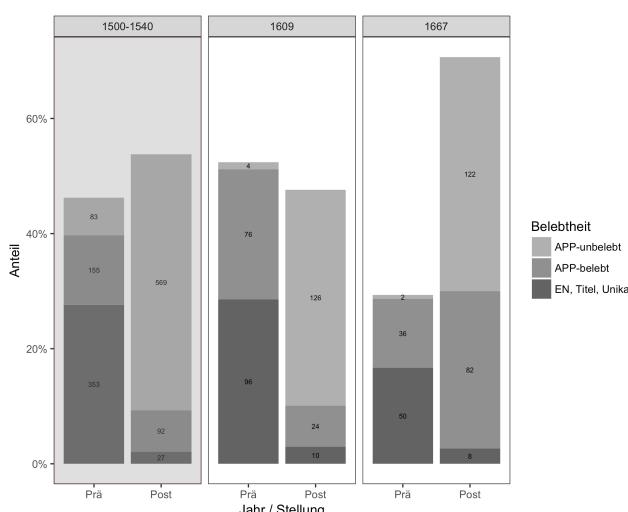
Mit einigen zusätzlichen Argumenten können wir den Plot allerdings noch etwas „aufpeppen“ und dadurch nicht nur ansehnlicher, sondern letztlich auch informativer machen. So kann es nicht schaden, den Balken unterschiedliche Farben zu geben. Auch das ist sehr einfach – zumindest, wenn man eines der Grundprinzipien von *ggplot* verstanden hat: Prinzipiell sollten sich alle *aesthetics*, also alle ästhetischen Parameter wie Farbe, Form usw., aus den Daten ableiten lassen. Wir können *ggplot* sehr einfach sagen, dass die Füllfarbe den Textsorten entsprechen soll, indem wir das *aes*-Argument am Anfang damit „füttern“:

```
ggplot(mannfrau, aes(x = Genre, fill = Genre)) + geom_bar()
```

Nun sieht das Ganze schon etwas bunter aus:



Allerdings ist es auch etwas redundant, weil nun die Textsorten sowohl als „Argumentnamen“ (um den vorhin gebrauchten Begriff aufzugreifen) unter den Balken verwendet als auch in der Legende erklärt werden. Das liegt daran, dass in vielen Fällen *x* und *fill* unterschiedliche Argumente haben, etwa wenn es sich um „gestapelte“ Barplots (*stacked barplots*) handelt, also so etwas wie das hier (zur Genitivstellung):



In unserem Fall aber brauchen wir die Legende nicht und können sie mit Hilfe des *guide*-Arguments deaktivieren. (Wenn man eine Legende will und braucht, kann man sie mit Hilfe des *guide*-Arguments auch anpassen und ggf. z.B. eine neue Überschrift für die Legende wählen; vgl. die Hilfe zu `?guide`).

```
ggplot(mannfrau, aes(x = Genre, fill = Genre)) + geom_bar() +
  guides(fill = FALSE)
```

Jetzt ist die Legende weg – die Regenbogenfarben aber sind noch da, was auch sehr schön ist, aber hinderlich sein kann, wenn wir die Grafik in Schwarzweiß publizieren wollen. Zum Glück gibt es die *scale*-Funktionen, mit denen man die einzelnen Argumente (wie *fill*) den eigenen Bedürfnissen anpassen kann. So können wir mit *scale_fill_grey* die Füllwerte zu unterschiedlichen Graustufen ändern:

```
ggplot(mannfrau, aes(x = Genre, fill = Genre)) + geom_bar() +
  guides(fill = FALSE) + scale_fill_grey()
```

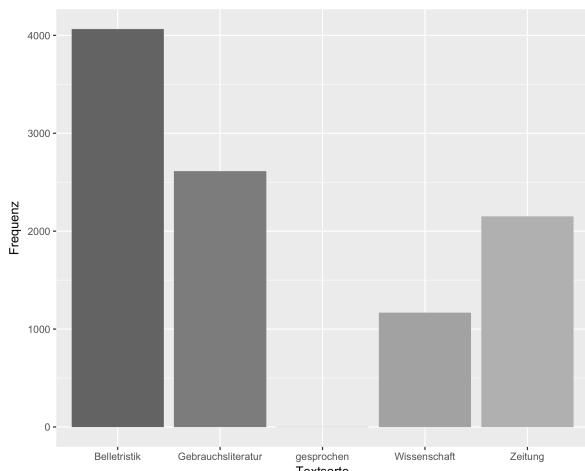
In manchen Fällen ist in diesem Fall der hellste Grau-Wert zu hell oder der dunkelste zu dunkel, vor allem dann, wenn man noch Text oder Zahlen auf die Balken drucken möchte (dazu kommen wir später). Praktischerweise kann man in *scale_fill_grey* mit den *start*- und *end*-Argumenten spezifizieren, wie dunkel der dunkelste Grauton bzw. wie hell der hellste Grauton sein soll, z.B.

```
ggplot(mannfrau, aes(x = Genre, fill = Genre)) + geom_bar() +
  guides(fill = FALSE) + scale_fill_grey(start = 0.4, end = 0.7)
```

Will man statt Graustufen andere Farben wählen, lohnt sich auch ein Blick auf *scale_fill_manual* sowie *scale_fill_brewer* (Letzteres erfordert das Paket *RColorBrewer*), womit man Zugriff auf verschiedene Farbpaletten hat – mit deren Hilfe kann man z.B. Plots generieren, die auch für farbenblinde Menschen gut erkennbar sind.

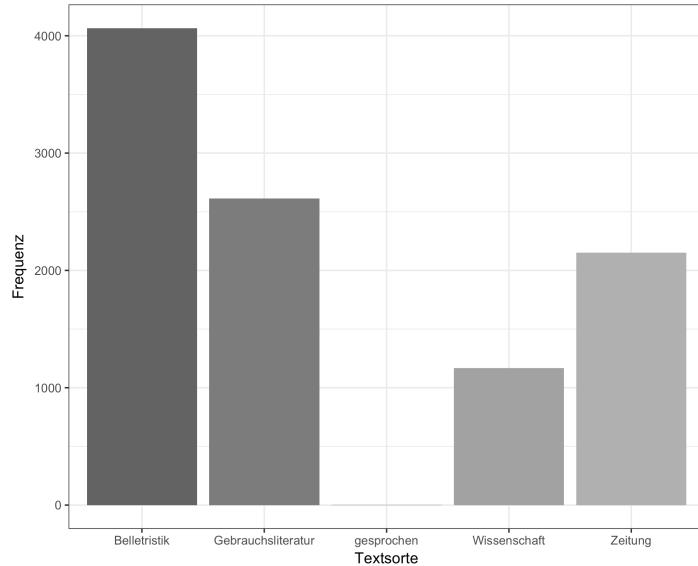
Weiterhin wollen wir u.U. auch noch die Beschriftung der x- und/oder der y-Achse verändern, z.B. um die Default-Beschriftung „count“ durch etwas Bedeutungsvolleres zu ersetzen. Auch das ist extrem einfach: Mit den Argumenten *xlab* und *ylab* können wir die Beschriftungen anpassen:

```
ggplot(mannfrau, aes(x = Genre, fill = Genre)) + geom_bar() +
  guides(fill = FALSE) + scale_fill_grey(start = 0.4, end = 0.7) +
  xlab("Textsorte") + ylab("Frequenz")
```



Das für *ggplot* charakteristische grau-weiße Gitternetz im Hintergrund ist für schwarz-weiße Darstellungen ebenfalls nur bedingt geeignet. Doch auch das sog. Thema (*theme*) lässt sich mit dem entsprechenden Argument sehr einfach ändern; wenn Sie wissen wollen, welche *theme*-Möglichkeiten es gibt, können Sie in RStudio einmal *theme* eingeben und die Tabulatortaste drücken, um die entsprechenden Vorschläge zu sehen. Im Folgenden benutzen wir das für Schwarzweißdarstellungen optimierte Thema *theme_bw()*:

```
ggplot(mannfrau, aes(x = Genre, fill = Genre)) + geom_bar() +
  guides(fill = FALSE) + scale_fill_grey(start = 0.4, end = 0.7) +
  xlab("Textsorte") + ylab("Frequenz") + theme_bw()
```



Jetzt fehlt noch eine Überschrift! Auch ein Plottitel ist extrem einfach zu kreieren; wenn wir jedoch das Erscheinungsbild des Titels anpassen wollen, dann stoßen wir u.U. an unsere Grenzen. Beginnen wir aber mit dem einfachsten Teil und fügen einfach einen Titel hinzu:

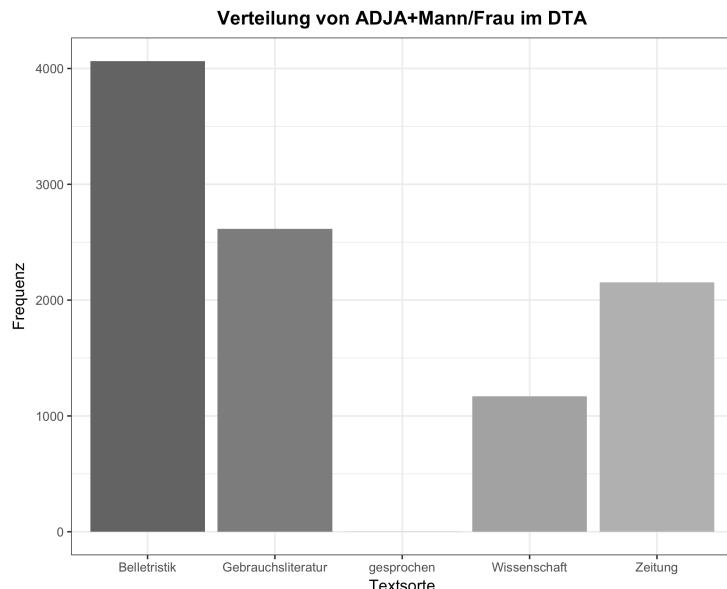
```
ggplot(mannfrau, aes(x = Genre, fill = Genre)) + geom_bar() +
  guides(fill = FALSE) + scale_fill_grey(start = 0.4, end = 0.7) +
  xlab("Textsorte") + ylab("Frequenz") + theme_bw() +
  ggtitle("Verteilung von ADJA+Mann/Frau im DTA")
```

Im Unterschied zum Titel bei den Base Graphics, der, wie wir oben gesehen haben, zentriert und in Fettdruck erscheint, ist der *ggplot*-Titel defaultmäßig linksbündig und nicht fett gedruckt. Das können wir ändern, indem wir ein Themaelement hinzufügen. Themaelemente kontrollieren das Aussehen von Elementen, die sich nicht unmittelbar aus den Daten ergeben (also z.B. Größe der Achsenbeschriftungen oder eben Aussehen und Position der Überschrift), und jedes Themaelement ist verbunden mit einer Elementfunktion, die beschreibt, wie genau das Element aussehen soll (vgl. Wickham 2016: 173).

```
ggplot(mannfrau, aes(x = Genre, fill = Genre)) + geom_bar() +
  guides(fill = FALSE) + scale_fill_grey(start = 0.4, end = 0.7) +
  xlab("Textsorte") + ylab("Frequenz") + theme_bw() +
  ggtitle("Verteilung von ADJA+Mann/Frau im DTA") +
  theme(plot.title = element_text(face = "bold", hjust = 0.5))
```

Die letzte Zeile im obigen Code ist das Themaelement, während *element_text()* die Elementfunktion ist. Die Tatsache, dass man diese Elementfunktion braucht und nicht einfach

*`+plot.title(face = "bold")` o.ä. angeben kann, mag für AnfängerInnen zunächst verwirrend sein; wenn man aber erst einmal ein paar Plots gemacht hat, gewöhnt man sich daran.
Nun sieht unser Plot also so aus:



Das ist schon einigermaßen veröffentlichtsreif (zumindest visuell; die Daten an sich sind nur bedingt interessant). Aber wir wollen perfektionistisch sein: Der linguistischen Konvention folgend, sollte *Mann/Frau* in der Überschrift kursiv stehen. Das ist eine kleine Herausforderung, die sich aber ebenfalls lösen lässt, und zwar mit Hilfe der Funktion `expression()`. An dieser Stelle ist nicht wichtig zu wissen, was diese Funktion tut (eine relativ technische Erklärung von Expressions findet sich bei Wickham 2013); wichtig ist, dass sie es erlaubt, Funktionen wie `italic()`, `bold()` oder `bolditalic()` zu benutzen (die eigentlich für mathematische Ausdrücke gedacht sind und daher zur Funktion `plotmath` gehören). Weil nicht die gesamte Überschrift in `italics()` stehen soll, sondern nur ein Teil davon, brauchen wir zudem die Funktion `paste()`. Was diese für die Verarbeitung von Text in R sehr praktische Funktion tut, können Sie ausprobieren, indem Sie z.B. eingeben:

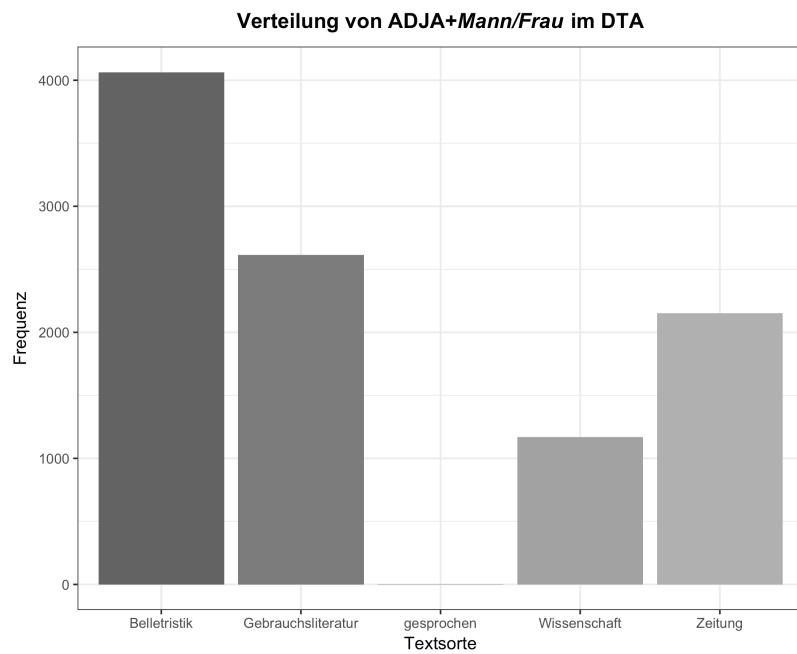
```
> paste(c("Die", "Kreiszahl", "pi", "hat", "den", "wert", "pi), collapse = " ")
[1] "Die Kreiszahl pi hat den Wert 3.14159265358979"
```

Der Code, mit dem wir den Plot mit teilweise kursiver Überschrift generieren können, sieht dann so aus:

```
ggplot(mannfrau, aes(x = Genre, fill = Genre)) + geom_bar() +
  guides(fill = FALSE) + scale_fill_grey(start = 0.4, end = 0.7) +
  xlab("Textsorte") + ylab("Frequenz") + theme_bw() +
  ggtitle(expression(paste(bold("Verteilung von ADJA+"), bolditalic("Mann/Frau"),
    bold(" im DTA"), collapse = ""))) +
  theme(plot.title = element_text(hjust = .5))
```

Sie werden bemerkt haben, dass hier der Fettdruck komplett über die `plotmath`-Funktionen `bold()` bzw. `bolditalic()` gesteuert wird. Das ist deshalb so, weil der oben benutzte Befehl `face = "bold"` nicht funktioniert, wenn wir `expression(paste(...))` in der Plot-Überschrift verwenden.

Hier ist nun der finale Plot:



Wenn Sie das Buch gelesen haben, zu dem dieses Begleitmaterial gehört, werden Sie bemerken, dass die meisten Plots darin so aussehen – insofern spiegelt dieses Tutorial natürlich ein Stückweit auch meine ganz persönliche Präferenz wider. Das Schöne an *ggplot* ist, dass Sie Ihren eigenen bevorzugten Stil finden können und dass Sie Ihre Grafiken auch an die Art und Weise der Präsentation anpassen können: In einer Powerpoint-Präsentation zum Beispiel kann ein farbiger Plot seine Wirkung besser entfalten als in einem Zeitschriftenartikel, der in schwarzweiß gedruckt wird.

Und wenn Sie einen Stil gefunden haben, der Ihnen zusagt und den Sie öfter benutzen wollen (oder auch mehrere), dann ist es kein Problem, das Ganze auch als Funktion zu speichern. Das ist zugegebenermaßen etwas für Fortgeschrittene, aber im Skript *visualisierung.R* zeige ich dennoch, wie es geht – Interessierte können sich dann ggf. etwas tiefer einlesen und den Code für eigene Zwecke adaptieren. Erfahrungsgemäß ist die Zeit, die man braucht, um sich die entsprechenden Fähigkeiten anzueignen, relativ kurz im Vergleich zu den Unmengen an Zeit, die man mit Hilfe dieser kleinen Tricks sparen kann, wenn man viel mit R-Grafiken arbeitet.

Plots exportieren

Will man Plots speichern, so ist davon abzuraten, sie im „Plot“-Fenster von RStudio manuell zu speichern: Leider generiert diese Methode derzeit nur Plots mit zu niedriger Auflösung, nämlich 72dpi. Publikationsfähige Grafiken benötigen aber mindestens 300dpi. Deshalb sollte man Grafiken idealerweise mit Hilfe der Funktion *png* (oder *jpg*, *bmp*, *tiff*) speichern. Das geht wie folgt:

```
png("plot_gross.png", width=12, height=10, un="in", res=300)
mannfrau$Genre %>% table %>% barplot
dev.off()
```

Die Parameter *width* und *height* sind selbsterklärend; *un* gibt die Einheit an, in der gemessen wird (hier: Inch), und das *res*-Argument spezifiziert die Auflösung, hier: 300dpi. Die erste Zeile des obigen Codes öffnet quasi das png-„Graphics Device“. Alles, was folgt (das können durchaus auch sehr viele Zeilen sein, die zusammen einen Plot konstituieren) geht, um es vereinfacht bildlich dazustellen, in dieses Device hinein. Am Ende muss das Device dann mit *dev.off()* geschlossen werden.

Eine weitere Möglichkeit besteht darin, Plots als Vektorgrafiken im *svg*-Format zu exportieren. Das hat den Vorteil, dass man sich über die Auflösung keine Gedanken machen muss, bringt aber den Nachteil mit sich, dass nicht alle Programme, mit denen Sie evtl. Ihren Text erstellen, das *svg*-Format unterstützen. Zum Beispiel ist in Microsoft Word SVG-Support erst ab Version 2016 und auch nur auf dem Windows-Betriebssystem implementiert. Ältere Word-Versionen und Word für Mac unterstützen nur das Microsoft-eigene Vektorgrafik-Format *emf*.

Etwas einfacher ist der Export bei *ggplot2*-Grafiken: Hier kann man die Funktion *ggsave* verwenden. Sobald man einen Plot erstellt hat, kann man anschließend

```
ggsave("dateiname.png")
```

verwenden, und die Grafik wird meist in einer geeigneten Größe und auch defaultmäßig schon mit 300dpi gespeichert. Allerdings unterstützt *ggsave* nativ kein *svg*; will man *svg*-Dateien mit *ggsave* exportieren, muss man zunächst ein entsprechendes Paket installieren oder aber auf die oben beschriebenen Standard-Exportoptionen zurückgreifen, die sich auf *ggplot*-Grafiken genauso anwenden lassen wie auf Base Graphics.

Gestapelte Barplots (stacked barplots) mit Base Graphics

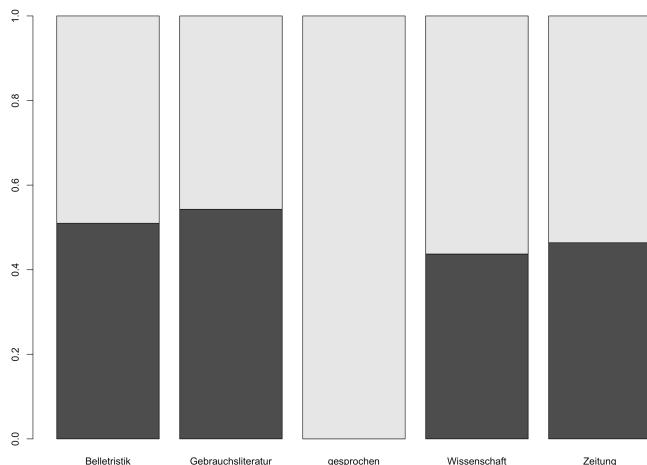
Manchmal möchte man auch mit Balkendiagrammen arbeiten, bei denen nicht nur einzelne Balken nebeneinanderstehen, sondern die – wie oben das Diagramm zur Genitivstellung – mehrere Kategorien in einem Balken repräsentieren. Zum Beispiel könnten wir in den oben erstellten Plots nach *Mann* und *Frau* trennen wollen. Die absoluten Frequenzen lassen sich denkbar einfach plotten:

```
# Stacked Barplot mit Base Graphics
mannfrau %>% select(df, Genre) %>% table %>% barplot()
```

Wenn wir die relativen Frequenzen darstellen möchten, können wir das ebenfalls recht einfach tun, indem wir in die obige Sequenz noch die Funktion *prop.table()* einfügen. Dabei ist es wichtig, dass wir in der Funktion *prop.table()* zusätzlich das *margin*-Argument spezifizieren, um sicherzustellen, dass der relative Wert nicht für die einzelnen Zeilen berechnet wird, sondern für die einzelnen Spalten (denn die Spalten sind es ja, die dann als Balken repräsentiert werden, und die einzelnen „gestapelten“ Bestandteile der Balken sollen sich ja auf 1 aufsummieren).

```
mannfrau %>% select(df, Genre) %>% table %>% prop.table(margin = 2) %>%
barplot()
```

Hier das Ergebnis:



Nun ist ein Barplot, der lediglich relative Frequenzen zeigt, nicht unbedingt immer aussagekräftig – zum Beispiel haben wir ja schon mehrfach gesehen, dass die Kategorie „gesprochen“ in unseren Daten nur eine einzige Beobachtung enthält und deshalb eigentlich vernachlässigbar ist. Wenn die Leserin oder der Leser nur das Balkendiagramm sieht, wie wir es eben geplottet haben, wird diese Information schlichtweg unterschlagen. Deshalb kann es sinnvoll sein, die absoluten Zahlen in die Balken zu schreiben. Das ist durchaus möglich (Gries 2013 zeigt für einfache Barplots mit Base Graphics, wie das geht), aber mit *ggplot2* so viel einfacher, dass wir uns besser gleich der alternativen Variante zuwenden.

Stacked Barplots mit *ggplot2*

Das Paket *ggplot2* bietet einige sehr nützliche Funktionen zur Erstellung informativer Barplots. Um sie voll nutzen zu können, ist es sinnvoll, die Daten zunächst in tabellarische Form zu bringen, anstatt sie durch die *stat_count*-Funktion von *ggplot* zählen zu lassen, wie wir es oben beim einfachen Barplot gemacht haben. Unser Ziel ist daher zunächst, eine Tabelle zu erstellen, die sowohl die absoluten als auch die relativen Frequenzen enthält. Beim Erstellen des Stacked Barplot mit Base Graphics haben wir beide Tabellen getrennt voneinander schon erstellt und jeweils als Input für eine Grafik verwendet. Damit haben wir schon einen guten Startpunkt. Fangen wir an mit den absoluten Frequenzen:

```
mf1 <- mannfrau %>% select(df, Genre) %>% table() %>% as.data.frame
```

Wir überführen die Daten in einen Dataframe, weil das der Input ist, den *ggplot2* braucht: Mit Tabellen (Matrizen) kann das Paket nicht umgehen. Anders als oben (Abschnitt 2.3.1 Einfache tabellarische Auswertung) verwenden wir nicht die Funktion *as.data.frame.matrix()*, weil *ggplot2* die Daten in dem „langen“ Format braucht, das *as.data.frame()* generiert. Wenn Sie sich den Dataframe *mf1* anschauen, werden Sie merken, dass es keine eigene Spalte für *Mann* und *Frau* gibt, sondern vielmehr jede Kombination von Merkmalsausprägungen eine eigene Zeile hat, also z.B. *Mann-Belletristik* und *Frau-Belletristik* in einer eigenen Zeile stehen. Genau das ist mit dem „langen“ Format gemeint; beim „breiten“ Format haben wir weniger Zeilen, dafür aber mehr Spalten.

Mit der Funktion *prop.table*, die wir oben bereits kennengelernt haben, können wir nun auf die gleiche Weise auch die relativen Frequenzen ausgeben lassen.

```
mf2 <- mannfrau %>% select(df, Genre) %>% table() %>% prop.table(mar=2) %>% as.data.frame
```

Im nächsten Schritt können wir die relativen Frequenzen, die jetzt im neu generierten Dataframe *mf2* zu finden sind, mit Hilfe der *mutate*-Funktion dem zuvor generierten Dataframe *mf1* als neue Spalte hinzufügen:

```
mf1 <- mutate(mf1, Abs_Freq = mf2$Freq)
```

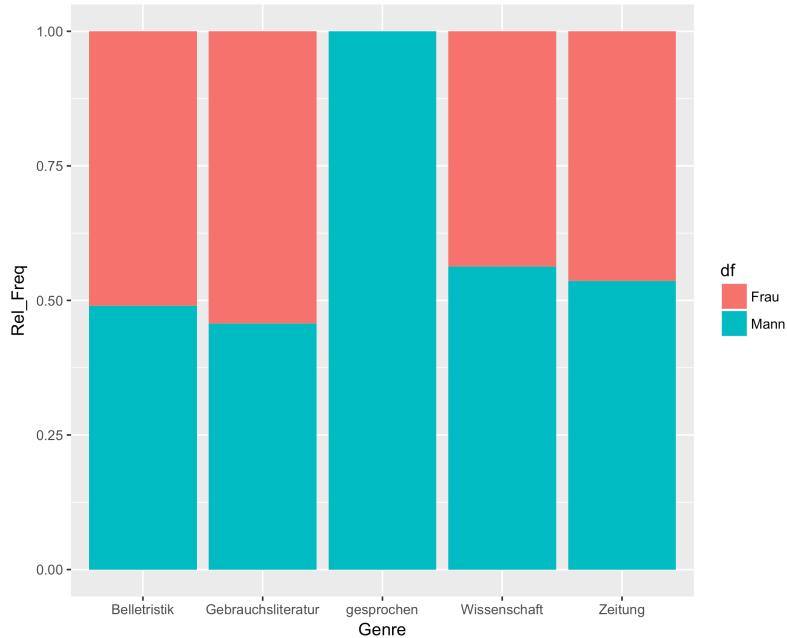
Nun enthält *mf1* alle Informationen, die wir brauchen, um einen Barplot zu kreieren, der die relativen Frequenzen zeigt und zugleich die absoluten Frequenzen in Textform darstellt. Basteln wir zunächst den einfachen Barplot, noch ohne die absoluten Frequenzen.

```
ggplot(mf1, aes(x = Genre, y = Rel_Freq, fill = df)) +  
  geom_col()
```

Wie wir oben gesehen haben, müssen wir die *aesthetics* spezifizieren, die bestimmen, was genau geplottet wird. Beim letzten *ggplot*-Barplot mit *geom_bar()* haben wir noch mit dem Dataframe selbst gearbeitet, haben also quasi *ggplot* das Auszählen der Daten überlassen. Deshalb mussten wir nur angeben, was auf der x-Achse geplottet werden soll, denn auf der y-Achse wurden automatisch die Frequenzen (*counts*) dargestellt. Das ist jetzt anders, denn wir arbeiten nun mit aggregierten Daten, haben also das Auszählen schon selbst übernommen. Deshalb geben wir an, was auf der x-Achse geplottet werden soll (Die Spalte *Genre* aus dem Dataframe *mf1*) und was auf der y-Achse (die *Freq*-Spalte). Zudem geben wir an, dass die *df*-Spalte aus dem Dataframe *mf1* die farbliche Füllung bestimmen soll. Anstelle von *geom_bar()* verwenden wir diesmal *geom_col()*. Das ist quasi das Äquivalent für *geom_bar()* für bereits aggregierte Daten, wie Sie auch aus der Hilfefunktion (?*geom_bar*) erfahren können:

`geom_bar` makes the height of the bar proportional to the number of cases in each group [...]. If you want the heights of the bars to represent values in the data, use `geom_col` instead.

Alternativ kann man auch `geom_bar(stat="identity")` verwenden und kommt zum gleichen Ergebnis.



Nun wollen wir noch die absoluten Frequenzen darstellen, indem wir Zahlen auf die Balken schreiben. Dafür müssen wir bei den *aesthetics* das *label*-Argument spezifizieren und zudem die `geom_text()`-Funktion hinzufügen:

```
ggplot(mf1, aes(x = Genre, y = Rel_Freq, fill = df, label = Freq)) +
  geom_col() +
  geom_text(data = subset(mf1, Freq>0),
            position=position_stack(vjust=0.5))
```

label = Freq dürfte mittlerweile selbsterklärend sein: Wir geben an, dass die Informationen für das Plotting der Labels aus der Spalte *Freq* des Dataframes *mf1* genommen werden sollen. Die einzelnen Argumente der `geom_text`-Funktion bedürfen jedoch noch der Erläuterung. Zunächst stelle ich hier mit der `subset`-Funktion sicher, dass nur Daten mit einer absoluten Frequenz größer als null geplottet werden, da es sonst aussieht wie im rechten Panel von Fig. 12 unten, wo die 0 im dritten Balken doch eher störend ist.

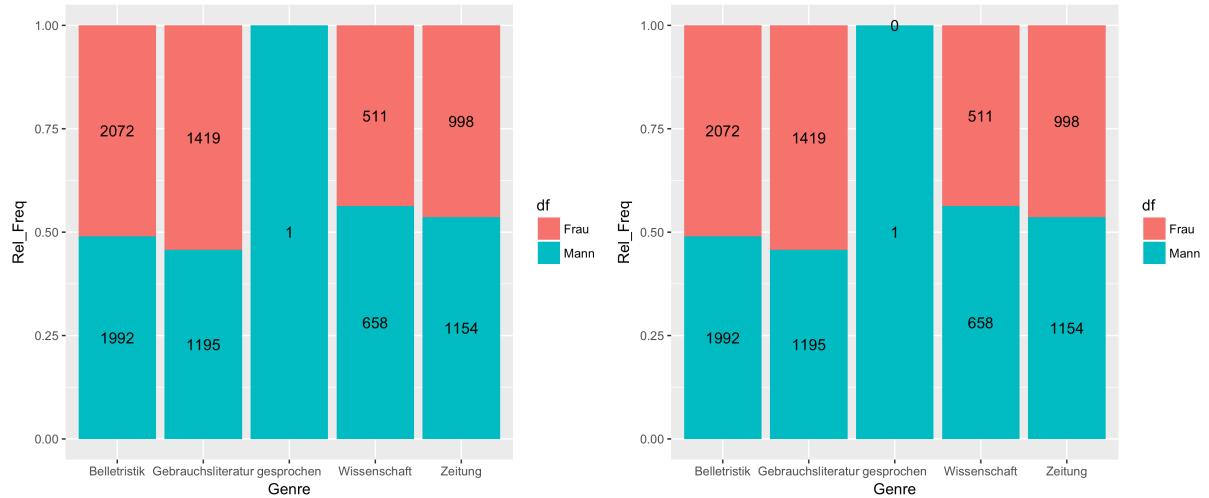
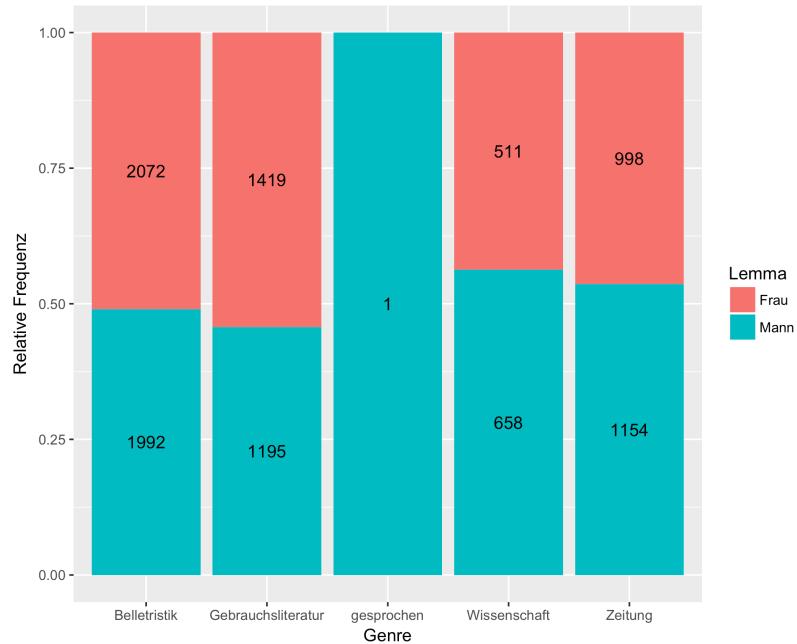


Fig. 12: Stacked Barplot mit absoluten Frequenzen, links: mit Frequenzbeschränkung > 0 , rechts: ohne Frequenzbeschränkung.

Nun können wir den Plot noch etwas verfeinern, indem wir die Beschriftung der y-Achse ändern (das haben wir oben bereits kennengelernt) und den Titel der Legende (das ist neu und ein bisschen umständlich):

```
ggplot(mf1, aes(x = Genre, y = Rel_Freq, fill = df, label = Freq)) +
  geom_col() +
  geom_text(data = subset(mf1, Freq>0),
            position=position_stack(vjust=0.5)) +
  ylab("Relative Frequenz") +
  guides(fill=guide_legend(title="Lemma"))
```

In der letzten Zeile zeigt sich wieder die für einige BenutzerInnen zunächst etwas kontraintuitive Kombination von Element und Elementfunktion, die für *ggplot* so charakteristisch ist. Mit *guides* können wir für die einzelnen Elemente, die wir geplottet haben, Legenden einfügen oder unterdrücken (Letzteres haben wir weiter oben schon gemacht). Mit *fill* geben wir an, worauf sich die Legende bezieht, nämlich auf die Füllfarben. Mit der Elementfunktion *guide_legend()* können wir nun das Aussehen der Legende manipulieren und z.B. den Titel ändern, weil der Spaltenname *df* doch eher wenig aussagekräftig ist. Hier habe ich mich für *Lemma* entschieden. Der finale Plot sieht dann so aus:



Boxplots

Wie oben bereits erwähnt, genießen Barplots teilweise einen etwas zweifelhaften Ruf, weil sie die in den Daten vorhandenen Informationen oft genug nur unvollständig wiedergeben.
Alternative Visualisierungsvarianten

2.3 Einfache statistische Tests

Die Visualisierung ist ein wichtiger und zumeist unentbehrlicher erster Schritt bei der Datenauswertung. Allerdings fängt hier die statistische Analyse erst an, zumal Visualisierungen bisweilen suggestiv sein können. So suggerieren sehr viele der Barplots mit relativen Frequenzen, die wir oben erstellt haben, dass die Verteilung von *Frau* und *Mann* nach attributivem Adjektiv in gesprochenen Texten völlig anders ist als in geschriebenen, nämlich dahingehend, dass sich in geschriebenen Textsorten ADJA+*Mann* und ADJA+*Frau* mehr oder weniger die Waage halten, während im gesprochenen Teilkorpus ausschließlich ADJA+*Mann* vorkommt. Nun braucht man keine besonders tiefgehenden statistischen Kenntnisse, um zu dem Schluss zu kommen, dass die Daten aus dem gesprochenen Korpus mit nur einem einzigen Beleg nicht wirklich belastbar sind. In vielen anderen Fällen ist jedoch weniger klar zu entscheiden, ob ein Unterschied in der Stichprobe dem Zufall zuzuschreiben ist oder tatsächlich auf einen tatsächlichen Unterschied in der Grundgesamtheit schließen lässt. Hier kommen statistische Tests ins Spiel.

Im Folgenden werde ich lediglich für einige sehr grundlegende statistische Tests zeigen, wie man sie in R durchführen kann. Dabei kann ich nur am Rande auf die Theorie hinter den Tests bzw. hinter hypothesentestender Statistik allgemein eingehen; um sich tiefer einzulesen, sollte man hier zu einschlägigen Einführungen greifen.

Auch werde ich mich auf Beispiele dafür beschränken, wie man auf signifikante Unterschiede testet. Oft genug wollen wir auch auf signifikante Zusammenhänge testen, z.B. mit linearen Modellen; dafür empfehle ich die exzellenten Tutorials von Bodo Winter (Winter 2013).

2.3.1 Vorab: Skalenniveaus

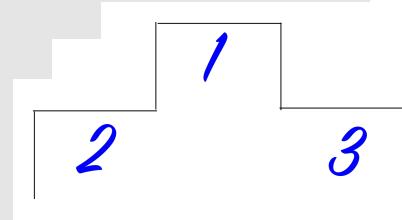
Auch wenn wir uns im Folgenden nur auf sehr basaler Ebene mit statistischen Tests beschäftigen werden, kommen wir nicht umhin, zunächst kurz etwas Theorie abzuhandeln und einen Blick auf die sog. Skalenniveaus zu werfen. Die Wahl statistischer Tests hängt nämlich immer davon ab, wie die entsprechenden Daten skaliert sind: Manche Daten kann man, salopp gesagt, in „Schubladen“ stecken, etwa „ja/nein“, „deutsch/englisch/französisch“. Das sind kategoriale Variablen. Dazu gehören nominal und ordinal skalierte Variablen (s. Liste unten). Andere Variablen lassen sich nicht in Schubladen stecken, sondern sind kontinuierlich; man spricht auch von kardinal skalierten Daten. Hier unterscheidet man zwischen intervall- und verhältnisskalierten Daten je nachdem, ob es einen natürlichen Nullpunkt gibt oder nur einen willkürlich festgelegten Nullpunkt. In vielen Fällen kann man übrigens darüber streiten, welche Skala die richtige ist: Während z.B. Schulnoten relativ klar der Ordinalskala zuzuordnen sind, kann man bei Punkten, die z.B. auf eine Prüfung vergeben werden, darüber diskutieren, ob sie ordinal-, intervall- oder verhältnisskaliert sind (teilweise natürlich auch abhängig von der Vergabepraxis; wenn ich Punkte nach Gutdünken vergabe, sind sie ordinal skaliert; wenn ich für jede richtig beantwortete Frage einen Punkt vergabe und das dann am Ende summiere, haben wir es schon eher mit verhältnisskalierten Daten zu tun).

Nominalskala

- Klassifizierung ohne Ordnungsrelation
- z.B. Familienstand: ledig, verheiratet, verwitwet, geschieden
- keine "Hierarchie": ledig ist nicht "besser" oder "größer" als verheiratet, geschweige denn "halb so gut" oder "doppelt so gut" o.ä.
- Auch **binäre** Variablen sind nominalskaliert, z.B. "lebendig" vs. "tot"

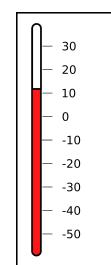
Ordinalskala

- Ordnungsrelation: z.B. Gold > Silber > Bronze
- keine Angaben, **um wie viel** Gold "besser" ist als Silber, Silber besser als Bronze etc.



Intervallskala

- metrische Skala: Intervalle zwischen einzelnen Punkten sind gleich groß
- z.B. Temperatur: Abstand zwischen 10°C und 20°C so groß wie zwischen 20°C und 30°C
- Jedoch: 40°C ist **nicht** doppelt so warm wie 20°C und 20°C **nicht** viermal so warm wie 5°C!
- Grund: Nullpunkt willkürlich festgelegt



Verhältnisskala

- alle intervallskalierten Variablen mit **natürlichem Nullpunkt**
- z.B. Lebensalter, Temperatur in Kelvin (beginnt mit -273,15°C, dem absoluten Nullpunkt)

2.3.2 Chi-Quadrat-Test

Ein sehr einfacher Test, den man auch z.B. in Seminararbeiten gut anwenden kann, ist der Chi-Quadrat-Test. Dabei handelt es sich um einen Kreuztabellentest, wobei beide Variablen, die quasi „über Kreuz“ gerechnet werden, kategorial sein müssen. Der Chi-Quadrat-Test lässt sich mit einem einfachen Beispiel illustrieren (aus Hartmann 2018, Kap. 7): Angenommen, an Ihrer Universität werden zwei Kurse zur Einführung in die historische Sprachwissenschaft angeboten. Nehmen wir, natürlich idealisierend, an, dass der einzige Unterschied zwischen den beiden Kursen der Dozent ist: Sie werden von zwei verschiedenen Dozenten geleitet, aber das Themenspektrum ist das gleiche, die Zusammensetzung der Studierenden ist im Blick auf ihre Vorkenntnisse, ihre Intelligenz, ihre Herkunft usw. die gleiche, die Rahmenbedingungen sind die gleichen, und am Ende schreiben alle denselben Test, der von nicht von den beiden Dozenten, sondern von einer dritten, unabhängigen Instanz bewertet wird. Am Ende ergibt sich folgendes Bild: In Kurs A bestehen 25 von 30 Studierenden die Prüfung, 5 fallen durch. In Kurs B bestehen nur 17 von 30 Studierenden, 13 hingegen nicht. Diese Ergebnisse lassen sich in einer Vier-Felder-Kontingenztabelle wie Tab. 1 anordnen.

	Kurs A	Kurs B
bestanden	25	17
nicht bestanden	5	13

Tab. 1: Kontingenztabelle zum fiktiven Beispiel „Prüfung in historischer Sprachwissenschaft“.

Auf den ersten Blick wird klar, dass in Kurs B mehr Studierende durchgefallen sind als in Kurs A. Aber heißt das, dass der Dozent in Kurs A schlechtere Arbeit geleistet hat – oder könnte das Ergebnis durch Zufall zustandegekommen sein, etwa weil in Kurs B gerade sieben Studierende mehr als in Kurs A einen schlechten Tag erwischt hatten? Mit Hilfe statistischer Methoden lässt sich herausfinden, wie wahrscheinlich es ist, dass eine Verteilung wie diejenige, die wir beobachten, durch Zufall zustandekommt.

Das nutzen Mehrfeldertests wie der Chi-Quadrat-Test, die Auskunft darüber geben, wie wahrscheinlich es ist, dass eine Verteilung z.B. in einer 2x2-Tabelle wie Tab. 1 zustandekommt. Dafür werden die beobachteten Werte verglichen mit den Werten, die man bei einer Zufallsverteilung erwarten würde. In unserem Beispiel etwa würde man bei einer Zufallsverteilung erwarten, dass das Verhältnis zwischen Studierenden, die die Prüfung bestehen, und Studierenden, die die Prüfung nicht bestehen, in beiden Kursen gleich ist. Um diesen Erwartungswert zu berechnen, werfen wir quasi die beiden Gruppen zusammen und teilen sie so auf, dass wir in beiden Gruppen gleich viele „Besteher“ und „Nichtbesteher“ haben. Mathematisch erreichen wir das, indem wir für jede Zelle die Zeilensumme mit der Spaltensumme multiplizieren und sie durch die Gesamtsumme, die sich aus allen vier Zellen ergibt, teilen, also z.B. für die erste Zelle: $(25+17)*(25+5) / (25+5+17+13)$. Wiederholen wir das für die drei anderen Zellen, ergibt sich das Bild in Tab. 2.

	Kurs A	Kurs B
bestanden	21	21
nicht bestanden	9	9

Tab. 2: Erwartete Frequenzen bei Zufallsverteilung.

Der Chi-Quadrat-Wert wird berechnet, indem man in jeder Zelle den erwarteten Wert vom beobachteten Wert abzieht, das Resultat quadriert, es durch den erwarteten Wert teilt und die vier Werte, die sich dadurch ergeben (einer für jede Zelle) aufsummiert. Für die erste Zelle

wäre das also z.B. $(25-21)^2 / 21$. Insgesamt ergibt sich für unser Beispiel so ein Chi-Quadrat-Wert von 3,89. Dieser Wert kann dann mit einer Wahrscheinlichkeitsverteilung abgeglichen werden, sodass man mit Hilfe des Chi-Quadrat-Werts (sowie der sog. Freiheitsgrade; für eine gute Erklärung dieses Konzepts vgl. z.B. Field et al. 2012: 38) einen Wahrscheinlichkeitswert ermitteln kann. Früher musste man das noch manuell mit Hilfe von Tabellen tun (wie sie z.B. in der lesewerten Einführung von Butler 1985 abgedruckt sind), heute kann man dafür auf Statistikprogramme wie R zurückgreifen. Konkret kann man dort die Funktion *chisq.test* verwenden (der Code findet sich im Skript *statistische_tests* im Ordner *R_Einstieg*):

```
# fiktiven Datensatz generieren
df <- data.frame(KURS_A=c(BESTANDEN=25, NICHT_BESTANDEN=5),
                  KURS_B=c(BESTANDEN=17, NICHT_BESTANDEN=13))
chisq.test(as.matrix(df))
```

Mit diesem Code können wir sowohl den Chi-Quadrat-Wert als auch den *p*-Wert, also den Wahrscheinlichkeitswert, einfach anhand der Daten aus Tab. 1 herausfinden. In unserem Beispiel ergibt sich ein Wert von $p = 0,049$. Dieser Wert liegt knapp unter der häufig verwendeten (natürlich willkürlichen!) Signifikanzschwelle von 0,05. Das bedeutet, dass wir die Nullhypothese, dass die Verteilung durch Zufall zustandegekommen ist, zurückweisen können. Das heißt natürlich nicht, dass die Verteilung nicht trotzdem durch Zufall zustande gekommen sein kann. Somit können wir die Schuld zwar mit einiger Gewissheit auf den Dozenten schieben, aber eben nicht mit absoluter Sicherheit.

Weiterhin ist zu bedenken, dass das Ergebnis des Chi-Quadrat-Tests von der Stichprobengröße abhängt – wenn wir eine zehnmal größere Stichprobe mit genau der gleichen Verteilung haben, ist der *p*-Wert plötzlich hochsignifikant:

```
df2 <- data.frame(KURS_A=c(BESTANDEN=250, NICHT_BESTANDEN=50),
                     KURS_B=c(BESTANDEN=170, NICHT_BESTANDEN=130))
chisq.test(as.matrix(df2))
```

Hier zeigt R den *p*-Wert 1.96E-12 an; das ist die sog. wissenschaftliche Notation für 1.96 mal 10^{-12} , also 0.0000000000196. Deshalb ist es wichtig, beim Chi-Quadrat-Test – und auch bei anderen Tests, die sich auf Kontingenztabellen stützen, etwa dem Fisher Exact Test – nicht nur auf den *p*-Wert zu schauen, sondern auch die Effektstärke zu berücksichtigen. Um die Effektstärke *unabhängig* von der Stichprobengröße zu quantifizieren, greift man zumeist auf den Korrelationskoeffizienten Phi zurück (wenn der Kreuztabellentest nicht eine 2x2-Tabelle, sondern z.B. eine 2x3-Tabelle zugrundeliegt, dann spricht man von Cramer's V; Phi und Cramer's V liegt jedoch die gleiche Formel zugrunde, vgl. Gries 2013: 186).

Glücklicherweise kann man sich mit der Funktion *assocstats()* aus dem Paket *vcd* die Effektstärke zusätzlich zum Chi-Quadrat-Wert und zum *p*-Wert ausgeben lassen; die Funktion verlangt allerdings eine Matrix als Input, sodass wir unseren Dataframe mit der Funktion *as.matrix* transformieren müssen. Wir sehen, dass sich die Effektstärken von *df* und *df2* (wir erinnern uns: Letzterer ist der Dataframe mit den sehr viel größeren Zahlen, die zu einem sehr viel niedrigeren *p*-Wert führen) nicht unterscheiden:

```
library(vcd)
assocstats(as.matrix(df))
assocstats(as.matrix(df2))
```

Hier bekommen wir in beiden Fällen einen Phi-Wert von 2.9. Praktisch an *assocstats* ist überdies, dass es auch die Log-Likelihood-Ratio mit ausgibt, ein anderes Assoziationsmaß. Ein weiterer verbreiteter Kreuztabellentest ist der Fisher Exact Test, den man in R mit *fisher.test()* ausführen kann. Einen Vergleich der unterschiedlichen Assoziationsmaße – jeweils mit Blick auf Assoziations- und Dissoziationsmuster bei Wort- und n-Gramm-

Frequenzdistributionen – bieten z.B. Evert (2005), Wiechmann (2008) und Bubenhofer (2009).

2.3.3 *t*-Test und Mann-Whitney U-Test (= Wilcoxon Rank-Sum Test)

Beim Chi-Quadrat-Test haben wir es mit zwei kategorialen Variablen zu tun, in unserem konkreten Beispiel sogar mit zwei binären: Für alle Studierenden gilt „Kurs A = ja“ oder „Kurs A = nein“, und für alle gilt „bestanden = ja“ oder „bestanden = nein“. In vielen Fällen wollen wir aber etwas differenziertere Daten differenzieren (und damit: Daten auf anderen Skalenniveaus). Gehen wir jetzt also davon aus, dass die Studierenden in Kurs A und Kurs B Punkte auf ihre Arbeiten bekommen haben. Gehen wir weiterhin davon aus, dass die Punkte so vergeben wurden, dass sie mindestens intervallskaliert sind - denn die Tests, mit denen wir uns nun beschäftigen, vergleichen Mittelwerte, und von ordinalskalierten Variablen lässt sich kein sinnvoller Mittelwert errechnen (auch wenn das bei Schulnoten, die ordinal skaliert sind, in der Praxis oft gemacht wird).

Ähnlich wie bei den Tests oben wollen wir auch hier wissen, ob zwischen den beiden Gruppen ein signifikanter Unterschied besteht. Dafür wollen wir, wie bereits gesagt, die Mittelwerte vergleichen. Viele statistische Tests sind allerdings an bestimmte Voraussetzungen geknüpft. Insbesondere ist zu unterscheiden zwischen parametrischen und non-parametrischen oder verteilungsfreien Tests. Erstere sind an bestimmte Verteilungen (z.B. die Normalverteilung) geknüpft und setzen daher voraus, dass die Daten dieser Verteilung entsprechen (vgl. ausführlich Meindl 2011: 159–162). Ist das nicht der Fall oder haben wir es mit kleinen Datenmengen (meist definiert als $n < 30$) zu tun, dann müssen wir auf die etwas weniger robusten verteilungsfreien Testverfahren zurückgreifen.

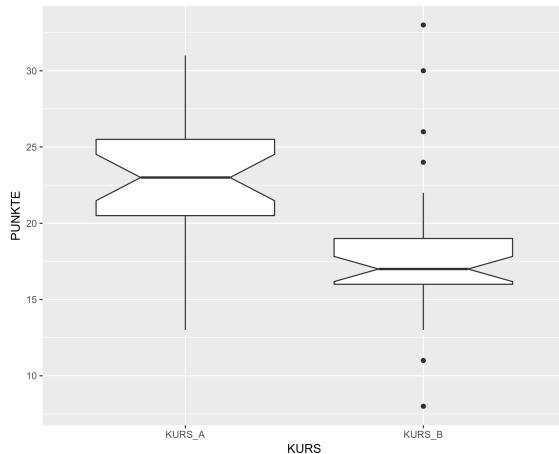
Wenn wir Mittelwerte vergleichen, können wir zum einen auf den parametrischen *t*-Test, zum anderen auf den non-parametrischen Mann-Whitney U-Test zurückgreifen.

Im Folgenden generieren wir zunächst wieder fiktive Daten für unsere beiden Kurse - den Code, der folgt, müssen Sie nicht verstehen, aber wenn Sie ihn verstehen, umso besser... Das `set.seed()` am Anfang dient übrigens dazu, sicherzustellen, dass immer die gleichen Zufallsdaten generiert werden, d.h. Sie werden die gleichen Ergebnisse bekommen wie ich.

```
# fiktive Daten generieren
set.seed(1138)
df <- round(rnorm(60, mean = 20, sd = 5))
df <- sort(df)
df <- as.data.frame(df)
df$KURS <- c(sample(c("KURS_A", "KURS_B"), 30, prob = c(0.2,0.8),
                  replace = T),
              sample(c("KURS_A", "KURS_B"), 30, prob = c(0.8,0.2),
                  replace = T))
colnames(df) <- c("PUNKTE", "KURS")
```

Da Visualisierung, wie wir oben gesehen haben, immer sehr wichtig ist, plotten wir zunächst die Daten mit `ggplot2`:

```
# als Boxplot:
ggplot(df, aes(x = KURS, y = PUNKTE)) + geom_boxplot(notch = TRUE)
```



Die Grafik zeigt, dass Kurs B in Sachen Punktzahl deutlich schlechter abschneidet Kurs A - aber auch *signifikant* schlechter? Um das herauszufinden, wollen wir vorzugsweise den t-Test verwenden. Dieser setzt voraus, dass die Daten normalverteilt sind. Das wiederum können wir mit dem Shapiro-Wilk Test herausfinden. Dieser geht von der Nullhypothese aus, dass die Daten aus einer Grundgesamtheit stammen, die der Normalverteilung entspricht.

```
> shapiro.test(df$PUNKTE)
Shapiro-Wilk normality test
data: df$PUNKTE
W = 0.98221, p-value = 0.5286
```

Wir haben einen sehr hohen p-Wert, können also die Nullhypothese nicht ablehnen. Anders als in vielen anderen Fällen wollen wir das hier auch gar nicht: Vielmehr freuen wir uns, dass die Daten offenbar normalverteilt sind und wir den t-Test somit anwenden können. Nun gibt es zwei t-Tests, einen für abhängige und einen für unabhängige Stichproben. Wir haben es mit zwei unterschiedlichen Studierendengruppen zu tun, können also den t-Test für unabhängige Stichproben verwenden. Hätten wir es hingegen z.B. mit zweimal der gleichen Studierendengruppe zu tun, die den Test einmal hellwach morgens um elf, einmal schlaftrunken nachts um vier macht, dann müssten wir den t-Test für abhängige Stichproben verwenden.⁹

Die entsprechende Funktion in R nimmt als Argumente zwei Vektoren, die jeweils die Werte der beiden Gruppen, die wir miteinander vergleichen wollen, enthalten:

```
t.test(subset(df, KURS == "KURS_A")$PUNKTE,
       subset(df, KURS == "KURS_B")$PUNKTE)
```

Wir sehen: Der p-Wert ist enorm niedrig, der Unterschied ist hochsignifikant. Als nächstes generieren wir einen fiktiven Datensatz, der der Normalverteilung *nicht* entspricht:

⁹ Dafür müssten wir in die *t.test*-Funktion unten einfach nur das Argument *paired = TRUE* einfügen. Die Unabhängigkeit der Stichproben ist übrigens bei sehr vielen statistischen Tests eine wichtige Voraussetzung!

```
# fiktive Daten generieren
set.seed(1138)
df <- data.frame(KURS_A = c(round(rnorm(25, mean = 10, sd = 5)),
                             round(rnorm(5, mean = 25, sd = 5))),
                  KURS_B = c(abs(round(rnorm(17, mean = 10, sd = 5))),
                             abs(round(rnorm(13, mean = 25, sd = 5)))))
```

Erneut wollen wir die Daten zunächst visualisieren, wofür wir diesmal den Dataframe ins „lange“ Format überführen müssen (beim Beispiel oben haben wir ihn direkt im langen Format erstellt):

```
# ins lange Format ueberfuehren:
df2 <- melt(df)

# als Boxplot:
ggplot(df2, aes(x = variable, y = value)) + geom_boxplot(notch = TRUE)
```

Erneut testen wir, ob Normalverteilung vorliegt:

```
> shapiro.test(df2$value)
Shapiro-Wilk normality test
data: df2$value
W = 0.92287, p-value = 0.001002
```

Diesmal liegt der p-Wert deutlich unter der üblichen Schwelle von 0,05, weshalb wir die Nullhypothese ablehnen und davon ausgehen können, dass die Daten *nicht* normalverteilt sind. Deshalb müssen wir diesmal auf den verteilungsfreien Mann-Whitney U-Test ausweichen. Eine Funktion dafür werden Sie in R wahrscheinlich vergeblich suchen; aber zum Glück ist der Mann-Whitney U-Test äquivalent mit dem Wilcoxon Rank-Sum Test (vgl. Field et al. 2013: 655), und der ist in R implementiert. Die entsprechende Funktion funktioniert ebenso wie die für den t-Test, wir brauchen also je einen Vektor mit den Werten der beiden Gruppen, die wir vergleichen:

```
# Wilcoxon Rank-Sum Test
wilcox.test(subset(df2, variable=="KURS_A")$value,
            subset(df2, variable=="KURS_B")$value, exact = F)
```

Diesmal ist der p-Wert mit 0,34 recht hoch; somit können wir die Nullhypothese, dass die Verteilung durch Zufall zustandegekommen ist, nicht ablehnen.

Zum Weiterlesen

In diesem Tutorial bin ich nur sehr oberflächlich auf viele Aspekte eingegangen - wer sich tiefer einlesen möchte, kann z.B. zu Gries (2013) und Levshina (2015) greifen. Das Gries-Buch eignet sich gut zum Durcharbeiten; leider ist es etwas unübersichtlich und hat keinen Index, weshalb es schwer fällt, einzelne Aspekte (z.B. bestimmte statistische Tests) nachzuschlagen. Levshina ist etwas übersichtlicher, stellt aber auch viele Methoden vor, die bei Gries nicht vorkommen, was im Umkehrschluss heißt, dass die grundlegenden statistischen Tests insgesamt etwas kürzer abgehandelt werden als bei Gries. Nicht speziell für Linguisten, aber sehr ausführlich (teils zu ausführlich) und recht unterhaltsam ist Field et al. (2013) - hier lernt man nicht nur viel über Statistik, sondern auch über Populärkultur. Zum Beispiel habe ich nur durch diese Einführung erfahren, dass es in Großbritannien seit 2002 die Show „I'm a celebrity - get me out of here!“ gibt. Dass eine Sendung, in der es vor allem um ekliges Essen geht, aus Großbritannien kommen muss, hätte ich mir ja denken können; aber ich hatte zuvor immer gedacht, der merkwürdige Titel sei eine deutsche Erfindung. Außerdem lässt sich das Buch dank seines Umfangs gut als Bremsklotz oder Briefbeschwerer zweckentfremden.

Literatur

- Bubenhofer, Noah. 2009. *Sprachgebrauchsmuster. Korpuslinguistik als Methode der Diskurs- und Kulturanalyse*. Berlin, Boston: de Gruyter.
- Chang, Winston. 2013. *R graphics cookbook*. Cambridge: O'Reilly.
- Evert, Stefan. 2005. *The statistics of word cooccurrences. Word pairs and collocations*. Stuttgart: Institut für maschinelle Sprachverarbeitung.
- Gries, Stefan Th. & Anatol Stefanowitsch. 2004. Extending Collostructional Analysis: A Corpus-Based Perspective on “Alternations.” *International Journal of Corpus Linguistics* 9(1). 97–129.
- Flach, Susanne. 2017. collostructions: An R Implementation for the Family of Collostructional Methods. R package version 0.0.10. www.bit.ly/sflach
- Hartmann, Stefan. 2018. Deutsche Sprachgeschichte. Grundzüge und Methoden. Tübingen: Narr.
- Levshina, Natalia. 2015. *How to do linguistics with R. Data exploration and statistical analysis*. Amsterdam, Philadelphia: John Benjamins.
- Meindl, Claudia. 2011. *Methodik für Linguisten: Eine Einführung in Statistik und Versuchsplanung*. Tübingen: Narr.
- Murrell, Paul. 2006. *R graphics*. Boca Raton: CRC.
- Wiechmann, Daniel. 2008. On the computation of collostruction strength: Testing measures of association as expressions of lexical bias. *Corpus Linguistics and Linguistic Theory* 4(2). 253–290.
- Wickham, Hadley. 2015. *Advanced R*. Boca Raton: CRC Press.
- Wickham, Hadley. 2016. *ggplot2. Elegant graphics for data analysis*. 2nd ed. Berlin: Springer.
- Winter, Bodo. 2013. Linear models and linear mixed effects models in R with linguistic applications. arXiv:1308.5499. [<http://arxiv.org/pdf/1308.5499.pdf>]