Computational Methods in der politischen Kommunikationsforschung

Methodische Vertiefung: Computational Methods mit R und RStudio

Julian Unkel

Contents

Einführung

Zuletzt aktualisiert: 2020-11-18 17:41:16. Dies ist ein *Work-in-Progress* und wird laufend aktualisiert.

Seminarinformationen

- Dozent: Julian Unkel, unkel@ifkw.lmu.de
- Zeit und Ort: Donnerstags, 12-14 Uhr, ~~Oe 057~~ (bis auf weiteres findet das Seminar rein digital statt)
- Moodle: https://moodle.lmu.de/course/view.php?id=8250

Ablauf des Kurses

Aufgrund der aktuellen Situation wird dieses Seminar in einen Online-Kurs überführt. Alle Seminarinhalte werden in Textform aufbereitet und nach und nach diesem Online-Kurs hinzugefügt. Auf Basis des Kurses sollen die Seminarinhalte selbstständig und mit weitestgehend eigenem Lerntempo erarbeitet werden.

In jedem Kapitel werden hierzu zunächst die wesentlichen Konzepte und Inhalte erläutert. Jedes Kapitel schließt mit einigen Übungsaufgaben, die über Moodle abgegeben werden müssen. Deadlines für die Übungsaufgaben werden ebenfalls über Moodle kommuniziert, Lösungen im Anschluss an die Deadlines im Kurs hinzugefügt.

Jeden Donnerstag zum regulären Seminartermin findet von 12-14 Uhr eine Online-Sprechstunde via Zoom statt. Hier können Fragen zu den Seminarinhalten, Übungsaufgaben etc. gestellt und diskutiert werden.

In Moodle stehen zudem zwei Foren zur Verfügung, in dem Sie 1) allgemeine Fragen zu R und RStudio sowie 2) spezifische Fragen / alternative Lösungen zu den Übungsaufgaben (vor-)stellen und diskutieren können. Scheuen Sie sich bitte nicht, auch selbst auf Fragen und Probleme von Kommiliton*innen einzugehen.

Neben den regulären Übungsaufgaben werden Sie bisweilen auch optionale, besonders knifflige Aufgaben vorfinden, die ich in der Sprache meiner Ahnen als *Käpseles-Aufgaben* kennzeichnen werde. Diese sind nicht verpflichtend, können Ihnen aber als Gradmesser dienen, ob Sie die jeweiligen Inhalte auch eigenständig und in leicht abgewandelter Form anwenden können.

Motivation und Ziele des Seminars

Das Ziel des Kurses ist es, methodische Kenntnisse zur Anwendung computationaler Methoden zu vermitteln. Hierzu werden wir uns zunächst allgemein mit der Datenbearbeitung und -analyse mit der statistischen Programmiersprache R auseinandersetzen. Es folgen dann spezifischere Verfahren der computationalen Datenerhebung und -analyse.

Dabei stehen insbesondere folgende Inhalte im Vordergrund:

- Einführung in *R* und die Arbeit mit *RStudio*
- Datenmanagement in *R*
- Computationale Datenerhebung mit *R*
- Datenvisualisierung mit *R*
- Automatisierte Inhaltsanalyse mit *R*

Zudem wird darauf eingegangen, wie mittels R und RStudio Kommunikationsforschung transparent, nachvollziehbar und reproduzierbar gestaltet werden kann.

Es werden keine Vorkenntnisse in R vorausgesetzt; die Inhalte der Veranstaltung *15424 Datenanalyse* werden als bekannt vorausgesetzt.

Bevor es jedoch ans Eingemachte geht, ein paar Worte zur Motivation hinter diesem Seminar: Warum lohnt es sich überhaupt, eine Programmiersprache für die quantitativ-wissenschaftliche Arbeit zu lernen? Und warum ausgerechnet R?

Warum also eine Programmiersprache für Datenanalyse lernen?

Wenn Sie bisher Daten statistisch ausgewertet haben, etwa im Rahmen von Forschungsseminaren oder der Bachelorarbeit, wird das in der Regel mit einem Programm mit grafischer Oberfläche erfolgt sein, etwa mit *Microsoft Excel* oder mit *IBM SPSS*. Diese Programme haben viele Vorteile: sie sind meist auf spezifische Funktionen zugeschnitten, in ihrer Aufmachung an typische Computersoftware angepasst und entsprechend intuitiv zu bedienen - ein paar Klicks,

und schon gibt SPSS eine Regressionstabelle mit allen relevanten Informationen aus. Für die meisten Anwendungsfälle im KW-Studium bieten genannten Programme leicht zu erlernende und umzusetzende Lösungen an. Programmiersprachen haben hingegen eine zweifellos höhere Einstiegshürde, deren Bewältigung für viele Anwendungsfälle in der Kommunikationswissenschaft auf den ersten Blick keinen größeren Nutzen verspricht.

Vielleicht ist im Studium aber auch schon eine Situation aufgetreten, in der SPSS keine Hilfe bot. Eine Effektstärke für einen Mittelwertvergleich? Die bietet SPSS zwar in Form von η_p^2 für die ANOVA an, nicht jedoch Cohen's d für den t-Test. Sie haben zusammen mit Komilliton*innen eine Inhaltsanalyse geplant und möchten vorab einen Intercoderreliabilitätstest durchführen? SPSS kennt weder die Reliabilität nach Holsti noch Krippendorff's α. Und auch wenn SPSS Grafiken ausgeben kann, so hat es doch einen Grund, warum man diese selten in wissenschaftlichen Veröffentlichungen (und hoffentlich auch in studentischen Arbeiten) findet.

Benötigt man also eine Funktion, die in der gewählten Softwarelösung nicht vorhanden ist, so muss man auf eine andere ausweichen. Programmiersprachen bieten hier deutlich mehr Flexibilität - ist die gewünschte Funktion nicht vorhanden, so schreibt man sie eben selbst (bzw. hat dies in aller Regel schon jemand anderes, der ebenfalls vor diesem Problem stand, für Sie getan). Dies gilt natürlich umso mehr, je weniger standardisiert die zu analysierenden Daten und gewählten Analyseverfahren sind. Beschäftigen wir uns beispielsweise mit Onlinetexten oder digitalen Spurendaten, dann liegen diese oftmals nicht in vorstrukturierter Form vor, müssen erst über Schnittstellen abgerufen, automatisiert heruntergeladen und/oder für die weitere Nutzung aufbereitet werden. Computationale Analyseverfahren wie beispielsweise Verfahren zur automatisierten Inhaltsanalyse werden beständig weiterentwickelt und angepasst. Die Flexibilität, die skriptbasierte Datenanalyse bietet, ist daher einer der Hauptgründe, warum nicht nur in der Wissenschaft, sondern auch in anderen professionellen Kontexten, etwa der Markt- und Medienforschung, wo Lösungen für vielseitige datenanalytische Fragen gesucht werden, die Bedeutung von Programmiersprachen zur Datenanalyse zunimmt.

Zugleich ist der Einstieg in das Programmieren deutlich einfacher geworden. Für viele Programmiersprachen stehen sogenannte Integrierte Entwicklungsumgebungen (IDEs) zur Verfügung, die mittels grafischer Benutzeroberflächen, intuitiver Bedienung und Hilfswerkzeugen (z. B. der automatischen Vervollständigung von Funktionsnamen) den Umgang mit Programmiersprachen deutlich erleichtern und komfortabler gestalten.

Ein weiterer entscheidender Vorteil der *programmatischen*[1] Datenanalyse ist, dass Skripte und Code alle Analyseschritte nachvollziehbar, transparent

[1] d. h. skript- bzw. codebasiert; im Englischen wird *programmatically* verwendet, um auszudrücken, dass etwas 'durch Code' und nicht durch Klicken von Knöpfen in einem Computerprogramm erfolgt ist, im Deutschen ist diese Wortbedeutung außerhalb von Informatikkreisen (noch) kaum geläufig; siehe auch diese Diskussion zur Wortbedeutung.

und reproduzierbar gestalten (entsprechend wurden Sie in der Datenanalyse-Ausbildung vermutlich auch dazu angehalten, in SPSS stets die Syntax zu nutzen). Einmal durchgeführte Arbeiten können somit jederzeit und problemlos von anderen und auch Ihnen selbst wiederholt und angepasst werden.

Schließlich können auch karriereteчnische Überlegungen eine Rolle spielen. Viele Unternehmen setzen für datenanalytische Tätigkeiten die Kenntnis einer einschlägigen Programmiersprache inzwischen zwingend voraus. Und natürlich spiegelt sich das auch im Gehalt wider: das Vergleichsportal *PayScale* gibt beispielweise für *Data Analysts*, die die statistische Programmiersprache R beherrschen, ein um rund 5.000 US-Dollar höheres Jahresdurchschnittsgehalt an als für diejenigen Data Analysts, die mit SPSS arbeiten.

Warum *R* lernen?

Bisher wurde allgemein von Programmiersprachen gesprochen. In der Datenanalyse-Praxis sind viele unterschiedliche Programmiersprachen gängig, z. B. *Python*, *R*, *SQL* und *Julia*. Wir werden in den kommenden zwei Semestern mit R arbeiten. Dies hat einige Gründe:

- R ist eine speziell auf statistische und datenanalytische Anwendungen ausgelegte Programmiersprache (auch wenn die Anwendungsbereiche inzwischen darüber hinausgehen). Das bedeutet, dass viele gängige statistische Verfahren bereits in der Basis-Version vorhanden sind und ohne weitere Anpassungen genutzt werden können.
- In der *Scientific Community* ist R inzwischen sehr weit verbreitet und wird durch diese kontinuierlich weiterentwickelt. Das bedeutet auch, dass neue Verfahren, sowohl zur Datenerhebung als auch zur Datenanalyse, meist sehr schnell auch in R verfügbar sind.
- Zugleich gibt es durch die weite Verbreitung auch vielzählige Hilfsangebote. In Communities wie Stack Overflow und durch googeln werden Sie für nahezu jedes Problem, das sich Ihnen bei der Arbeit mit R stellt, schnell eine Lösung finden.
- R ist komplett kostenlos und für jedes Betriebssystem verfügbar.
- Mit *RStudio* steht eine ebenfalls kostenfreie IDE zur Verfügung, die die ehemals hohen Einstiegshürden erheblich senkt.
- R und RStudio decken durch Erweiterungen nahezu alle Schritte ab, die für die wissenschaftliche Arbeit erforderlich sind. Das reicht vom Datenabruf aus Befragungssoftware sowie der Datenerhebung durch Programmierschnittstellen oder Web Scraping über die Datenbearbeitung, -bereinigung und -analyse bis hin zur Erstellung von Manuskripten und publikationsfähigen Grafiken. Auch dieser Kurs ist komplett in RStudio erstellt.

Auch wenn sich R in einigen Aspekten von den oben genannten Programmiersprachen unterscheidet, so sind viele der Konzepte, die wir in den kommenden

zwei Semestern lernen werden, auch in anderen Programmiersprachen gleich oder zumindest ähnlich umgesetzt. Ihnen wird es in Zukunft also auch leichter fallen, sich bei Bedarf in andere Programmiersprachen einzuarbeiten.

Hinweise zur Nutzung des Online-Kurses

- In der Onlineversion können Sie mit den Cursortasten ← und → durch die Seiten des Kurses blättern.
- In der oberen Leiste finden Sie einen Download-Knopf, mit dem Sie sich die aktuelle Version des Kurses als *PDF* oder *EPUB* (für E-Reader) herunterladen können. Bitte achten Sie in diesem Fall darauf, regelmäßig die aktuellste und somit vollständigste Version herunterzuladen. Oben auf dieser Seite ist angegeben, wann der Kurs zuletzt aktualisiert wurde.
- Früher oder später wird etwas in Ihrem Code nicht so funktionieren, wie Sie sich das vorstellen oder wünschen. Hier greift die 15-Minuten-Regel: Versuchen Sie zunächst, 15 Minuten lang das Problem selbst zu lösen - in dem Sie das Problem in kleinere Schritte zerlegen, den Code nach Tippfehlern durchsuchen, nochmals Hilfsdokumente konsultieren etc. Sind Sie nach 15 Minuten noch nicht weitergekommen, fragen Sie um Hilfe - z. B. in unseren Moodle-Foren.
- Der Witz, wonach Programmieren zu 70% aus Googeln bestehe, hat einen wahren Kern. Es ist nicht verwerflich, im Internet nach Hilfestellungen und Lösungen zu suchen und Code-Schnipsel von anderen zu verwenden - ganz im Gegenteil, gezieltes Suchen stellt einen wesentlichen Teil der Problemlösekompetenz dar. Auch wenn es jedoch verlockend und einfach erscheinen mag, Code von StackOverflow und vergleichbaren Portalen zu kopieren, sollten Sie immer versuchen, den Code und damit die Lösung auch nachvollziehen zu können.

Beginnen wir mit der Installation von R und RStudio sowie ersten Schritten.

Figure 1: Illustration von @allison_horst: https://twitter.com/allison_horst

Part I

Eine kurze Einführung in R

Chapter 1

Installation und erste Schritte

In diesem Kapitel installieren wir die notwendige Software und machen uns mit der Benutzeroberfläche von RStudio vertraut.

1.1 R installieren

Zunächst benötigen wir natürlich R. Die aktuellste Version erhalten wir immer über CRAN (*C*omprehensive *R A*rchive *N*etwork).

Unter "Download and Install R" wählen wir zunächst unser Betriebssystem. Im Falle von Windows wählen wir zusätzlich auf der folgenden Seite noch "base" (die Basisversion) aus. Es sollte dann ein Download-Link für die aktuellste Version erscheinen (3.6.3, Stand 9. April 2020). Der Installationsprozess selbst läuft wie bei anderer Software auch ab.

Neben einem *Interpreter*, einem Programm, das Code (in diesem Fall also Code, der in R geschrieben wurde) für unseren Computer in ausführbare Befehle übersetzt, umfasst die Installation von R auch schon eine (sehr) rudimentäre grafische Benutzeroberfläche, die aber nur wenig komfortabel und nutzerfreundlich ist. Als nächstes installieren wir daher noch RStudio.

1.2 RStudio installieren

RStudio ist eine grafische Benutzeroberfläche für R, die die Arbeit mit der Programmiersprache deutlich erleichert. Auch RStudio ist für Privatanwender kom-

plett kostenfrei nutzbar. Die aktuellste Version kann über https://rstudio.com/products/rstudio/download/#download heruntergeladen werden.

Für den Rest des Kurses arbeiten wir immer mit RStudio (und nicht direkt mit der Oberfläche von R). Öffnen wir also zum ersten Mal RStudio.

1.3 Die Benutzeroberfläche von RStudio

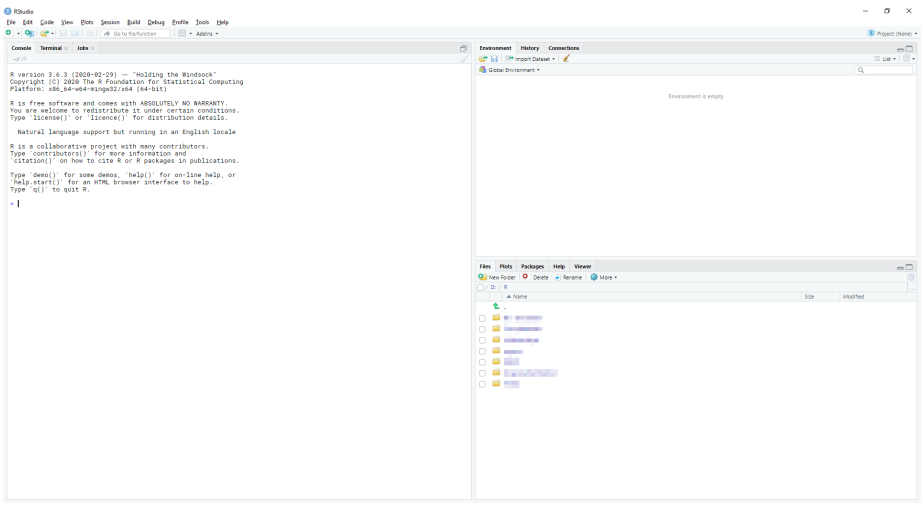


Figure 1.1: Die Benutzeroberfläche von RStudio

Nach dem Starten von RStudio sollte sich das Programm Ihnen wie oben präsentieren - mit einer Dreiteilung in drei abgetrennte Bereiche. Wir beginnen mit dem großen, aktuell noch weitestgehend leeren Bereich auf der linken Seite, der Konsole.

1.3.1 Konsole

Die Konsole ist zugleich das Eingabe- und das Ausgabefenster von R bzw. RStudio. Befehle, die wir hier eingeben, werden durch Druck auf die `Eingabe`/`Enter`-Taste direkt ausgeführt. Die Konsole signalisiert uns, dass sie bereit ist, einen Befehl zu empfangen, durch ein vorangestelltes `>`. Wir können dies mit simplen Berechnungen ausprobieren:

(Zur Darstellung in diesem Kurs: die erste hellgraue Box umfasst hier und im Folgenden jeweils die Befehle, die wir eingeben - in diesem Fall also den Befehl `1 + 2`. Die zweite hellgraue Box enthält dann immer die Ausgabe in der Konsole, gekennzeichnet durch zwei vorangestellte Rautensymbole `##`.)

```
1 + 2
```

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

Die Konsole spuckt also direkt das Ergebnis aus – in diesem Fall `3` – und wartet auf den nächsten Befehl, wieder zu erkennen am `>`. Die `[1]` links neben dem Ergebnis gibt an, dass es sich hierbei um den ersten (und einzigen) Ausgabewert handelt. Wir werden aber noch zahlreiche Befehle kennenlernen, bei denen mehr als nur ein Wert ausgegeben wird.

Mit den Cursortasten `↑` und `↓` können wir in der Konsole durch bisher eingegebene Befehle schalten. Ein Druck auf `↑` sollte also den ersten und bisher einzigen Befehl - `1 + 2` - anzeigen.

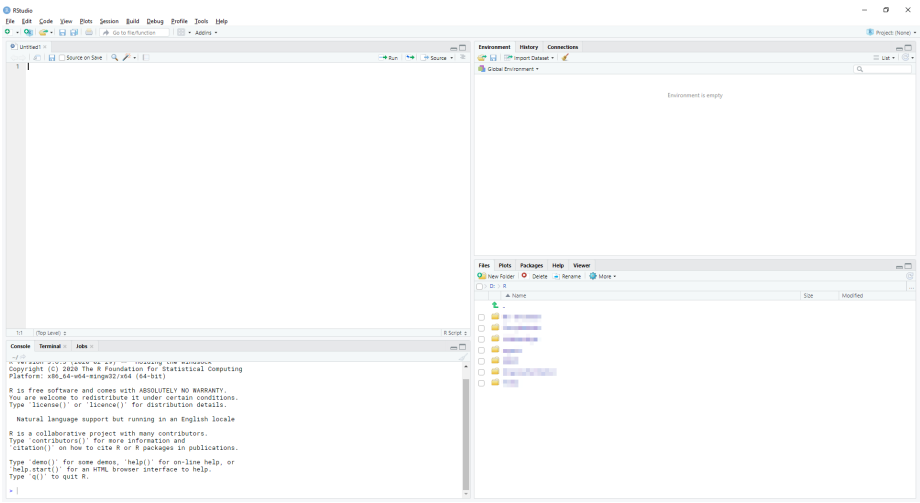
Ist ein Befehl noch nicht vollständig, signalisiert uns dies die Konsole durch ein vorangestelltes `+`. Wir können dies ausprobieren, in dem wir beispielsweise eine unvollständigen Additionsbefehl eingeben: `3 +`. Die Konsole wartet nun auf den restlichen Befehl - in diesem Fall können wir eine weitere Zahl eingeben und den Befehl abschließen. Alternativ können wir den unvollständigen Befehl durch Druck der `ESC`-Taste abbrechen.

In der Praxis passiert dies vor allem, wenn in einem längeren Befehl eine Klammer `)` oder Anführungszeichen `"` fehlt. Sollte die Konsole also einmal die Arbeit verweigern, liegt das oft daran, dass noch ein unvollständiger Befehl vorhanden ist.

Prinzipiell könnten wir alle Arbeitsschritte über die Konsole ausführen. Das ist in der Praxis aber wenig sinnvoll, da wir im Normalfall längere und mehrere Befehle hintereinander ausführen und diese auch festhalten möchten. Wir arbeiten daher mit Skript-Dateien.

1.3.2 R-Skripte

Um eine neue Skriptdatei zu erstellen, klicken wir entweder links oben auf das Symbol mit der leeren Seite und dem grünen Plus und anschließend auf R-Script, auf *File - New File - R-Script* oder drücken die Tastenkombination `Strg/Cmd + Shift + N`. Es sollte sich im links-oberen Bildschirmviertel eine leere Skriptdatei öffnen und unser RStudio-Fenster somit in ein viergeteiltes Layout übergehen:



Hier können wir nun alle Befehle der Reihen der Reihe nach schreiben und gesammelt abspeichern. Einzelne Befehlszeilen lassen sich über die Tastenkombination `Strg/Cmd + Eingabe/Enter` ausführen. Das Ergebnis des Befehls erscheint dann in der Konsole. Wir können auch mehrere Zeilen auf einmal markieren und gemeinsam über dieselbe Tastenkombination ausführen. Schreiben wir z. B. mehrere Rechenoperationen hintereinander, so erscheinen deren Ergebnisse in der Ausführungsreihenfolge in der Konsole:

```
1 + 3
12 * 25
17 / 4
```

```
## [1] 4
## [1] 300
## [1] 4.25
```

Längere Skriptdateien werden schnell unübersichtlich. Wir können aber an jeder Stelle Kommentare einfügen, indem wir eine Raute `#` voran stellen - alles was in dieser Zeile *hinter* dem Symbol steht, wird von R beim Ausführen ignoriert, wir können also sowohl ganze Zeilen *auskommentieren* als auch hinter R-Befehlen eine kurze Erklärung hinzufügen. Außerdem können jederzeit Leerzeilen eingefügt werden, um das Skript etwas aufzulockern:

```
# Zunächst ein wenig Addition
2 + 5
6 + 12

# Dann ein wenig Multiplikation
21 * 35
```

```
2345 * 1.6

# Und zum Schluss etwas komplexere Rechenoperationen
(2 + 3) ^ 3 # Das ^ steht für Exponentiation, hier also 5 hoch 3
```

```
## [1] 7
## [1] 18
## [1] 735
## [1] 3752
## [1] 125
```

Skriptdateien können und sollten natürlich abgespeichert werden - entweder über *File - Save* oder die Tastenkombination `Strg/Cmd + S`. R-Skriptdateien erhalten die Dateiendung `.R`.

1.3.3 Environment

Im rechten oberen Bildschrimbereich öffnet sich standardmäßig das *Environment* an, in dem RStudio alle derzeit angelegten und somit verfügbaren Objekte anzeigt. Mit Objekten werden in uns im nächsten Kapitel genauer auseinandersetzen.

Weitere Registerkarten in diesem Bereich sind die *History* (eine Auflistung sämtlicher ausgeführter Zeilen der aktuellen Sitzung) sowie *Connections* und *Build*, die für uns aber vorerst keine Rolle spielen werden.

1.3.4 Files

Der rechte untere Bildschirmbereich zeigt standardmäßig einen Dateibrowser (*Files*) an, der das aktuelle Arbeitsverzeichnis zeigt. Auch damit setzen wir uns in den kommenden Kapiteln ausführlicher auseinander.

Weitere Registerkarten in diesem Bereich sind:

- *Plots*: hier werden Grafiken angezeigt, wenn wir diese in R erstellen.
- *Packages*: Eine Übersicht aller installieren Packages (kurz gesagt Sammlungen von R-Funktionen, die nicht in der Basisversion enthalten sind). Auch mit Packages beschäftigen wir uns in einem eigenen Kapitel.
- *Help*: Hier wird die Dokumentation einzelner Funktionen angezeigt, sobald wir diese anfordern. Diesen Bereich sehen wir uns an, sobald wir uns mit Funktionen beschäftigen.
- *Viewer*: Hier kann RStudio Webinhalte anzeigen, die mit R-Funktionen erstellt wurden. Dies wird vorab keine Rolle für uns spielen.

1.4 RStudio anpassen

Unter *Tools - Global Options* können wir RStudio nach unseren Wünschen anpassen. Die einzelnen Einstellungsmöglichkeiten sollen hier nicht ausführlich diskutiert werden; hier jedoch einige sinnvolle Einstellungen:

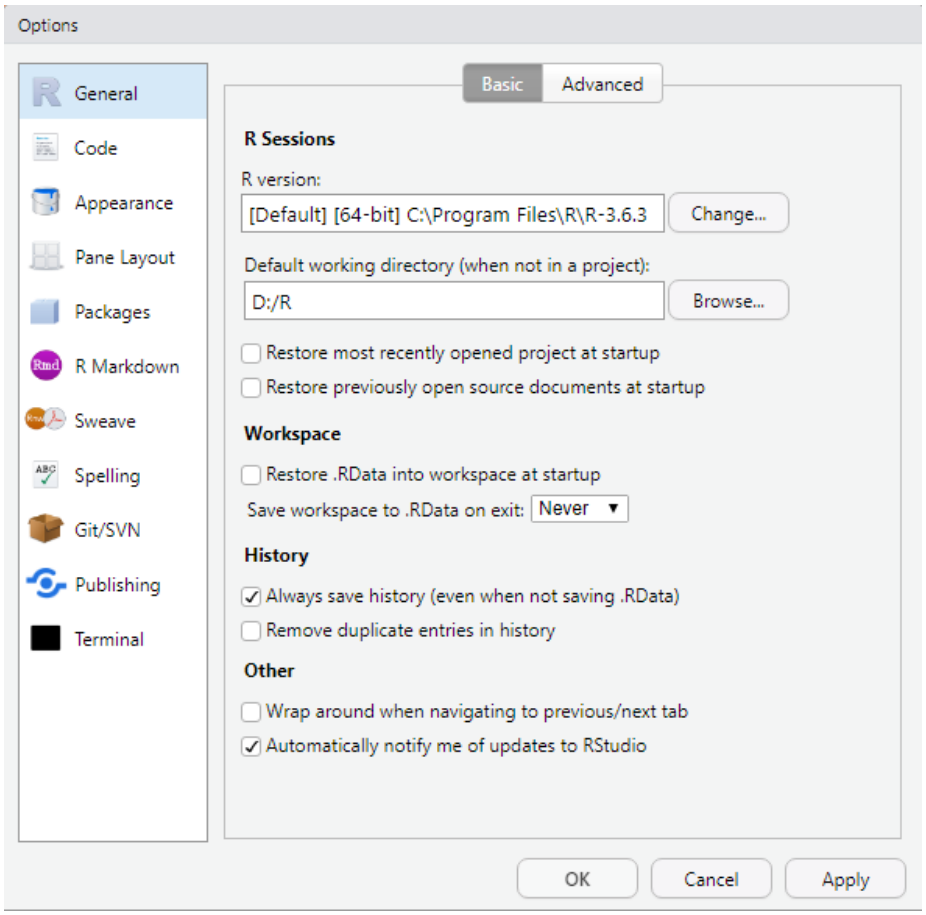


Figure 1.2: Global Options in RStudio

Im Bereich *General* ist es sinnvoll, zwei Anpassungen vorzunehmen. Zum einen können wir unter *R Sessions* ein *Default working directory* (also ein standardmäßiges Arbeitsverzeichnis) einstellen. Dieses Verzeichnis öffnet R dann beim Start automatisch. Hier bietet es sich an, einen eigenen Ordner anzulegen.

Unter *Workspace* entfernen Sie bitte, falls vorhanden, das Häkchen bei *Restore .RData into workspace at startup* und stellen *Save workspace to .RData on exit* auf *Never*. Zwar mag es praktisch erscheinen, dass RStudio automatisch die zuletzt bearbeitete *Session* wiederherstellt, das führt in der Praxis aber gerne

zu Konflikten und Problemen – und letztlich ist es längerfristig auch sinnvoller, sich einen Arbeitsprozess anzueignen, bei dem Skripte schnell den jeweiligen Arbeitsstand wiederherstellen.

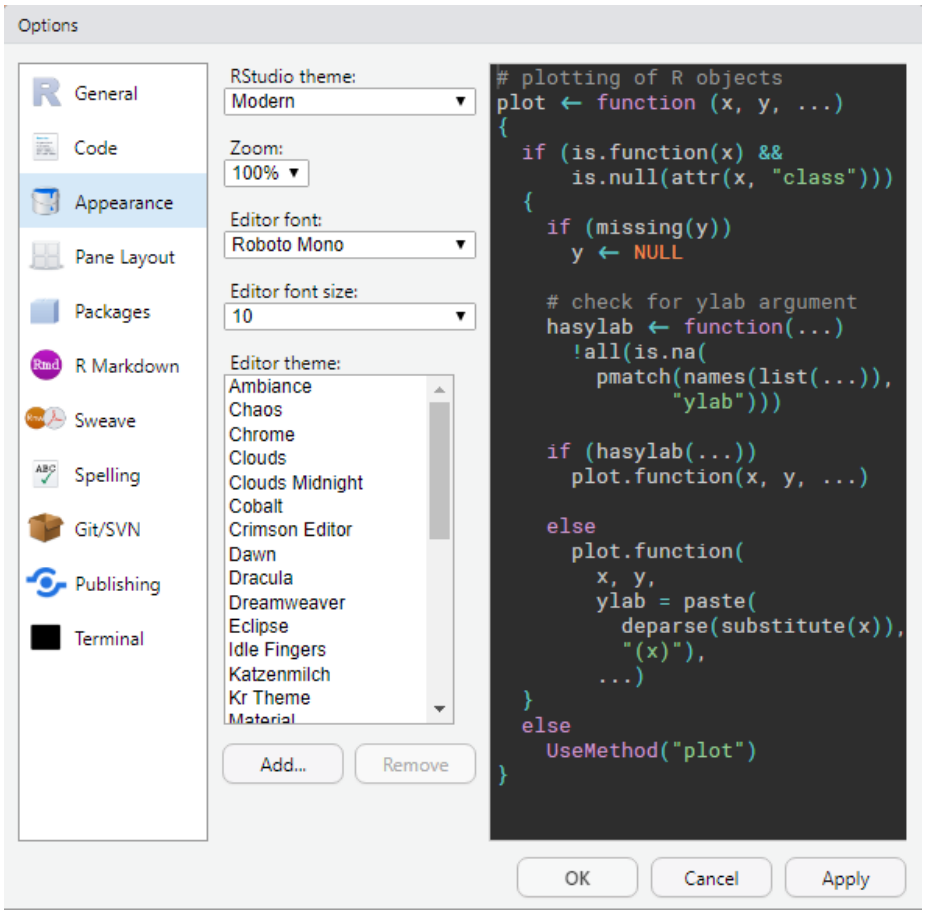


Figure 1.3: Anzeigeeinstellungen in RStudio

Eher Geschmackssache sind die Anzeigeeinstellungen, die Sie unter *Appearance* vornehmen können. Hier können Sie die Schriftart und -größe im Skripteditor einstellen sowie unter verschiedenen *Themes* (Farbschemata) wählen (darunter auch "dunkle" Themes, also solche, die hellen Text auf dunklem Hintergrund bieten). Am besten, Sie probieren hier unterschiedliche Einstellungen aus, bis Sie ein subjektiv angenehmes Anzeigebild von RStudio gefunden haben.

1.5 Übungsaufgaben

Diese Aufgaben sollen Sie lediglich mit den grundlegendsten Funktionen von RStudio vertraut machen. Sie müssen daher keine Dateien abgeben.

Übungsaufgabe 1.1. Installieren Sie R (siehe 1.1) und RStudio (1.2) und nehmen Sie die Einstellungen unter 1.4 vor.

Übungsaufgabe 1.2. Öffnen Sie RStudio und führen Sie ein paar simple Berechnungen in der Konsole durch.

Übungsaufgabe 1.3. Erstellen Sie eine neue Skriptdatei und fügen dort mindestens sechs Berechnungen hinzu. Gliedern Sie die Skriptdatei durch einige Kommentare.

Chapter 2

Objekte und Datenstrukturen

Wir können R bzw. RStudio nun als Taschenrechner verwenden und uns arithmetische Operationen direkt in der Konsole ausgeben lassen:

```
1 + 2 # Addition
1 - 2 # Subtraktion
2 * 2 # Multiplikation
4 / 2 # Division
2 ^ 5 # Exponentiation
```

```
## [1] 3
## [1] -1
## [1] 4
## [1] 2
## [1] 32
```

Wirklich sinnvoll ist dies aber nicht: wir wollen Ergebnisse ja auch speichern und weiterverwenden können. Hierfür benötigen wir Variablen, also Namen, denen wir (veränderliche) Werte zuordnen können.

In R lassen sich Variablen erstellen, indem wir einer Zeichenkette (zu den Benennungsregeln kommen wir gleich) einen Wert mittels `<-` zuordnen (hierfür entweder die Zeichen `<` und `-` eingeben oder die Tastenkombination `Alt/Option + -` drücken).[1] Dies erstellt ein *Objekt*[2] mit eben diesem Namen und der entsprechenden Zuordnung:

[1] Es ist prinzipiell auch möglich, die Zuordnung mittels einem `=` vorzunehmen. Da das `=` aber auch in anderen Kontexten benötigt wird, erzeugt dies Verwirrung, sodass wir Objekte immer mit `<-` zuordnen sollten.

[2] Die Begriffe Variable und Objekt können für unsere Zwecke weitestgehend synonym ver-

```
x <- 2
```

Führen wir diesen Befehl aus, erstellen wir das Objekt `x` und ordnen den Wert `2` zu. Wir sollten diese neue Zuordnung zudem im rechten oberen *Environment*-Bereich sehen können.

Wir können nun mit diesem Objekt weiterrechnen:

```
x * 2
x + 5
x / 2
```

```
## [1] 4
## [1] 7
## [1] 1
```

Um den aktuellen Wert eines Objektes anzuzeigen, können wir auch einfach das Objekt ausführen:

```
x
```

```
## [1] 2
```

Generell wird bei der Zuordnung immer zunächst der Teil rechts vom `<-` ausgeführt und dann zugeordnet. Wir können also auch komplexere Befehle ausführen und diese Zuordnen:

```
y <- (2 + 4) * (3 - 1) / 2
y
```

```
## [1] 6
```

Objekte sind veränderlich und können jederzeit neu zugeordnet werden - und dabei auch selbst bei der Zuordnung verwendet werden:

```
x <- 2
x
x <- 3
x
x <- x - 1
x
```

wendet werden. Wir werden aber gleich noch sehen, das so ziemlich alles in R ein Objekt sein kann.

```
## [1] 2
## [1] 3
## [1] 2
```

Schauen wir uns das einmal in einem etwas komplexeren Beispiel an - welchen Wert hat `b` am Ende dieser Befehlskette?

```
a <- 10
b <- a / 2
a <- b * 2 + a
b <- a - b
```

Die Antwort lautet `15`. Gehen wir das der Reihe nach durch:

1. Zunächst ordnen wir `a` den Wert `10` zu.
2. Dann orden wir `b` den Wert `a / 2` zu. Da `a` in diesem Schritt `10` zugeordnet ist, wird `10 / 2` gerechnet. `b` entspricht nun also dem Wert `5`.
3. Wir ordnen nun `a` den Wert `b * 2 + a` zu. Der gesamte Teil rechts vom `<-` wird zuerst ausgeführt und dann zugeordnet, hier also `5 * 2 + 10`. `a` entspricht nun dem Wert `20`, b weiterhin dem Wert `5`.
4. Zuletzt ordnen wir `b` das Ergebnis von `a - b` zu, was vor dieser Zuordnung `20 - 5` bedeutet. `b` entspricht schlussendlich also `15`.

Nochmals die wichtigsten Punkte zusammengefasst:

- Mit `<-` erstellen wir Objekte und ordnen diesen Werte zu.
- Alle Objekte sind veränderlich und können überschrieben werden.
- Bei einer Zuordnung wird der gesamte Teil rechts vom Zuordnungspfeil `<-` zuerst ausgeführt und dann die Zuordnung vorgenommen.

2.1 Objektnamen

Für die obigen Beispiele haben wir nur einzelne Buchstaben für Objekte verwendet. In der Praxis können und sollten wir längere Objektnamen verwenden. Dabei gelten folgende Regeln:

- Objektnamen können Groß- und Kleinbuchstaben, Ziffern sowie Punkte `.` und Unterstriche `_` beinhalten. Andere Sonderzeichen, Umlaute und Leerzeichen sind nicht gestattet.
- Objektnamen können mit einem Buchstaben oder einem `.` beginnen, nicht jedoch mit Ziffern oder `_`.
- Objektnamen sind *case-sensitive*, d. h. unterscheiden zwischen Groß- und Kleinschreibung. `myVar` und `myvar` sind also unterschiedliche Objekte.

Es ist sinnvoll, Objekten "sprechende" Namen zu geben, sodass andere (und auch Sie zu einem späteren Zeitpunkt) nachvollziehen können, was sich dahinter verbirgt, auch ohne den gesamten Code zu lesen.

```
# Gute Objektnamen
mittelwert <- 2.5
mein_alter <- 32
groesse_in_cm <- 175

# Schlechte Objektnamen
x1 <- 2.5
var2 <- 32
asdasdasd <- 175
```

Es gibt außerdem unterschiedliche Konventionen, um mehrere Wörter in Objektnamen aneinanderzuhängen. `mein_alter` ist ein Beispiel für den sogenannten *snake_case*: Alle Wörter kleingeschrieben und durch einen Unterstrich `_` miteinander verbunden. Einen Überblick über verbreitete Konventionen der Objektbenennung gibt folgende Illustration:

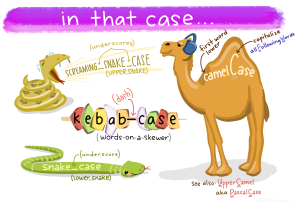


Figure 2.1: Illustration von @allison_horst: https://twitter.com/allison_horst

Was auch immer Sie wählen - wichtig ist vor allem, dass Sie einheitlich vorgehen.[3]

2.2 Objekttypen

Bisher haben wir lediglich Zahlen Objekten zugewiesen. Natürlich können Daten aber auch in anderen Formen vorliegen; wir sprechen daher von verschiedenen *Objekttypen*.[4]

[3] Aufmerksamen Leser*innen dürfte zudem aufgefallen sein, dass *kebab-case* in R nicht möglich ist.

[4] Tatsächlich ist die Sache etwas komplexer: es gibt in R einige wenige Kernobjekttypen, die wiederum mit bestimmten Attributen versehen werden können, um daraus zusätzliche Objekttypen abzuleiten. So kann etwa eine Zahlenfolge mit einem zusätzlichen Attribut als Datumsangabe interpretiert werden. Für unsere Zwecke spielt diese Unterscheidung jedoch keine Rolle.

2.2.1 Numerische Objekte

Zahlenwerte werden in R als `numeric` bezeichnet. Wir können hier zudem zwischen den Typen `integer` (ganze Zahlen) und `double` (Kommazahlen[5]) unterscheiden.

Grundsätzlich ordnet R Zahlen als `double` zu, auch wenn nur ganze Zahlen zugeordnet werden.

```
x <- 4
typeof(x) # Mit dieser Funktion können wir den Objekttyp anfordern
```

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

Um explizit den Typ `integer` anzufordern, muss Zahlenwerten ein `L` nachgestellt werden:[6]

```
x <- 4L
typeof(x)
```

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

In der Praxis macht es aber kaum einen Unterschied, ob eine ganze Zahl als `integer` oder `double` abgespeichert wird – `integer` verbraucht weniger Speicherplatz, aber das wird erst bei *sehr* großen Datensätzen relevant. Wir können die Unterscheidung also guten Gewissens ignorieren und von numerischen Objekten sprechen.

2.2.2 Textobjekte

Wir können Objekten auch Text zuordnen - diese Objekte haben dann den Typ `character` (Textvariablen werden zudem häufig als "string" bezeichnet). Um ein `character`-Objekt zu erstellen, müssen wir die Zeichenkette in einfache `''` oder doppelte `""` Anführungszeichen setzen:

```
text1 <- "Guten Morgen!"
text2 <- 'Einfache Anführungszeichen sind sinnvoll, wenn im "Text" ebenfalls Anführungszeichen vo
```

Natürlich können auch Zahlen als Text gespeichert werden - werden dann aber natürlich auch als Text behandelt, sodass man nicht mehr mit ihnen rechnen kann.

[5] Der Typenbezeichnung leitet sich von Gleitkommazahlen mit doppelter Genauigkeit ab.
[6] Warum `L`? Hier gibt es unterschiedliche Erklärungsansätze, die beispielsweise hier nachgelesen werden können

```
zahl_als_text <- "123"
```

2.2.3 Logicals (logische Objekte)

Der dritte Kernobjekttyp heißt `logical` und kann nur zwei Werte annehmen: `TRUE` (wahr) oder `FALSE` (falsch). Logicals entstehen durch logische Vergleiche zweier Objekte, wobei u.a. folgende Operatoren verwendet werden können:

Table 2.1: Logische Operatoren in R

Operator	Vergleich	Beispiele
`==`	ist gleich	`1 == 1` (ergibt `TRUE`)`"a" == "b"` (ergibt `FALSE`)
`!=`	ist nicht gleich	`1 != 1` (ergibt `FALSE`)`"a" != "b"` (ergibt `TRUE`)
`<`	ist kleiner als	`1 < 2` (ergibt `TRUE`)`2 < 2` (ergibt `FALSE`)
`>`	ist größer als	`2 > 1` (ergibt `TRUE`)`2 > 2` (ergibt `FALSE`)
`<=`	ist kleiner gleich	`1 <= 2` (ergibt `TRUE`)`2 <= 2` (ergibt `TRUE`)
`>=`	ist größer gleich	`2 >= 1` (ergibt `TRUE`)`2 >= 2` (ergibt `TRUE`)

Die Zuordnung erfolgt wie bei anderen Objekten auch:

```
x <- "a" == "b"
x
```

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

Logicals werden vor allem bei Wenn-Dann-Bedingungen benötigt, mit denen wir uns im übernächsten Kapitel auseinandersetzen werden.

2.2.4 Weitere Objekttypen

Diese drei Objekttypen (`numeric`, `character`, `logical`) bilden die Basis fast aller Objekte in R. Durch zusätzliche Attribute können jedoch noch zusätzliche Objekttypen erzeugen, die den Umgang mit bestimmten Daten erleichtern. Für kategoriale Variablen kennt R beispielsweise den Typ `factor`, für Datumsangaben den Typ `date`. Diese werden bei der Zuordnung nicht automatisch erkannt und müssen stattdessen durch bestimmte Funktionen erzeugt werden.

Erzeugen wir beispielsweise ein Objekt mit einer Zeichenfolge, die ein Datum repräsentiert (z. B. `date1 <- "2020-05-05"`, im Format *YYYY_MM_DD*, also Jahr-Monat-Tag), speichert R dies zunächst als `character` ab. Wir können aber R explizit sagen, dass er dies als Datum behandeln soll:

```
date2 <- as.Date("2020-05-05")
```

Dies hat nun u.a. den Vorteil, dass wir im Gegensatz zu `character`-Objekten auch arithmetische Operationen durchführen können, also beispielsweise zwei Datums-Objekte voneinander subtrahieren, um die zeitliche Differenz zu berechnen.

Wir werden uns im späteren Verlauf noch ausführlicher mit diesen spezielleren Objekttypen beschäftigen – bis jetzt nehmen Sie vor allem mit, dass sowohl kategoriale Variablen als auch Datumsangaben kein Problem für R darstellen.

2.2.5 Fehlende Werte

Fehlende Werte werden in R als `NA` angegeben. Es ist sinnvoll, fehlende Werte immer explizit als `NA` zu kennzeichnen und nicht etwa durch einen negativen Wert bei numerischen Variablen (z. B. `-9`) oder durch einen leeren String bei Textvariablen (`""`), damit sichergestellt ist, dass Funktionen den fehlenden Wert auch entsprechend als einen solchen behandeln.

2.2.6 Objekttypen ändern

Bisweilen wird es relevant sein, Objekttypen zu ändern - etwa weil Zahlen fälschlicherweise als Text eingelesen wurden. Hierfür bietet R Funktionen an, die allesamt nach dem Schema `as.[Objekttyp]()` aufgebaut sind: mit `as.numeric()` wandeln wir Objekte in numerische Objekte um (genauer gesagt in `double`), mit `as.character()` in Textobjekte und, wie im vorigen Abschnitt gesehen, mit `as.Date()` in ein Datumsobjekt. Der Fachbegriff hierfür lautet *Coercion*, wir *zwingen* R also dazu, ein Objekt als einen bestimmten Typ zu behandeln, auch wenn R automatisch einen anderen Typus bestimmt hätte.

```
x1 <- "25"
x1
typeof(x1)
x2 <- as.numeric("25")
x2 # Beachten Sie, dass in der Ausgabe nun die Anführungszeichen fehlen
typeof(x2)
```

```
## [1] "25"
## [1] "character"
## [1] 25
## [1] "double"
```

Natürlich klappt das nur, solange die Umwandlung auch sinnvoll durchführbar ist – in allen anderen Fällen wird eine Warnung ausgegeben und es werden fehlende Werte erzeugt.

```
x <- as.numeric("Dieser Text kann nicht sinnvoll als Zahl interpretiert werden")
```

```
## Warning: NAs introduced by coercion
```

```
x
```

```
## [1] NA
```

2.3 Datenstrukturen

Bisher haben wir einem Objekt immer nur einen einzigen Wert zugeordnet. Der Fachbegriff hierfür lautet *Skalar* und beschreibt somit die einfachst mögliche Datenstruktur, eben dass einem Objekt nur ein einziger Wert zugeordnet wurde. Objekte können in R jedoch auch mehrere Werte enthalten und somit komplexere Datenstrukturen erzeugen.

Im Folgen betrachten wir daher die vier wichtigsten komplexeren Datenstrukturen in R. Diese unterscheiden sich zum einen in ihrer Dimensionalität (also ob sie ein- oder zweidimensional sind) und zum anderen, ob sie homogene (also nur dieselben) oder heterogene (also unterschiedliche) Objekttypen beinhalten können:

Table 2.2: Datenstrukturen in R

Datenstruktur	Dimensionalität	Objekttypen
Vektor	eindimensional	homogen
Liste	eindimensional	heterogen
Matrix	zweidimensional	homogen
Dataframe	zweidimensional	heterogen

2.3.1 Vektoren

Vektoren sind Objekte, die mehrere Werte desselben Typs beinhalten. Wir erzeugen Vektoren über die Funktion `c()` (von *concatenate*, also verketten). Die einzelnen Elemente des Vektors werden durch Kommas `,` getrennt.

```
gerade_zahlen <- c(2, 4, 6, 8)
gerade_zahlen
ungerade_zahlen <- c(1, 3, 5, 7, 9)
ungerade_zahlen
simpsons <- c("Homer Simpson", "Marge Simpson", "Bart Simpson", "Lisa Simpson", "Maggie Simpson")
simpsons
```

```
## [1] 2 4 6 8
## [1] 1 3 5 7 9
## [1] "Homer Simpson"  "Marge Simpson"  "Bart Simpson"   "Lisa Simpson"   "Maggie Simpson"
```

Wir können auch Vektoren über `c()` mit einander verketten:

```
zahlen <- c(gerade_zahlen, ungerade_zahlen)
zahlen
```

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

Beachten Sie, dass die Verkettung immer in der angegebenen Reihenfolge erfolgt – R sortiert die Elemente also nicht automatisch.

2.3.1.1 Vektorelemente auswählen

Um bestimmte Elemente eines Vektors auszuwählen, können wir die gewünschten Elemente in eckigen Klammern `[]` hinter einem Vektor definieren. Hier geben wir nur das zweite Element des oben erzeugten **`zahlen`**-Vektors aus:

```
zahlen[2]
```

```
## [1] 4
```

Um mehrere Elemente eines Vektors auszugeben, benötigen wir wiederum einen Vektor mit den gewünschten Positionen – hier geben wir uns beispielsweise das erste, dritte und fünfte Element aus:

```
zahlen[c(1, 3, 5)]
```

```
## [1] 2 6 1
```

2.3.1.2 Vektorelemente benennen

Elemente in Vektoren können benannt werden, indem beim Erstellen die Namen der Elemente mit einem `=` angegeben werden:

```
homer <- c(nachname = "Simpson", vorname = "Homer", wohnort = "Springfield")
homer
```

```
##      nachname       vorname       wohnort
##     "Simpson"       "Homer" "Springfield"
```

Benannte Elemente können dann auch über den Namen ausgewählt werden:

```
homer["wohnort"]
```

```
##       wohnort
## "Springfield"
```

Auch dies funktioniert mit mehreren Elementen gleichzeitig:

```
homer[c("vorname", "nachname")]
```

```
##   vorname  nachname
##   "Homer" "Simpson"
```

Alternativ können Elementnamen im Nachhinein über die Funktion `names()` hinzugefügt werden:

```
marge <- c("Simpson", "Marge", "Springfield")
names(marge) <- c("nachname", "vorname", "wohnort")
marge
```

```
##      nachname       vorname       wohnort
##     "Simpson"       "Marge" "Springfield"
```

2.3.1.3 Mit Vektoren rechnen

Mit numerischen Vektoren können arithmetische Berechnungen durchgeführt werden:

```
zahlen + 1
zahlen * 2
```

```
## [1]  3  5  7  9  2  4  6  8 10
## [1]  4  8 12 16  2  6 10 14 18
```

Berechnungen werden dabei der Reihe nach für jedes einzelne Vektorelement durchgeführt. Es ist auch möglich, Vektoren gleicher Länge zu addieren, subtrahieren etc. – im Falle einer Addition wird dann das erste Element des ersten Vektors zum ersten Element des zweiten Vektors addiert, dann das zweite Element des ersten Vektors zum zweiten Element des zweiten Vektors usw.:

```
x1 <- c(1, 3, 5)
x2 <- c(2, 3, 4)
x1 * x2
```

```
## [1]  2  9 20
```

2.3.1.4 Nützliche Vektorfunktionen

Abschließend einige nützliche Funktionen für den Umgang mit Vektoren:

- `length()` gibt die Anzahl der Elemente eines Vektors aus.
- Die unter 2.2.6 eingeführten *Coercion*-Funktionen (`as.numeric()`, `as.character()` usw.) können auch auf Vektoren angewendet werden und wandeln so jedes Vektorelement um.
- Für numerische Vektoren stehen zahlreiche statistische Funktionen bereit, z. B. zur Berechnung der Summe (`sum()`), des arithmetischen Mittels (`mean()`) und der Standardabweichung (`sd()`)

```
x <- c(10, 24, 32, 999)
length(x)
sum(x)
mean(x)
sd(x)
```

```
## [1] 4
## [1] 1065
## [1] 266.25
## [1] 488.5846
```

Oftmals benötigen wir aufsteigende Zahlenfolgen für Vektoren, beispielsweise für laufende Nummern. Dies lässt sich über die Funktion `:` abkürzen, die einen Vektor `Startwert:Endwert` erstellt:

```
eins_bis_zehn <- 1:10
eins_bis_zehn
```

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

2.3.2 Listen

Listen ähneln zunächst Vektoren und werden mit der Funktion `list()` erzeugt. Auch die Benennung von Listenelementen erfolgt analog zu Vektoren entweder beim Erstellen der Liste mit `=` oder im Nachhinein mit `names()`:

```
munich_facts <- list(name = "München", bundesland = "Bayern", bezirk = "Oberbayern")
munich_facts
```

```
## $name
## [1] "München"
##
## $bundesland
## [1] "Bayern"
##
## $bezirk
## [1] "Oberbayern"
```

Wir sehen aber bereits am Konsolenoutput, dass die Darstellung von Vektoren abweicht: anstatt alle Elemente nebeneinander angezeigt zu bekommen, werden die einzelnen Elemente untereinander angezeigt.

Das rührt daher, dass Listen deutlich mächtiger und flexibler sind als Vektoren. Nicht nur können wir in den einzelnen Elementen Objekttypen mischen, wir sind auch nicht auf einzelne Werte (Skalare) als Elemente beschränkt. Tatsächlich kann so gut wie jedes Objekt ein Listenelement sein – also auch Vektoren, ganze Datensätze und sogar Listen (die wiederum eigene Listen enthalten können – wir können hier also Daten prinzipiell endlos verschachteln). Erweitern wir dazu die Liste von oben:

```
munich_facts <- list(
  namen = c(hochdeutsch = "München", englisch = "Munich", bairisch = "Minga"),
  bundesland = "Bayern",
  gruendungsjahr = 1158,
  daten = list(
    einwohner = 1471508,
    geographie = c(flaeche_in_km2 = 310.7, hoehe_NHN_in_m = 519)
  )
)
munich_facts
```

```
## $namen
## hochdeutsch    englisch    bairisch
##   "München"    "Munich"     "Minga"
##
## $bundesland
## [1] "Bayern"
##
## $gruendungsjahr
## [1] 1158
##
## $daten
## $daten$einwohner
## [1] 1471508
##
## $daten$geographie
## flaeche_in_km2 hoehe_NHN_in_m
##          310.7          519.0
```

Mit ihrer Flexibilität stellen Listen in R die Basis für nahezu alle komplexeren Datenstrukturen – auch bei Datensätzen, Regressionsmodellen etc. handelt es sich um Listen, die mit bestimmten Attributen versehen wurden.

2.3.2.1 Listenelemente auswählen

Auch die Auswahl von Listenelementen funktioniert ähnlich wie bei Vektoren über die numerische Position oder den Elementnamen:

```
munich_facts[1]
```

```
## $namen
## hochdeutsch    englisch    bairisch
##   "München"    "Munich"     "Minga"
```

```
munich_facts["daten"]
```

```
## $daten
## $daten$einwohner
## [1] 1471508
##
## $daten$geographie
## flaeche_in_km2 hoehe_NHN_in_m
##          310.7          519.0
```

Vielleicht ist Ihnen bereits das Dollarsymbol `$` vor den Elementnamen aufgefallen – dieses verweist auf eine Funktion, mit der Listenelemente noch komfortabler ausgewählt werden können:

```
munich_facts$bundesland
```

```
## [1] "Bayern"
```

Nicht nur sparen Sie sich ein paar Zeichen bei der Eingabe, RStudio macht Ihnen auch automatisch Vorschläge, sobald Sie das Dollarzeichen eingetippt haben (im Beispiel also ab `munich_facts$`), welche Elemente sie auswählen können. Auch tiefer verschachtelte Elemente können so ausgewählt werden:

```
munich_facts$daten$einwohner
```

```
## [1] 1471508
```

2.3.2.2 Nützliche Listenfunktionen

Auch für Listen stehen einige nützliche Funktionen zur Verfügung, die die Arbeit mit ihnen erleichtern.

`length` gibt wie auch schon bei Vektoren die Anzahl der Elemente aus (wobei auch komplexere Elemente, also z. B. Vektoren, Listen etc., jeweils als ein Element gezählt werden):

```
length(munich_facts)
```

```
## [1] 4
```

Spezifisch für Listen relevant ist die Funktion `str()`, die zusätzliche Informationen über die Struktur der Liste ausgibt, was gerade bei verschachtelteren Listen den Überblick erleichtert:

```
str(munich_facts)
```

```
## List of 4
##  $ namen         : Named chr [1:3] "München" "Munich" "Minga"
##   ..- attr(*, "names")= chr [1:3] "hochdeutsch" "englisch" "bairisch"
##  $ bundesland    : chr "Bayern"
##  $ gruendungsjahr: num 1158
##  $ daten         :List of 2
##   ..$ einwohner : num 1471508
##   ..$ geographie: Named num [1:2] 311 519
##   .. ..- attr(*, "names")= chr [1:2] "flaeche_in_km2" "hoehe_NHN_in_m"
```

Wir sehen, dass die Liste `munich_facts` aus 4 Elementen besteht `List of 4`. Das erste Element trägt den Namen `namen` ist ein benannter `character`-Vektor (abgekürzt durch `chr`) mit 3 Elementen (Angabe `[1:3]`) usw.

Durch die komplexe Struktur sind Listen jedoch nicht so einfach zu handhaben. Mittels der Funktion `unlist()` können Listen daher in Vektoren (inkl. Elementnamen, soweit vorhanden) umgewandelt werden:

```
munich_facts_vector <- unlist(munich_facts)
munich_facts_vector
```

```
##               namen.hochdeutsch                  namen.englisch                   namen.bairisc
##                       "München"                        "Munich"                          "Minga
##                  gruendungsjahr                daten.einwohner daten.geographie.flaeche_in_km
##                          "1158"                       "1471508"                          "310.7
```

2.3.3 Matrizen

Matrizen sind Vektoren, die in eine zweidimensionale Struktur, also Zeilen und Spalten, überführt werden und können mit der Funktion `matrix()` erstellt werden. Hierzu ist zusätzlich noch die Anzahl an Zeilen, in die der Vektor aufgeteilt werden soll, nötig:

```
x <- 1:10 # Zahlen von 1 bis 10 als Vektor
m <- matrix(x, nrow = 2) # Vektor in Matrix mit zwei Zeilen aufteilen
m
```

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

Alternativ können wir mehrere Vektoren mit den Funktionen `cbind()` spaltenweise (von *c*olumn) und `rbind()` zeilenweise (von *r*ow) zu einer Matrix "zusammenkleben".

```
x1 <- 1:4
x2 <- 5:8
x3 <- 0:3
m <- cbind(x1, x2, x3)
m
```

```
##      x1 x2 x3
## [1,]  1  5  0
## [2,]  2  6  1
## [3,]  3  7  2
## [4,]  4  8  3
```

2.3.3.1 Matrizen benennen

Auch Matrizen können benannt werden. Da wir nun aber eine zweidimensionale Struktur haben, können wir entsprechend auch Zeilen und Spalten einzeln benennen. Hierfür gibt es die Funktionen `rownames()` und `colnames()`, die analog zu `names()` verwendet werden:

```
colnames(m) <- c("spalte_1", "spalte_2", "spalte_3")
rownames(m) <- c("zeile_a", "zeile_b", "zeile_c", "zeile_d")
m
```

```
##         spalte_1 spalte_2 spalte_3
## zeile_a        1        5        0
## zeile_b        2        6        1
## zeile_c        3        7        2
## zeile_d        4        8        3
```

2.3.3.2 Nützliche Matrixfunktionen

`length()` funktioniert auch für Matrizen und gibt die Anzahl aller Elemente an. Interessieren wir uns dagegen für die Anzahl an Zeilen und Spalten, gibt die Funktion `dim()` Aufschluss, die einen Vektor mit der Anzahl der Zeilen und der Anzahl der Spalten ausgibt:

```
length(m)
dim(m)
```

```
## [1] 12
## [1] 4 3
```

Die Funktion `t()` transponiert die Matrix, dreht die Matrix also um 90 Grad und vertauscht somit Zeilen und Spalten:

```
t(m)
```

```
##          zeile_a zeile_b zeile_c zeile_d
## spalte_1       1       2       3       4
## spalte_2       5       6       7       8
## spalte_3       0       1       2       3
```

2.3.4 Dataframes

Mit Matrizen kommen wir der Datensatzstruktur, wie wir sie von SPSS oder Excel kennen, schon recht nahe. Allerdings repräsentieren Matrizen Vektoren und sind daher auf einen Objekttyp beschränkt. Diese Einschränkung hebt die Datenstruktur *Dataframe* auf, mit der gleich lange Vektoren unterschiedlichen Typs kombiniert werden können. Wir erstellen Dataframes mit der gleichnamigen Funktion `data.frame()`:

```
beatles_data <- data.frame(
  name = c("John", "Paul", "George", "Ringo"),
  surname = c("Lennon", "McCartney", "Harrison", "Starr"),
  born = c(1940, 1942, 1943, 1940)
)
beatles_data
```

```
##     name   surname born
## 1   John    Lennon 1940
## 2   Paul McCartney 1942
## 3 George  Harrison 1943
## 4  Ringo     Starr 1940
```

Wenn wir mit tabellarischen Daten arbeiten, geschieht das also in der Regel mit Dataframes. Natürlich wäre es nicht zielführend, wenn wir diese immer von Hand erstellen müssten. Es gibt daher Funktionen, mit denen wir externe Dateien (z. B. CSV-, Excel- und sogar SPSS-Dateien) als Dataframes in R laden können. Wie wir externe Dateien laden, schauen wir uns zu einem späteren Zeitpunkt genauer an.

2.3.4.1 Beispiel-Dataframes

R enthält einige eingebaute Beispiel-Dataframes, mit denen Funktionen demonstriert und geübt werden können. Keiner davon hat auch nur einen geringen KW-Bezug, aber damit wir nun auch ohne große Erstellungs-Arbeit mit Dataframes arbeiten können, nutzen wir diese dennoch. Wir werden aber bald auch mit für Sie relevanteren Daten arbeiten, versprochen.

Diese Beispiel-Datensätze sind direkt als Objekte hinterlegt und können durch Eingabe des Objektnamens genutzt werden. Wir arbeiten nun mit dem Datensatz `iris`, der Blütenblatt- und Kelchblatt-Daten zu je 50 Exemplaren dreier Spezies von Schwertlilien (englisch *iris*) umfasst. Wie gesagt, keinerlei KW-Bezug, aber immerhin ein Grund, um den Text kurz durch einige Blumenfotos aufzulockern.

Figure 2.2: Drei Schwertlilien-Spezies im `iris`-Datensatz, von links nach rechts: Borsten-Schwertlilie (*iris setosa*), verschiedenfarbige Schwertlilie (*iris versicolor*) und Virginia-Schwertlilie (*iris virginica*). Fotos: Radomił Binek, Danielle Langlois und Eric Hunt.

2.3.4.2 Arbeiten mit Dataframes

Dataframes basieren auf Listen und Vektoren (genau genommen ist ein Dataframe eine Liste von gleich langen Vektoren, die zweidimensional dargestellt wird). Entsprechend können wir eine Vielzahl der Funktionen, die wir bei Listen und Vektoren kennengelernt haben, auch auf Dataframes anwenden.

In der Regel haben wir Datensätze, die mehr als nur ein paar Fälle umfassen. Sie in der Konsole ausgeben zu lassen, ist daher nur bedingt sinnvoll. Besser ist es, Informationen über die Struktur des Datensatzes über Funktionen abzufragen.

Die `length()`-Funktion gibt bei Dataframes die Anzahl der Spalten zurück (wir erinnern uns: Dataframes sind Listen, deren Elemente die Spaltenvektoren sind; entsprechend ermittelt `length()` daher die Anzahl der Vektoren). Die Anzahl der Zeilen – mithin die Anzahl der Fälle – gibt die Funktion `nrow()` aus. Beide Werte gemeinsam können wir erneut über `dim()` ausgeben.

```
length(iris)
nrow(iris)
dim(iris)
```

```
## [1] 5
## [1] 150
## [1] 150   5
```

Wir sehen, dass der `iris`-Datensatz 5 Spalten (= Variablen) und 150 Zeilen (= Fälle) umfasst. Für einen Überblick bietet sich wie auch schon bei Listen die `str()`-Funktion an.

```
str(iris)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

Der Datensatz umfasst also 4 numerische Variablen (jeweils Länge und Breite des Blüten- (*sepal*) und des Kelchblattes (*petal*)) sowie eine kategoriale Faktorvariable mit 3 Stufen, in der die Spezies des jeweiligen Exemplars festgehalten ist.

150 Fälle à 5 Variablen wären im Konsolenoutput sehr lang und unübersichtlich. Wollen wir dennoch in unsere Daten spähen, können wir uns mittels `head()` und `tail()` die ersten bzw. letzten Zeilen des Datensatzes ausgeben. Standardmäßig werden bei beiden Funktionen 6 Fälle angezeigt.

```
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

R bietet auch einen Viewer, mit dem der gesamte Datensatz ähnlich wie ein Tabellenblatt in Excel oder die Datenansicht in SPSS angezeigt wird. Hierzu führen wir die Funktion `View()` aus (großes `V` beachten), woraufhin in RStudio ein eigener Reiter mit der Datenansicht geöffnet wird.

```
View(iris)
```

In der Datenansicht können wir den Datensatz nach Variablen sortieren oder bestimmte Wertebereiche je Variable filtern; in der Fußzeile werden Strukturinformationen (Zeilen- und Spaltenanzahl) angezeigt. Fährt man mit dem Mauszeiger über die Kopfzeilen der Variablen, werden zudem weitere Informationen (Objekttyp und Wertebereich) eingeblendet.

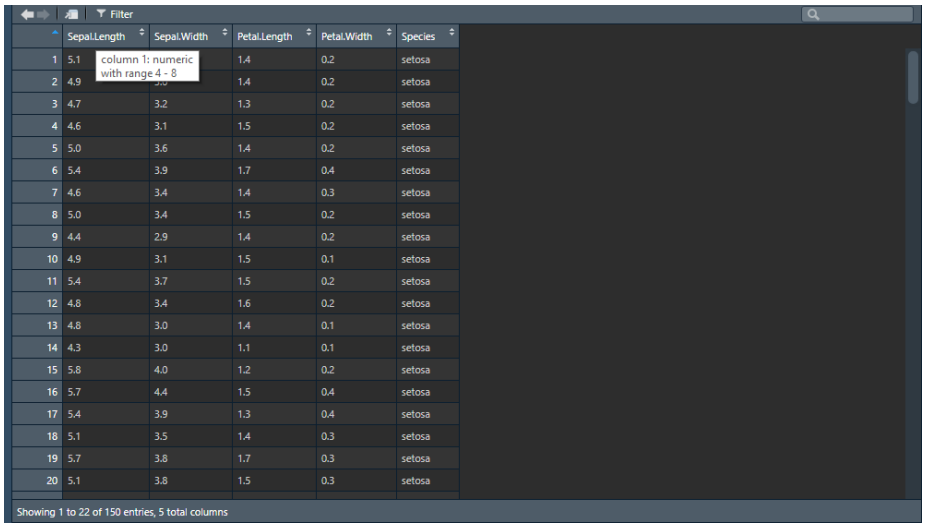


Figure 2.3: Die Datenansicht von R

Um einzelne Variablen (also Spalten bzw. die dahinterliegenden Vektoren) auszuwählen, können wir wie auch bei Listen das `$`-Zeichen nutzen:

```
iris$Sepal.Length
```

```
##   [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4 5.1 5.7 5
##  [34] 5.5 4.9 5.0 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0 6.4 6
##  [67] 5.6 5.8 6.2 5.6 5.9 6.1 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4 6
## [100] 5.7 6.3 5.8 7.1 6.3 6.5 7.6 4.9 7.3 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7
## [133] 6.4 6.3 6.1 7.7 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8 6.7 6.7 6.3 6.5 6.2 5.9
```

Das Resultat hat den Objekttyp Vektor. Wir können daher die uns bekannten Vektorfunktionen auf das Resultat anwenden. Um beispielsweise den Mittelwert der Kelchblattbreite zu erhalten, geben wir folgendes ein:

```
mean(iris$Petal.Width)
```

```
## [1] 1.199333
```

Auch die eckigen Klammern `[]` können genutzt werden, um flexibler nur bestimmte Teile des Datensatzes anzuzeigen. Dabei ist die zweidimensionale Struktur zu beachten – wir können zwei Werte bzw. Vektoren, getrennt durch ein `,`, übergeben, die dann die Zeilen respektive die Spalten anwählt. Um etwa die ersten zehn Zeilen der Variablen `Sepal.Length` und `Petal.Length` auszuwählen, ist folgender Code nötig:

```
iris[1:10, c("Sepal.Length", "Petal.Length")]
```

```
##    Sepal.Length Petal.Length
## 1           5.1          1.4
## 2           4.9          1.4
## 3           4.7          1.3
## 4           4.6          1.5
## 5           5.0          1.4
## 6           5.4          1.7
## 7           4.6          1.4
## 8           5.0          1.5
## 9           4.4          1.4
## 10          4.9          1.5
```

Sollen lediglich Zeilen oder nur Spalten gefiltert werden, lassen wir den jeweiligen Wert vor (Zeilen) oder nach dem `,` (Spalten) leer. Folgender Code gibt die Zeilen 5-10 und alle Spalten aus:

```
iris[5:10,]
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 5           5.0         3.6          1.4         0.2  setosa
## 6           5.4         3.9          1.7         0.4  setosa
## 7           4.6         3.4          1.4         0.3  setosa
## 8           5.0         3.4          1.5         0.2  setosa
## 9           4.4         2.9          1.4         0.2  setosa
## 10          4.9         3.1          1.5         0.1  setosa
```

Auch die Benennung von Dataframes läuft analog zu den bisherigen Objekttypen ab. Um etwa die Variablen einzudeutschen, können wir wieder `names()` nutzen[7]:

```
names(iris) <- c("bluetenblatt_laenge", "bluetenblatt_breite",
                 "kelchblatt_laenge", "kelchblatt_breite",
                 "spezies")
head(iris)
```

```
##   bluetenblatt_laenge bluetenblatt_breite kelchblatt_laenge kelchblatt_breite spezies
## 1                 5.1                3.5              1.4               0.2  setosa
```

[7] Da Dataframes eine zweidimensionale Struktur darstellen, können wir auch die Funktionen `colnames()` und `rownames()`, die wir von den Matrizen kennen, verwenden, um die Spalten respektive Zeilen umzubenennen. `colnames()` und `names()` sind bei Dataframes äquivalent; Zeilennamen sind eher unüblich. In der Praxis wird daher meistens lediglich `names()` verwendet.

```
## 2          4.9         3.0          1.4         0.2  setos
## 3          4.7         3.2          1.3         0.2  setos
## 4          4.6         3.1          1.5         0.2  setos
## 5          5.0         3.6          1.4         0.2  setos
## 6          5.4         3.9          1.7         0.4  setos
```

Falls Ihnen das nun umständlich erscheint – keine Sorge, wir werden, sobald wir uns ans richtige Datenmanagement begeben, Funktionen kennenlernen, die die obigen Schritte deutlich erleichtern. Für das Grundverständnis ist es aber wichtig, auch diese Art der Arbeit mit Dataframes kennenzulernen.

2.4 Übungsaufgaben

Erstellen Sie für die folgenden Übungsaufgaben eine eigene Skriptdatei und speichern diese als `ue2_nachname.R` ab. Antworten auf Fragen können Sie direkt als Kommentare in das Skript einfügen.

Übungsaufgabe 2.1. Objekttypen und Datenstrukturen I:

Erstellen Sie ein Objekt `myself`, das folgende Elemente enthält:

- `name`: Ihren Namen
- `born`: Ihr Geburtsjahr
- `from_bavaria`: Sind Sie in Bayern geboren?

Welche Datenstruktur ist hierfür am sinnvollsten? Welche Objekttypen haben die einzelnen Elemente?

Übungsaufgabe 2.2. Vektorfunktionen, Objekttypen und -umwandlung:

Führen Sie folgenden Code aus:

```
values <- c(1.2, 1.3, 0.8, 0.7, 0.7, 1.5, 1.1, 1.0, 1.1, 1.2, 1.1)
average <- mean(values)
above_average <- values > average
sum(above_average) / length(values)
```

Beschreiben Sie in eigenen Worten, was hier in jeder Zeile passiert. Was bedeutet das Resultat der letzten Codezeile? Warum wird hier überhaupt ein Resultat ausgeben? Schauen Sie sich hierzu an, was passiert, wenn Sie `above_average` in einen numerischen Vektor umwandeln.

Übungsaufgabe 2.3. Arbeiten mit Dataframes:

Im Beispiel-Datensatz `mtcars` sind einige Daten zu verschiedenen KfZ-Modellen hinterlegt. Beantworten Sie die folgenden Fragen zum Datensatz mittels Funktionen:

- Wie viele Variablen und Fälle befinden sich in dem Datensatz?
- Welche der drei Objekttypen (`numeric`, `character`, `logical`) kommen in dem Datensatz vor?
- Wie viele Zylinder haben die enthaltenen Fahrzeuge im Durchschnitt? (Zylinder: `cyl`)
- Erstellen Sie einen neuen Datensatz `cars_short`, der lediglich die Variablen `mpg` und `hp` enthält.

Chapter 3

Funktionen

Wir haben bereits im vorherigen Kapitel einige Funktionen eingesetzt. Tatsächlich ist *alles*, was in R ausgeführt wird, eine Funktion. Zeit also, dass wir uns etwas detaillierter mit Funktionen auseinandersetzen.

3.1 Funktionen aufrufen

In den allermeisten Fällen rufen wir Funktionen nach dem folgenden Schema auf:

```
funktionsname(argument1 = wert1, argument2 = wert2, argument3 = wert3, ...)
```

Der Funktionsname wird gefolgt von Klammern `()`, in die in aller Regel mindestens ein *Argument* übergeben wird.[1] Die Funktion gibt daraufhin ein *Resultat* oder eine *Fehlermeldung* zurück, wobei das Resultat jede Datenstruktur in R annehmen kann – also sowohl einzelne Werte als auch komplexe Listen oder Dataframes.

Da Funktionen im Erfolgsfall immer exakt ein Resultat ausgeben, können wir dieses problemlos wiederum einem Objekt zuweisen und weiterverwenden:

[1] Da der Funktionsname vor den Argumenten steht, wird hierbei auch von *Prefix*-Funktionen gesprochen. Wir haben im vorherigen Kapitel auch schon zwei andere Arten von Funktionen kennengelernt: *Infix*-Funktionen, bei dir der Funktionsname zwischen zwei Argumenten steht (dazu zählen beispielsweise alle arithmetischen Operatoren, z. B. `+`, `-`, `*` und `/`), und *Replacement*-Funktionen, die Teile eines bestehenden Objekts direkt verändern (z. B. `names(x) <-`). Das sind aber Spezialfälle – im Alltag werden wir vorrangig Prefix-Funktionen verwenden.

```
mean_petal_length <- mean(iris$Petal.Length)
```

3.1.1 Funktionsargumente

Funktionen benötigen mindestens ein Argument, oft auch mehrere Argumente, um korrekt ausgeführt zu werden. Oftmals umfassen Funktionsargumente zum einen (Daten-)Objekte, die von der Funktion verwendet werden sollen, zum anderen "Optionen", die die Funktion berücksichtigen soll.

Die Funktion `round()`, die numerische Werte rundet, beispielsweise hat zwei Argumente:

1. `x`: ein numerischer Vektor, der gerundet werden soll.
2. `digits`: ein Integer, der angibt, auf wie viele Dezimalstellen gerundet werden soll.

Der vollständige Funktionsaufruf läuft dementsprechend in der Form `round(x, digits)` ab:

```
round(pi, 1) # Die Kreiszahl Pi ist direkt als Objekt in R hinterlegt
round(pi, 3)
```

```
## [1] 3.1
## [1] 3.142
```

3.1.1.1 Funktionsargumente übergeben

Funktionsargumente können auf zwei Arten übergeben werden: zum einen implizit, wie im obigen Beispiel, durch die Reihenfolge der Funktionsargumente; zum anderen explizit über den Namen des Funktionsarguments in der Form `argument = wert`:

```
round(pi, 2)
round(pi, digits = 2)
```

```
## [1] 3.14
## [1] 3.14
```

In diesem Fall macht es keinen Unterschied, ob wir explizit `digits =` angeben oder nicht. In der Praxis ist es jedoch üblich, maximal die ersten ein oder zwei Funktionsargumente unbenannt zu übergeben, alle anderen Argumente jedoch benannt zu übergeben. Gerade bei Funktionen mit vielen Argumenten müssen

Sie so zum einen nicht die exakte Reihenfolge der Argumente einhalten, zum anderen wird ihr Code auch für andere nachvollziehbarer und lesbarer.

Schauen wir uns hierzu als zweites Beispiel die schon bekannte `mean()`-Funktion an und nutzen sie mit einigen zusätzlichen, uns noch nicht bekannten Argumenten. Was passiert wohl hier?

```
mean(iris$Petal.Length, 0.1, TRUE)
```

```
## [1] 3.76
```

Ohne Kenntnis der Funktionsargumente ist dieser Aufruf schwer nachzuvollziehen – wir berechnen wohl den Mittelwert der Variablen `Petal.Length` im Datensatz `iris`, aber was bedeutet der Rest? Durchschaubarer wird es, wenn wir die beiden hinteren Argumente mit ihrem Namen übergeben:

```
mean(iris$Petal.Length, trim = 0.1, na.rm = TRUE)
```

```
## [1] 3.76
```

Hier können wir anhand der Argumentnamen Vermutungen anstellen, auch ohne die genaue Funktion zu kennen. `trim` wird wohl bedeuten, dass ein bestimmter Anteil der Werte getrimmt, also abgeschnitten wird. Tatsächlich gibt `trim` den Anteil der niedrigsten und höchsten Werte an, der nicht bei der Berechnung berücksichtigt wird – nützlich, um den Einfluss von Ausreißern auf den Mittelwert abzuschwächen. In unserem Fall, `trim = 0.1`, schließen wir also die untersten und obersten 10% der Werte aus. Mit `na.rm = TRUE` geben wir an, dass etwaige fehlende Werte `NA` vor der Berechnung aus dem Vektor entfernt werden sollen (`rm` steht hier also für "remove").

Die Benennung erlaubt es uns nun auch, die Reihenfolge der Argumente im Funktionsaufruf zu tauschen:

```
mean(iris$Petal.Length, na.rm = TRUE, trim = 0.1)
```

```
## [1] 3.76
```

Natürlich könnten wir auch das erste Argument benennen, aber das verspricht – auch ob des generischen Namens `x` – keinen zusätzlichen Erkenntnisgewinn und wäre wohl eher Overkill:

```
mean(x = iris$Petal.Length, trim = 0.1, na.rm = TRUE)
```

```
## [1] 3.76
```

Die mittlere Variante – implizite Übergabe des ersten Funktionsarguments, explizite Nennung der weiteren Funktionsargumente – stellt also das gesunde Mittelmaß aus Kürze und Lesbarkeit dar.

3.1.1.2 Default-Werte von Argumenten

Wenn die `mean()`-Funktion drei Argumente besitzt, wie war es uns dann bisher möglich, sie mit nur einem Argument zu übergeben? Funktionsargumente, die eher Optionen als zwingende Voraussetzung für eine Funktion darstellen, haben in R oftmals *Default*-Werte – Standardwerte, auf die die Funktion zurückgreift, wenn diese nicht im Funktionsaufruf angegeben wurden. So ist es auch bei der `mean()`-Funktion, die wie folgt definiert ist: `mean(x, trim = 0, na.rm = FALSE)`:

- `x` hat keinen Default-Wert und muss daher zwingend angegeben werden. Ohne Angabe von `x` wird eine Fehlermeldung zurückgegeben.
- `trim` hat einen Default-Wert von `0`, ohne Angabe von `trim` werden also keine Werte abgeschnitten.
- `na.rm` hat einen Default-Wert von `FALSE`, standardmäßig werden fehlende Werte also nicht ausgeschlossen.

```
mean()
```

```
## Error in mean.default(): argument "x" is missing, with no default
```

```
mean(c(1, 2, 3, NA))
```

```
## [1] NA
```

```
mean(c(1, 2, 3, NA), na.rm = TRUE)
```

```
## [1] 2
```

In der Regel haben insbesondere Argumente, die eher eine Option denn zwingende Voraussetzung für die Ausführung einer Funktion darstellen, Default-Werte, sodass diese bei der gebräuchlichsten Verwendung weggelassen werden können.

3.1.2 Hilfestellungen zu Funktionen

R und RStudio bieten verschiedene Möglichkeiten, Hilfestellungen zu Funktionen anzuzeigen – also z. B. zur Verwendung der Funktion, ihren Argumenten, den zugehörigen Default-Werten usw.

Tippen Sie einen Funktionsnamen ein, so macht Ihnen R Vorschläge, welche Funktion Sie meinen könnten. Daneben wird ein Tooltip angezeigt, der die

Funktion beschreibt. Geben Sie beispielsweise die Buchstaben `rou` in ein R-Skript oder die Konsole ein, sollten Sie nach kurzer Zeit eine Auswahl von Funktionen sehen, die mit eben diesen Buchstaben beginnen, darunter an erster Stelle die `round()`-Funktion. Die anderen vorgeschlagenen Funktionen können Sie mit den Cursortasten ↑ und ↓ wählen. Neben der ausgewählten Funktion wird die Funktionsbeschreibung angezeigt.

Drücken Sie nun die `tab`-Taste, wird nicht nur die Funktion inklusive Klammern `()` eingefügt; ein Tooltip zeigt Ihnen zudem noch die Argumente am. Im Falle der `round()`-Funktion sollte `round(x, digits = 0)` angezeigt werden. Daran sehen Sie, dass `round()` zwei Argumente annimmt – ein Argument `x` ohne Default-Wert und ein Argument `digits` mit dem Default-Wert `0` – ohne Angabe des `digits`-Arguments rundet `round()` also standardmäßig auf ganze Zahlen ohne Nachkommastellen.

Oftmals benötigen Sie jedoch noch mehr Informationen zu einer Funktion. In diesem Fall können Sie vor einen beliebigen Funktionsnamen ein `?` stellen und ausführen – dieser Befehl öffnet die Dokumentationsseite der jeweiligen Funktion im Bereich rechts unten in RStudio:

```
?mean()
```

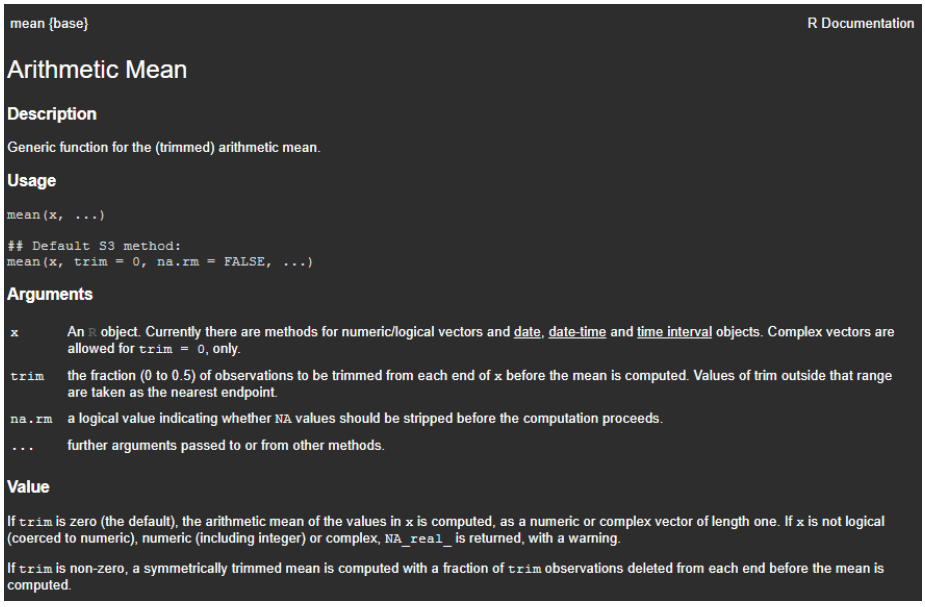


Figure 3.1: Dokumentation der `mean()`-Funktion

Dokumentationsseiten in R sind zumeist nach demselben Schema aufgebaut:

- Beschriftung der Funktion. In Falle von `mean()` also “Arithmetic Mean”.

- *Description*: Kurze Beschreibung der Funktion.
- *Usage*: Verwendung der Funktion. Anhand von `mean(x, trim = 0, na.rm = FALSE, ...)` können wir die Argumente und deren Default-Werte ablesen. Die `...` weisen hier auf einen Spezialfall hin, dass bestimmte Objekttypen und Datenstrukturen noch zusätzliche Argumente für die Funktion bereitstellen können – dies braucht uns an dieser Stelle aber nicht weiter zu kümmern.
- *Arguments*: Hier werden alle Argumente der Funktion aufgelistet und ausführlicher beschrieben. Wir erfahren im Falle von `mean()` beispielsweise, welche Objekttypen für `x` unterstützt werden, was `trim` bedeutet etc.
- *Value*: Hier wird das Resultat, das die Funktion ausgibt, beschrieben.

Außerdem können Funktionsdokumentationen noch weitere Abschnitte, beispielsweise Verwendungsbeispiele oder Literaturverweise, enthalten. Es ist daher immer sinnvoll, vor der ersten Verwendung einer Funktion die zugehörige Dokumentation zu konsultieren.

3.2 Eigene Funktionen erstellen

Wir wissen nun einiges über Funktionen; tatsächlich reicht unser Wissen sogar schon aus, um eigene Funktionen zu *programmieren*. Wir beschäftigen uns daher abschließend in diesem Kapitel mit der Frage, warum und wann wir eigene Funktionen schreiben sollten, und schauen uns an, wie wir unseren Code in Funktionen übersetzen und so weiterverwertbar machen.

3.2.1 Funktionen erstellen: wann und warum?

Viele Schritte in der Datenaufbereitung und -analye wiederholen sich: oft müssen wir mehrere Variablen transformieren, meistens führen wir zur Deskription die gleichen oder zumindest ähnliche Schritte durch, interessieren uns bei Auswertungsverfahren immer wieder für die gleichen Kennwerte.

Funktionen sind das zentrale Element des Programmierens. Im Prinzip handelt es sich bei Funktionen um Code-Fragmente oder Skripte, die durch die Verwendung von "Platzhaltern" (= Variablen, Argumente) so angepasst werden, dass sie immer wieder auf vergleichbare Problemstellungen angewendet werden können.

Nehmen wir das Beispiel der Variablendeskription. Wenn wir Daten erhoben haben, sind wir bei metrischen Variablen in der Regel zunächst an den selben Kennwerten interessiert: der Anzahl der Werte, dem arithmetischen Mittel, der Standardabweichung, dem Minimum und Maximum, eventuell noch dem Median oder anderen Perzentilen. Mit unserem bisherigen Wissen (und drei neuer, aber ebenso einfacher Funktionen) können wir diese Kennwerte einzeln

anfordern, zum Beispiel für die Variable `Sepal.Length` im Beispiel-Datensatz `iris`:

```
length(iris$Sepal.Length) # Anzahl Werte
mean(iris$Sepal.Length) # Arithmetisches Mittel
sd(iris$Sepal.Length) # Standardabweichung
min(iris$Sepal.Length) # Minimum
max(iris$Sepal.Length) # Maximum
median(iris$Sepal.Length) # Median
```

```
## [1] 150
## [1] 5.843333
## [1] 0.8280661
## [1] 4.3
## [1] 7.9
## [1] 5.8
```

Die Ausgabe ist jedoch etwas unübersichtlich. Schöner wird es, wenn wir alle Werte in einem benannten Vektor ablegen:

```
sepal_length_descriptives <- c(
  n = length(iris$Sepal.Length),
  M = mean(iris$Sepal.Length),
  SD = sd(iris$Sepal.Length),
  Minimum = min(iris$Sepal.Length),
  Maximum = max(iris$Sepal.Length),
  Median = median(iris$Sepal.Length)
)

sepal_length_descriptives
```

```
##           n           M          SD     Minimum     Maximum      Median
## 150.0000000   5.8433333   0.8280661   4.3000000   7.9000000   5.8000000
```

Das sieht doch gleich übersichtlicher aus – in einem Vektor haben wir alle relevanten Werte gespeichert und können diese auf einen Schlag anzeigen.

Nun haben wir im `iris`-Datensatz aber noch weitere Variablen, die wir ebenfalls auf diese Art beschreiben möchten (ganz zu schweigen von zukünftigen Datensätzen, für deren Variablen diese Beschreibung ebenfalls relevant ist). Wir könnten natürlich den obigen Code für jede weitere Variablen kopieren und den Variablennamen austauschen; das würde aber unser Skript unnötig aufblähen und unübersichtlicher machen.

Sobald Sie Code(-Fragmente) mehrfach verwenden möchten, ist daher der Zeitpunkt gekommen, darübernachzudenken, ob es nicht sinnvoll ist, ihn in eine Funktion umzuwandeln. Dies hat einige entscheidende Vorteile:

- Code ist einfacher wiederzuverwenden. Anstatt mehrerer Codezeilen ist in Zukunft für dasselbe Ergebnis dann nur noch eine Codezeile – der Funktionsaufruf – erforderlich.
- Skripte werden weniger redundant und dadurch übersichtlicher.
- Fehlerkorrekturen und Anpassungen werden vereinfacht – möchten wir im obigen Beispiel etwa noch einen weiteren Kennwert (z. B. die Spannweite) ergänzen, müssten wir den Code für jede einzelne Variable erneut anpassen; hätten wir eine entsprechende Funktion, wäre die Anpassung nur einmal im Funktionscode nötig.

Schauen wir uns also an, wie wir aus obigen Code eine Funktion erstellen können.

3.2.2 Aufbau und Erstellung von Funktionen

Funktionen in R sind ebenfalls Objekte und werden dementsprechend erstellt: indem einem Objektnamen durch den Zuweisungs-Operator `<-` die Funktion zugewiesen wird. Neben dem Namen haben Funktionen zwei zentrale Komponenten:

- die Funktionsargumente, wie wir sie auch schon vom Aufruf von eingebauten Funktionen kennen
- den *Body* der Funktion, der den Code enthält, den die Funktion ausführt

Die Erstellung von Funktionen läuft dabei nach folgendem Schema ab:

```
funktionsname <- function(argument_1, argument_2) {
  # Body: Code, der ausgeführt wird
}
```

Wir weisen also einem Objektnamen ein Funktionsobjekt zu; dies geschieht mit der Funktion `function()`, wobei wir innerhalb der Klammern die *Argumente* definieren, die wir im Code der Funktion verwenden möchten. Zwischen zwei geschweiften Klammern `{}` steht dann der Code, den wir ausführen wollen.

Sehen wir uns nochmals unser obiges Beispiel an:

```
sepal_length_descriptives <- c(
  n = length(iris$Sepal.Length),
  M = mean(iris$Sepal.Length),
  SD = sd(iris$Sepal.Length),
  Minimum = min(iris$Sepal.Length),
  Maximum = max(iris$Sepal.Length),
  Median = median(iris$Sepal.Length)
)
```

Um den Code auf eine andere Variable anzuwenden, müssten wir jeweils `iris$Sepal.Length` ersetzen – z. B. durch `iris$Sepal.Width`, `iris$Petal.Length` oder eine Variable aus einem anderen Datensatz. Wir möchten diesen Teil also durch einen Platzhalter ersetzen, den wir dann als Funktionsargument übergeben können. Mit einem generischen Platzhalter, den wir der Einfachheit halber (und vollkommen willkürlich!) als `x` bezeichnen, sähe der Code also wie folgt aus:

```
descriptives_vector <- c(
  n = length(x),
  M = mean(x),
  SD = sd(x),
  Minimum = min(x),
  Maximum = max(x),
  Median = median(x)
)
```

Zuletzt müssen wir diesen Code nun nur noch als Funktion einem Objekt zuweisen – und dabei unserer Funktionen einen treffenden Namen geben, z. B. `descriptives`, und dabei `x` als Funktionsargument definieren:

```
descriptives <- function(x) { # Wir definieren 'x' als Argument
  descriptives_vector <- c(
    n = length(x),
    M = mean(x),
    SD = sd(x),
    Minimum = min(x),
    Maximum = max(x),
    Median = median(x)
  )
}
```

Einmal ausgeführt, taucht unsere neue Funktion nun in unserer Arbeitsumgebung auf (siehe den Bereich *Environment* rechts oben in RStudio, in dem `descriptives` nun unter *Functions* erscheinen sollte) und wir können Sie auf beliebige Vektoren anwenden:

```
descriptives(iris$Sepal.Length)
```

Nur warum sehen wir keine Ausgabe? Das liegt daran, dass wir noch nicht definiert haben, was die Funktion zurückgeben soll. Wir erstellen derzeit lediglich den Vektor `descriptives_vector`, machen aber noch nichts mit ihm – ähnlich, wie uns die Konsole noch keine Ausgabe anzeigt wenn wir ein Objekt lediglich zuweisen.

3.2.2.1 Rückgabe-Werte

Es gibt zwei Möglichkeiten, die Ausgabe einer Funktion zu definieren:

- generell wird das zuletzt im Funktionscode ausgegebene Objekt zurückgegeben. Wir könnten also im Funktionscode ganz am Ende noch eine Zeile hinzufügen, in der unser neu erstellter Vektor ausgeben wird, in dem wir einfach den Objektnamen tippen `descriptives_vector`.
- mit der speziellen Funktion `return()` können wir im Funktionscode explizit ein im Funktionscode erstelltes Objekt angeben, das zurückgegeben werden soll.

Wir müssen also noch eine kleine Anpassung vornehmen:

```
descriptives <- function(x) {
  descriptives_vector <- c(
    n = length(x),
    M = mean(x),
    SD = sd(x),
    Minimum = min(x),
    Maximum = max(x),
    Median = median(x)
  )

  return(descriptives_vector) # Oder alternativ: lediglich 'descriptives_vector'
}
```

Unsere Funktion ist nun flexibel für numerische Vektoren einsetzbar:

```
descriptives(iris$Sepal.Length)
```

```
##           n           M          SD     Minimum     Maximum      Median
## 150.0000000   5.8433333   0.8280661   4.3000000   7.9000000   5.8000000
```

```
descriptives(iris$Petal.Length)
```

```
##           n           M          SD     Minimum     Maximum      Median
## 150.000000   3.758000   1.765298   1.000000   6.900000   4.350000
```

```
descriptives(c(1, 2, 3, 4, 5, 6))
```

```
##        n        M       SD  Minimum  Maximum   Median
## 6.000000 3.500000 1.870829 1.000000 6.000000 3.500000
```

3.2.2.2 Funktionsargumente und Default-Werte

Unsere Funktion funktioniert also schon ganz gut; im nächsten Schritt möchten wir sie aber noch verbessern, um auch Problemfälle abzudecken. Schauen wir uns einmal an, was passiert, wenn wir einen numerischen Vektor mit fehlenden Werten übergeben:

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

```
##        n        M       SD Minimum Maximum  Median
##        5       NA       NA      NA      NA      NA
```

Wir sehen, dass alle Kennwerte (außer der Anzahl der Werte) ebenfalls `NA` sind. Wir erinnern uns, dass Funktionen wie `mean()` oder `sd()` bei vorhandenen fehlenden Werten auch einen fehlenden Wert zurückgeben – es sei denn, wir übergeben zusätzlich das Argument `na.rm = TRUE`, sodass fehlende Werte vor der Berechnung gelöscht werden. Wir möchten dieses Argument nun auch in unserer Funktion nutzen.

Hierzu benötigen wir ein weiteres Funktionsargument, das wir dann als Platzhalter für das `na.rm`-Argument der jeweiligen Kennwert-Funktionen verwenden können. Aus Konsistenzgründen nennen wir es ebenfalls `na.rm`:

```
descriptives <- function(x, na.rm) { # Wir fügen ein weiteres Argument 'na.rm' hinzu...
  descriptives_vector <- c(
    n = length(x),
    M = mean(x, na.rm = na.rm),       # ... und übergeben dessen Wert an
    SD = sd(x, na.rm = na.rm),        # das jeweilige 'na.rm'-Argument der
    Minimum = min(x, na.rm = na.rm), # Kennwert-Funktionen
    Maximum = max(x, na.rm = na.rm),
    Median = median(x, na.rm = na.rm)
  )

  return(descriptives_vector)
}
```

Wir können unsere Funktion nun auch für Vektoren mit fehlenden Werten verwenden, indem wir analog zu `mean()`, `sd()` usw. das Argument `na.rm = TRUE` übergeben.

```
descriptives(c(1, 2, 3, 4, NA), na.rm = TRUE)
```

```
##        n        M       SD  Minimum  Maximum   Median
## 5.000000 2.500000 1.290994 1.000000 4.000000 2.500000
```

Allerdings ist unsere Funktion nun weniger komfortabel einzusetzen, wenn wir dieses Argument weglassen möchten:

```
descriptives(iris$Sepal.Length)
```

```
## Error in mean(x, na.rm = na.rm): argument "na.rm" is missing, with no default
```

Wie wir unter 3.1.1.2 gesehen haben, haben viele Funktionsargumente, die eher optionalen Charakter aufweisen, Default-Werte. Wir können diese ganz einfach bei der Erstellung definieren, indem wir den Default-Wert per = dem jeweiligen Funktionsargument zuweisen:

```
descriptives <- function(x, na.rm = FALSE) { # Default-Wert für 'na.rm' = FALSE
  descriptives_vector <- c(
    n = length(x),
    M = mean(x, na.rm = na.rm),
    SD = sd(x, na.rm = na.rm),
    Minimum = min(x, na.rm = na.rm),
    Maximum = max(x, na.rm = na.rm),
    Median = median(x, na.rm = na.rm)
  )

  return(descriptives_vector)
}
```

Dieser Default-Wert wird nun also immer verwendet, wenn wir das Argument nicht angegeben haben. Unsere Funktion ist somit noch flexibler geworden:

```
descriptives(iris$Petal.Width)
```

```
##           n           M          SD     Minimum     Maximum      Median
## 150.0000000   1.1993333   0.7622377   0.1000000   2.5000000   1.3000000
```

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

```
##       n       M      SD Minimum Maximum  Median
##       5      NA      NA      NA      NA      NA
```

```
descriptives(c(1, 2, 3, 4, NA), na.rm = TRUE)
```

```
##        n        M       SD  Minimum  Maximum   Median
## 5.000000 2.500000 1.290994 1.000000 4.000000 2.500000
```

3.3 U̇bungsaufgaben

Erstellen Sie für die folgenden Übungsaufgaben eine eigene Skriptdatei und speichern diese als `ue3_nachname.R` ab. Antworten auf Fragen können Sie direkt als Kommentare in das Skript einfügen.

U̇bungsaufgabe 3.1. Funktionen aufrufen und Dokumentation konsultieren:

Die Funktion `seq()` kann verwendet werden, um Zahlenfolgen zu erstellen. Lesen Sie sich die Dokumentationsseite der Funktion durch. Wie müssen Sie die Funktion aufrufen (und welche Argumente benötigen Sie dafür), um eine Zahlenfolge von `0` bis `100` in 5er-Schritten zu erzeugen?

U̇bungsaufgabe 3.2. Funktionen erstellen:

Erstellen Sie eine Funktion `fahrenheit_to_celsius`, die eine Temperaturangabe in Grad Fahrenheit (als numerischen Wert, also z. B. `80`) in Grad Celsius umrechnet und diesen Wert zurückgibt.

Die Formel zur Umrechnung von Fahrenheit in Celsius lautet řC = (řF − 32)Ö5/9.

U̇bungsaufgabe 3.3. Funktionen erstellen für Fortgeschrittene:

Käpseles-Aufgabe (optional)

Erweitern Sie die in Kapitel 3.2.2.2 erstellte Funktion `descriptives` um folgende Features:

- Die Kennwerte sollen um die Anzahl der fehlenden Werte ergänzt werden (dieses Element soll den Namen `missing` tragen). Hierfür benötigen Sie die Funktion `is.na()`, die für jeden Wert eines Vektors prüft, ob es sich dabei um `NA` handelt oder nicht und entsprechend `TRUE` oder `FALSE` zurückgibt.
- Die Funktion soll alle Kennwerte einheitlich auf eine gewünschte Anzahl an Nachkommastellen, standardmäßig auf 2 Nachkommastellen runden.

Hier nochmals der bisherige Funktionscode:

```r
descriptives <- function(x, na.rm = FALSE) {
  descriptives_vector <- c(
    n = length(x),
    M = mean(x, na.rm = na.rm),
    SD = sd(x, na.rm = na.rm),
    Minimum = min(x, na.rm = na.rm),
    Maximum = max(x, na.rm = na.rm),
    Median = median(x, na.rm = na.rm)
  )

  return(descriptives_vector)
}
```

Chapter 4

Kontrollstrukturen

Mittels *Kontrollstrukturen* können wir definieren, ob und wie oft Code ausgeführt wird; unterschieden wird hierbei vorrangig zwischen *Bedingungen* – d. h. Code, der nur ausgeführt wird, wenn eine bestimmte Bedingung erfüllt ist – und *Iterationen* (wiederholtes ausführen von Code, auch als *Schleifen* bezeichnet).

Bedingungen und Iterationen sind Konzepte, die ebenso wie Funktionen in quasi allen Programmiersprachen zu finden sind. Ziel dieses Kapitels ist es, ein Grundverständnis beider Konzepte zu erlangen; relevant werden diese spätestens, wenn wir über APIs oder Webscraping Daten erheben werden.

4.1 Bedingungen

Bedingungen führen Code abhängig davon aus, ob eine definierte Bedingung `TRUE` oder `FALSE` ist.

4.1.1 `if`-Bedingungen

Bedingungen werden in R – ebenso wie in vielen anderen Programmiersprachen – über das Schlüsselwort `if` umgesetzt, dem eine in runde Klammern `()` gesetzte, logische Bedingung folgt. Wie auch bei Funktionen wird der bedingt auszuführende Code in geschweifte Klammern gesetzt `{}`:

```
if (bedingung) {
  # Code der ausgeführt wird, wenn bedingung = TRUE ist
}
```

Die `bedingung` muss dabei in dem Objekttypen `logical` (also `TRUE` oder `FALSE`, siehe Kapitel 2.2.3) resultieren; in den meisten Fällen wird daher ein logischer Vergleich in den Klammern `()` durchgeführt:

```
x <- 2
aktion <- "verdoppeln"

if (aktion == "verdoppeln") {
  x <- x * 2
}

x
```

```
## [1] 4
```

4.1.2 `if-else`-Bedingungen

Mit dem Schlüsselwort `else` können wir anschließend einen Codeblock definieren, der ausgeführt werden soll, wenn die Bedingung nicht zutrifft:

```
punktzahl <- 45

if (punktzahl > 50) {
  status <- "bestanden"
} else {
  status <- "nicht bestanden"
}

status
```

```
## [1] "nicht bestanden"
```

4.1.3 Bedingungen verketten

Durch die Kombination von `else` und `if` können wir auch beliebig viele Bedingungen hintereinander prüfen:

```
steak_temperatur <- 56

if (steak_temperatur < 45) {
  garstufe <- "raw"
} else if (steak_temperatur < 53) {
  garstufe <- "rare"
```

```
} else if (steak_temperatur < 57) {
  garstufe <- "medium rare"
} else if (steak_temperatur < 63) {
  garstufe <- "medium"
} else {
  garstufe <- "well done"
}

garstufe
```

```
## [1] "medium rare"
```

4.1.4 Mehrere Bedingungen

Mittels Boolescher Operatoren können wir mehrere Bedingungen miteinander verknüpfen, um etwa zu prüfen ob alle Bedingungen zutreffen (`UND`-Verknüpfung), mindestens eine Bedinung zutrifft (`ODER`-Verknüpfung), oder das Gegenteil einer Bedingung zutrifft (`NICHT`-Verknüpfung). Die gebräuchlichsten Operatoren sind:

Table 4.1: Boolesche Operatoren in R

Operator	Verknüpfung	Beispiele
`&`	und	`1 == 1 & 2 == 2` ergibt `TRUE``1 == 1 & 1 == 3` ergibt `FALSE`
`|`	oder	`1 == 1 | 2 == 2` ergibt `TRUE``1 == 1 | 1 == 3` ergibt `TRUE`
`!`	nicht	`!(1 == 1)` ergibt `FALSE` (`!` wird der Bedingung vorangestellt)`!(1 == 3)` ergibt `TRUE`

Als Beispiel verknüpfen wir die Bedingungen für ein Schaltjahr. Damit ein Jahr ein Schaltjahr ist, müssen folgende Bedingungen erfüllt sein:

- Die Jahreszahl ist durch `400` teilbar `ODER`
- Die Jahreszahl ist durch `4` teilbar `UND` ist gleichzeitig `NICHT` durch `100` teilbar.

Ob eine Zahl durch eine andere Zahl teilbar ist, können wir mit dem Modulo-Operator `%%` prüfen, der den ganzzahligen Rest der Division ausgibt – mit anderen Worten: wenn das Ergebnis der Modulo-Operation `0` ist, dann ist Zahl 1 durch Zahl 2 teilbar, ansonsten nicht.

```
year <- 2020

year %% 400 == 0 | (year %% 4 == 0 & !(year %% 100 == 0))
# Natürlich könnten wir hinten auch prüfen, ob year %% 100 != 0 ist und
# nur die UND-Verknüpfung nutzen, aber ich wollte alle drei Booleschen
# Operatoren in einer Prüfung unterbringen
```

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

Oder als Funktion verpackt:

```
is_leap_year <- function(year) {
  year %% 400 == 0 | (year %% 4 == 0 & !(year %% 100 == 0))
}

is_leap_year(c(1900, 2000, 2016, 2018, 2020))
```

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

4.1.5 Mehrere Prüfwerte

Oft wollen wir prüfen, ob ein Wert zu einer Reihe an Werten gehört – beispielsweise wenn wir Werte kategorisieren möchten. Wir können dies mit einer `ODER`-Verknüpfung erreichen:

```
food <- "Banane"

if (food == "Apfel" | food == "Orange" | food == "Banane") {
  food_category <- "Obst"
}
```

Allerdings wird diese Verknüpfung schnell umständlich, wenn wir viele Prüfwerte haben, im Beispiel also nicht nur 3 Obstsorten, sondern 5, 10 oder 123. Hier hilft uns der Operator `%in%`, der testet, ob ein Wert in einem Vektor an Werten vorhanden ist:

```
"Banane" %in% c("Apfel", "Orange", "Banane", "Zitrone", "Mango", "Kumquat")
```

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

Das funktioniert auch mit mehreren Werten auf der linken Seite der Prüfung:

```
c("Banane", "Mango", "Leberkäse") %in% c("Apfel", "Orange", "Banane", "Zitrone", "Mango", "Kumqua
```

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

Und natürlich können wir den Prüfvektor vorab zuweisen:

```
obstsorten <- c("Apfel", "Orange", "Banane", "Zitrone", "Mango", "Kumquat")
food <- "Kumquat"

if (food %in% obstsorten) {
  food_category <- "Obst"
} else {
  food_category <- "Kein Obst"
}

food_category
```

```
## [1] "Obst"
```

4.2 Iterationen

Mittels Iterationen führen wir ein Code-Fragment wiederholt für verschiedene Input-Objekte aus. R bietet viele unterschiedliche Möglichkeiten für Iterationen – für den Anfang genügen wir uns mit vektorisierten Funktionen, `for`-Loops und `while`-Loops:

4.2.1 Vektorisierte Funktionen

Tatsächlich haben wir unbewusst bereits mehrfach mit Iterationen gearbeitet, indem wir *vektorisierte* Funktionen eingesetzt haben – Funktionen, die automatisch auf jedes Element eines Vektors angewendet werden. Dazu zählen beispielsweise alle arithmetischen Operatoren und nahezu alle Funktionen in der Basis-Version von R:

```
zahlen <- c(5, 10, 42)
zahlen - 10
zahlen * 3
log(zahlen) # Berechnet den natürlichen Logarithmus
tolower(c("Text 1", "TEXT 2", "TEXT DREI")) # wandelt Text in Kleinbuchstaben um
```

```
## [1] -5  0 32
## [1]  15  30 126
## [1] 1.609438 2.302585 3.737670
## [1] "text 1"    "text 2"    "text drei"
```

Wir haben außerdem in Kapitel 2.3.1.3 gesehen, dass wir so auch mit zwei gleichlangen Vektoren effizient rechnen können:

```
c(2, 3, 4) * c(2, 5, 10)
```

```
## [1]  4 15 40
```

Was passiert, wenn beide Vektoren nicht gleichlang sind? Hier tritt eine Eigenschaft von R zu Tage, die sich *Recycling* nennt: Ist der längere Vektor durch den kürzeren Vektor teilbar, wiederholt R den kürzeren Vektor einfach entsprechend oft:

```
c(2, 4) * c(2, 3, 5, 10)
```

```
## [1]  4 12 10 40
```

Ist das nicht der Fall, produziert R hingegen eine Fehlermeldung:

```
c(2, 4) * c(2, 3, 5)
```

```
## Warning in c(2, 4) * c(2, 3, 5): longer object length is not a multiple of shorter o
```

```
## [1]  4 12 10
```

4.2.2 `for`-Loops

`for`-Loops führen (beliebig viele Zeilen) Code für jedes Element eines Vektors durch. Die Grundform eines `for`-Loops sieht wie folgt aus:

```
for (element in vektor) {
  # Body: Code, der ausgeführt wird
}
```

`element` ist hierbei ein Objekt, dem nach jeder Ausführung des Codes in den geschweiften Klammern `{}` das nächste Element aus dem angegebenen Vektor zugewiesen wird. Wir können dem Element einen beliebigen Objektnamen geben und es dann ähnlich wie in Funktionen als Platzhalter im Loop-Code verwenden. Für einfache Loops wird meistens der Objektname `i` verwendet:

```
zahlen <- 1:5

for (i in zahlen) {
  neue_zahl <- i * i - 1
  print(neue_zahl) # print() schreibt ein Objekt in den Konsolenoutput
}
```

```
## [1] 0
## [1] 3
## [1] 8
## [1] 15
## [1] 24
```

Nutzen wir ein etwas anwendungsbezogeneres Beispiel. Nehmen wir an, wir möchten den Mittelwert aller (numerischen) Variablen in einem Datensatz ausgeben. Wir könnten die `mean()`-Funktion natürlich einfach händisch für jede Variable anfordern:

```
mean(iris$Sepal.Length)
mean(iris$Sepal.Width)
mean(iris$Petal.Length)
# usw.
```

```
## [1] 5.843333
## [1] 3.057333
## [1] 3.758
```

Im Falle von `iris` bei nur vier numerischen Variablen wäre das noch problemlos möglich, bei längeren Datensätzen hätten wir aber schnell sehr viel zu tun – und in allen Fällen produzieren wir sehr viel redundanten Code. Eleganter lösen wir das mit einem `for`-Loop und einem Vektor, der alle uns interessierenden Variablennamen enthält:

```
variablen <- c("Sepal.Length", "Sepal.Width", "Petal.Length", "Petal.Width")

mittelwerte <- c()

for (variable in variablen) {
 mittelwerte[variable] <- mean(iris[[variable]])
}

mittelwerte
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##     5.843333     3.057333     3.758000     1.199333
```

Was passiert hier?

1. Wir erstellen einen Vektor mit den uns interessierenden Variablen `variablen` sowie einen leeren Vektor `mittelwerte`.
2. Der `for`-Loop beginnt: `variable` bekommt das erste Element aus `variablen`, also `"Sepal.Length"` zugewiesen. Dann wird der Code in den geschweiften Klammern `{}` ausgeführt:
   - `iris[[variable]]` extrahiert aus dem `iris`-Datensatz die Spalte mit dem Namen, der in `variable` gespeichert ist – aktuell also `"Sepal.Length"`. (Wir kennen bisher nur einfache eckige Klammern `[]` zur Extraktion; dabei wird der Objekttyp `data.frame` beibehalten und wir können auch mehrere Variablen extrahieren. Mit den doppelten eckigen Klammern `[[]]` wird hingegen nur eine einzige Variable extrahiert und in den Objekttyp `vector` umgewandelt. Diesen Objekttypen benötigen wir für die `mean()`-Funktion.)
   - Wir berechnen davon den Mittelwert mittels `mean()`.
   - Der Vektor `mittelwerte` erhält ein Element mit dem Namen, der in `variable` gespeichert ist – aktuell also ebenfalls `"Sepal.Length"`. Diesem Element weisen wir den berechneten Mittelwert zu.
3. Der Loop ist nun einmal durchlaufen und beginnt von vorne. Dabei wird `variable` nun das zweite Element von `variablen`, "Sepal.Width", zugewiesen. Dann wird der Code in den geschweiften Klammern erneut ausgeführt.
4. Diese Schritte werden so oft wiederholt, bis wir am Ende von `variablen` angekommen sind und jedes Element aus `variablen` einmal `variable` zugewiesen wurde.
5. Als Resultat erhalten wir einen benannten Vektor `mittelwerte`, der alle Mittelwerte enthält.

Mittels `for`-Loops können wir also sehr schnell Teile unseres Codes automatisieren und als Grundprinzip finden sich `for`-Loops in nahezu allen Programmiersprachen. Wir werden jedoch in Kürze noch Funktionen kennenlernen, die uns Iterationen nochmals deutlich komfortabler gestalten.

4.2.3 `while`-Loops

Bei `for`-Loops wissen wir vorab, wie oft der Loop ausgeführt wird – nämlich für jedes Element, das der Vektor, über den wir loopen, enthält. Manchmal ist es uns aber nicht vorab bewusst, wie oft ein Loop ausgeführt werden soll. In diesem Fall können wir `while`-Loops verwenden, die so lange ausgeführt werden, wie eine Bedingung als `TRUE` erfüllt ist:

```
while (bedingung) {
  # Body: Code, der ausgeführt wird
}
```

Entsprechend benötigen wir eine Bedingung, die bei jeder Iteration wieder überprüft wird. Das Prüfkriterium sollten wir also im Body des Loops auch anpassen, da der Loop sonst unendlich läuft.

Als Beispiel schreiben wir einen Loop, der in 5er-Schritten von `50` bis `100` zählt:

```
x <- 50

while (x <= 100) {
  print(x)
  x <- x + 5
}
```

```
## [1] 50
## [1] 55
## [1] 60
## [1] 65
## [1] 70
## [1] 75
## [1] 80
## [1] 85
## [1] 90
## [1] 95
## [1] 100
```

Was passiert hier?

1. Wir weisen `x` den Ausgangswert `50` zu.
2. Der `while`-Loop beginnt. Wir prüfen zunächst ob `x` kleiner gleich `100`, was aktuell der Fall ist. Dann wird der Code ausgeführt:
   - Wir schreiben zunächst den aktuellen Wert von `x` in die Konsole.
   - Dann addieren wir `5` zu `x`. `x` ist nun `55`.
3. Der Loop ist nun einmal durchlaufen und beginnt von vorne. Erneut wird geprüft, ob `x` kleiner gleich `100` ist. Dies ist weiterhin der Fall, der Code wird also erneut ausgeführt.
4. Dies wird so lange wiederholt, bis die Bedingung nicht mehr erfüllt ist. Dies geschieht, nachdem `100` in die Konsole geschrieben wurde, da danach auf `x` nochmals `5` addiert werden und `x` am Ende des Loops folglich `105` ist. Die nächste Prüfung `105 <= 100` resultiert in `FALSE`, der Loop wird abgebrochen.

Einen häufigen Anwendungsfall für `while`-Loops lernen wir kennen, sobald wir mit APIs arbeiten. Wollen wir etwa Tweets zu einem bestimmten Hashtag herunterladen, wissen wir vorab nicht, um wie viele Tweets es sich handelt. Mit einem `while`-Loop könnten wir daher festlegen, dass wir den Code zum Tweets-aus-der-API-ziehen ausführen, bis diese keine weiteren zurückgibt.

Herzlichen Glückwunsch, Sie beherrschen nun die zentralen Grundlagen von R (und fast jeder anderen Programmiersprache) und könnten theoretisch alle weiteren Funktionen von Hand schreiben. In der Praxis wurde aber vermutlich so gut wie jedes Problem, das Ihnen im datenanalytischen Kontext begegnet, schon von jemand anderem gelöst. Wir schauen uns also als nächstes an, wie wir auf Funktionen von anderen in Form von Packages zugreifen können.

4.3 Übungsaufgaben

Erstellen Sie für die folgenden Übungsaufgaben eine eigene Skriptdatei und speichern diese als `ue4_nachname.R` ab. Antworten auf Fragen können Sie direkt als Kommentare in das Skript einfügen.

Übungsaufgabe 4.1. Bedingungen:

Wir haben in einer Studie das Nachrichtennutzungsverhalten erhoben und möchten dieses nun basierend auf zwei Variablen in einer neuen Variablen `news_category` kategorisieren:

- Falls in der Variable `news_channel` nicht `"Internet"` angeben wurde, soll die neue Variable `news_category` den Wert `"Offline"` lauten.
- Falls dort "`Internet`" angegeben wurde, steht eine weitere Unterteilung an:
  - Falls in `news_website` die Werte `"Twitter"`, `"Facebook"` oder `"Instagram"` angegeben wurden, soll `news_category` den Wert `"Online: SNS"` haben.
  - Bei allen anderen Werten von `news_website` soll `news_category` den Wert `"Online: Sonstige"` bekommen

Bilden Sie im folgenden Codebeispiel diesen Entscheidungsbaum mit Bedingungen nach:

```
news_channel <- "Internet"
news_website <- "Facebook"
```

```
#       Ihr Code hier
#
#  _.-      _,-'""`-._
# (,-.`._,'(       |\`-/|
#     `-.-' \ )-`( , o o)
#           `-    \`_`"'-
#
# (Diese Katze sieht in R kopiert besser aus als hier in der Webansicht)

news_category # Wenn alles geklappt hat, sollte "Online: SNS" herauskommen
```

Übungsaufgabe 4.2. Iterationen:

Vervollständigen Sie in der folgenden Funktion alle Platzhalter `___`, sodass diese für alle numerischen Variablen eines Datensatzes Mittelwert und Standardabweichung ausgibt:

```
numeric_summary <- function(data) {

  # Alle Variablennamen in Vektor speichern
  variables <- names(data)

  # Leere Liste für Ausgabe vorbereiten
  summary_list <- list()

  # Über alle Variablen iterieren
  ___ (___) { # Hier die ___ ersetzen
    variable_vector <- data[[___]] # Und hier ebenfalls

    if (is.numeric(variable_vector)) { # Prüfen ob die Variable numerisch ist

      # Mittelwert und Standardabweichung dieser Variablen der summary_list hinzufügen
      summary_list[[___]] <- c( # Hier wieder die ___ ersetzen
        M = mean(variable_vector),
        SD = sd(variable_vector)
      )
    }

  }

  # Summary List ausgeben
  return(summary_list)
}
```

Testen Sie die fertige Version mit den `iris`- und `mtcars`-Datensätzen.

Chapter 5

Packages

Bereits mit der Basisversion von R kommen wir schon recht weit. Der Erfolg von R geht aber nicht zuletzt darauf zurück, dass sich die Basisversion sehr einfach durch *Packages* erweitern lässt. Dabei handelt es sich um Sammlungen von Funktionen, deren Dokumentation und Zusatzinhalten (z. B. Beispieldaten), die R beispielsweise um neue Analyse- und Erhebungsverfahren erweitern oder bestehende Funktionalitäten vereinfachen.

Auf CRAN, dem *C*omprehensive *R* *A*rchive *N*etwork – Sie erinnern sich, von dort haben Sie auch R installiert –, stehen aktuell rund 15.500 Packages zur Verfügung, die so ziemlich jede Funktionalität abdecken, die man sich wünschen kann.

5.1 Packages installieren

Packages lassen von sich von CRAN sehr einfach direkt in R bzw. RStudio über die Funktion `install.packages("package_name")` installieren. Mit folgendem Befehl installieren wir beispielsweise das Paket `tibble`:

```
install.packages("tibble")
```

R lädt automatisch die benötigten Dateien (und Packages, sollte das gewünschte Package andere Packages voraussetzen) herunter und installiert diese, sodass Sie meist nach wenigen Sekunden mit dem Package arbeiten können. Einmal installierte Packages bleiben Ihnen erhalten, bis Sie den Computer wechseln oder zu einer neuen R-Version (siehe nächstes Kapitel) wechseln.

5.2 Packages nutzen

Packages werden über die Funktion `library(package_name)` für die aktuelle R-Session geladen, sodass Sie auf die enthaltenen Funktionen zugreifen können. Es bietet sich daher an, die `library()`-Befehle immer ganz oben in ein R-Skript zu schreiben, da von ihnen dann die Funktionalität des restlichen Skripts abhängig ist.

Laden wir einmal das `tibble`-Package:

```
library(tibble)
```

Was kann dieses Package nun? In der Regel haben wir davon natürlich schon eine Vorstellung, da wir das neue Package vermutlich über eine Google-Suche nach dessen Funktionalität gefunden haben (ganz allgemein funktioniert eine Suche nach *"R + [Name des gesuchten Verfahrens]"* meist gut). Wie auch bei den Basis-Funktionen können wir auch bei Funktionen aus Packages über ein vorangestelltes `?` die Dokumentationsseite der jeweiligen Funktion anzeigen.

Viele Packages kommen zudem mit einer oder mehreren *Vignetten*, längeren Dokumentationen, die die Funktionen eines Packages genauer erläutern, oft mit Anwendungsbeispielen. Wir können diese Vignetten über den Befehl `vignette()` öffnen, wobei wir dafür den Namen der Vignette kennen müssen – viele Packages bieten daher eine Vignette an, die nach dem Package selbst benannt ist. Alternativ öffnet sich über `browseVignettes("package_name")` in einem Webbrowser eine Übersicht über alle Vignetten, die zu einem bestimmten Package gehören.

Öffnen wir einmal die Vignette unseres neues Packages `tibble`:

```
vignette("tibble")
```

Wir erfahren also, dass das `tibble`-Package einen "modern take on data frames" einführt. Neben einigen technischen Details zeichnen sich "Tibbles" vor allem durch eine schickere Darstellung in der R-Konsole aus. Schauen wir uns das doch einmal an – die Funktion, um einen Dataframe in ein Tibble umzuwandeln lautet `as_tibble()`:

```
as_tibble(iris)
```

```
## # A tibble: 150 x 5
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##           <dbl>       <dbl>        <dbl>       <dbl> <fct>
##  1          5.1         3.5          1.4         0.2 setosa
##  2          4.9         3            1.4         0.2 setosa
```

```
##  3          4.7         3.2          1.3         0.2 setosa
##  4          4.6         3.1          1.5         0.2 setosa
##  5          5           3.6          1.4         0.2 setosa
##  6          5.4         3.9          1.7         0.4 setosa
##  7          4.6         3.4          1.4         0.3 setosa
##  8          5           3.4          1.5         0.2 setosa
##  9          4.4         2.9          1.4         0.2 setosa
## 10          4.9         3.1          1.5         0.1 setosa
## # ... with 140 more rows
```

Im Gegensatz zu Dataframes sehen wir auf den ersten Blick folgende Unterschiede:

- Wir erhalten zunächst eine Zeile mit den Dimensionen des Datensatzes: `A tibble: 150 x 5`
- Unter der Kopfzeile sind die Objekttypen der jeweiligen Spalten aufgelistet, hier also `dbl` (für `double`) und `fct` (für `factr`).
- Standardmäßig werden nur die ersten 10 Zeilen des Datensatzes angezeigt, sodass uns bei langen Datensätzen nicht die Konsole vollläuft.
- hier nicht sichtbar: negative Zahlen und fehlende Werte werden farbig hervorgehoben

Wir werden zukünftig viel mit Tibbles arbeiten, doch nun erst einmal zurück zur Package-Nutzung: Bisweilen lohnt es sich nicht, das gesamte Package zu laden, da Sie nur einmal eine einzige Funktion daraus benötigen. In diesem Fall lassen sich die Funktionen installierter Packages auch ohne `library()` nutzen, indem die Funktion über das Format `package_name::funktionsname()` aufgerufen wird:

```
tibble::as_tibble(iris)
```

```
## # A tibble: 150 x 5
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##           <dbl>       <dbl>        <dbl>       <dbl> <fct>
##  1          5.1         3.5          1.4         0.2 setosa
##  2          4.9         3            1.4         0.2 setosa
##  3          4.7         3.2          1.3         0.2 setosa
##  4          4.6         3.1          1.5         0.2 setosa
##  5          5           3.6          1.4         0.2 setosa
##  6          5.4         3.9          1.7         0.4 setosa
##  7          4.6         3.4          1.4         0.3 setosa
##  8          5           3.4          1.5         0.2 setosa
##  9          4.4         2.9          1.4         0.2 setosa
## 10          4.9         3.1          1.5         0.1 setosa
## # ... with 140 more rows
```

Dies ist vor allem dann sinnvoll, wenn Sie in einem Skript viele verschiedene Packages nutzen: Da es durchaus vorkommen kann, dass mehrere Packages denselben Funktionsnamen nutzen, kann das Laden vieler unterschiedlicher Packages dazu führen, dass die eigentlich gewünschte Funktion aus Package 1 durch eine gleichnamige Funktion aus Package 2 “überschrieben” wird – eine häufige Fehlerquelle.

Chapter 6

Workflow

Zum Abschluss der Einführung in die Grundlagen von R schauen wir uns noch einige Möglichkeiten an, effizient(er) mit R und RStudio arbeiten zu können.

6.1 Arbeitsverzeichnisse

Abgesehen von den Skripten für die Übungsaufgaben haben wir bisher noch nicht mit Dateien gearbeitet. In Zukunft sieht dies natürlich anders aus: wir laden Datensätze auf unserer Festplatte, erstellen Skripte und speichern unsere Ergebnisse, z. B. in Form von Tabellen oder Abbildungen.

Sobald wir daher mit Dateien arbeiten, wird das Konzept des Arbeitsverzeichnisses relevant, da R basierend auf diesem nach Dateien sucht. Unser aktuelles Arbeitsverzeichnis können wir mit dem Befehl `getwd()` (für *w*orking *d*irectory) anzeigen.

Nehmen wir einmal an, unser Arbeitsverzeichnis lautet `C:/Projekte/R`. R wird nun Pfadangaben zu Dateien *relativ* zu diesem Verzeichnis betrachten. Eine Datensatz-Datei `beispiel.csv`, die in diesem Verzeichnis liegt (deren gesamter Dateipfad also `C:/Projekte/R/beispiel.csv` ist), könnten wir mit der Funktion `read.csv()`, die zum Einlesen von CSV-Dateien verwendet wird, daher über `read.csv("beispiel.csv")` laden. Läge die Datei in einem Unterordner `data` (wäre der vollständige Dateipfad also `C:/Projekte/R/data/beispiel.csv`), müssten wir entsprechend `read.csv("data/beispiel.csv")` angeben. Gleiches gilt auch für Dateien, die wir mit R erstellen – geben wir keinen weiteren Pfad an, so landen diese im Hauptordner des aktuellen Arbeitsverzeichnisses.

Umgekehrt bedeutet dies auch, dass es umständlich wird, Dateien zu laden, die außerhalb unseres Arbeitsverzeichnisses (also im Beispiel in einem anderen Ordner auf `C:`) liegen. Zwar kann immer auch der gesamte Dateipfad angegeben

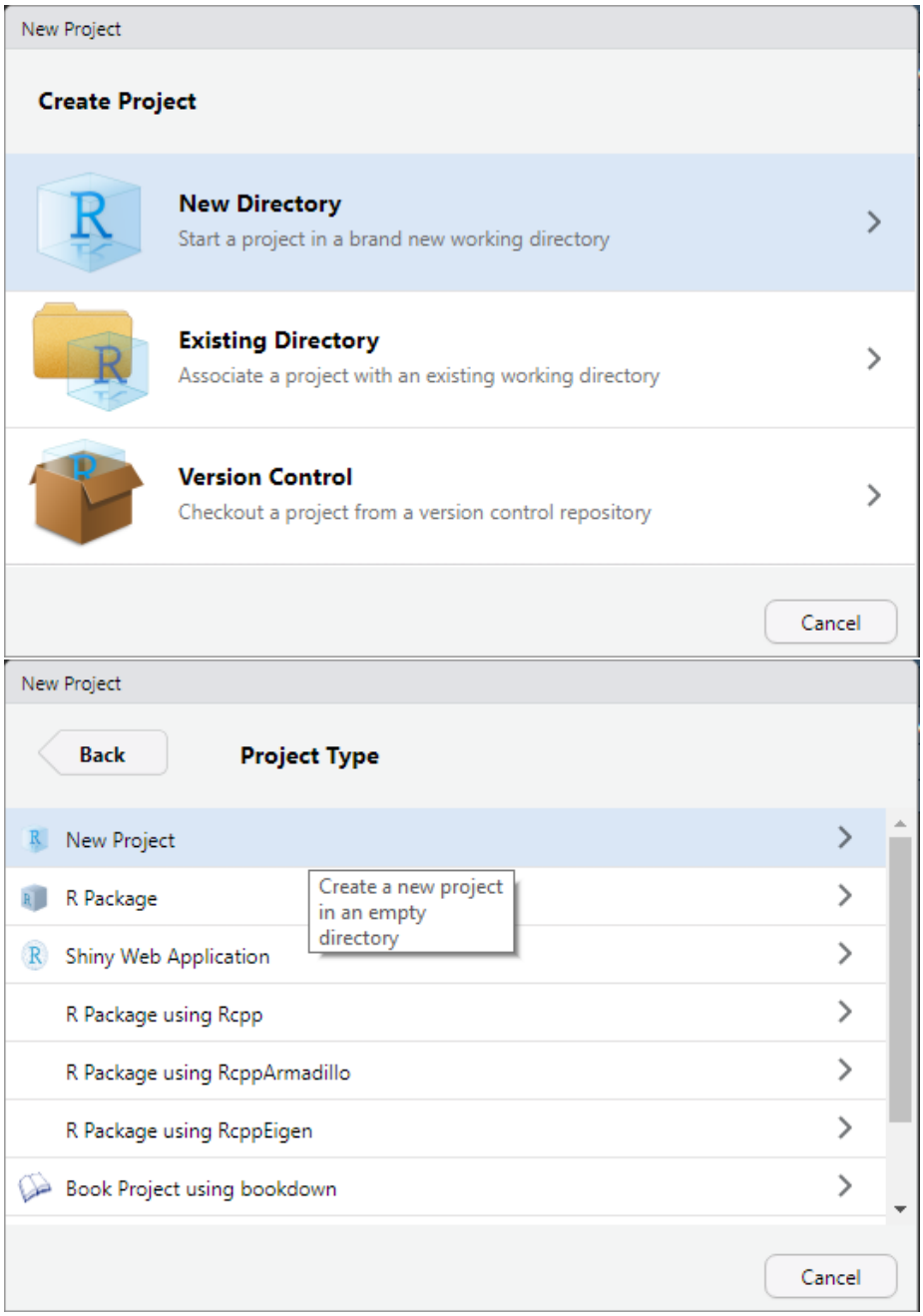
werden (z. B. `read.csv("C:/Projekte/R/data/beispiel.csv")`), das hat aber die Nachteile, dass Sie immer sehr viel tippen müssen und vor allem, dass Ihre Skripte nur noch auf Ihrem eigenen Rechner laufen, da Sie auf anderen Rechnern mit hoher Wahrscheinlichkeit auch eine andere Ordnerstruktur haben.

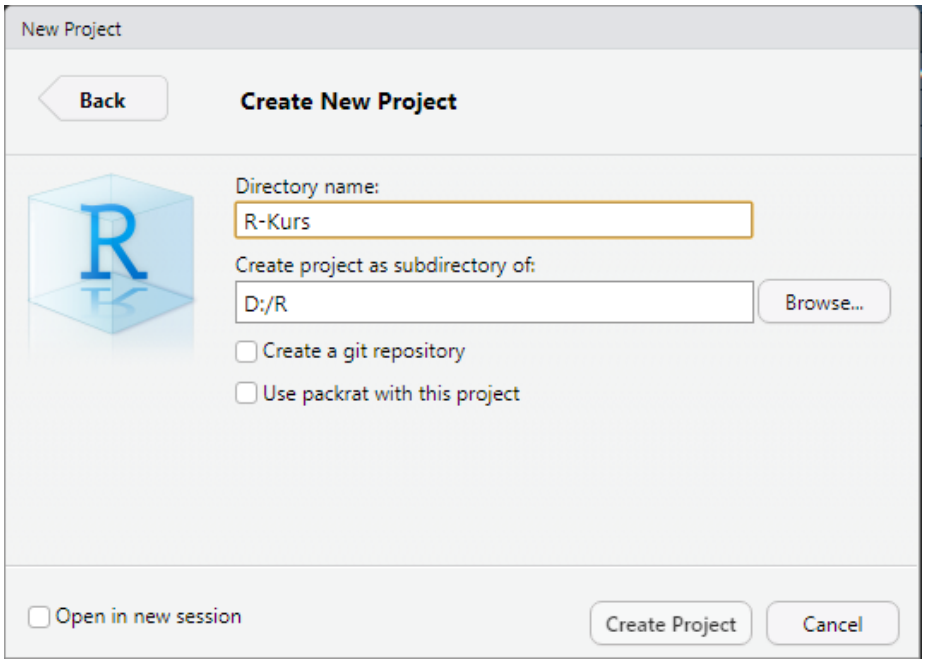
Es ist daher wichtig, sich a) eine sinnvolle Ordnerstruktur zu überlegen und b) sicherzustellen, dass R auch das korrekte Arbeitsverzeichnis nutzt. Das Arbeitsverzeichnis kann mit der Funktion `setwd()` festgelegt bzw. geändert werden – allerdings bietet R-Studio noch eine Funktionalität, die das nochmals deutlich erleichert:

6.2 Projekte und Ordnerstrukturen

RStudio bietet mittels Projekten eine einfache Möglichkeit, alle relevanten Projektdateien zusammenzuhalten und das Arbeitsverzeichnis automatisch zu setzen. Projektdateien (Dateieindung `.rproj`) sind prinzipiell nichts weiter als "Markierungen" für RStudio, dass es sich bei einen Ordner auf der Festplatte (und den darin enthaltenen Dateien) um ein Analyseprojekt mit R handelt.

Erstellen wir also unser erstes R-Projekt für diesen Kurs: Klicken Sie in RStudio auf *File - New Project* und wählen Sie im sich nun öffnenden Fenster *New Directory* und anschließend *New Project* aus. Im folgenden Fenster können Sie Ihrem Projekt einen sinnvollen Namen (z. B. "R-Kurs") geben und das Verzeichnis auswählen, in dem Ihr Projekt erstellt werden soll (RStudio erzeugt darin einen Unterordner mit dem Namen des Projekts):


New Project
Create Project
New Directory
Start a project in a brand new working directory
Existing Directory
Associate a project with an existing working directory
Version Control
Checkout a project from a version control repository
Cancel
New Project
Back
Project Type
New Project
R Package
Shiny Web Application
R Package using Rcpp
R Package using RcppArmadillo
R Package using RcppEigen
Book Project using bookdown
Create a new project in an empty directory
Cancel



Schließen Sie nun RStudio und navigieren zu dem Projektordner auf Ihrer Festplatte. Durch einen Doppelklick auf die `.rproj`-Datei öffnet sich RStudio mit diesem Projekt; das Arbeitsverzeichnis wird automatisch auf den Hauptordner des Projektes gesetzt, der *Files*-Browser im rechten unteren Bildschirmbereich zeigt ebenfalls dieses an.

Am rechten oberen Bildschirmrand wird Ihnen außerdem das aktuelle Projekt angezeigt – bearbeiten Sie mehrere Projekte gleichzeitig, können Sie hier auch direkt von einem Projekt in ein anderes wechseln (oder sogar mehrere gleichzeitig in unterschiedlichen RStudio-Sessions öffnen).

Schließlich ist nun auch ein guter Zeitpunkt, sich eine Ordnerstruktur zu überlegen. Ich persönlich lege (nummerierte) Skriptdateien meist direkt im Hauptordner des Projektes ab, speichere alle relevanten Datensätze in einem Unterordner `data`, und alle Ergbnisse in entsprechend benannten Unterordnern (z. B. `tables` und `figures`). Finden Sie hier vor allem eine Struktur, in der Sie sich – und im besten Falle auch andere, wenn Sie Daten aus einem Projekt anderen zur Verfügung stellen – intuitiv zurechtfinden.

6.3 R Markdown

R Markdown ist ein Dateiformat (Endung: `.Rmd`) von RStudio, das es erlaubt, Code, Ergebnisse und freien Text in nur einem Dokument zu kombinieren. Es ist dementsprechend gut geeignet, um Analyseschritte und die dahinterstehenden

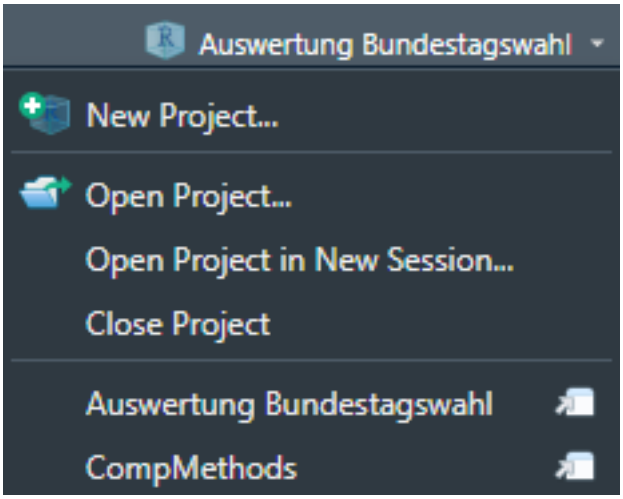


Figure 6.1: Auswahl von Projekten in RStudio

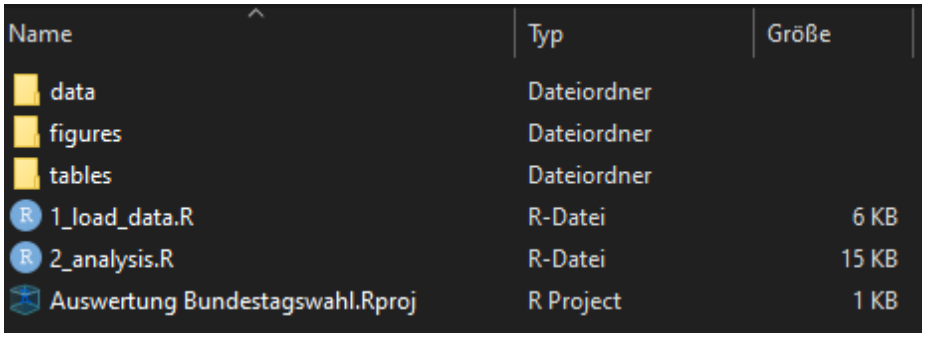


Figure 6.2: Ein – besonders für meine Verhältnisse – gut aufgeräumter Projektordner

Entscheidungen und Gedanken festzuhalten. Zudem können solche Dokumente schnell in andere Dokumenttypen – z. B. HTML-Websiten, PDF-Dateien oder Word-Dokumente – umgewandelt werden.

Wir erstellen R-Markdown-Dokumente über *File - New File - R Markdown*. Im folgenden Fenster können wir einen Dokumenttitel, Ausgabeformat usw. einstellen – all dies lässt sich aber auch später noch bearbeiten, sodass Sie fürs erste die Standardeinstellungen übernehmen können. Die erzeugte Datei ist bereits mit etwas Beispieltext gefüllt und besteht aus drei Komponenten:

6.3.1 YAML-Header

Ganz oben steht ein (optionaler) Block in der Auszeichnungssprache *YAML* (für *Y*et *A*nother *M*arkdown *L*anguage), abgetrennt durch `---`, der die Outputoptionen festlegt.

```
---
title: "Auswertung"
author: "Julian Unkel"
date: "4/29/2020"
output: html_document
---
```

Je nach Ausgabeformat können hier noch weitere Optionen angegeben werden, beispielsweise ob automatisch ein Inhaltsverzeichnis erzeugt werden soll. Auch hier reichen die Standardeinstellungen fürs erste jedoch vollkommen aus.

6.3.2 Freitext in Markdown

Freier Text kann jeder Stelle in das Dokument eingefügt werden. Dabei nutzt das Format die Formatierungssyntax *Markdown* (daher auch der Dateiformatname), mit der sich Text schnell mittels bestimmter Symbole formatieren lässt. Die Grundidee dahinter ist, dass man sich beim Schreiben auf das Schreiben konzentriert und alle wesentlichen Formatierungen im Nachhinein automatisiert auf Basis von Vorlagen erfolgen. Einige Formatierungsoptionen sind:

Table 6.1: Einige Formatierungsoptionen in R Markdown

Syntax	Erzeugt
Klartext	Klartext
`_kursiv_*kursiv*`	*kursivkursiv*
`__fett__**fett**`	**fettfett**

Syntax	Erzeugt
`# Überschrift## Überschrift### Überschrift`	Überschrift erster EbeneÜberschrift zweiter EbeneÜberschrift dritter Ebene
`- Listenpunkt 1- Listenpunkt 2`	- Listenpunkt 1- Listenpunkt 2
`1. nummerierte Liste2. Listenpunkt 2`	1. nummerierte Liste2. Listenpunkt 2

Ein gelungener Spickzettel für R Markdown findet sich zudem hier.

6.3.3 Code Chunks

R-Code kann in speziellen Blöcken hinzugefügt werden, die durch ```` ```{r} ```` (und einem optionalen Namen hinter `r`) eingeleitet werden und abschließend ```` ``` ```` begrenzt werden. Alles innerhalb dieser Blöcke ("Chunks") wird als R-Code interpretiert. Einfacher können R-Code-Chunks mit der Tastenkombination `Strg/Cmd + Alt + I` eingefügt werden.

Einzelne Codezeilen können wie gehabt mit `Strg/Cmd + Enter/Return` ausgeführt werden; mittels `Strg/Cmd + Shift + Enter/Return` oder durch Klick auf den grünen "Play"-Knopf am rechten oberen Ende des Chunks wird der gesamte Code des Chunks auf einmal ausgeführt. Die Besonderheit dabei ist, dass das Ergebnis direkt unter dem Chunk – und nicht in der Konsole – angezeigt wird, sodass Sie Code damit auch sehr schön gliedern können. Die Codebeispiele in der erzeugten R-Markdown-Datei verdeutlichen dies – einzelne Werte, Tabellen und auch Grafiken können so direkt in dem Dokument angezeigt werden.

6.3.4 Ausgabedateien erzeugen

Mit einem Klick auf den "Knit"-Knopf (mit Stricksymbol) am oberen Rand wird die im Header definierte Ausgabedatei erzeugt – standardmäßig also eine HTML-Datei, die im Arbeitsverzeichnis abgelegt wird und mit jedem Browser geöffnet werden kann. Auch dies können Sie einmal mit dem Beispielcode ausprobieren.

Diese Einführung kann nicht alle Ausgabeformate in Detail ansprechen – hier gibt es aber inzwischen vielfältige Möglichkeiten, die quasi den gesamten wissenschaftlichen Prozess abdecken, aber natürlich auch etwas Einarbeitung erfordern: mittels spezieller Packages lassen sich so unter anderem Präsentationen, PDF-Berichte (inklusive Literaturverwaltung und automatischer Formatierung von Ergebnistabellen nach gängigen wissenschaftlichen Standards) oder ganze Webseiten erstellen (auch dieser Kurs bzw. diese Website ist komplett in R Markdown erstellt).

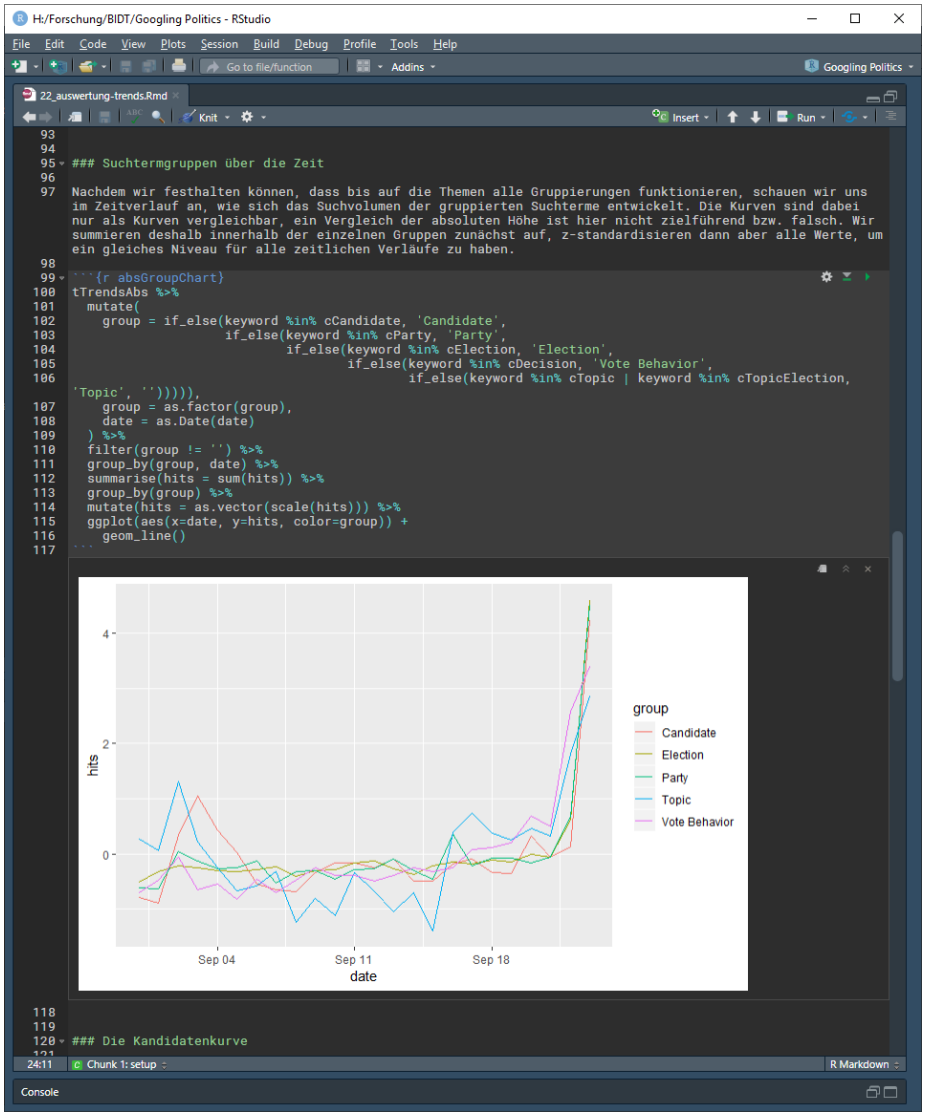


Figure 6.3: Ein R Markdown-Dokument mit Text, Code und Ergebnissen – in diesem Fall – Grafiken

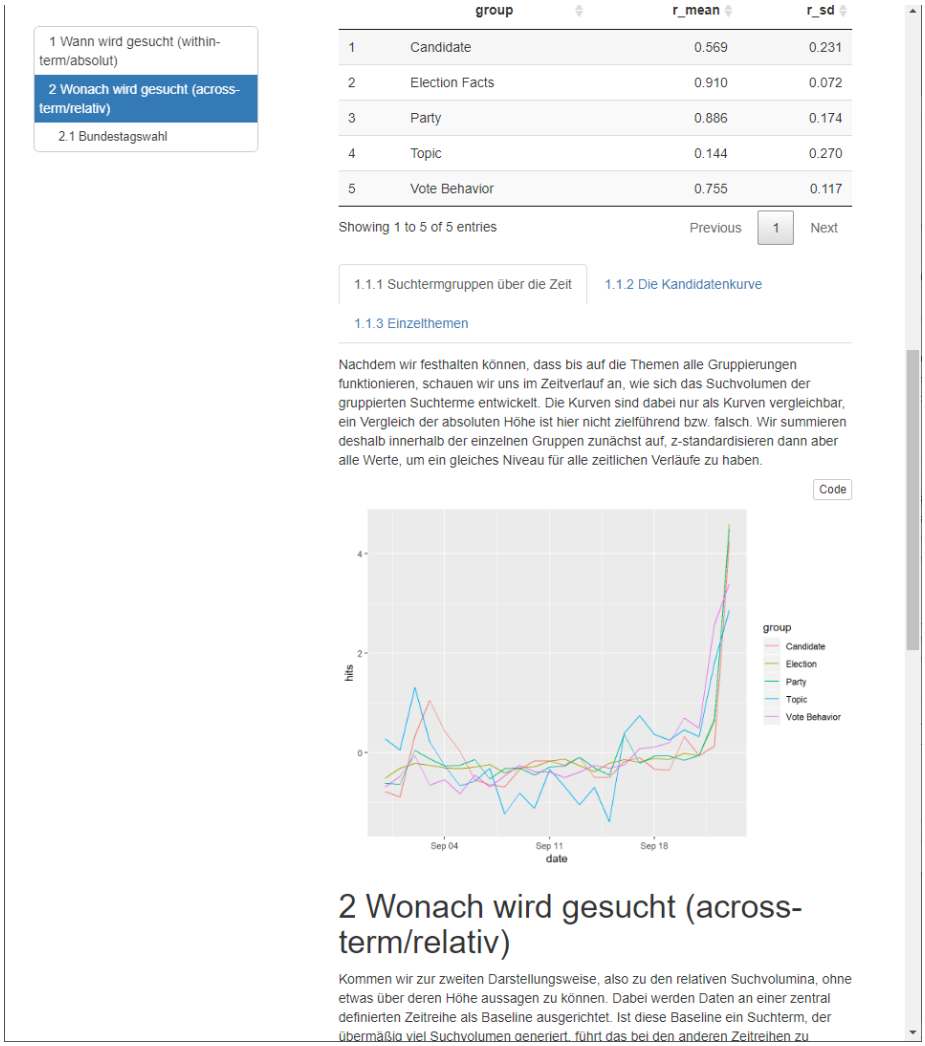


Figure 6.4: . . . und dasselbe Dokument als HTML-Seite konvertiert

Übungsaufgaben können Sie zukünftig sowohl als R-Skript (`.R`) als auch als R Markdown (`.Rmd`) abgeben.

Figure 6.5: Illustration von @allison_horst: https://twitter.com/allison_horst

6.4 R aktuell halten

Wie bei jeder anderen Software auch sollte die Arbeitsumgebung in R durch regelmäßige Updates aktuell gehalten werden, um neue Features, Fehlerkorrekturen, Performance-Verbesserungen und Beseitigungen von Sicherheitslücken zu erhalten. Da wir mit unterschiedlichen Komponenten – R, RStudio und jeder Menge Packages – arbeiten, müssen all diese Komponenten auch getrennt geupdated werden.

6.4.1 Packages aktualisieren

Am häufigsten sollten Sie Ihre installierten Packages aktualisieren. Dies geht in RStudio sehr komfortabel über *Tools - Check for Package Updates*, wodurch alle Packages angezeigt werden, für die Updates bereitstehen. Nach einer Auswahl der gewünschten Packages können Updates gesammelt über den Button *Install Updates* heruntergeladen und installiert werden.

Um nur ein einzelnes Packages zu aktualisieren, kann dieses auch einfach über die Funktion `install.packages("package_name")` mit der aktuellsten Version

neu installiert werden.

6.4.2 RStudio aktualisieren

Auch RStudio kann direkt aus dem Programm heraus aktualisiert werden. Hier empfiehlt es sich, regelmäßig unter *Help - Check for Updates* zu überprüfen, ob eine Aktualisierung bereitsteht.

6.4.3 R aktualisieren

Etwas unkomfortabler ist es, R selbst zu aktualisieren. Dass eine neue Version bereitsteht, erfährt man häufig dadurch, dass Packages beim Laden darauf aufmerksam machen, dass Sie mit einer aktuelleren Version von R als der aktuell installierten erstellt wurden. Das liest sich in der Konsole dann so:

```
Warning message:
package 'package_name' was built under R version x.x.x
```

Das Updaten funktioniert dabei ebenso wie eine Neuinstallation, wie sie in Kapitel 1.1 beschrieben wurde – RStudio sollte die neue R-Version dann automatisch erkennen (die aktuell installierte R-Version wird stets beim Starten von R/RStudio in der Konsole angezeigt). Dabei ist zu beachten, dass installierte Packages nicht von einer R-Version zur nächsten kopiert werden; es ist also entweder manuelles Kopieren oder eine Neuinstallation der Packages nötig.

Unter Windows kann zudem das Paket `installr` genutzt werden, dessen Funktion `updateR()` prüft, ob eine neue R-Version bereitsteht, diese herunterlädt und installiert sowie die Möglichkeit bietet, automatisch Packages der alten Installation für die neue Installation zu kopieren (was nicht immer auch funktioniert). Allerdings wird empfohlen, diese Funktion nicht in RStudio, sondern direkt in R zu nutzen.

Da vorige Woche eine neue Hauptversion von R erschienen ist (`R 4.0.0`) und wir aktuell noch kaum Packages installiert haben, die wir durch ein Update verlieren könnten, ist nun ein guter Zeitpunkt, um diese Version zu installieren. Falls Sie einen Mac nutzen, führen Sie also eine Neuinstallation von R wie unter Kapitel 1.1 beschrieben durch.

Auch unter Windows können Sie diesen Weg wählen – oder Sie probieren das `installr`-Package aus. Hierzu installieren Sie das Package zunächst über `install.packages("installr")`. Schließen Sie dann RStudio und öffnen R (Sie finden R in der Regel im Startmenü unter R; das gibt Ihnen auch die Gelegenheit, wertzuschätzen, wie viel komfortabler RStudio ist). In die Konsole können Sie nun `installr::updateR()` ein und folgen den Anweisungen des Installationsprogramms.

In beiden Fällen sollte beim nächsten Start von RStudio in der ersten Konsolenzeile die neue R-Version angezeigt werden:

```
R version 4.0.0 (2020-04-24) -- "Arbor Day"
```

Part II

Datenhandling in R

Chapter 7

Einführung in das Datenhandling

Das Paretoprinzip – 80% der Ergebnisse werden mit 20% des Aufwands, 20% der Ergebnisse mit 80% des Aufwands erreicht – behält auch im datenanalytischen Kontext seine Gültigkeit: ein Großteil der Arbeitszeit wird darauf verwendet, Daten zu finden, zu importieren, zu säubern, zu transformieren, zu modifizieren und zu explorieren; die tatsächliche Analyse der Daten hingegen ist dann verhältnismäßig schnell erledigt. Wir fassen diese Schritte unter dem Begriff des *Datenhandlings* zusammen.

7.1 Daten- und Dateiformate tabellarischer Daten

Wenn wir von Datensätzen sprechen, denken wir zumeist direkt an tabellarische Daten. In den kommenden Kapiteln werden wir auch zunächst lediglich mit tabellarischen Daten arbeiten. Daten können und werden aber auch in anderen Datenformaten repräsentiert, zum Beispiel als verschachtelte und hierarchisch strukturierte Daten (dieses Datenformat wird uns u.a. bei der Arbeit mit APIs begegnen) oder als unstrukturierte Textdokumente – dazu an gegegeber Stelle mehr.

7.1.1 Tidy data

Dieselben Daten können in Tabellen unterschiedlich repräsentiert werden. Schauen wir uns dazu einen Datensatz an – was könnten hier Probleme sein?

```
## # A tibble: 8 x 3
##   name             variable value
##   <chr>            <chr>    <dbl>
## 1 Anakin Skywalker height     188
## 2 Anakin Skywalker mass        84
## 3 Leia Organa      height     150
## 4 Leia Organa      mass        49
## 5 Luke Skywalker   height     172
## 6 Luke Skywalker   mass        77
## 7 Obi-Wan Kenobi   height     182
## 8 Obi-Wan Kenobi   mass        77
```

Diese Datenanordnung ist in dreierlei Hinsicht nicht optimal, wobei alle Probleme miteinander verbunden sind:

1. Zwei Variablen sind in einer Spalte hinterlegt: `value` enthält sowohl Werte, die sich auf die Körpergröße als auch auf das Gewicht beziehen.
2. Entsprechend ist die Spalte `value` abhängig von der Spalte `variable` – allein anhand der Werte `188`, `84`, `150` etc. wissen wir nicht, ob diese sich auf die Größe `height` oder das Gewicht `mass` beziehen.
3. Daraus folgt, dass wir Probleme mit vektorisierten Funktionen – wir erinnern uns, in R sind Spalten in Datensätzen Vektoren – bekommen: wir können beispielsweise nicht einfach die `mean()`-Funktion auf die Spalte `value` anwenden, um das Durchschnittsgewicht der Star-Wars-Charaktere zu berechnen, da dort auch die Werte für die Körpergröße enthalten wären.

Schauen wir uns eine zweite Version derselben Daten an:

```
## # A tibble: 4 x 2
##   name             height_and_mass
##   <chr>            <chr>
## 1 Anakin Skywalker 188;84
## 2 Leia Organa      150;49
## 3 Luke Skywalker   172;77
## 4 Obi-Wan Kenobi   182;77
```

Auch hier ergeben sich zwei miteinander verbundene Probleme:

1. In einer Zelle stehen zwei Werte, die je eine unterschiedliche Variable abbilden.
2. Sowohl bei Körpergröße als auch Gewicht handelt es sich um numerische Werte, sie werden aber hier als `character` gespeichert, wodurch wir keine Berechnungen durchführen können.

Dieselben Daten können wir jedoch auch besser tabellarisch abbilden:

```
## # A tibble: 4 x 3
##   name             height  mass
##   <chr>             <int> <dbl>
## 1 Anakin Skywalker    188    84
## 2 Leia Organa         150    49
## 3 Luke Skywalker      172    77
## 4 Obi-Wan Kenobi      182    77
```

Daten, die so aufbereitet, werden als *tidy data* bezeichnet. Sie befolgen drei Regeln:

- Jede Variable steht in einer eigenen Spalte
- Jede Beobachtung/jeder Fall steht in einer eigenen Zeile
- Jeder Wert steht in einer Zelle

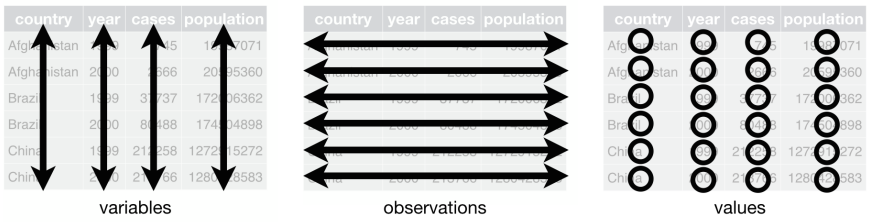


Figure 7.1: Tidy data (Quelle: R for Data Science)

Im Umkehrschluss bedeutet dies, dass Daten die nicht diesen Regeln folgen, als *messy data* vorliegen und entsprechend aufbereitet werden sollten. Tidy data hat zwei entscheidende Vorteile:

1. Konsistent aufbereitete Daten sind leichter zu lesen, zu verarbeiten, zu laden und zu speichern.
2. Viele Verfahren (bzw. die zugehörigen Funktionen) in R setzen diese Art der Daten voraus.

Wenn Sie bisher vor allem mit Daten aus Befragungssoftware o.ä. gearbeitet haben, dann werden Ihnen diese Grundprinzipien bereits vertraut sein, da der Datenexport aus Befragungssoftware in der Regel diesen Prinzipien folgt. Tatsächlich liegen in der "Realität" Daten aber häufig nicht in dieser Form vor, beispielsweise weil die Eingabe oder Speicherung der Daten anderen Prinzipien als einer möglichst einfachen Analyse folgt (z. B. möglichst einfache Dateneingabe; Abruf von bestimmten Werten).

7.1.2 Dateiformate

Sie kennen vermutlich bereits eine Vielzahl an Dateiformaten, in denen tabellarische Daten gespeichert werden können. Einige Beispiel sind Excel-Dateien mit den Endungen `.xls` bzw. `.xlsx` oder SPSS-Dateien mit der Endung `.sav`. Hierbei handelt es sich um sogenannte *proprietäre* Dateiformate, sich im Eigentum von Unternehmen befindliche und oft softwarespezifische Dateiformaten. Ein Vorteil dieser Formate ist es, dass oft relevante Zusatzinformationen für die jeweilige Software mitgespeichert werden, etwa Formatierungen in Excel oder Wertebeschriftungen in SPSS. Zugleich entsteht hierbei der große Nachteil, dass der Austausch der Daten zwischen verschiedenen Programmen oft recht kompliziert, nur durch Zusatzpakete oder Zwischenschritte möglich ist.

Ein *offenes* und sehr simples Dateiformat ist `.csv` (für *c*omma-*s*eparated *v*alues), das tabellarische Daten als Klartext abspeichert, von so gut wie jedem Programm auf jedem Computer der Welt gelesen werden kann und daher das Standardformat für tabellarische Daten ist. Dabei werden Kommas zu Trennung von Spalten und Zeilenumbrüche zur Trennung von Zeilen genutzt, wobei in der ersten Zeile optional Spaltenüberschriften stehen können. Die Daten aus obigen Beispiel sähen als `.csv` also so aus:

```
name,height,mass
Anakin Skywalker,188,84
Leia Organa,150,49
Luke Skywalker,172,77
Obi-Wan Kenobi,182,77
```

Soll das Trennzeichen in den einzelnen Text-Werten vorkommen (beispielsweise als Komma in einem Satz), werden Textbegrenzungszeichen, häufig doppelte Anführungszeichen `"` verwendet:

```
name,height,mass
"Skywalker, Anakin",188,84
"Organa, Leia",150,49
"Skywalker, Luke",172,77
"Kenobi, Obi-Wan",182,77
```

In verallgemeinerter Form kann letztlich jedes Zeichen als Trennzeichen genutzt werden; im deutschsprachigen Raum wird in `.csv`-Dateien häufig ein Semikolon `;` statt einem Komma genutzt `,` (z. B. wenn Sie Daten aus SPSS oder Excel als `.csv` exportieren):

```
name;height;mass
Anakin Skywalker;188;84
Leia Organa;150;49
```

```
Luke Skywalker;172;77
Obi-Wan Kenobi;182;77
```

Zwar kann R mittels unterschiedlicher Packages inzwischen auch relativ komfortabel Excel- und SPSS-Dateien einlesen; aus oben genannten Gründen werden wir zum Laden und Speichern jedoch vorrangig `.csv`-Dateien nutzen.

7.2 Willkommen im Tidyverse

Die Idee, Datenhandling komplett codebasiert durchzuführen, erscheint zunächst vielleicht wenig komfortabel – vor allem wenn wir daran denken, dass das Auswählen und Verarbeiten von Dataframes bisher wenig intuitiv ablief (oder wissen Sie auf Anhieb noch, was der Unterschied zwischen `iris[, "Sepal.Width"]`, `iris$Sepal.Width` und `iris[["Sepal.Width"]]` ist?).[1]

Seit einigen Jahren hat sich jedoch das Tidyverse als Standard des Datenhandlings in R durchgesetzt, was die mit dem Datenhandling verbundenen Schritte deutlich vereinfacht. Dabei handelt es sich um ein sogenanntes Meta-Package, also eine Sammlung von verschiedenen Packages, die allesamt den gleichen Designprinzipien folgen, um Datenhandling zu vereinfachen und den zugehörigen Code möglichst lesbar (für Menschen) zu machen. So haben beispielsweise Funktionen im Tidyverse in der Regel Verben als Namen, die genau das beschreiben, was die Funktion macht.

> The tidyverse is an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures.

Das Tidyverse umfasst hierzu unter anderem Pakete für:

- Datenstrukturen (`tibble`)
- Einlesen von Daten (z. B. `readr` für CSV, `haven` für SPSS, Stata und SAS, `readxl` für Excel)
- Datentransformation und -modifikation (`tidyr`, `dplyr`)
- Spezielle Objekttypen (z. B. `stringr` für die Arbeit mit Textobjekten, `forcats` für Faktoren, `lubridate` für Zeitdaten)
- Programmieren mit R (`purrr`)
- Grafik-/Diagrammerstellung (`ggplot2`)

Alle Pakete lassen sich gesammelt installieren und laden[2]:

[1] Die erste Variante resultiert in einem Dataframe, der lediglich `Sepal.Width` enthält, Varianten zwei und drei extrahieren diese Spalte als Vektor.

[2] `library(tidyverse)` lädt dabei genauer gesagt die Kernpakete des Tidyverse, die die alltäglich am häufigsten genutzten Funktionen bereitstellen. Tidyverse-Packages für eher spezifische Anwendungsgebiete – z. B. `lubridate` für Zeitdaten oder `haven` für SPSS-Datensätze – müssen einzeln über den `library()`-Befehl geladen werden.

```
install.packages("tidyverse") # Dies muss natürlich nur einmal ausgeführt werden
library(tidyverse)
```

Die Standard-Datenstruktur des Tidyverse ist das Tibble, das wir bereits aus Kapitel 5.2 kennen. Zur Erinnerung, dabei handelt es sich im Wesentlichen um Dataframes mit einigen technischen und kosmetischen Verbesserungen:

```
as_tibble(iris)
```

```
## # A tibble: 150 x 5
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##           <dbl>       <dbl>        <dbl>       <dbl> <fct>
##  1          5.1         3.5          1.4         0.2 setosa
##  2          4.9         3            1.4         0.2 setosa
##  3          4.7         3.2          1.3         0.2 setosa
##  4          4.6         3.1          1.5         0.2 setosa
##  5          5           3.6          1.4         0.2 setosa
##  6          5.4         3.9          1.7         0.4 setosa
##  7          4.6         3.4          1.4         0.3 setosa
##  8          5           3.4          1.5         0.2 setosa
##  9          4.4         2.9          1.4         0.2 setosa
## 10          4.9         3.1          1.5         0.1 setosa
## # ... with 140 more rows
```

Im folgenden Kapitel werden wir uns nun ansehen, wie wir mittels Tidyverse-Funktionen Datensätze laden und modifizieren können.

Chapter 8

Daten laden, modifizieren und speichern

In diesem Kapitel sehen wir uns grundlegende Arbeitsschritte und Funktionen des Datenhandlings: das Einlesen von Daten, einfaches Modifizieren von Datensätzen und das Abspeichern der Ergebnisse. Für all diese Schritte arbeiten wir mit Funktionen aus dem Tidyverse – falls noch nicht geschehen, sollten Sie das Package jetzt also installieren.

```
install.packages("tidyverse")
```

Und dann laden wir das Package zu Beginn unseres Auswertungsskripts:

```
library(tidyverse)
```

8.1 Daten laden

Wir sprechen von *lokalen* Daten, wenn wir diese in Form einer Datei auf unserer Festplatte gespeichert haben. *Externe* Daten liegen beispielsweise auf Webservern oder sind in Packages enthalten. Zunächst laden wir nur lokale Daten.

8.1.1 CSV-Dateien einlesen

Wenn Sie den Schritten in Kapitel 6.2 gefolgt sind, haben Sie ein R-Projektverzeichnis auf des Festplatte. Auf Moodle finden Sie den Datensatz `facebook_europawahl.csv`, der Informatationen zu Facebook-Posts der

deutschen Parteien im Vorfeld der Europawahl 2019 enthält.[1] Speichern Sie diesen Datensatz in Ihrem Projektverzeichnis ab – im Beispiel liegt der Datensatz im Unterordner `data`.

Funktionen zum Einlesen von Daten folgen im Tidyverse dem Namensschema `read_`, wobei nach dem Unterstrich der Dateityp folgt. Für CSV-Dateien sind zwei Funktionen relevant:

- `read_csv()` liest CSV-Dateien, die ein Komma `,` als Spalten- und einen Punkt `.` als Dezimaltrennzeichen verwenden
- `read_csv2()` liest CSV-Dateien, die ein Semikolon `;` als Spalten- und das Komma `,` als Dezimaltrennzeichen verwenden

Bei beiden Funktionen handelt es sich um spezifische Varianten der Funktion `read_delim()`, bei der Trennzeichen etc. einzeln definiert werden können. In der Regel sollten aber die beiden oben genannten Funktionen ausreichen. Im Zweifelsfall können CSV-Dateien durch anklicken im *Files*-Bereich in RStudio geöffnet werden, sodass ersichtlich wird, wie diese aufgebaut sind und welches Trennzeichen verwendet wird.

Alle Funktionen aus der `read_`-Familie benötigen als erstes (und oft auch einziges) Argument den Dateipfad (relativ zum Arbeitsverzeichnis) als Textobjekt. Da unser Datensatz im Unterordner `data` liegt, lautet der gesamte Dateipfad also `"data/facebook_europawahl.csv"`. Natürlich sollten wir das Resultat der Funktion einem treffend benannten Objekt zuweisen.[2]

```
df_fb_eu <- read_csv("data/facebook_europawahl.csv")
```

```
##
## -- Column specification --------------------------------------------------------
## cols(
##   id = col_double(),
##   URL = col_character(),
##   party = col_character(),
##   timestamp = col_datetime(format = ""),
##   type = col_character(),
##   message = col_character(),
##   link = col_character(),
##   comments_count = col_double(),
##   shares_count = col_double(),
##   reactions_count = col_double(),
##   like_count = col_double(),
```

[1] Vielen Dank an den Kollegen Jörg Haßler!

[2] Häufig verwendete Objektnamen für Datensätze sind `df` und `data`, aber es schadet auch nicht, etwas spezifischere Namen zu vergeben, besonders wenn mit mehreren Datensätzen gearbeitet wird.

```
##   love_count = col_double(),
##   wow_count = col_double(),
##   haha_count = col_double(),
##   sad_count = col_double(),
##   angry_count = col_double()
## )
```

Die Funktion teilt uns direkt mit, welche Objekttypen für welche Variable verwendet wurden, sodass wir hier auch direkt sehen können, ob z. B. eine numerische Variable als Text eingelesen wurde. Zudem ist der eingelesene Datensatz direkt ein *Tibble*, wir müssen also nicht mehr durch `as_tibble()` konvertieren.[3]

Schauen wir uns unseren gerade geladenen Datensatz einmal an – da unser Datensatz als Tibble vorliegt, erhalten wir die wichtigsten Informationen zu Struktur und einen Einblick in die Daten direkt in Konsole, wenn wir das Datensatz-Objekt aufrufen:

```
df_fb_eu
```

```
## # A tibble: 902 x 16
##       id URL   party timestamp           type  message link  comments_count shares_count react
##    <dbl> <chr> <chr> <dttm>              <chr> <chr>   <chr>          <dbl>        <dbl>
##  1     1 http~ oedp~ 2019-04-28 09:00:00 video "Guido~ http~              0            4
##  2     2 http~ tier~ 2019-04-28 13:57:00 photo "Aus u~ http~             17          130
##  3     3 http~ B90D~ 2019-04-28 06:00:01 video "Beim ~ http~             70           28
##  4     4 http~ FDP   2019-04-28 11:49:59 photo "Unser~ http~             16            9
##  5     5 http~ tier~ 2019-04-28 08:24:15 link  "Eine ~ http~              6           46
##  6     6 http~ CDU   2019-04-28 09:12:19 video "Freih~ http~            239          136
##  7     7 http~ SPD   2019-04-28 13:06:09 photo "Katar~ http~            180           54
##  8     8 http~ Pira~ 2019-04-28 17:36:30 video "Unser~ http~              0           NA
##  9     9 http~ DieP~ 2019-04-28 07:44:28 link  "Der a~ http~             35           76
## 10    10 http~ CSU   2019-04-28 08:21:00 photo "#Klar~ http~            174           61
## # ... with 892 more rows, and 3 more variables: haha_count <dbl>, sad_count <dbl>, angry_count
```

Wir haben also einen Datensatz mit 902 Zeilen bzw. Fällen – im diesen Falle also Facebook-Posts – und 16 Spalten bzw. Variablen. Darunter sind:

- eine numerische `id`, die `URL` und ein Zeitstempel (`timestamp`) des Posts
- Die Parteiseite `party` von der der Post abgesetzt wurde

[3] Auch die Basisversion von R bietet Funktionen zum Einlesen von CSV-Dateien, die `read.csv()`, `read.csv2()` etc. heißen. Diese erzeugen einen Dataframe und sind weniger gut für große Dateien optimiert, sodass ich empfehle, immer direkt die Tidyverse-Funktionen zu nutzen. Generell erkennen Sie Tidyverse-Varianten von Funktionen der R-Basisversion daran, dass diese einen Unterstrich anstatt eines Punkts zur Worttrennung nutzen.

- Der Typ (`type`) des Posts (Video, Photo, Link oder Status)
- Der Text (`message`) des Posts und ein etwaiger enthaltener `link`
- Die Anzahl verschiedener Facebook-Metriken, darunter Kommentare, Shares sowie Reactions gesamt und getrennt in einzelne Typen, allesamt auf `_count` endend

8.1.2 Andere Dateiformate

Andere Dateiformate funktionieren analog – in der Regel reicht es, die korrekte Funktion zu verwenden und den Dateipfad anzugeben. Allerdings müssen für proprietäre Dateiformate erst die – mit dem Tidyverse bereits installierten – Packages geladen werden:

- das Paket `readxl` bietet Funktionen zum Import von Excel-Dateien, z. B. `readxl::read_xlsx()`
- das Paket `haven` deckt den Import von Datensätzen aus anderer Statistiksoftware (SAS, Stata, SPSS) ab, z. B. `haven::read_sav()` für SPSS-Datensätze

8.2 Daten modifizieren

Zur Datenmodifikation betrachten wir sechs zentrale Funktionen (+ einige zugehörige Hilfsfunktionen bzw. Variationen davon), die das Tidyverse – genauer gesagt das Teilpaket `dplyr`[4] – zur Verfügung stellt:

- `select()` zum Auswählen von Variablen (spaltenweise)
- `filter()` zum Filtern von Variablen (zeilenweise)
- `arrange()` zum Sortieren des Datensatzes
- `mutate()` zum Erzeugen neuer Variablen
- `summarize()` zum Zusammenfassen von Variablen
- `group_by` zum Gruppieren von Variablen

Alle Funktionen haben dabei gemeinsam (und das trifft auf nahezu alle Funktionen des Tidyverse zu), dass das erste Argument der Datensatz selbst (als Tibble) ist und auch das Resultat der Funktion wiederum ein Tibble ist.

8.2.1 Variablen spaltenweise auswählen mit `select()`

Mit `select()` können wir bestimmte Spalten eines Datensatzes auswählen. Hierzu übergeben wir nach dem Datensatz einfach alle Variablen, die wir benöti-

[4] Wobei umstritten ist, ob man das Paket *dee_plier* oder *deeply_ar* ausspricht.

Figure 8.1: Illustration von @allison_horst: https://twitter.com/allison_horst

gen, direkt als Objektnamen – in unserem Beispiel `id`, `URL`, `party` usw. – ohne Anführungszeichen durch Kommas getrennt:[5]

```
# Wähle nur die Variablen id, party und timestamp aus
select(df_fb_eu, id, party, timestamp)
```

```
## # A tibble: 902 x 3
##       id party            timestamp
##    <dbl> <chr>            <dttm>
##  1     1 oedp.de          2019-04-28 09:00:00
##  2     2 tierschutzpartei 2019-04-28 13:57:00
##  3     3 B90DieGruenen    2019-04-28 06:00:01
##  4     4 FDP              2019-04-28 11:49:59
##  5     5 tierschutzpartei 2019-04-28 08:24:15
##  6     6 CDU              2019-04-28 09:12:19
##  7     7 SPD              2019-04-28 13:06:09
##  8     8 Piratenpartei    2019-04-28 17:36:30
##  9     9 DiePARTEI        2019-04-28 07:44:28
## 10    10 CSU              2019-04-28 08:21:00
## # ... with 892 more rows
```

Durch ein vorangstelltes `-` werden Variablen ausgeschlossen:

```
# Entferne die id und die URL-Variable
select(df_fb_eu, -id, -URL)
```

```
## # A tibble: 902 x 14
##    party timestamp           type  message link  comments_count shares_count reactio
##    <chr> <dttm>              <chr> <chr>   <chr>          <dbl>        <dbl>
##  1 oedp~ 2019-04-28 09:00:00 video "Guido~ http~             0            4
##  2 tier~ 2019-04-28 13:57:00 photo "Aus u~ http~            17          130
##  3 B90D~ 2019-04-28 06:00:01 video "Beim ~ http~            70           28
##  4 FDP   2019-04-28 11:49:59 photo "Unser~ http~            16            9
##  5 tier~ 2019-04-28 08:24:15 link  "Eine ~ http~             6           46
##  6 CDU   2019-04-28 09:12:19 video "Freih~ http~           239          136
##  7 SPD   2019-04-28 13:06:09 photo "Katar~ http~           180           54
##  8 Pira~ 2019-04-28 17:36:30 video "Unser~ http~             0           NA
##  9 DieP~ 2019-04-28 07:44:28 link  "Der a~ http~            35           76
## 10 CSU   2019-04-28 08:21:00 photo "#Klar~ http~           174           61
## # ... with 892 more rows, and 2 more variables: sad_count <dbl>, angry_count <dbl>
```

[5] Und natürlich müssen wir das Resultat der Funktionen immer einem Objekt zuweisen, wenn wir damit weiterarbeiten wollen – zu Demonstrationszwecken reicht aber der reine Aufruf der Funktion.

Durch ein vorangestelltes `neuer_objektname =` können wir Variablen beim Auswählen auch direkt umbenennen:

```
# Benenne party und message beim Auswählen um in partei respektive inhalt
select(df_fb_eu, partei = party, inhalt = message)
```

```
## # A tibble: 902 x 2
##    partei          inhalt
##    <chr>           <chr>
##  1 oedp.de         "Guido #Klamt aus #Ludwigsburg, Listenplatz 5 auf der Kandidatenliste der #
##  2 tierschutzpart~ "Aus unserem Europawahlprogramm, Kapitel 7: Gesundheits- und Sozialpolitik:
##  3 B90DieGruenen   "Beim Wahlkampf-Camp in Berlin waren gestern hunderte Freiwillige, die sich
##  4 FDP             "Unser neuer Bundesvorstand \U0001f389\U0001f38a\U0001f388 #ChancenNutzen \
##  5 tierschutzpart~ "Eine neue Studie der Universität Oxford zeigt, dass eine vegane Ernährung
##  6 CDU             "Freiheit ist nicht selbstverständlich. #UnserEuropa steht für freiheitlich
##  7 SPD             "Katarina Barley sagt: Eine weitere Koalition mit der #EVP will ich nicht -
##  8 Piratenpartei   "Unser Spitzenkandidat Patrick Breyer mit einem Update zum EU19 Workshop in
##  9 DiePARTEI       "Der absolut härteste „Martin-Sonneborn-Moment" kommt für Watson.de nach de
## 10 CSU             "#Klartext von Bayerns Ministerpräsident und CSU-Chef Markus Söder in der \
## # ... with 892 more rows
```

Um Variablen basierend auf Namensbestandteilen auszuwählen, sind einige Hilfsfunktionen - z. B. `starts_with()`, `ends_with()` und `contains()` - verfügbar. Da in unserem Beispiel alle Facebook-Metriken auf `_count` enden, können wir diese gesammelt mit `ends_with("count")` auswählen:

```
# Wähle party und alle Facebook-Metriken aus
select(df_fb_eu, party, ends_with("count"))
```

```
## # A tibble: 902 x 10
##    party            comments_count shares_count reactions_count like_count love_count wow_coun
##    <chr>                     <dbl>        <dbl>           <dbl>      <dbl>      <dbl>     <dbl
##  1 oedp.de                       0            4               9          9          0
##  2 tierschutzpartei             17          130             395        354         23
##  3 B90DieGruenen                70           28             215        174         14
##  4 FDP                          16            9             262        254          7
##  5 tierschutzpartei              6           46             145        129         14
##  6 CDU                         239          136             398        292          8
##  7 SPD                         180           54             699        576         34
##  8 Piratenpartei                 0           NA               7          6          0
##  9 DiePARTEI                    35           76             612        509         49
## 10 CSU                         174           61             458        334          3
## # ... with 892 more rows
```

Schließlich kann die Hilfsfunktion `everything()` (ohne Argumente) genutzt werden, um sämtliche nicht zuvor angegebenen Variablen auswählen – das ist hilfreich, wenn nur bestimmte Variablen z. B. umbenannt oder an den Anfang des Datensatzes gestellt werden sollen, aber man nicht alle anderen Variablen von Hand tippen möchte:

```
# Stelle party umbenannt in Partei an den Anfang und hänge alle verbleibenden Variable
select(df_fb_eu, Partei = party, everything())
```

```
## # A tibble: 902 x 16
##    Partei    id URL   timestamp           type  message link  comments_count shares_
##    <chr>  <dbl> <chr> <dttm>              <chr> <chr>   <chr>          <dbl>
##  1 oedp.~     1 http~ 2019-04-28 09:00:00 video "Guido~ http~              0
##  2 tiers~     2 http~ 2019-04-28 13:57:00 photo "Aus u~ http~             17
##  3 B90Di~     3 http~ 2019-04-28 06:00:01 video "Beim ~ http~             70
##  4 FDP        4 http~ 2019-04-28 11:49:59 photo "Unser~ http~             16
##  5 tiers~     5 http~ 2019-04-28 08:24:15 link  "Eine ~ http~              6
##  6 CDU        6 http~ 2019-04-28 09:12:19 video "Freih~ http~            239
##  7 SPD        7 http~ 2019-04-28 13:06:09 photo "Katar~ http~            180
##  8 Pirat~     8 http~ 2019-04-28 17:36:30 video "Unser~ http~              0
##  9 DiePA~     9 http~ 2019-04-28 07:44:28 link  "Der a~ http~             35
## 10 CSU       10 http~ 2019-04-28 08:21:00 photo "#Klar~ http~            174
## # ... with 892 more rows, and 3 more variables: haha_count <dbl>, sad_count <dbl>, a
```

8.2.2 Variablen zeilenweise filtern mit `filter()`

Um nur bestimmte Zeilen auswählen, können wir mittels `filter()` eine oder mehrere Bedingungen übergeben, die analog zu den `if`-Bedingungen in Kapitel 4.1 angegeben werden. Zuerst wird erneut der Datensatz übergeben:

```
# Wähle alle Facebookposts mit mindestens einem Kommentar
filter(df_fb_eu, comments_count > 0)
# Achten Sie in der Ausgabe auf die veränderte Zeilenanzahl
```

```
## # A tibble: 832 x 16
##       id URL   party timestamp           type  message link  comments_count shares_
##    <dbl> <chr> <chr> <dttm>              <chr> <chr>   <chr>          <dbl>
##  1     2 http~ tier~ 2019-04-28 13:57:00 photo "Aus u~ http~             17
##  2     3 http~ B90D~ 2019-04-28 06:00:01 video "Beim ~ http~             70
##  3     4 http~ FDP   2019-04-28 11:49:59 photo "Unser~ http~             16
##  4     5 http~ tier~ 2019-04-28 08:24:15 link  "Eine ~ http~              6
##  5     6 http~ CDU   2019-04-28 09:12:19 video "Freih~ http~            239
##  6     7 http~ SPD   2019-04-28 13:06:09 photo "Katar~ http~            180
##  7     9 http~ DieP~ 2019-04-28 07:44:28 link  "Der a~ http~             35
```

```
##  8    10 http~ CSU   2019-04-28 08:21:00 photo "#Klar~ http~            174           61
##  9    11 http~ DieP~ 2019-04-28 08:11:28 video "Anste~ http~             61          312
## 10    12 http~ alte~ 2019-04-28 14:55:00 link  "++ Ha~ http~           1163         1499
## # ... with 822 more rows, and 3 more variables: haha_count <dbl>, sad_count <dbl>, angry_count
```

Natürlich können auch Boolesche Operatoren (`!` für `NICHT`, `&` für `UND`, `|` für `ODER`) verwendet werden. Mehrere Bedingungen können auch per `,` getrennt werden (`UND`-Verknüpfung):

```
# Wähle nur Video-Posts der großen Koalition, die keine fehlenden Werte bei den Shares haben
filter(df_fb_eu, party %in% c("CDU", "CSU", "SPD"), type == "video", !is.na(shares_count))
```

```
## # A tibble: 62 x 16
##       id URL   party timestamp           type  message link  comments_count shares_count react
##    <dbl> <chr> <chr> <dttm>              <chr> <chr>   <chr>          <dbl>        <dbl>
##  1     6 http~ CDU   2019-04-28 09:12:19 video "Freih~ http~            239          136
##  2    18 http~ CDU   2019-04-29 11:38:01 video "Live:~ http~            140           17
##  3    26 http~ SPD   2019-04-29 09:02:51 video "Press~ http~            174           67
##  4    31 http~ CDU   2019-04-29 09:58:45 video "Da st~ http~            274           76
##  5    93 http~ CDU   2019-05-01 09:00:47 video "Heute~ http~            305          113
##  6   122 http~ CDU   2019-05-02 18:00:01 video "Heute~ http~            122           97
##  7   141 http~ CDU   2019-05-03 14:06:29 video "Anneg~ http~            158           65
##  8   152 http~ SPD   2019-05-03 14:00:10 video "Jetzt~ http~            515          114
##  9   178 http~ CSU   2019-05-05 08:30:00 video "Die E~ http~            153           55
## 10   197 http~ CSU   2019-05-06 11:04:28 video "Press~ http~             98           23
## # ... with 52 more rows, and 3 more variables: haha_count <dbl>, sad_count <dbl>, angry_count
```

8.2.3 Daten sortieren mit `arrange()`

Um den Datensatz für die Ansicht umzusortieren, wird die Funktion `arrange()` genutzt, die aufsteigend nach den angegebenen Variablen sortiert:

```
# Sortiere aufsteigend nach Datum
arrange(df_fb_eu, timestamp)
```

```
## # A tibble: 902 x 16
##       id URL   party timestamp           type  message link  comments_count shares_count react
##    <dbl> <chr> <chr> <dttm>              <chr> <chr>   <chr>          <dbl>        <dbl>
##  1     3 http~ B90D~ 2019-04-28 06:00:01 video "Beim ~ http~             70           28
##  2    13 http~ FDP   2019-04-28 06:18:18 photo "Wir w~ http~             47          110
##  3     9 http~ DieP~ 2019-04-28 07:44:28 link  "Der a~ http~             35           76
##  4    11 http~ DieP~ 2019-04-28 08:11:28 video "Anste~ http~             61          312
##  5    10 http~ CSU   2019-04-28 08:21:00 photo "#Klar~ http~            174           61
##  6     5 http~ tier~ 2019-04-28 08:24:15 link  "Eine ~ http~              6           46
```

```
##  7     1 http~ oedp~ 2019-04-28 09:00:00 video "Guido~ http~              0
##  8     6 http~ CDU   2019-04-28 09:12:19 video "Freih~ http~            239
##  9    15 http~ FDP   2019-04-28 10:40:57 photo "Weil ~ http~             14
## 10     4 http~ FDP   2019-04-28 11:49:59 photo "Unser~ http~             16
## # ... with 892 more rows, and 3 more variables: haha_count <dbl>, sad_count <dbl>, a
```

Werden mehrere Variablen angegeben, wird zunächst nach der ersten Variablen, dann innerhalb der ersten Variablen nach der zweiten Variablen usw. sortiert. Soll eine Variable stattdessen absteigend sortiert werden, wird der Variablenname in die Hilfsfunktion `desc()` gepackt:

```
# Sortiere alphabetisch aufsteigend nach Partei
# und innerhalb von Parteien absteigend nach Kommentaranzahl
arrange(df_fb_eu, party, desc(comments_count))
```

```
## # A tibble: 902 x 16
##       id URL   party timestamp           type  message link  comments_count shares_
##    <dbl> <chr> <chr> <dttm>              <chr> <chr>   <chr>          <dbl>        <
##  1   870 http~ alte~ 2019-05-26 15:50:19 video "+++ H~ http~          4829
##  2   752 http~ alte~ 2019-05-23 17:24:43 video "+++ E~ http~          3294
##  3   616 http~ alte~ 2019-05-20 09:49:00 photo "++ Me~ http~          3100
##  4   802 http~ alte~ 2019-05-24 17:01:12 video "+++ S~ http~          2874
##  5    44 http~ alte~ 2019-04-30 19:02:07 video "2. Te~ http~          2862
##  6   335 http~ alte~ 2019-05-10 17:27:58 video "+++ H~ http~          2425
##  7   343 http~ alte~ 2019-05-11 13:12:00 photo "++ Di~ http~          2363
##  8    50 http~ alte~ 2019-04-30 17:18:13 video "+++ H~ http~          2246
##  9   198 http~ alte~ 2019-05-06 09:15:00 photo "++ Tä~ http~          1947
## 10   868 http~ alte~ 2019-05-26 11:19:19 photo "++ Wi~ http~          1744
## # ... with 892 more rows, and 3 more variables: haha_count <dbl>, sad_count <dbl>, a
```

8.2.4 Neue Variablen hinzufügen mit `mutate()`

Mit `mutate()`, dem vielleicht einzigen nicht selbsterklärenden Funktionsnamen der sechs diskutierten Funktionen, können wir Datensätzen neue Variablen hinzufügen (oder alte überschreiben). Hierzu geben wir den neuen Variablennamen an, gefolgt von einem `=` und der Berechnung bzw. Konstruktion der neuen Variablen. Wird als Variablenname ein schon im Datensatz bestehender Variablenname verwendet, so wird diese Variable überschrieben. Mit Kommas getrennt können auch mehrere neue Variablen erstellt werden.

```
# Wir erstellen eine neue Variable comments_centered,
# die die Kommentarzahl am allgemeinen Mittelwert zentriert
# indem wir von jedem Wert den Mittelwert der Kommentarzahl abziehen
# und wandeln die bestehende Variable message in Kleinschreibung
```

```
# mittels der Funktion tolower() um.
#
# Zur Darstellung werden die beiden 'mutierten' Variablen
# anschließend mit select() ausgewählt

df_mutated <- mutate(df_fb_eu,
       comments_centered = comments_count - mean(comments_count, na.rm = TRUE),
       message = tolower(message))

select(df_mutated, comments_centered, message)
```

```
## # A tibble: 902 x 2
##    comments_centered message
##                <dbl> <chr>
##  1            -159.  "guido #klamt aus #ludwigsburg, listenplatz 5 auf der kandidatenliste der
##  2            -142.  "aus unserem europawahlprogramm, kapitel 7: gesundheits- und sozialpoliti
##  3             -89.2 "beim wahlkampf-camp in berlin waren gestern hunderte freiwillige, die si
##  4            -143.  "unser neuer bundesvorstand \U0001f389\U0001f38a\U0001f388 #chancennutzer
##  5            -153.  "eine neue studie der universität oxford zeigt, dass eine vegane ernährun
##  6              79.8 "freiheit ist nicht selbstverständlich. #unsereuropa steht für freiheitli
##  7              20.8 "katarina barley sagt: eine weitere koalition mit der #evp will ich nicht
##  8            -159.  "unser spitzenkandidat patrick breyer mit einem update zum eu19 workshop
##  9            -124.  "der absolut härteste „martin-sonneborn-moment" kommt für watson.de nach
## 10              14.8 "#klartext von bayerns ministerpräsident und csu-chef markus söder in der
## # ... with 892 more rows
```

8.2.5 Variablen zusammenfassen mit `summarize()`

Mit `summarize()`[6] fassen wir Variablen zusammen, indem wir Funktionen auf eine Variable anwenden. Das Resultat ist ein neues Tibble, das die zusammengefassten Variablen als Spalten enthält. Die Funktionsweise ist ähnlich wie bei `mutate()`:

```
# Mittelwert der drei zentralen Facebook-Metriken berechen
summarize(df_fb_eu,
          mean_comments = mean(comments_count, na.rm = TRUE),
          mean_shares = mean(shares_count, na.rm = TRUE),
          mean_reactions = mean(reactions_count, na.rm = TRUE))
```

```
## # A tibble: 1 x 3
##   mean_comments mean_shares mean_reactions
##           <dbl>       <dbl>          <dbl>
## 1          159.        249.           846.
```

[6] Wer *the King's English* bevorzugt: `summarise()` funktioniert auch.

8.2.6 Variablen gruppieren mit `group_by()`

Mittels `group_by()` können wir unseren Datensatz nach einer oder mehrerer Variablen gruppieren. Das Resultat ist erstmal ein Tibble, das nicht weiter von unserem Ausgangs-Tibble unterscheidet. Die Gruppierung wird dann jedoch bei folgenden Funktionen wie `mutate()` oder `summarize()` berücksichtig.

```
# Wir berechnen erneut die zentralen Facebook-Metriken
# mit summarize(), gruppieren aber zuvor nach Partei

grouped_df <- group_by(df_fb_eu, party)

summarize(grouped_df, mean_comments = mean(comments_count, na.rm = TRUE),
          mean_shares = mean(shares_count, na.rm = TRUE),
          mean_reactions = mean(reactions_count, na.rm = TRUE))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## # A tibble: 14 x 4
##    party                            mean_comments mean_shares mean_reactions
##    <chr>                                    <dbl>       <dbl>          <dbl>
##  1 alternativefuerde                       863.         1796.          4032.
##  2 B90DieGruenen                           106.          184.           660.
##  3 CDU                                     349.           87.1          628.
##  4 CSU                                     136.           57.1          499.
##  5 DiePARTEI                                60.4         160.          1343.
##  6 FamilienParteiDeutschlands                0.633        16.0           10.5
##  7 FDP                                      70            67.3          447.
##  8 freie.waehler.bundesvereinigung          25.3          43.1          141.
##  9 linkspartei                             116.          218.           935.
## 10 oedp.de                                   6.58         29.5          110.
## 11 Piratenpartei                            11.4          42.5          124.
## 12 SPD                                     220.          149.           719.
## 13 tierschutzpartei                         67.4         391.           979.
## 14 VoltDeutschland                          15.6          39.8          230.
```

Analog wird auch bei `mutate()` die Gruppierung in den Berechnungen berücksichtigt. Wenden wir die oben durchgeführte Mittelwert-Zentrierung der Kommentaranzahl auf unseren gruppierten Datensatz an, wird durch die `mean()`-Funktion der Mittelwert *innerhalb* der Gruppen (hier also der Parteien) berechnet. Im Ergebnis bekommen wir also für jeden Facebook-Post einen Wert, wie dieser von der durchschnittlichen Kommentaranzahl auf dieser Parteienseite abweicht:

```
mutated_df <- mutate(grouped_df,
                     comments_group_centered = comments_count - mean(comments_count, na.rm = TRUE

select(mutated_df, party, comments_group_centered, comments_count)
```

```
## # A tibble: 902 x 3
## # Groups:   party [14]
##    party            comments_group_centered comments_count
##    <chr>                              <dbl>          <dbl>
##  1 oedp.de                            -6.58              0
##  2 tierschutzpartei                  -50.4              17
##  3 B90DieGruenen                     -35.5              70
##  4 FDP                               -54                16
##  5 tierschutzpartei                  -61.4               6
##  6 CDU                              -110.              239
##  7 SPD                               -40.4             180
##  8 Piratenpartei                     -11.4               0
##  9 DiePARTEI                         -25.4              35
## 10 CSU                                37.7             174
## # ... with 892 more rows
```

Wir können auch nach mehreren Variablen gruppieren:

```
# Wir berechnen erneut die zentralen Facebook-Metriken
# mit summarize(), gruppieren aber zuvor nach Partei UND Post-Typ

grouped_df <- group_by(df_fb_eu, party, type)

summarize(grouped_df, mean_comments = mean(comments_count, na.rm = TRUE),
          mean_shares = mean(shares_count, na.rm = TRUE),
          mean_reactions = mean(reactions_count, na.rm = TRUE))
```

```
## `summarise()` regrouping output by 'party' (override with `.groups` argument)
```

```
## # A tibble: 46 x 5
## # Groups:   party [14]
##    party             type   mean_comments mean_shares mean_reactions
##    <chr>             <chr>          <dbl>       <dbl>          <dbl>
##  1 alternativefuerde link           826.        1341.          2953
##  2 alternativefuerde photo          860.        2466.          5269.
##  3 alternativefuerde video          875.         847.          2340.
##  4 B90DieGruenen     photo          134.         183.           902.
##  5 B90DieGruenen     video           77.9        184.           425.
##  6 CDU               photo          394.         114.           810.
```

```
##  7 CDU               video          293.         53.0           400.
##  8 CSU               link            25          11             132.
##  9 CSU               photo          143.         63.8           566.
## 10 CSU               status         416.        112.           1106
## # ... with 36 more rows
```

Wir sehen hier also, dass die AfD im Mittel 826.14 Kommentare auf Links bekommt, 860.49 auf Photos usw.

Gruppierungen können (und sollten) im Anschluss mittels `ungroup()` wieder entfernt werden (auch hier wird der Datensatz als Argument übergeben), um Probleme bei der weiteren Datentransformation zu vermeiden.

Eine besondere Variante von `group_by()` ist `rowwise()`, die den Datensatz zeilenweise gruppiert; dies ermöglicht zeilenweise Berechnungen mit Funktionen über mehrere Variablen hinweg, z. B. die Erstellung von Mittelwerts-Indizes:

```
# Gruppiere den Datensatz zeilenweise, um für jeden Post
# den Mittelwert der einzelnen Reactions (Like, Love etc.)
# zu berechnen

rowwise_df <- rowwise(df_fb_eu)

mutated_df <- mutate(rowwise_df,
                     mean_reactions = mean(c(like_count, love_count, wow_count, haha_c
                                           na.rm = TRUE))

select(mutated_df, mean_reactions)
```

```
## # A tibble: 902 x 1
## # Rowwise:
##    mean_reactions
##             <dbl>
##  1           1.5
##  2          65.8
##  3          35.8
##  4          43.7
##  5          24.2
##  6          66.3
##  7         116.
##  8           1.17
##  9         102
## 10          76.3
## # ... with 892 more rows
```

Eine Funktion, die einen häufigen Anwendungsfall von `group_by()`, `summarize()` und `ungroup()` kombiniert, ist `count()`, die die Fallzahl

einer oder mehrerer Gruppierungsvariablen ausgibt. Mit dem Argument `sort = TRUE` kann die Ausgabe zudem direkt absteigend nach Anzahl sortiert werden:

```
# Zähle Posts pro Partei
count(df_fb_eu, party)
```

```
## # A tibble: 14 x 2
##    party                              n
##    <chr>                          <int>
##  1 alternativefuerde                 79
##  2 B90DieGruenen                     67
##  3 CDU                               64
##  4 CSU                              103
##  5 DiePARTEI                         96
##  6 FamilienParteiDeutschlands        30
##  7 FDP                               94
##  8 freie.waehler.bundesvereinigung   30
##  9 linkspartei                       38
## 10 oedp.de                           71
## 11 Piratenpartei                     73
## 12 SPD                               49
## 13 tierschutzpartei                  33
## 14 VoltDeutschland                   75
```

```
# Zähle Posts pro Partei und Post-Typ und sortiere absteigend nach Anzahl
count(df_fb_eu, party, type, sort = TRUE)
```

```
## # A tibble: 46 x 3
##    party             type      n
##    <chr>             <chr> <int>
##  1 CSU               photo    76
##  2 FDP               photo    67
##  3 DiePARTEI         photo    53
##  4 alternativefuerde photo    45
##  5 oedp.de           photo    40
##  6 Piratenpartei     photo    39
##  7 B90DieGruenen     video    35
##  8 CDU               photo    35
##  9 VoltDeutschland   photo    35
## 10 SPD               photo    33
## # ... with 36 more rows
```

8.3 Daten speichern

Wenn wir unsere Dateien modifiziert haben, möchten wir diese wohl auch speichern bzw. exportieren.

8.3.1 Tabellarische Daten exportieren

Analog zu den `read_`-Funktionen stehen daher Exportfunktionen nach dem Schema `write_` zur Verfügung. Als Argumente werden dabei der Datensatz, der gespeichert werden soll, sowie der Dateipfad der zu speichernden Datei übergeben. Haben wir durch Modifikation beispielsweise das Tibble `df_modified` erstellt und möchten es in der Datei `datensatz_modifiziert.csv` im Unterordner `data` abspeichern, führen wir die `write_csv()`-Funktion aus:

```
write_csv(df_modified, "data/datensatz_modifiziert.csv")
```

`write_csv()` nutzt dabei das Komma `,` als Spalten- und einen Punkt `.` als Dezimaltrennzeichen. Möchten wir stattdessen das in Deutschland gebräuchliche Format mit Semikolon `;` als Spalten- und Komma `,` als Dezimaltrennzeichen haben, verwenden wir analog zu `read_csv2()` `write_csv2()`.

Da Excel öfters Probleme mit dem Einlesen von CSV-Dateien hat, können, soll die Datei danach in Excel betrachtet werden, auch die Funktionen `write_excel_csv()` bzw. `write_excel_csv2()` verwendet werden. Dies fügt ein spezielles Zeichen hinzu, das Excel den Datenimport erleichert.

8.3.2 R-Objekte exportieren

Beim Export als CSV gehen unweigerlich auch Informationen verloren – bei unserem Datensatz beispielsweise die Objekttypen, die R den jeweiligen Variablen zugeordnet hat. Wollen wir R-Objekte daher für die zukünftige Verwendung in R abspeichern, lohnt es sich, direkt das jeweilige R-Objekt zu exportieren. Hierfür steht das Dateiformat `.rds` zur Verfügung, mit dem beliebige R-Objekte – neben Datensätzen also auch Vektoren, Listen, statistische Modelle etc. - gespeichert werden können.

Die zugehörige Funktion lautet `saveRDS()` und wird analog zu den `write_`-Funktionen verwendet:

```
saveRDS(df_mutated, "data/datensatz_modifiziert.rds")
```

RDS-Dateien können dann jederzeit mit der Funktion `readRDS()` wieder geladen werden.

Sollen mehrere R-Objekte exportiert werden – also beispielsweise ein Ausgangsdatensatz, ein modifizierter Arbeitsdatensatz und zugehörige statistische Modelle – kann das Dateiformat `.RData` und die Funktion `save()` verwendet werden. Dabei werden alle zu speichernden Objekte gefolgt von dem benannten Argument `file =`, das den Dateipfad angibt, in dem Funktionsaufruf genannt:

```
save(df_fb_eu, df_mutated, "data/eu_file.RData")
```

So exportiere Objekte können dann gesammelt über die `load()`-Funktion, die den Dateipfad als Argument benötigt, wieder geladen werden, was sehr praktisch ist, um direkt den gesamten Arbeitsstand wiederherzustellen.

8.4 Übungsaufgaben

Erstellen Sie für die folgenden Übungsaufgaben eine eigene Skriptdatei oder eine R-Markdown-Datei und speichern diese als `ue8_nachname.R` bzw. `ue8_nachname.Rmd` ab.

Übungsaufgabe 8.1. Daten laden:

Laden Sie die Datei `facebook_europawahl.csv` aus Moodle in Ihr Projektverzeichnis herunter und laden Sie den Datensatz in R.

Übungsaufgabe 8.2. Daten modifizieren und speichern:

Erstellen Sie einen Teildatensatz, der:

- nur Posts der aktuell im Bundestag vertretenen Parteien enthält (CDU, CSU, SPD, FDP, Linke, Grüne, AfD); Tipp: Betrachten Sie vorab die Schreibweise der Parteien (bzw. deren Facebook-Accounts)
- nur die Variablen `party`, `timestamp`, `type` sowie alle Facebook-Metriken enthält
- eine neue Variable `total_count` enthält, in der für jeden Post die Gesamtzahl der Kommentare, Shares und Reactions angegeben ist

Speichern Sie diesen Teildatensatz sowohl als CSV- als auch als RDS-Datei.

Übungsaufgabe 8.3. Daten modifizieren und zusammenfassen:

Nutzen Sie die oben vorgestellten Funktionen, um pro Partei Mittelwert und Standardabweichung der drei Facebook-Metriken (Kommentare, Shares, Reactions) aller Posts zu berechnen, die in der Woche vor der Wahl (also nach dem 19.05.2019) erschienen sind.

Tipp: Logische Operatoren funktionieren auch mit Datums- und Zeitvariablen; Text, der wie ein Datum aussieht, wird dabei automatisch in ein Datum bzw. eine Zeitangabe konvertiert.

Chapter 9

Der Pipe-Operator %>%

Neben vielen praktischen Funktionen und der Datenstruktur Tibbles führt das Tidyverse auch ein neuens Syntax-Konzept in R ein: den sogenannten Pipe-Operator `%>%`, mit dem Argumente auf eine andere Art und Weise an Funktionen übergeben werden.[1]

9.1 Lesbarkeit verschachtelter Funktionen

Hierzu rufen wir uns zunächst noch einmal in Erinnerung, wie Funktionen in R grundsätzlich aufgerufen werden:

```
funktionsname(argument1 = wert1, argument2 = wert2, argument3 = wert3, ...)
```

Wir haben außerdem bereits gesehen, dass wir Funktionen ineinander verschachteln können, wenn wir mehrere Funktionen hintereinander aufrufen möchten:

```
round(mean(iris$Sepal.Length), 2)
```

```
## [1] 5.84
```

Das wird jedoch irgendwann sehr unübersichtlich und anfällig für Fehler – bereits bei diesem Beispiel müssen wir darauf achten, dass die Klammern zur richtigen Zeit geöffnet und vor allem wieder geschlossen werden, und wir müssen "Klammern zählen", wenn wir wissen wollen, zu welcher der aufgerufenen Funktionen

[1] Das Konzept ist aus anderen Programmiersprachen entlehnt und wurde ursprünglich durch das Package `magrittr` in R eingeführt; soll der Pipe-Operator `%>%` ohne das Tidyverse-Package genutzt werden, kann also dieses Package geladen werden: `library(magrittr)`.

das Argument 2 gehört. Zusätzlich ergibt sich durch die Verschachtelung die unnatürliche Lesereihenfolge von innen nach außen, was komplexeren Code schwer nachvollziehbar macht.

9.2 Ein Beispiel in Pseudo-Code

Um dies zu verdeutlichen, stellen wir uns einmal vor, eine typische Morgenroutine bestünde aus "Funktionen", die wir der Reihe nach "aufrufen":

1. Aufstehen
2. Frühstücken
3. Zähne putzen
4. Duschen
5. Anziehen

In R-Code ausgedrückt würde das also wie folgt aussehen:

```
einsatzbereit <- anziehen(duschen(zaehne_putzen(fruehstuecken(aufstehen(ich), food = "m
```

Wir könnten das ganze durch Einrückungen zumindest etwas lesbarer gestalten:

```
einsatzbereit <- anziehen(
  duschen(
    zaehne_putzen(
      fruehstuecken(
        aufstehen(ich),
        food = "muesli")
      ), wasser_temperatur = "warm")
  )
```

Das ist schon etwas besser, aber immer noch nicht sonderlich intuitiv zu lesen – und schließen wir nur eine Klammer an der falschen Stelle oder vergessen sie gar ganz, fliegt uns der gesamte Code um die Ohren.

Natürlich könnten wir die Schritte der Morgenroutine auch einzeln durchgehen und jeweils einem neuen "Objekt" zuweisen:

```
wach <- aufstehen(ich)
satt <- fruehstuecken(wach, food = "muesli")
sauber1 <- zaehe_putzen(satt)
sauber2 <- duschen(sauber1, wasser_temperatur = "warm")
einsatzbereit <- anziehen(sauber2)
```

Das erzeugt aber viele Objekte, die wir gar nicht weiter benötigen, da wir nur an `einsatzbereit` interessiert sind. Wir könnten natürlich auch immer das gleiche Objekt wieder und wieder überschreiben, darunter leidet dann aber erneut die Lesbarkeit.

Mit dem Pipe-Operator `%>%` können wir diese Schritte in einer logischen Lesereihenfolge ohne die Erstellung von redundaten Objekten durchführen:

```
einsatzbereit <- ich %>%
  aufstehen() %>%
  fruehstuecken(food = "muesli") %>%
  zaehne_putzen() %>%
  duschen(wasser_temperatur = "warm") %>%
  anziehen()
```

9.3 Formale Definition

Formal ausgedrückt übergibt der Pipe-Operator `%>%` das links von ihm stehende Objekt als erstes Argument an die rechts von ihm stehende Funktion:

```
# Die folgenden beiden Zeilen sind analog
f(x)
x %>% f()

# Oder anhand einer echten Funktion
mean(x) # ist analog zu
x %>% mean()
```

Weitere Funktionsargumente können regulär entweder positional oder explizit durch Namensnennung an die Funktion übergeben werden:

```
# Die folgenden beiden Zeilen sind wiederum analog
f(x, y, z)
x %>% f(y, z)

# Und wieder am Beispiel der mean()-Funktion
mean(x, na.rm = TRUE) # ist analog zu
x %>% mean(na.rm = TRUE)
```

9.4 Einsatz von Pipes im Tidyverse

Besonders sinnvoll sind *Pipes* dann, wenn wir viele Funktionen hintereinander am gleichen Ausgangsobjekt aufrufen wollen, z. B. wenn wir unterschiedliche

Schritte der Datenmodifikation an einem Datensatz vornehmen möchten. Bei den Tidyverse-Funktionen wissen wir, dass

1. das erste Argument immer der Datensatz, also ein Tibble, ist und
2. das Resultat der Funktion auch immer ein Datensatz, also ein Tibble ist.

Daher können wir diese Schritte schnell aneinanderreihen. Nutzen wir als Beispiel nochmals den Datensatz aus Kapitel 8:

```
# Wir laden den Datensatz
df_fb_eu <- read_csv("data/facebook_europawahl.csv")

# Wir erstellen einen modifizierten Datensatz, indem wir:
# 1. nur die Video-Posts auswählen
# 2. nur die Variablen id, party und comment_count auswählen
# 3. Nach Partei gruppieren
# 4. Eine neue Variable erstellen, die für jeden Post angibt,
#    welchen Anteil dieser an allen Kommentaren unter Post der
#    jeweiligen Partei hatte
# 5. heben die Gruppierung wieder auf und
# 6. weisen das Resultat dieser 'Pipe' dem Objekt modified_df zu

modified_df <- df_fb_eu %>% # Wir definieren die Zuweisung und übergeben df_fb_eu an
  filter(type == "video") %>% # die filter()-Funktion; der resultierende Datensatz wir
  select(id, party, comments_count) %>% # select() übergeben; das Resultat wiederum wi
  group_by(party) %>%                   # gruppiert etc.
  mutate(comment_percentage = comments_count / sum(comments_count)) %>%
  ungroup()

modified_df
```

```
## # A tibble: 272 x 4
##       id party         comments_count comment_percentage
##    <dbl> <chr>                  <dbl>              <dbl>
##  1     1 oedp.de                    0          NA
##  2     3 B90DieGruenen             70          NA
##  3     6 CDU                      239          NA
##  4     8 Piratenpartei              0           0
##  5    11 DiePARTEI                 61           0.0371
##  6    14 FDP                      358           0.117
##  7    16 DiePARTEI                 15           0.00912
##  8    18 CDU                      140          NA
##  9    26 SPD                      174           0.0599
## 10    31 CDU                      274          NA
## # ... with 262 more rows
```

Auch für schnelle deskriptive Auswertungen können wir Pipes gut nutzen – z. B. um uns schnell die Mittelwerte bestimmter Variablen gruppiert nach anderen Variablen anzuzeigen:

```
df_fb_eu %>%
  group_by(party, type) %>%
  summarize(mean_comments = mean(comments_count, na.rm = TRUE),
            mean_shares = mean(shares_count, na.rm = TRUE),
            mean_reactions = mean(reactions_count, na.rm = TRUE))
```

```
## `summarise()` regrouping output by 'party' (override with `.groups` argument)
```

```
## # A tibble: 46 x 5
## # Groups:   party [14]
##    party             type   mean_comments mean_shares mean_reactions
##    <chr>             <chr>          <dbl>       <dbl>          <dbl>
##  1 alternativefuerde link           826.       1341.           2953
##  2 alternativefuerde photo          860.       2466.           5269.
##  3 alternativefuerde video          875.        847.           2340.
##  4 B90DieGruenen     photo          134.        183.            902.
##  5 B90DieGruenen     video           77.9       184.            425.
##  6 CDU               photo          394.        114.            810.
##  7 CDU               video          293.         53.0           400.
##  8 CSU               link            25          11             132.
##  9 CSU               photo          143.         63.8           566.
## 10 CSU               status         416.        112.           1106
## # ... with 36 more rows
```

Praktisch, oder? Bleibt noch die eine Hürde, dass `%>%` eher kompliziert zu tippen ist – dankenswerterweise stellt RStudio aber auch hier eine Tastenkombination zur Verfügung: `Strg/Cmd + Shift + M` fügt den gesamten Operator ein.

9.5 Übungsaufgaben

Erstellen Sie für die folgende Übungsaufgabe eine eigene Skriptdatei oder eine R-Markdown-Datei und speichern diese als `ue9_nachname.R` bzw. `ue9_nachname.Rmd` ab.

Übungsaufgabe 9.1. Pipes:

Lösen Sie die Übungsaufgaben 8.2 und 8.3 erneut, aber verwenden Sie Pipes, um den Code lesbarer und mit weniger redundanten Zwischenobjekten zu gestalten. An welchen Stellen ist es sinnvoll bzw. weniger sinnvoll, Pipes zu verwenden?

Chapter 10

Daten umstrukturieren und zusammenfügen

Datenhandling umfasst oft auch etwas komplexere Schritte, als lediglich relevante Variablen zu selektieren und nach bestimmten Bedingungen zu filtern. In diesem Kapitel sehen wir uns daher an, wie wir Daten umstrukturieren und mehrere Datensätze zusammenfügen können.

10.1 Daten umstrukturieren

Wir haben in Kapitel 7.1.1 gesehen, dass dieselben Daten ganz unterschiedlich tabellarisch abgebildet werden können. Oftmals ist es jedoch so, dass wir unsere Daten in einem ganz bestimmten Format benötigen, um mit diesen weiterarbeiten zu können – beispielsweise um diese an eine Funktion, die ein statistisches Verfahren implementiert, zu übergeben oder diese grafisch darzustellen. Hierfür bietet das Tidyverse einige Funktionen, um Datensätze schnell umzustrukturieren.

10.1.1 Wide vs. Long Data

Einer der häufigsten Fälle der Umstrukturierung von Datensätzen betrifft das Konvertieren von *Wide Data* in *Long Data* und umgekehrt.

Nehmen wir als Beispiel einen Datensatz, der die Auflagezahlen verschiedener politischer Wochenmagazine beinhält:

```
## # A tibble: 3 x 4
##   medium      auflage_2018 auflage_2019 auflage_2020
```

```
##   <chr>              <dbl>        <dbl>        <dbl>
## 1 Der Spiegel       708077       701337       685799
## 2 Stern             539191       476097       422156
## 3 Focus             425737       373847       328587
```

Man spricht hierbei von *Wide* Data: Mehrere Beobachtungen desselben Wertetyps stehen in unterschiedlichen Spalten – in diesem Fall haben wir drei Spalten, in denen jeweils Auflagenzahlen stehen. Unsere Fallebene ist das einzelne Medium: jeweils eine Zeile für *Spiegel*, *Stern* und *Focus*.

Wir könnten dieselben Daten aber auch so darstellen:

```
## # A tibble: 9 x 3
##   medium       jahr auflage
##   <chr>       <int>   <dbl>
## 1 Der Spiegel  2018  708077
## 2 Der Spiegel  2019  701337
## 3 Der Spiegel  2020  685799
## 4 Stern        2018  539191
## 5 Stern        2019  476097
## 6 Stern        2020  422156
## 7 Focus        2018  425737
## 8 Focus        2019  373847
## 9 Focus        2020  328587
```

Hierbei handelt es sich um *Long* Data: alle Beobachtungen desselben Wertetyps, also z. B. die Auflagenzahlen, stehen in einer Spalte, die weiteren Spalten dienen zur Identifikation dieser Beobachtungen, hier also, für welches Jahr und für welches Medium diese gelten. Unsere Fallebene wäre also *Medium und Jahr*: jeweils eine Zeile pro Medium und Jahr.

Wide Data ist häufig intuitiver zu lesen, insbesondere wenn es sich um bivariate Verteilungen handelt, da die Darstellung einer Kreuztabelle gleicht. Tatsächlich würden wir diese erste tabellarische Darstellung aber nicht als *tidy* bezeichnen, da derselbe Werte- bzw. Beobachtungstyp in unterschiedlichen Spalten steht.

Viele Funktionen in R sind jedoch auf Long Data ausgelegt. Häufig müssen wir daher Wide Data in Long Data transformieren:

10.1.2 Wide Data in Long Data transformieren mit `pivot_longer()`

Mittels `pivot_longer()` können wir Wide Data in Long Data umwandeln. Als erstes Argument nutzen wir hierbei wie immer den Datensatz, den wir transformieren möchten, gefolgt von einem Vektor, der alle Spalten umfasst, die "länger" gemacht werden sollen.

Als Beispiel nutzen wir den ersten Auflagen-Datensatz aus dem vorherigen Unterkapitel, der über folgenden Code erstellt wird:

```
auflagen_wide <- tibble(
  medium = c("Der Spiegel", "Stern", "Focus"),
  auflage_2018 = c(708077, 539191, 425737),
  auflage_2019 = c(701337, 476097, 373847),
  auflage_2020 = c(685799, 422156, 328587)
)
```

Wir möchten nun alle Auflagenspalten (`auflage_2018`, `auflage_2019`, `auflage_2020`) transformieren, sodass alle Auflagenwerte in einer Spalte stehen. Wir übergeben diese Spalten daher als zweites Argument als `pivot_longer()`:

```
auflagen_wide %>%
  pivot_longer(c(auflage_2018, auflage_2019, auflage_2020))
```

```
## # A tibble: 9 x 3
##   medium      name          value
##   <chr>       <chr>         <dbl>
## 1 Der Spiegel auflage_2018 708077
## 2 Der Spiegel auflage_2019 701337
## 3 Der Spiegel auflage_2020 685799
## 4 Stern       auflage_2018 539191
## 5 Stern       auflage_2019 476097
## 6 Stern       auflage_2020 422156
## 7 Focus       auflage_2018 425737
## 8 Focus       auflage_2019 373847
## 9 Focus       auflage_2020 328587
```

Das führt uns schon (fast) zum gewünschten Ergebnis. Wie wir sehen, stehen nun alle Auflagenzahlen in der Spalte `value`, die Information aus den Spaltennamen stehen in der Spalte `name`. Wir können den Aufruf aber noch etwas anpassen, um die Ausgabe zu optimieren:

- Mit nur drei Variablen war es kein großer Aufwand, alle Variablen zum umtransformieren einzeln anzugeben. Wenn wir jedoch Auflagenzahlen der vergangenen 50 Jahre hätten, wäre dies sehr viel Tipparbeit. Wir können jedoch alle Hilfsfunktionen, die auch bei `select()` (siehe Kapitel 8.2.1) zur Verfügung stehen verwenden. In unserem Fall könnten wir uns die Arbeit beispielsweise mit `starts_with("auflage")` abkürzen.[1]

[1] Da alle Spalten bis auf `medium` transformiert werden sollen, könnten wir alternativ auch mit `-medium` angeben, um alle Spalten außer eben dieser zu nutzen.

- Die neuen Variablen haben recht generische Namen. Hier können wir mit den Argumenten `names_to` (Name(n) der neuen Spalte(n) basierend auf den alten Spaltennamen) und `values_to` (Name der neuen Wertespalte) Text-Vektoren übergeben, die die neuen Spaltennamen beinhalten. In unserem Fall bieten sich daher `names_to = "jahr"` und `values_to = "auflage"` an.
- Der Bestandteil `"auflage_"` ist redundant; schöner wäre es wenn lediglich die Jahreszahl angeben wird. Hierfür können wir das Argument `names_prefix` nutzen und einen Textbestandteil der Spaltennamen angeben, der abgeschnitten werden soll – in unserem Fall also `names_prefix = "auflage_"`.

```
auflagen_wide %>%
  pivot_longer(starts_with("auflage"), names_to = "jahr", values_to = "auflage", names_
```

```
## # A tibble: 9 x 3
##   medium      jahr  auflage
##   <chr>       <chr>   <dbl>
## 1 Der Spiegel 2018   708077
## 2 Der Spiegel 2019   701337
## 3 Der Spiegel 2020   685799
## 4 Stern       2018   539191
## 5 Stern       2019   476097
## 6 Stern       2020   422156
## 7 Focus       2018   425737
## 8 Focus       2019   373847
## 9 Focus       2020   328587
```

Schon fast perfekt! Das einzige Schönheitsfehler ist, dass die Variable `jahr` nun vom Typ `character` ist – `pivot_longer()` weist diesen Typ allen Informationen zu, die aus den ursprünglichen Spaltennamen stammen. Sinnvoller wäre ein numerischer Objekttyp. Hier können wir dem Argument `names_transform` eine Liste übergeben (auch wenn dies nur eine Variable betrifft), die für alle aus den ursprünglichen Spaltennamen generierten Variablen jeweils den gewünschten Objekttyp angibt:

```
# Da nun alles passt, weisen wir unseren neuen Long-Datensatz auch einem Objekt zu
auflagen_long <- auflagen_wide %>%
  pivot_longer(starts_with("auflage"), names_to = "jahr", values_to = "auflage",
               names_prefix = "auflage_", names_transform = list(jahr = as.integer))

auflagen_long
```

```
## # A tibble: 9 x 3
```

```
##   medium       jahr auflage
##   <chr>       <int>   <dbl>
## 1 Der Spiegel  2018  708077
## 2 Der Spiegel  2019  701337
## 3 Der Spiegel  2020  685799
## 4 Stern        2018  539191
## 5 Stern        2019  476097
## 6 Stern        2020  422156
## 7 Focus        2018  425737
## 8 Focus        2019  373847
## 9 Focus        2020  328587
```

Betrachten wir ein zweites, noch etwas komplexeres Beispiel:

```
auflagen_wide2 <- tibble(
  medium = c("Der Spiegel", "Stern", "Focus"),
  auflage_2018_q1 = c(708077, 539191, 425737),
  auflage_2018_q2 = c(704656, 528860, 417759),
  auflage_2018_q3 = c(716663, 514889, 412165),
  auflage_2018_q4 = c(712268, 480739, 413276),
  auflage_2019_q1 = c(701337, 476097, 373847),
  auflage_2019_q2 = c(707459, 464489, 367101),
  auflage_2019_q3 = c(719326, 466019, 364254),
  auflage_2019_q4 = c(691451, 440284, 349944),
  auflage_2020_q1 = c(685799, 422156, 328587)
)

auflagen_wide2
```

```
## # A tibble: 3 x 10
##   medium auflage_2018_q1 auflage_2018_q2 auflage_2018_q3 auflage_2018_q4 auflage_2019_q1 aufla
##   <chr>            <dbl>           <dbl>           <dbl>           <dbl>           <dbl>
## 1 Der S~          708077          704656          716663          712268          701337
## 2 Stern           539191          528860          514889          480739          476097
## 3 Focus           425737          417759          412165          413276          373847
## # ... with 1 more variable: auflage_2020_q1 <dbl>
```

Nun haben in den Spaltennamen gleich zwei Informationen: das Jahr und das jeweilige Quartal. Es ist daher sinnvoll, diese Informationen beim Transformieren in getrennten Spalten zu speichern. Hierzu können wir zusätzlich das Argument `names_sep` heranziehen, mit dem wir ein(e) Zeichen(kette) übergeben – in unserem Fall den Unterstrich `_`, an der wir die ursprünglichen Spaltennamen weiter aufteilen möchten. Entsprechend müssen wir in `names_to` dann mehrere

neuen Spaltennamen angeben – in unserem Fall 2, da wir `jahr` und `quartal` trennen möchten:[2]

```
auflagen_long2 <- auflagen_wide2 %>%
  pivot_longer(starts_with("auflage"), names_to = c("jahr", "quartal"), values_to = "au
               names_prefix = "auflage_", names_sep = "_", names_transform = list(jahr

auflagen_long2
```

```
## # A tibble: 27 x 4
##    medium       jahr quartal auflage
##    <chr>       <int> <chr>     <dbl>
##  1 Der Spiegel  2018 q1       708077
##  2 Der Spiegel  2018 q2       704656
##  3 Der Spiegel  2018 q3       716663
##  4 Der Spiegel  2018 q4       712268
##  5 Der Spiegel  2019 q1       701337
##  6 Der Spiegel  2019 q2       707459
##  7 Der Spiegel  2019 q3       719326
##  8 Der Spiegel  2019 q4       691451
##  9 Der Spiegel  2020 q1       685799
## 10 Stern        2018 q1       539191
## # ... with 17 more rows
```

`pivot_longer()` bietet also eine mächtige, wenn auch anfangs nicht immer intuitive Syntax zum umformen von Datensätzen – aber auch ohne die Kenntnis aller Zusatzargumente hätten wir einen tranformierten Datensatz, den wir mit wenigen zusätzlichen Funktionen wie `select()` oder `rename()` sowie `mutate()` in unsere gewünschte Form bringen können.

10.1.3 Long Data in Wide Data transformieren mit `pivot_wider()`

Umgekehrt können wir natürlich auch Wide Data in Long Data transformieren. Die passende Funktion heißt `pivot_wider()`. Hier geben wir mittels `names_from` an, die Werte welcher Variablen in Spalten überführt werden sollen, und definieren mit `values_from`, welche Variable die Werte für die neu erzeugten Spalten liefert.

Nehmen wir Long-Variante unseres jährlichen Auflagen-Datensatzes:

[2] Noch schöner wäre es, wenn wir automatisch auch noch das `q` aus `q1`, `q2` etc. entfernen und das Quartal auch als `integer` speichern könnten, aber die nötigen Funktionen zum Umgang mit Textvariablen lernen wir erst in zwei Wochen. Möglich ist es aber.

```
auflagen_long
```

```
## # A tibble: 9 x 3
##   medium       jahr auflage
##   <chr>       <int>   <dbl>
## 1 Der Spiegel  2018  708077
## 2 Der Spiegel  2019  701337
## 3 Der Spiegel  2020  685799
## 4 Stern        2018  539191
## 5 Stern        2019  476097
## 6 Stern        2020  422156
## 7 Focus        2018  425737
## 8 Focus        2019  373847
## 9 Focus        2020  328587
```

Um diesen in den ursprünglichen Wide-Datensatz zu transformieren, wollen wir also die Werte in `jahr` in Spaltennamen umwandeln und die Werte in `auflage` auf diese neuen Spalten verteilen. Wir geben also `names_from = "jahr"` und `values_from = "auflage"` an:

```
auflagen_long %>%
  pivot_wider(names_from = jahr, values_from = auflage)
```

```
## # A tibble: 3 x 4
##   medium      `2018` `2019` `2020`
##   <chr>        <dbl>  <dbl>  <dbl>
## 1 Der Spiegel 708077 701337 685799
## 2 Stern       539191 476097 422156
## 3 Focus       425737 373847 328587
```

Da Spaltennamen, die nicht mit Buchstaben beginnen, suboptimal sind und wir nun nicht mehr direkt sehen können, dass es sich um Auflagen-Daten handelt, können und sollten wir mittels `names_prefix` auch wieder die ursprünglichen Spaltennamen herstellen[3]:

```
auflagen_long %>%
  pivot_wider(names_from = jahr, values_from = auflage, names_prefix = "auflage_")
```

```
## # A tibble: 3 x 4
##   medium      auflage_2018 auflage_2019 auflage_2020
##   <chr>              <dbl>        <dbl>        <dbl>
```

[3] Im Gegensatz zu `pivot_longer` entfernt `names_prefix` bei `pivot_wider` das Prefix nicht, sondern fügt dieses hinzu.

```
## 1 Der Spiegel       708077       701337       685799
## 2 Stern             539191       476097       422156
## 3 Focus             425737       373847       328587
```

Im zweiten Fall stehen in mehreren Spalten Informationen, die wir in Spaltennamen überführen wollen, nämlich `jahr` und `quartal`:

```
auflagen_long2
```

```
## # A tibble: 27 x 4
##    medium       jahr quartal auflage
##    <chr>       <int> <chr>     <dbl>
##  1 Der Spiegel  2018 q1       708077
##  2 Der Spiegel  2018 q2       704656
##  3 Der Spiegel  2018 q3       716663
##  4 Der Spiegel  2018 q4       712268
##  5 Der Spiegel  2019 q1       701337
##  6 Der Spiegel  2019 q2       707459
##  7 Der Spiegel  2019 q3       719326
##  8 Der Spiegel  2019 q4       691451
##  9 Der Spiegel  2020 q1       685799
## 10 Stern        2018 q1       539191
## # ... with 17 more rows
```

Zum Transformieren nutzen wir erneut `names_from`, nur geben dieses Mal beide Variablen an, die für die Spaltennamen kombiniert werden sollen, und übergeben mit `names_sep` wieder ein Trennzeichen[4]:

```
auflagen_long2 %>%
  pivot_wider(names_from = c(jahr, quartal), values_from = auflage, names_prefix = "au
```

```
## # A tibble: 3 x 10
##   medium auflage_2018_q1 auflage_2018_q2 auflage_2018_q3 auflage_2018_q4 auflage_20
##   <chr>            <dbl>           <dbl>           <dbl>           <dbl>          <
## 1 Der S~          708077          704656          716663          712268          70
## 2 Stern           539191          528860          514889          480739          47
## 3 Focus           425737          417759          412165          413276          37
## # ... with 1 more variable: auflage_2020_q1 <dbl>
```

10.1.4 Spalten aufteilen mit `separate()`

Schauen wir uns dieselben Daten nochmals in einer anderen Darstellung an. Mehrere Werte in einer Spalte – gruselig, aber kommt vor:

[4] Tatsächlich wäre der Unterstrich `_` aber auch bereits als Default-Wert von `names_sep` eingestellt.

```
auflagen_gruselig <- tibble(
  medium = c("Der Spiegel", "Stern", "Focus"),
  auflagen_18_bis_20 = c("708077;701337;685799",
                         "539191;476097;422156",
                         "425737;373847;328587"),
)

auflagen_gruselig
```

```
## # A tibble: 3 x 2
##   medium      auflagen_18_bis_20
##   <chr>       <chr>
## 1 Der Spiegel 708077;701337;685799
## 2 Stern       539191;476097;422156
## 3 Focus       425737;373847;328587
```

Hier hilft uns die Funktion `separate()`, mit der wir mit dem Argument `col` eine Spalte zum Trennen, mit dem Argument `into` einen Vektor mit neuen Spaltennamen und dem Argument `sep` ein Trennzeichen, an dem die bisherigen Werte getrennt werden sollen, mitteilen.

In unserem Fall möchten wir die Spalte `auflagen_18_bis_20` am Semikolon `;` aufteilen:

```
auflagen_gruselig %>%
  separate(col = auflagen_18_bis_20, into = c("auflage_2018", "auflage_2019", "auflage_2020"), se
```

```
## # A tibble: 3 x 4
##   medium      auflage_2018 auflage_2019 auflage_2020
##   <chr>       <chr>        <chr>        <chr>
## 1 Der Spiegel 708077       701337       685799
## 2 Stern       539191       476097       422156
## 3 Focus       425737       373847       328587
```

Das hat schon gut funktioniert, nur ist der Objekttyp nun jeweils `character`. Mit dem zusätzlichen Argument `convert = TRUE` versucht `separate()`, den neuen Spalten direkt einen passenderen numerischen Objekttyp zu geben:

```
auflagen_gruselig %>%
  separate(col = auflagen_18_bis_20, into = c("auflage_2018", "auflage_2019", "auflage_2020"), se
           convert = TRUE)
```

```
## # A tibble: 3 x 4
```

```
##   medium      auflage_2018 auflage_2019 auflage_2020
##   <chr>              <int>        <int>        <int>
## 1 Der Spiegel       708077       701337       685799
## 2 Stern             539191       476097       422156
## 3 Focus             425737       373847       328587
```

10.2 Daten zusammenfügen

In größeren Forschungsprojekten haben wir oftmals nicht nur einen Datensatz, sondern mehrere Datensätze – etwa weil verschiedene Teildatensätze getrennt erhoben wurden oder weil diese auf unterschiedlichen Ebenen liegen. Für weitere Analysen sollen diese nun zusammengefügt werden.

10.2.1 Teildatensätze zusammenfügen mit `bind_rows()` und `bind_cols()`

Beginnen wir mit dem einfachsten Fall, dem Zusammenfügen von gleichförmigen Datensätzen: wir wollen einem bestehenden Datensatz also entweder neue Zeilen (= neue Fälle) oder neue Spalten (=neue Variablen hinzufügen). Hierfür bietet das Tidyverse die Funktionen `bind_rows()` und `bind_cols()`. Bei beiden werden als Argumente einfach die zu verbindenden Datensätze angegeben.

Nehmen wir einmal an, wir hätten die Auflagendaten aus den vorherigen Beispielen getrennt für die Nachrichtenmedien erhoben:

```
# Aufteilen des Datensatzes

auflagen_spiegel <- auflagen_long %>%
  filter(medium == "Der Spiegel")

auflagen_stern <- auflagen_long %>%
  filter(medium == "Stern")

auflagen_focus <- auflagen_long %>%
  filter(medium == "Focus")
```

Vor uns liegen nun also drei Datensätze, die jeweils Daten eines Nachrichtenmediums enthalten:

```
auflagen_spiegel
```

```
## # A tibble: 3 x 3
##   medium       jahr auflage
```

```
##   <chr>       <int>   <dbl>
## 1 Der Spiegel  2018  708077
## 2 Der Spiegel  2019  701337
## 3 Der Spiegel  2020  685799
```

```
auflagen_stern
```

```
## # A tibble: 3 x 3
##   medium  jahr auflage
##   <chr>  <int>   <dbl>
## 1 Stern   2018  539191
## 2 Stern   2019  476097
## 3 Stern   2020  422156
```

```
auflagen_focus
```

```
## # A tibble: 3 x 3
##   medium  jahr auflage
##   <chr>  <int>   <dbl>
## 1 Focus   2018  425737
## 2 Focus   2019  373847
## 3 Focus   2020  328587
```

Praktischerweise sind die Datensätze gleichförmig: alle drei haben dieselben Variablen. Wir können diese also einfach mit `bind_rows()` untereinander kleben, um einen gemeinsamen Datensatz zu erstellen:

```
auflagen_spiegel %>%
  bind_rows(auflagen_stern, auflagen_focus)
```

```
## # A tibble: 9 x 3
##   medium       jahr auflage
##   <chr>       <int>   <dbl>
## 1 Der Spiegel  2018  708077
## 2 Der Spiegel  2019  701337
## 3 Der Spiegel  2020  685799
## 4 Stern        2018  539191
## 5 Stern        2019  476097
## 6 Stern        2020  422156
## 7 Focus        2018  425737
## 8 Focus        2019  373847
## 9 Focus        2020  328587
```

Fehlende Variablen in einem Teildatensatz werden automatisch durch fehlende Werte ersetzt:

```
auflagen_spiegel %>%
  select(-auflage) %>% # Wir entfernen zu Demonstrationszwecken die Auflage-Spalte
  bind_rows(auflagen_stern, auflagen_focus)
```

```
## # A tibble: 9 x 3
##   medium       jahr auflage
##   <chr>       <int>   <dbl>
## 1 Der Spiegel  2018      NA
## 2 Der Spiegel  2019      NA
## 3 Der Spiegel  2020      NA
## 4 Stern        2018  539191
## 5 Stern        2019  476097
## 6 Stern        2020  422156
## 7 Focus        2018  425737
## 8 Focus        2019  373847
## 9 Focus        2020  328587
```

Hier haben wir den Vorteil, dass die Identifikationsvariable `medium` bereits vorhanden ist, sodass wir die Auflagendaten auch im vollständigen Datensatz problemlos zuordnen können. Was, wenn das nicht der Fall ist?

```
# Wir entfernen zu Demonstrationszwecken die Medium-Variable
auflagen_spiegel <- select(auflagen_spiegel, -medium)
auflagen_stern <- select(auflagen_stern, -medium)
auflagen_focus <- select(auflagen_focus, -medium)

auflagen_spiegel
```

```
## # A tibble: 3 x 2
##    jahr auflage
##   <int>   <dbl>
## 1  2018  708077
## 2  2019  701337
## 3  2020  685799
```

Zwar können wir die Teildatensätze nun anhand des Namens gut identifizieren, bekommen aber bei einem Gesamtdatensatz Probleme, die Auflagenzahlen zuzuordnen:

```
auflagen_spiegel %>%
  bind_rows(auflagen_stern, auflagen_focus)
```

```
## # A tibble: 9 x 2
##    jahr auflage
##   <int>   <dbl>
## 1  2018  708077
## 2  2019  701337
## 3  2020  685799
## 4  2018  539191
## 5  2019  476097
## 6  2020  422156
## 7  2018  425737
## 8  2019  373847
## 9  2020  328587
```

In solchen Fällen können wir die Datensätze beim "zusammenkleben" benennen und über das `.id`-Argument einen Spaltennamen für die Identifikationsvariable festlegen:

```
bind_rows("Der Spiegel" = auflagen_spiegel,
          "Stern" = auflagen_stern,
          "Focus" = auflagen_focus,
          .id = "medium")
```

```
## # A tibble: 9 x 3
##   medium       jahr auflage
##   <chr>       <int>   <dbl>
## 1 Der Spiegel  2018  708077
## 2 Der Spiegel  2019  701337
## 3 Der Spiegel  2020  685799
## 4 Stern        2018  539191
## 5 Stern        2019  476097
## 6 Stern        2020  422156
## 7 Focus        2018  425737
## 8 Focus        2019  373847
## 9 Focus        2020  328587
```

Ganz ähnlich funktioniert auch `bind_cols()`, nur dass wir dieses Mal Variablen, also Spalten hinzufügen. Erinnern wir uns an den Wide-Datensatz der Auflagendaten:

```
auflagen_wide
```

```
## # A tibble: 3 x 4
##   medium      auflage_2018 auflage_2019 auflage_2020
##   <chr>              <dbl>        <dbl>        <dbl>
## 1 Der Spiegel       708077       701337       685799
## 2 Stern             539191       476097       422156
## 3 Focus             425737       373847       328587
```

Nun haben wir auch noch ein paar ältere Auflagendaten erhoben:

```
auflagen_alt <- tibble(
  medium = c("Der Spiegel", "Stern", "Focus"),
  auflage_2016 = c(793087, 719290, 474285),
  auflage_2017 = c(771066, 595729, 456020)
)

auflagen_alt
```

```
## # A tibble: 3 x 3
##   medium      auflage_2016 auflage_2017
##   <chr>              <dbl>        <dbl>
## 1 Der Spiegel       793087       771066
## 2 Stern             719290       595729
## 3 Focus             474285       456020
```

Wir ergänzen diese Variablen mittels `bind_cols()`:

```
auflagen_wide %>%
  bind_cols(auflagen_alt)
```

```
## New names:
## * medium -> medium...1
## * medium -> medium...5
```

```
## # A tibble: 3 x 7
##   medium...1  auflage_2018 auflage_2019 auflage_2020 medium...5  auflage_2016 auflag
##   <chr>              <dbl>        <dbl>        <dbl> <chr>              <dbl>
## 1 Der Spiegel       708077       701337       685799 Der Spiegel       793087
## 2 Stern             539191       476097       422156 Stern             719290
## 3 Focus             425737       373847       328587 Focus             474285
```

Wie wir sehen, hat das ganz gut funktioniert – die Mediumspalte ist nun doppelt vorhanden, aber mittels `select()` könnten wir diese schnell ausschließen. Generell ist aber bis auf in Ausnahmefällen vom Hinzufügen von Variablen mittel `bind_cols()` abzuraten, da die Prozedur sehr fehleranfällig ist – sind beide Datensätze nicht gleich sortiert, haben Sie schnell Daten miteinander verbunden, die gar nicht zusammengehören. Ist eine Identifikations-Variable vorhanden, so gibt es deutlich sinnvollere und sicherere Funktionen:

10.2.2 Relationale Daten zusammenführen

Nehmen wir an, dass wir ein Forschungsprojekt zu politischen Wochenmagazinen durchführen. Wir haben dazu neben den bereits bekannten Auflagenzahlen auch noch Daten zu den Magazinen selbst und einige Artikel erhoben und diese in den Datensätzen `auflagen`, `info` und `artikel` abgelegt.

Der Datensatz `auflagen` enthält die uns bereits bekannten Auflagenzahlen auf Ebene Medium pro Quartal und Jahr:

```
# Falls Sie direkt mitarbeiten möchten:
# Kopieren Sie bei den folgenden drei Code-Blöcken den Code
# um die Beispieldatensätze zu erstellen
auflagen <- auflagen_long2
auflagen
```

```
## # A tibble: 27 x 4
##    medium       jahr quartal auflage
##    <chr>       <int> <chr>     <dbl>
##  1 Der Spiegel  2018 q1       708077
##  2 Der Spiegel  2018 q2       704656
##  3 Der Spiegel  2018 q3       716663
##  4 Der Spiegel  2018 q4       712268
##  5 Der Spiegel  2019 q1       701337
##  6 Der Spiegel  2019 q2       707459
##  7 Der Spiegel  2019 q3       719326
##  8 Der Spiegel  2019 q4       691451
##  9 Der Spiegel  2020 q1       685799
## 10 Stern        2018 q1       539191
## # ... with 17 more rows
```

Der Datensatz `info` enthält allgemeine Informationen zu den Magazinen auf Mediums-Ebene:

```
info <- tibble(
  medium = c("Der Spiegel", "Stern", "Focus"),
```

```
  sitz = c("Hamburg", "Hamburg", "Berlin"),
  erscheinungstag = c("Samstag", "Donnerstag", "Samstag"),
  erstausgabe = lubridate::dmy(c("04-01-1947", "01-08-1948", "18-01-1993"))
)

info
```

```
## # A tibble: 3 x 4
##   medium      sitz    erscheinungstag erstausgabe
##   <chr>       <chr>   <chr>           <date>
## 1 Der Spiegel Hamburg Samstag         1947-01-04
## 2 Stern       Hamburg Donnerstag      1948-08-01
## 3 Focus       Berlin  Samstag         1993-01-18
```

Der Datensatz `artikel` schließlich enthält alle für das Forschungsprojekt relevanten Artikel auf Artikel-Ebene:

```
artikel <- tibble(
  titel = c("Ein spannender Artikel", "Noch ein Artikel", "Und noch ein Artikel", "Und
  autor_innen = c("Max Mustermann", "Erika Musterfrau", "John Doe; Jane Doe", "Mario R
  medium = c("Stern", "Stern", "Der Spiegel", "Focus"),
  ausgabe = c(1L, 1L, 1L, 1L),
  jahr = c(2020L, 2020L, 2020L, 2020L),
  seiten = c("20-22", "11", "17-25", "104-106")
)

artikel
```

```
## # A tibble: 4 x 6
##   titel                  autor_innen        medium      ausgabe  jahr seiten
##   <chr>                  <chr>              <chr>         <int> <int> <chr>
## 1 Ein spannender Artikel Max Mustermann     Stern             1  2020 20-22
## 2 Noch ein Artikel       Erika Musterfrau   Stern             1  2020 11
## 3 Und noch ein Artikel   John Doe; Jane Doe Der Spiegel       1  2020 17-25
## 4 Und noch einer         Mario Rossi        Focus             1  2020 104-106
```

Insgesamt haben wir also drei verschiedene Datensätze, deren Fälle allesamt auf unterschiedlichen Ebenen liegen. Allerdings hängen die Datensätze auch implizit zusammen: die Variable `medium` findet sich in allen drei Datensätzen; über sie können wir zwei oder alle drei Datensätze zusammenführen. Datensätze, die über Variablen miteinander verbunden werden können, nennt man *relationale* Daten, sie stehen also in einer Beziehung zueinander.

10.2.2.1 Schlüsselvariablen (Keys)

Variablen, über die Datensätze zusammengefügt werden, bezeichnet man als Schlüsselvariablen, im Englischen auch *Keys*. Schlüsselvariablen identifzieren Fälle in einem Datensatz *eindeutig*. Je nach Datenaufbereitung kann hierzu eine einzige Variable ausreichen; manchmal sind aber auch Kombinationen aus mehreren Variablen nötig, um Fälle eindeutig zuzuordnen.

Zudem werden zwei Typen von Schlüsselvariablen unterschieden:

- als *primary key* wird eine Variable bzw. eine Kombination von Variablen bezeichnet, die im *aktuellen* Datensatz einen Fall eindeutig identifizieren.
- als *foreign key* wird eine Variable bzw. eine Kombination von Variablen bezeichnet, die in einem *anderen* Datensatz Fälle eindeutig identifizieren.

Im Beispieldatensatz `info` ist `medium` der *primary key*. Jeder Wert von `medium` kommt exakt einmal vor und identifiziert einen Fall (= eine Zeile) somit eindeutig:

```
## # A tibble: 3 x 4
##   medium      sitz    erscheinungstag erstausgabe
##   <chr>       <chr>   <chr>           <date>
## 1 Der Spiegel Hamburg Samstag         1947-01-04
## 2 Stern       Hamburg Donnerstag      1948-08-01
## 3 Focus       Berlin  Samstag         1993-01-18
```

Wie sieht die Sache im Datensatz `artikel` aus:

```
## # A tibble: 4 x 6
##   titel                   autor_innen        medium      ausgabe  jahr seiten
##   <chr>                   <chr>              <chr>         <int> <int> <chr>
## 1 Ein spannender Artikel Max Mustermann     Stern             1  2020 20-22
## 2 Noch ein Artikel       Erika Musterfrau   Stern             1  2020 11
## 3 Und noch ein Artikel   John Doe; Jane Doe Der Spiegel       1  2020 17-25
## 4 Und noch einer         Mario Rossi        Focus             1  2020 104-106
```

Hier wäre `titel` der naheliegendste *primary key*, da darüber alle Fälle eindeutig identifiziert werden können. `medium` taugt hier hingegen nicht als *primary key*, da der Wert `"Stern"` nicht eindeutig ist – es gibt zwei Artikel vom Stern im Datensatz, entsprechend reicht die bloße Angabe `"Stern"` nicht aus, um einen Fall in `artikel` eindeutig zu identifizieren.

`medium` kann hier aber als *foreign key* für den Datensatz `info` dienen, da die Werte aus `medium` im Datensatz `artikel` sich eindeutig Fällen im Datensatz

`info` zuordnen lassen – jeder der drei einzigartigen Werte von `medium` im Datensatz `artikel` (`"Stern"`, `"Der Spiegel"` & `"Focus"`) kommt nur einmal in der Variable `medium` im Datensatz `info` vor.

Da `medium` in `info` *primary key* und in `artikel` *foreign key* ist, bilden diese beiden Datensätze über die Schlüsselvariablen `medium` eine *Relation*. Wir können nun Daten aus dem Datensatz `info` dem Datensatz `artikel` zuordnen und beispielsweise für jeden Artikel in `artikel` die Variable `erscheinungstag` hinzufügen; für alle Spiegel-Artikel erhalten wir dann den Wert `"Samstag"`, für alle Stern-Artikel den Wert `"Donnerstag"`, usw.

Bevor wir nun die Datensätze tatsächlich miteinander verbinden, schauen wir uns noch einen komplizierteren Fall an. Was ist der *primary key* im Datensatz `auflagen`:

```
## # A tibble: 27 x 4
##    medium       jahr quartal auflage
##    <chr>       <int> <chr>     <dbl>
##  1 Der Spiegel  2018 q1       708077
##  2 Der Spiegel  2018 q2       704656
##  3 Der Spiegel  2018 q3       716663
##  4 Der Spiegel  2018 q4       712268
##  5 Der Spiegel  2019 q1       701337
##  6 Der Spiegel  2019 q2       707459
##  7 Der Spiegel  2019 q3       719326
##  8 Der Spiegel  2019 q4       691451
##  9 Der Spiegel  2020 q1       685799
## 10 Stern        2018 q1       539191
## # ... with 17 more rows
```

Aktuell existiert keine einzelne Identifikationsvariable, mit der wir einen Eintrag in `auflagen` eindeutig identifzieren können – sowohl `medium` als auch `jahr` und `quartal` enthalten doppelte Werte.[5] Der *primary key* ist in diesem Fall eine Kombination aus den drei Variablen `medium`, `jahr` und `quartal`, da jede der Merkmalskombinationen dieser drei Variablen nur ein einziges Mal auftritt.

In solchen Fällen ist es sinnvoll, über eine laufende Nummer eine Variable zu erstellen, die als *primary key* fungieren kann. Dies können wir schnell mit den uns bereits bekannten Funktionen umsetzen:

```
auflagen %>%
  mutate(id = 1:nrow(auflagen))
```

[5] Tatsächlich sind die Werte in `auflage` aktuell noch eindeutig, aber Wertvariablen sind nicht gut als Schlüsselvariablen geeignet; sammeln wir noch mehr Auflagendaten aus früheren Jahren und von mehr Magazinen, steigt die Wahrscheinlichkeit, dass irgendwann auch mal ein doppelter Wert auftaucht.

```
## # A tibble: 27 x 5
##    medium       jahr quartal auflage    id
##    <chr>       <int> <chr>     <dbl> <int>
##  1 Der Spiegel  2018 q1       708077     1
##  2 Der Spiegel  2018 q2       704656     2
##  3 Der Spiegel  2018 q3       716663     3
##  4 Der Spiegel  2018 q4       712268     4
##  5 Der Spiegel  2019 q1       701337     5
##  6 Der Spiegel  2019 q2       707459     6
##  7 Der Spiegel  2019 q3       719326     7
##  8 Der Spiegel  2019 q4       691451     8
##  9 Der Spiegel  2020 q1       685799     9
## 10 Stern        2018 q1       539191    10
## # ... with 17 more rows
```

10.2.2.2 Join-Operationen

Um Datensätze nun miteinander zu verbinden, greifen wir auf `_join`-Funktionen zurück. Das Tidyverse orientiert sich dabei an Konzepten und Bezeichnungen, die auch in der Datenbanksprache SQL verwendet werden. Dabei werden zum Zusammenfügen vier Arten von *Joins* unterschieden (im Folgenden gehen wir davon aus, dass die beiden Datensätze, die wir verbinden möchten, `x` und `y` heißen):

- `inner_join()`: Alle Zeilen, die sowohl in `x` als auch in `y` vorkommen, und alle Spalten aus `x` und `y`
- `left_join()`: Alle Zeilen, die in `x` vorkommen und alle Spalten aus `x` und `y`
- `right_join()`: Alle Zeilen, die in `y` vorkommen und alle Spalten aus `x` und `y`
- `full_join()`: Alle Zeilen aus `x` und `y` und alle Spalten aus `x` und `y`

Man kann diese Unterschiede auch durch ein Venn-Diagramm verdeutlichen:

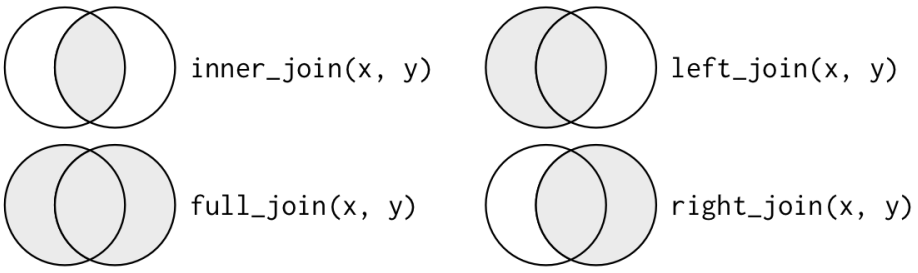


Figure 10.1: Join-Operationen. Quelle: R for Data Science

Daneben gibt es noch Join-Operationen, die weniger zum Zusammenfügen als zum Filtern von Datensätzen auf Basis von anderen Datensätzen zu gebrauchen sind, da dabei keine Spalten aus einem Datensatz in den anderen kopiert werden:

- `anti_join()`: Alle Zeilen in `x`, die *nicht* auch in `y` vorkommen, und alle Spalten aus `x` (nicht aber aus `y`)
- `semi_join()`: Alle Zeilen in `x`, die auch in `y` vorkommen, und alle Spalten aus `x` (nicht aber aus `y`)

In der Praxis benötigen wir vor allem `left_join()` und `inner_join()`, `anti_join()` wird uns aber bei der automatisierten Inhaltsanalyse wieder begegnen.

Alle `_join()`-Funktionen benötigen als Argumente die beiden Datensätze, die zusammengefügt (bzw. gefiltert) werden sollen. Wird kein weiteres Argument angegeben, werden automatisch alle gleichnamigen Variablen in beiden Datensätzen als Schlüsselvariablen verwendet.

```
artikel %>%
  left_join(info)
```

```
## Joining, by = "medium"
```

```
## # A tibble: 4 x 9
##   titel                 autor_innen        medium      ausgabe  jahr seiten  sitz
##   <chr>                 <chr>              <chr>         <int> <int> <chr>   <chr>
## 1 Ein spannender Artikel Max Mustermann     Stern             1  2020 20-22   Hambu
## 2 Noch ein Artikel       Erika Musterfrau   Stern             1  2020 11      Hambu
## 3 Und noch ein Artikel   John Doe; Jane Doe Der Spiegel       1  2020 17-25   Hambu
## 4 Und noch einer         Mario Rossi        Focus             1  2020 104-106 Berli
```

Wir sehen zunächst in der Konsole, dass die Join-Operation auf der Basis der Schlüsselvariable `medium` erfolgte, da diese als einzige in beiden Datensätzen zu finden ist. Das Ergebnis ist ein Datensatz, der alle Variablen aus beiden Datensätzen enthält und diese auf Basis der Schlüsselvariablen alle Spaltenwerte den richtigen Zeilen zugeordnet hat.

Mit dem Argument `by` können wir die Schlüsselvariable(n) auch explizit angeben; dies ist vor allem dann praktisch, wenn die Schlüsselvariablen in beiden Datensätzen unterschiedlich heißen. Auch bei gleichnamigen Variablen ist aber dennoch sinnvoll, die Schlüsselvariablen explizit zu nennen, um etwaigen Problemen vorzubeugen:

```
artikel %>%
  left_join(info, by = "medium")
```

```
## # A tibble: 4 x 9
##   titel                 autor_innen        medium      ausgabe  jahr seiten  sitz    erscheir
##   <chr>                 <chr>              <chr>         <int> <int> <chr>   <chr>   <chr>
## 1 Ein spannender Artikel Max Mustermann     Stern             1  2020 20-22   Hamburg Donnerst
## 2 Noch ein Artikel       Erika Musterfrau   Stern             1  2020 11      Hamburg Donnerst
## 3 Und noch ein Artikel   John Doe; Jane Doe Der Spiegel       1  2020 17-25   Hamburg Samstag
## 4 Und noch einer         Mario Rossi        Focus             1  2020 104-106 Berlin  Samstag
```

Zu beachten ist, dass `left_join()` auch Ergebnisse liefert, wenn die Schlüsselvariablen in `y` nicht eindeutig ist bzw. sind. Dies ist der Fall, wenn wir beide Datensätze im obigen Beispiel vertauschen. Zur Erinnerung, `medium` im `artikel`-Datensatz enthält Mehrfachwerte:

```
artikel
```

```
## # A tibble: 4 x 6
##   titel                 autor_innen        medium      ausgabe  jahr seiten
##   <chr>                 <chr>              <chr>         <int> <int> <chr>
## 1 Ein spannender Artikel Max Mustermann     Stern             1  2020 20-22
## 2 Noch ein Artikel       Erika Musterfrau   Stern             1  2020 11
## 3 Und noch ein Artikel   John Doe; Jane Doe Der Spiegel       1  2020 17-25
## 4 Und noch einer         Mario Rossi        Focus             1  2020 104-106
```

Welche der beiden `"Stern"`-Zeilen ordnet `left_join()` nun `info` zu?

```
info %>%
  left_join(artikel, by = "medium")
```

```
## # A tibble: 4 x 9
##   medium      sitz    erscheinungstag erstausgabe titel                 autor_innen        au
##   <chr>       <chr>   <chr>           <date>      <chr>                 <chr>
## 1 Der Spiegel Hamburg Samstag         1947-01-04  Und noch ein Artikel  John Doe; Jane Doe
## 2 Stern       Hamburg Donnerstag      1948-08-01  Ein spannender Artikel Max Mustermann
## 3 Stern       Hamburg Donnerstag      1948-08-01  Noch ein Artikel      Erika Musterfrau
## 4 Focus       Berlin  Samstag         1993-01-18  Und noch einer        Mario Rossi
```

Die Antwort: beide Zeilen – `left_join()` dupliziert also Zeilen in `x`, wenn es auf Basis der Schlüsselvariablen mehrere Entsprechungen in `y` gibt. Daher ist es nach einer Join-Operation immer sinnvoll zu prüfen, ob der resultierende Datensatz auch den gewünschten Umfang hat.

10.3 Übungsaufgaben

Erstellen Sie für die folgenden Übungsaufgaben eine eigene Skriptdatei oder eine R-Markdown-Datei und speichern diese als `ue10_nachname.R` bzw. `ue10_nachname.Rmd` ab.

Übungsaufgabe 10.1. Daten umstrukturieren:

Laden Sie den Datensatz `facebook_europawahl.csv`, der schon aus den vorigen Übungen bekannt ist. Wählen Sie zunächst die Variablen `id`, `party`, `timestamp` sowie `comments_count`, `shares_count` und `reactions_count` aus.

Wir möchten diesen Datensatz nun auf eine "Medium-pro-Tag-und-Facebook-Metrik"-Ebene umstrukturieren, sodass für jeden Post (zu erkennen an der Variablen `id`) drei Zeilen existieren, in der jeweils einmal die Anzahl der Kommentare, die Anzahl der Shares sowie die Anzahl der Reactions steht.

Der resultierende Datensatz sollte fünf Variablen haben: die Identifikationsvariablen `id`, `party`, `timestamp` (aus dem alten Datensatz) und `metric` (gibt an, um welche der drei Facebook-Metriken es sich handelt) sowie die Wertvariable `value` (die Anzahl der jeweiligen Facebook-Metrik).

Übungsaufgabe 10.2. Daten zusammenfügen:

Auf Moodle finden Sie den Datensatz `facebook_codings.csv`. Dieser enthält für die Facebook-Posts aus `facebook_europawahl.csv` manuelle Codierungen, ob bestimmte Themen in diesen Posts vorkommen oder nicht. Zudem ist die `id`-Variable der Facebook-Posts angegeben.

Laden Sie auch diesen Datensatz in R und fügen ihn mit `facebook_europawahl` zusammen, sodass für jeden Post neben den API-Informationen (`timestamp`, `message`, Facebook-Metriken etc.) auch die manuellen Codierungen ersichtlich sind.

Chapter 11

Daten visualisieren

Ein wichtiger Schritt bei der Datenexploration – und später auch bei der Veröffentlichung und Kommunikation – ist die Visualisierung von Daten, da wir so Auffälligkeiten und Zusammenhänge finden und verdeutlichen können, die uns durch reine deskriptive Auswertungen verborgen bleiben.

Die Basisversion von R bietet bereits einige Funktionen zur Datenvisualisierung, darunter die Funktion `plot()`, die je nach übergebenem Objekt (bzw. übergebenen Objekten) eine zugehörige Visualiserung erzeugt, beispielsweise Scatterplots für numerische Objekte oder diagnostische Plots für Regressionsmodelle.

Im Tidyverse übernimmt das Package `ggplot2` sämtliche Schritte der Datenvisualisierung und stellt hierfür eine eigene Syntax – ein "*grammar of graphics*" – bereit, die sehr viele Möglichkeiten und Flexibilität bei der Datenvisualisierung eröffnet, aber auch entsprechend eine eigene Lernkurve aufweist. Daher ist es sinnvoll, sich zunächst mit einigen Grundlagen der Datenvisualisierung vertraut zu machen.

11.1 Grundlagen der Datenvisualisierung

Auch wenn uns in den Medien häufig Datenvisualisierungen begegnen und wir im Studium auch schon selbst Datenvisualisierungen erstellt haben, lohnt es sich, die Entscheidungen, die hinter einer solchen Visualisierung stehen, zu verdeutlichen. Wir können solche Entscheidungen auf drei wesentliche Elemente einschränken:

- *Encodings* (bzw. *Aesthetics*): Welche Daten bzw. welche Variablen sollen in welche visuellen Elemente (z. B. X-Achse, Y-Achse, Farben) übertragen werden?

- *Geometrics*: Wie sollen diese visuellen Elemente in dem (zumeist) zweidimensionalen Raum, den uns eine grafische Darstellung zur Verfügung stellt, dargestellt werden (z. B. als Punkte, Linien, Balken)?
- *Scales*: Auf welchen Skalen sollen die betreffenden Skalen abgebildet werden?

Nehmen wir als aktuelles Beispiel Ergebnisse aus dem *ARD Deutschlandtrend*:

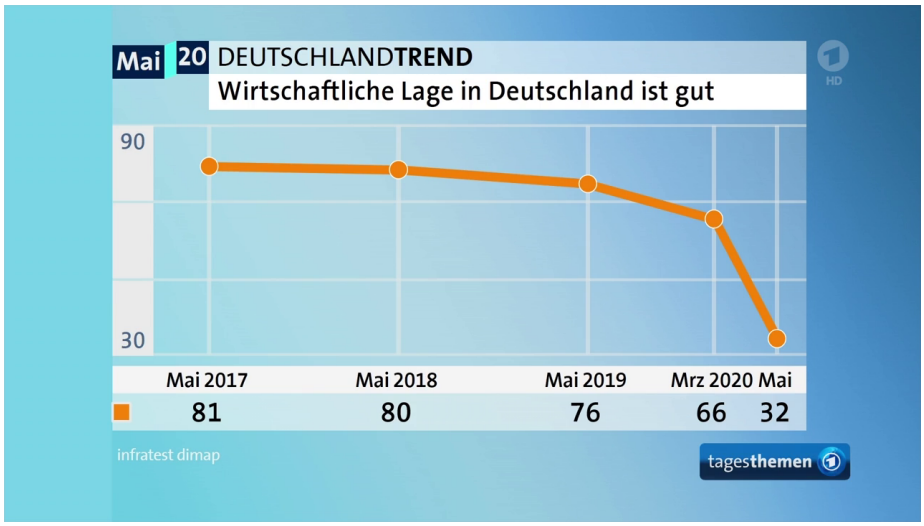


Figure 11.1: Quelle: Screenshot Das Erste tagesthemen, 22:30 Uhr, 07.05.2020

Wir können diese Grafik intuitiv vermutlich schnell erfassen und "lesen", ohne uns groß über diese Entscheidungen Gedanken zu machen. Versuchen wir dennoch, diese Visualisierung und die dahinterstehenden Daten zu entschlüsseln:

- Die Grafik basiert auf zwei Variablen: eine Zeitvariable (in Monaten) sowie eine numerische Variable, in der die prozentuale Zustimmung zur Aussage "Die wirtschaftliche Lage in Deutschland ist gut" festgehalten ist. Diese beiden Variablen sind in verschiedenen visuellen Elementen *kodiert* (→ *Encodings*): die Zeit ist auf der X-Achse abgetragen, die Zustimmung auf der Y-Achse.
- Die in diesen visuellen Elementen kodierten Werte werden nun durch geometrische Formen (→ *Geometrics*) dargestellt: die einzelnen Wert-Kombinationen aus beiden Variablen (z. B. Mai 2017 / 81 Prozent) sind als Punkte im Koordinatensystem abgetragen und werden zusätzlich durch Linien verbunden.
- Die Darstellung wird maßgeblich durch Entscheidungen zur Skalierung (→ *Scales*) beeinflusst: Die Zeitvariable (X-Achse) beginnt im Mai 2017 und endet im Mai 2020. Die Prozentvariable (Y-Achse) zeigt nicht den

gesamten möglichen Wertebereich, sondern umfasst lediglich den Bereich von 30 bis 90 Prozent.

Alle anderen Grafikbestandteile – Farben, Wertebeschriftungen etc. – lassen in diesem Fall keine zusätzlichen Aussagen über die zugrundeliegenden Daten aus bzw. kommunizieren keine weiteren Daten, sondern dienen dem Erscheinungsbild.

Ziehen wir noch eine zweite Visualisierung aus derselben *tagesthemen*-Ausgabe heran, vermutlich eine der bekanntesten Visualisierungen politischer Kommunikationsdaten in Deutschland: die Sonntagsfrage ("Welche Partei würden Sie wählen, wenn am nächsten Sonntag Bundestagswahl wäre?").

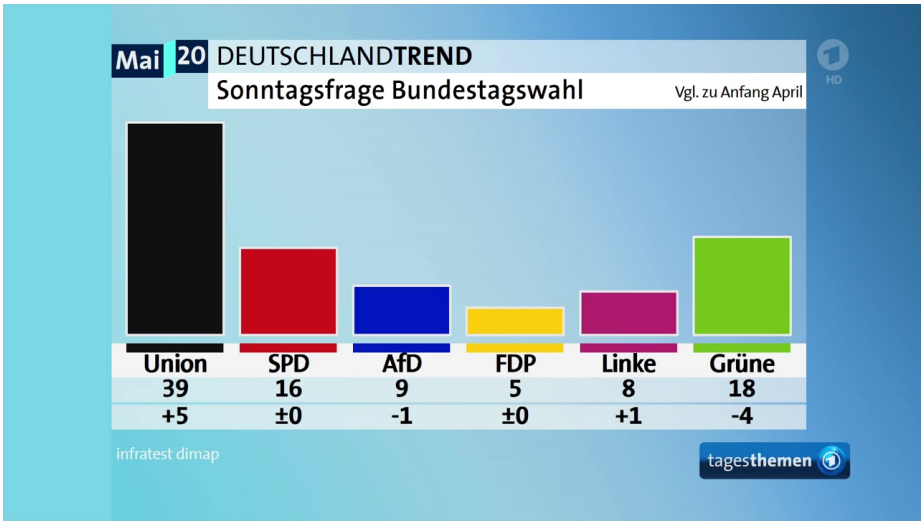


Figure 11.2: Quelle: Screenshot Das Erste tagesthemen, 22:30 Uhr, 07.05.2020

- in dieser Grafik sind erneut zwei Variablen kodiert: die kategoriale Variable Partei und die numerische Variable Wahlabsicht.[1] Partei ist dabei gleich *doppelt* visuell kodiert: zum einen auf der X-Achse, zum anderen auch farblich – eine solche doppelte Kodierung erfolgt aus Gründen der Ästhetik und Lesbarkeit, fügt der Visualisierung aber keine weiteren Informationen hinzu[2]. Wichtig ist, dass wir eine Variable zwar mehrfach

[1] Man könnte argumentieren, dass als dritte Variable zusätzlich noch die Veränderung der Wahlabsicht im Vergleich zum Vormonat ersichtlich ist; diese ist aber lediglich als (zusätzliche) Wertbeschriftung abgetragen, nicht jedoch im eigentlichen Diagrammbereich visuell kodiert.

[2] Wir können das leicht überprüfen, indem wir uns die Grafik jeweils ohne die andere Kodierung feststellen: auch wenn alle Balken gleichfarbig wären, könnten wir ohne Probleme dieselben Schlüsse aus der Grafik ziehen; würde die Kodierung "X-Achse" wegfallen, wären die Parteien also nicht nebeneinander auf der Achse abgetragen, sondern in einem Balken übereinandergestapelt, würden wir die einzelnen Balkenabschnitte durch die Farben auseinanderhalten können.

kodieren, niemals aber dieselbe visuelle Kodierung für mehrere Variablen verwenden können. Die aktuelle Wahlabsicht ist auf der Y-Achse abgetragen.
- als geometrische Form der Datenrepräsentation wurden Balken (bzw. Säulen) gewählt, deren Form und Aussehen entsprechend der visuellen Kodierungen durch Wahlabsicht (Höhe der Balken) und Partei (Farbe der Balken, Position auf der X-Achse) bestimmt wird.
- bei der Skalierung der X-Achse wurde eine Anordnung der kategorialen Ausprägungen – d.h., der Parteien – nach dem Ergebnis der Bundestagswahl 2017 (und nicht etwa alphabetisch, oder absteigend nach aktueller Wahlabsicht etc.) gewählt. Die Y-Achse reicht von 0 bis zum Maximalwert der Daten (in diesem Fall also 39 Prozent); und auch die Farbskalierung folgt einer bewussten Entscheidung – nämlich, dass den kategorialen Ausprägungen der Partei-Variablen die zugehörige Parteifarbe zugeordnet wurde.

Diese drei Schritte – *Encoding*, *Geometrics*, *Scales* – stellen also die wesentliche Entscheidungen der Datenvisualisierung dar. Schauen wir uns nun an, wir diese Schritte in R umsetzen können.

11.2 Datenvisualisierung mit `ggplot2`

Das Package `ggplot2` – das zu den Kernpackages des Tidyverse gehört und entsprechend mit `library(tidyverse)` direkt mitgeladen wird – orientiert sich an dieser schrittweisen Erstellung von Datenvisualisierungen. Vorab aber der Hinweis, dass es sich bei `ggplot2` um ein sehr umfangreiches und mächtiges Package handelt, das zudem mit einer eigenen Logik aufwartet und das man daher kaum an einem Nachmittag verinnerlichen wird. Es geht also zunächst darum, einige Grundprinzipien zu verstehen. Nicht zuletzt findet man online zahlreiche Vorlagen bzw. Code-Beispiele, sodass man in der Regel nach etwas herumprobieren zur gewünschten Darstellung kommt.

Wir nutzen erneut den Facebook-Europawahl-Datensatz, den wir wie gewohnt laden. Aus Gründen der Übersichtlichkeit filtern wir zudem erneut nur im Bundestag vertretenen Parteien an::

```
library(tidyverse)

bt_parteien <- c("alternativefuerde", "B90DieGruenen", "CDU", "CSU", "FDP", "linkspart

facebook_europawahl <- read_csv("data/facebook_europawahl.csv") %>%
  filter(party %in% bt_parteien)
```

Die Hauptfunktion im `ggplot2`-Package lautet `ggplot()` und benötigt wie alle

Figure 11.3: Illustration von @allison_horst: https://twitter.com/allison_horst

anderen Tidyverse-Funktionen auch zunächst den Datensatz, auf dessen Basis wir plotten möchten. Entsprechend können wir auch wieder Pipes verwenden:

```
facebook_europawahl %>%
  ggplot()
```


`ggplot()` initialisiert die Visualisierung mit einer leeren Diagrammfläche; Daten sind darauf natürlich noch nicht abgetragen, da wir noch keine Encodings, Geometrics und Scales angegeben haben.

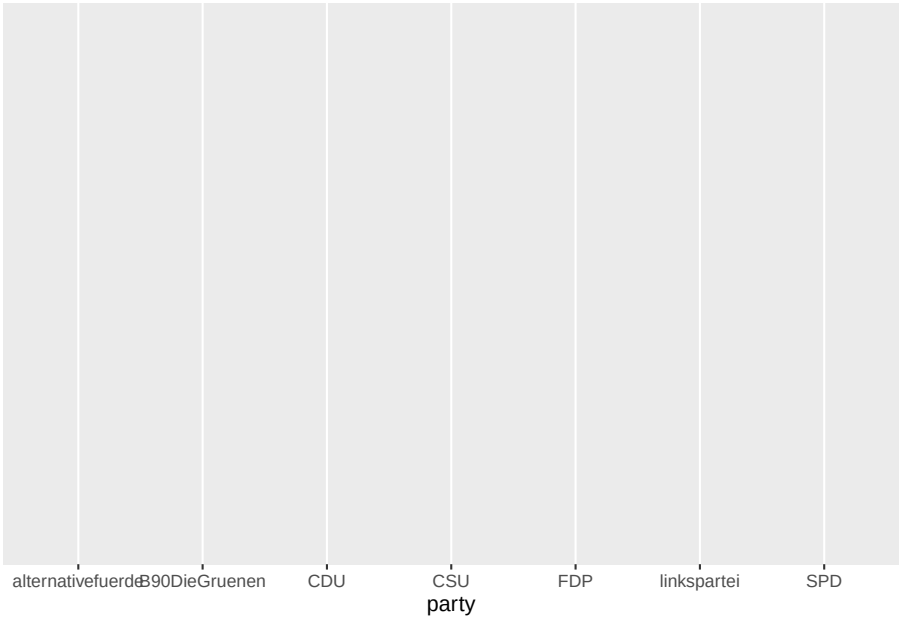
11.2.1 Encodings (`aesthetics`) definieren

Encodings heißen in `ggplot2` *Aesthetics* und werden innerhalb des Funktionsaufrufs von `ggplot()` als zweites Funktionsargument über die Funktion `aes()` angegeben. Optionen für Aesthetics sind unter anderem:

- `x`: Werte auf X-Achse
- `y`: Werte auf Y-Achse
- `color`: Linienfarben (für Punkte, Linien, Konturen von Flächen)
- `fill`: Füllfarben (für Flächen, z. B. Balken)
- `shape`: Punkform (nur bei Verwendung von Punktdiagrammen)
- `linetype`: Linienart (z. B. durchgezogen, gestrichelt etc.; entsprechend nur bei Liniendiagrammen)
- `alpha`: Transparenz

Um beispielsweise die Partei (Variable `party` im Datensatz) auf der X-Achse zu kodieren, geben wir an:

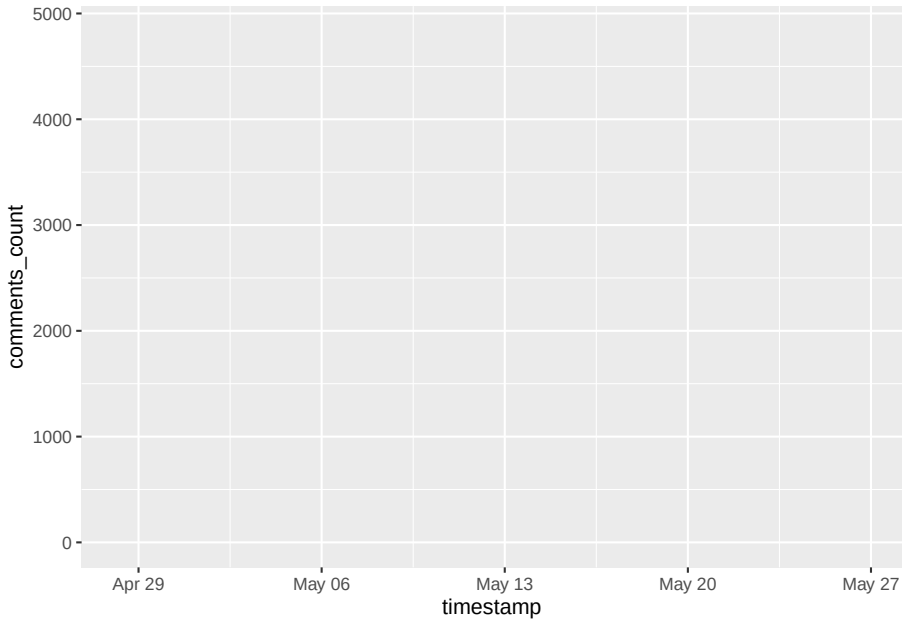
```
facebook_europawahl %>%
  ggplot(aes(x = party))
```



Wir sehen, dass die X-Achse nun schon für die Parteien vorbereitet wurde; mehr haben wir noch nicht definiert, entsprechend ist auch noch nicht mehr zu sehen.

Möchten wir stattdessen die Anzahl der Kommentare (Variable `comments_count`) im Zeitverlauf (Variable `timestamp`) und farblich getrennt nach Parteien (Variable `party`) zeigen, definieren wir die Aesthetics wie folgt:

```
facebook_europawahl %>%
  ggplot(aes(x = timestamp, y = comments_count, color = party))
```



Auch hier sehen wir, dass im Koordinatensystem bereits auf der X- und Y-Achse die entsprechenden Variablen eingetragen sind. Sichtbar ist aber weiterhin noch nicht viel mehr, da wir noch nicht definiert haben, in welchen Geometrics unsere Daten repräsentiert werden sollen:

11.2.2 Geometrics hinzufügen

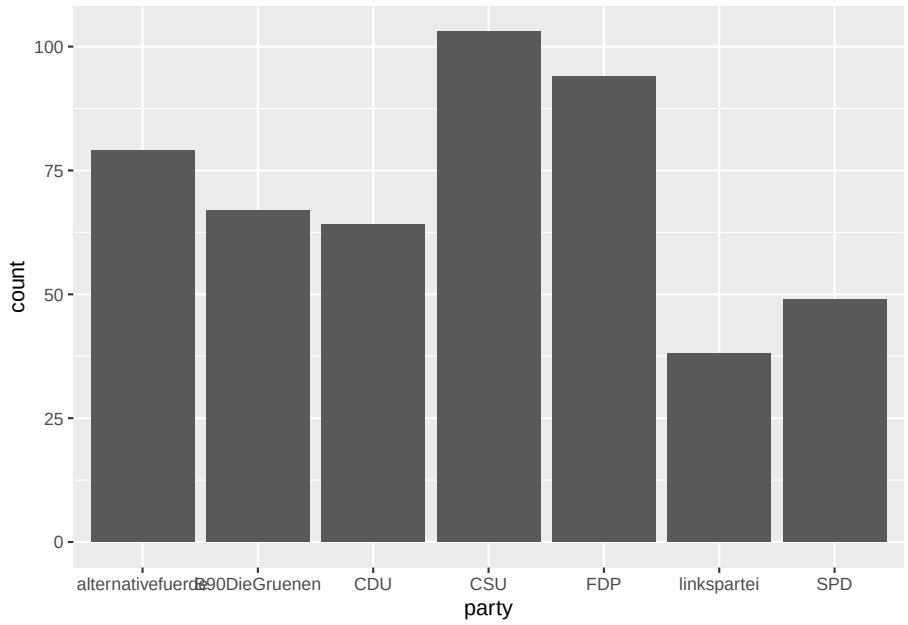
Dem Ausgangsaufruf von `ggplot()` können wir nun beliebige Diagramm-Elemente der Reihe nach hinzufügen – für Geometrics stehen hierbei verschiedene Funktionen zur Verfügung, die allesamt mit `geom_` beginnen. Dazu gehören:

- `geom_point()`: Punkte
- `geom_line()`: Linien
- `geom_bar()`: Balken für Häufigkeiten der `x`-Variable (`y`-Kodierung muss *nicht* angegeben werden)
- `geom_col()`: Balken, deren Höhe durch eine `y`-Variable spezifiziert wird
- `geom_hist()`: Histogramme
- `geom_boxplot()`: Boxplots

. . . und viele weitere. Eine Besonderheit ist, dass diese und alle weiteren Elemente dem Ausgangsplot nicht per Pipe `%>%`, sondern per `+` hinzugefügt werden.

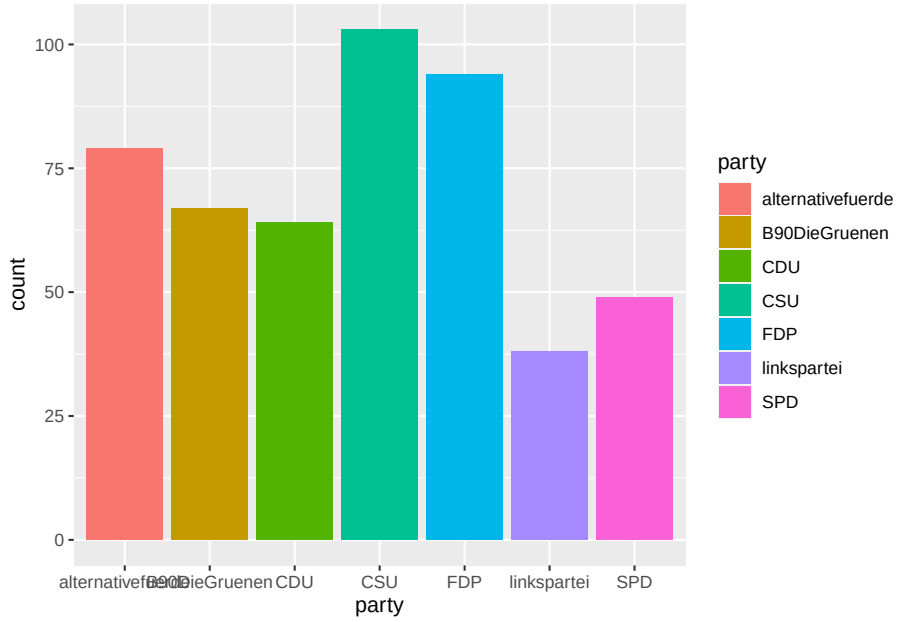
Um etwa ein einfaches Säulendiagramm für die Anzahl der Posts pro Partei zu erstellen, definieren wir `party` als `x`-Aesthetic und fügen ein `geom_bar()` hinzu:

```
facebook_europawahl %>%
  ggplot(aes(x = party)) +
  geom_bar()
```



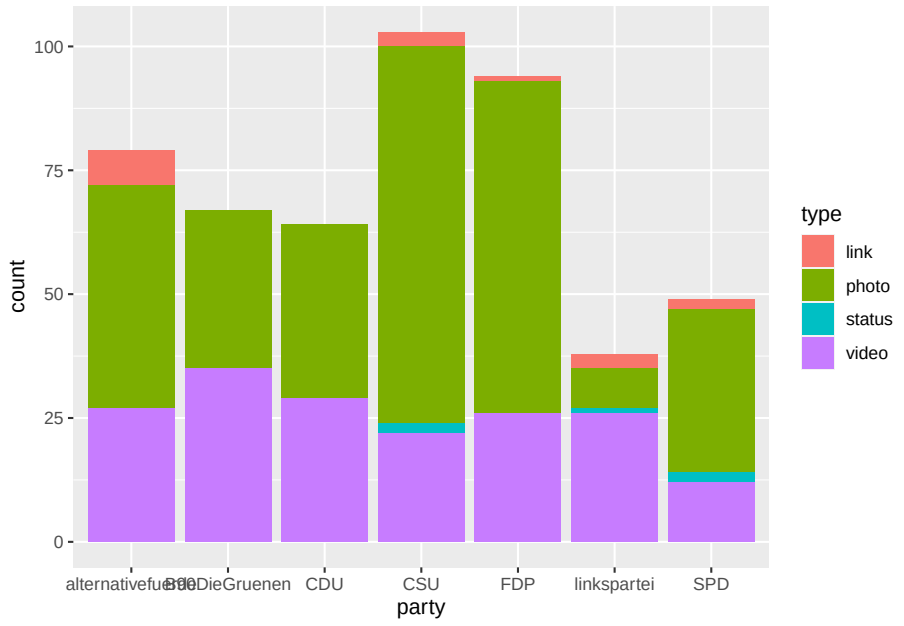
Wollen wir die Balken zusätzlich je nach Partei unterschiedlich einfärben, müssen wir wie im obigen tagesthemen-Beispiel `party` doppelt kodieren – sowohl als `x`, als auch als `fill` (die Füllfarbe der Balken):

```
facebook_europawahl %>%
  ggplot(aes(x = party, fill = party)) +
  geom_bar()
```



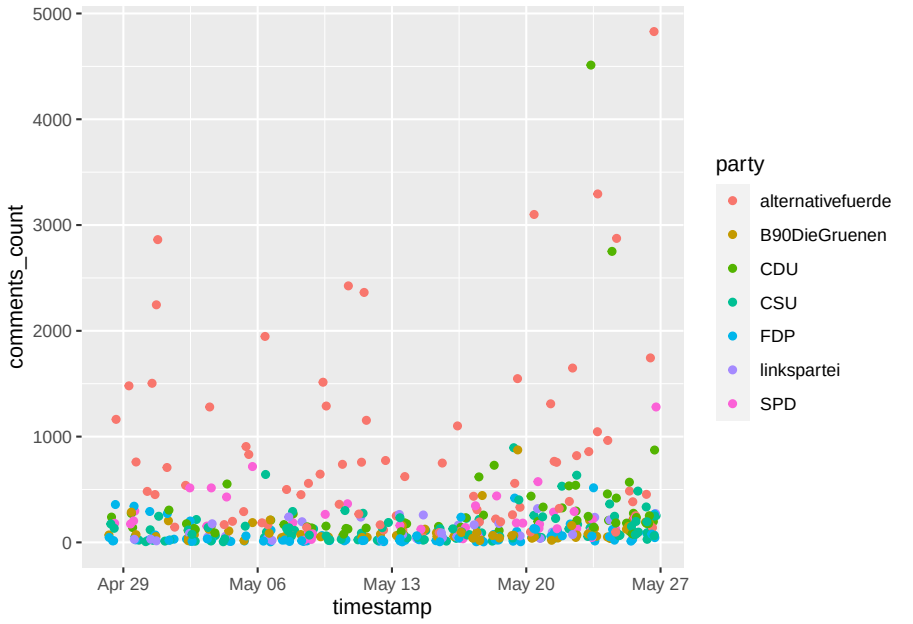
Ändern wir hingegen die Kodierung von `fill` in `type` (Art des Facebook-Posts), so haben wir diese Information auch im Balkendiagramm untergebracht:

```
facebook_europawahl %>%
  ggplot(aes(x = party, fill = type)) +
  geom_bar()
```



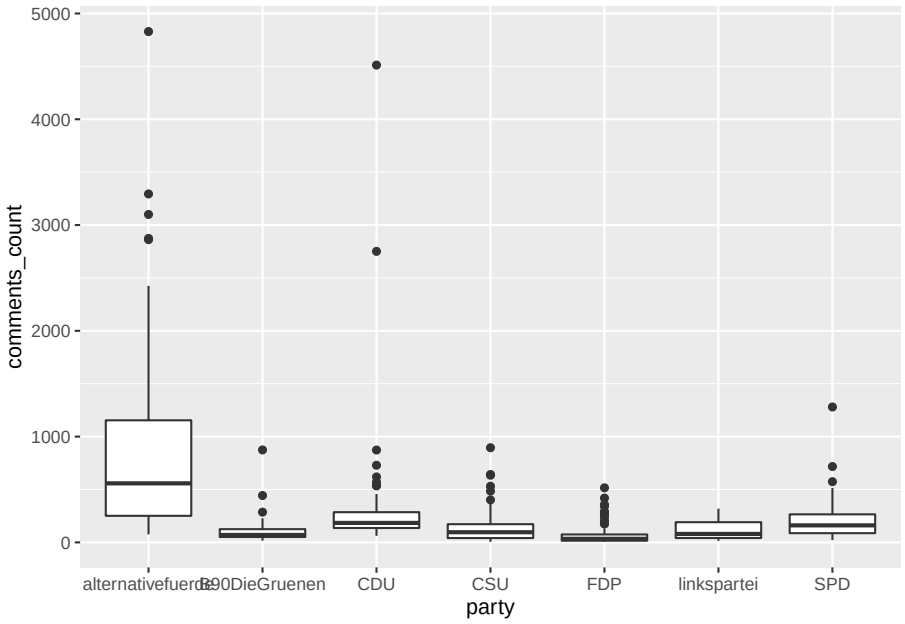
Im zweiten Beispiel möchten wir die Anzahl der Kommentare im Zeitverlauf und farblich nach Partei getrennt darstellen. Dazu bieten sich z. B. Punkte an:

```
facebook_europawahl %>%
  ggplot(aes(x = timestamp, y = comments_count, color = party)) +
  geom_point()
```



Oder wir stellen je Partei die Verteilung der Kommentaranzahl als Boxplot dar:

```
facebook_europawahl %>%
  ggplot(aes(x = party, y = comments_count)) +
  geom_boxplot()
```

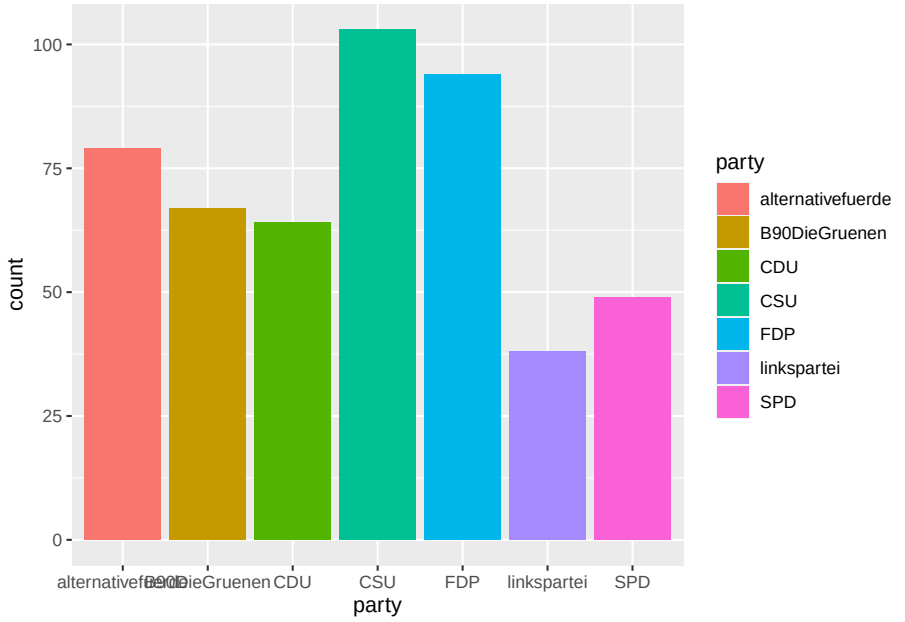


11.2.3 Skalierung anpassen

Jede Aesthetic, die wir in `ggplot()` definieren, resultiert in einer Skalierung der entsprechenden Variablen, die wir mittels `scale_`-Funktionen anpassen können. Alle `scale_`-Funktionen sind nach dem Schema `scale_[aesthetic]_[typ]` benannt. Mit `scale_y_continuous()` etwa passen wir kontinuierliche bzw. stetige, metrische Daten auf der Y-Achse an, während wir mittels `scale_fill_manual()` manuelle Werte für die Füllfarben vergeben können.

Schauen wir uns nochmals unser Säulendiagramm zur Anzahl der Posts an:

```
facebook_europawahl %>%
  ggplot(aes(x = party, fill = party)) +
  geom_bar()
```



Das sieht allgemein schon ganz ordentlich aus, wir möchten aber vielleicht noch ein paar weitere Anpassungen vornehmen:

- Die Y-Achse möchten wir in 10er-Schritten von 0 bis 110 einstellen.
- Die Anordnung der Parteien auf der X-Achse möchten wir – ähnlich wie im *tagesthemen*-Beispiel oben – entsprechend des Ergebnisses der Bundestagswahl vornehmen (stellen die beiden Unionsparteien aber nebeneinander).
- Und schließlich sollen die Parteien passende Farben erhalten.

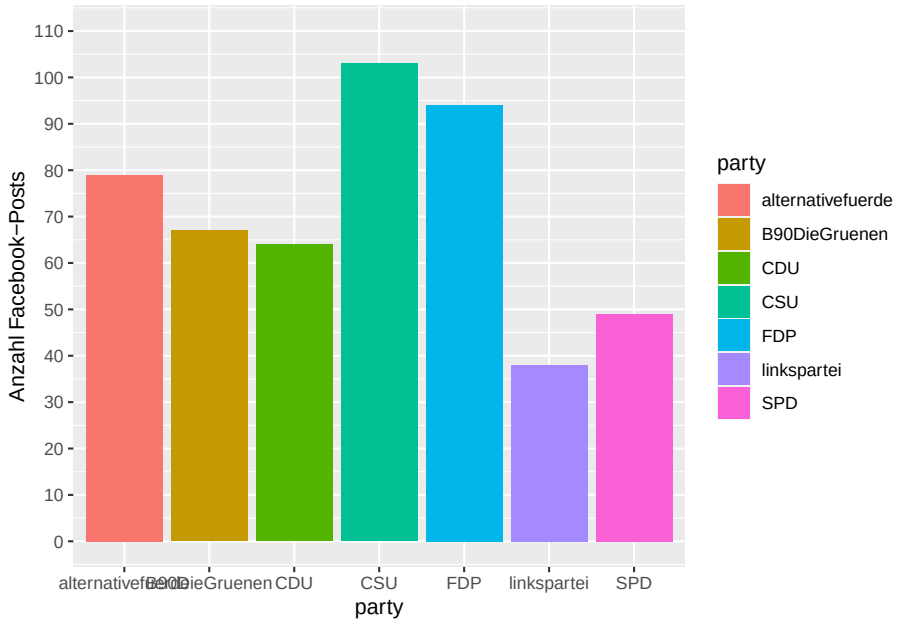
Zudem sollten die Beschriftungen der einzelnen Skalen etwas aussagekräftiger werden als nur `party` und `count`.

Wir beginnen mit der Y-Achse; es handelt sich um die Aesthetic `y` und eine numerische Skala. Die korrekte Skalenfunktion wäre daher `scale_y_continuous()`[3], die wir nun unserem Plot-Objekt per `+` hinzufügen können.

In dieser Funktion können wir nun Argumente definieren, wie die Skala verändert werden soll – zum Beispiel `name` für den Namen der Skala, `limits` für die untere und obere Begrenzung der Skala, und `breaks` für die Abschnitte der Skala:

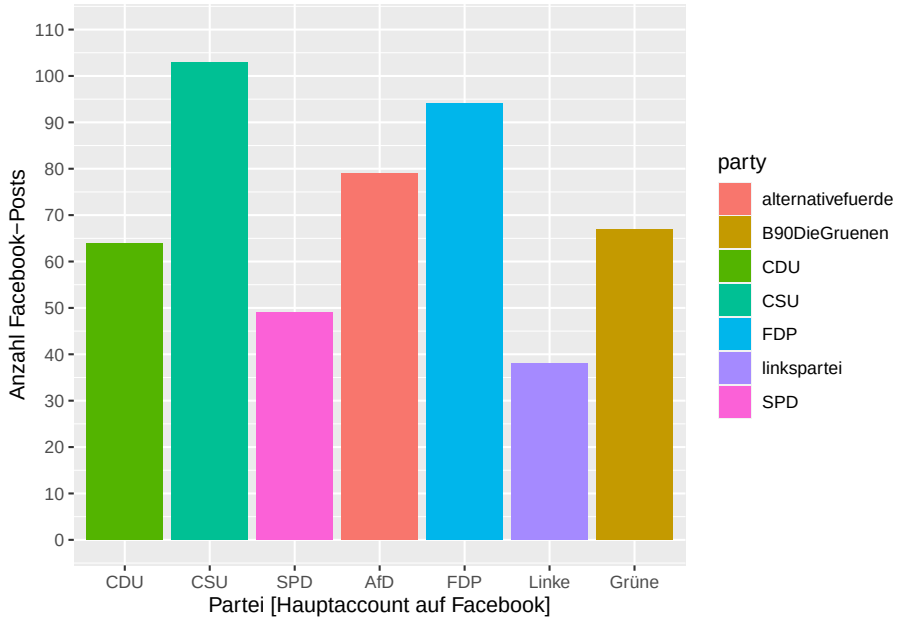
[3]Man könnte nun anmerken, dass es sich bei der Anzahl um eine diskrete Skala handelt, da diese nur ganzzahlige Werte annehmen kann; für numerische Daten empfehle ich jedoch, immer kontinuierliche Skalen zu verwenden.

```
facebook_europawahl %>%
  ggplot(aes(x = party, fill = party)) +
  geom_bar() +
  scale_y_continuous(name = "Anzahl Facebook-Posts", limits = c(0, 110), breaks = seq(0, 110, 10)
```



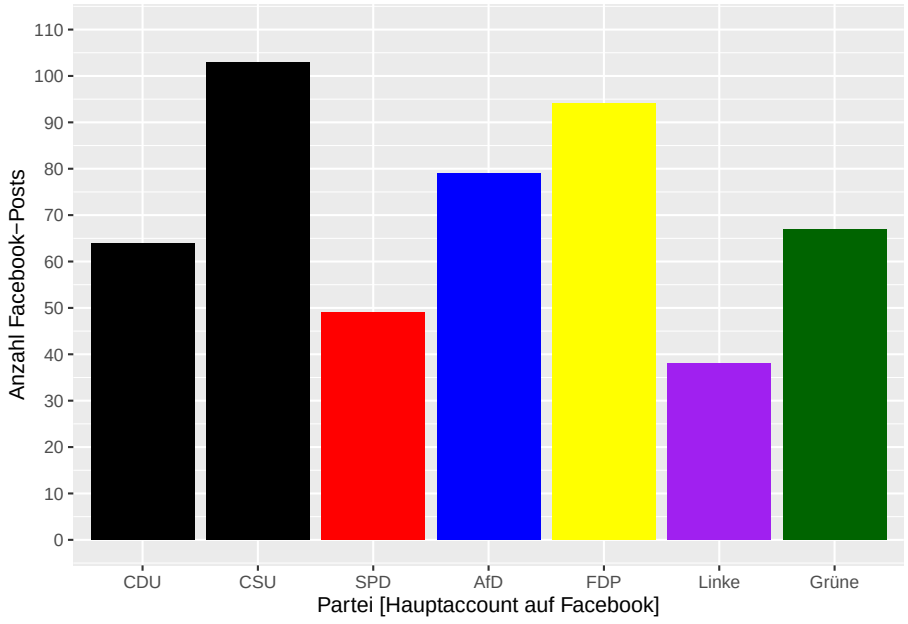
Und schon haben wir die Y-Achse angepasst. Weiter geht es mit der X-Achse. Hierbei handelt es sich um kategoriale, also auch diskrete Werte, die zugehörige Skalenfunktion ist also `scale_x_discrete()`. Auch hier übergeben wir einen neuen Namen und geben die Ausprägungen in der gewünschten Reihenfolge per `limits` an. Zudem passen wir mittels `labels` die Beschriftungen für die drei Facebook-Seiten `alternativefuerde`, `linkspartei` und `B90DieGruenen"` an:

```
facebook_europawahl %>%
  ggplot(aes(x = party, fill = party)) +
  geom_bar() +
  scale_y_continuous(name = "Anzahl Facebook-Posts", limits = c(0, 110), breaks = seq(0, 110, 10)
  scale_x_discrete(name = "Partei [Hauptaccount auf Facebook]",
                   limits = c("CDU", "CSU", "SPD", "alternativefuerde", "FDP", "linkspartei", "B9
                   labels = c("alternativefuerde" = "AfD", "linkspartei" = "Linke", "B90DieGruene
```



Schließlich passen wir die Parteifarben an. Diese werden über die Aesthetic `fill` (für Füllfarbe) definiert und sollen willkürlich festgelegt werden. Wir benötigen daher die Funktion `scale_fill_manual()`. Hier können wir mit dem Argument `values` einen Vektor mit den jeweiligen Wertausprägungen zugeordneten Farben übergeben. Zudem blenden wir mittels `guide = NULL` die Legende aus, da die Partei auch bereits in der X-Achse kodiert ist und die Legende somit redundant ist.

```
facebook_europawahl %>%
  ggplot(aes(x = party, fill = party)) +
  geom_bar() +
  scale_y_continuous(name = "Anzahl Facebook-Posts", limits = c(0, 110), breaks = seq(0
  scale_x_discrete(name = "Partei [Hauptaccount auf Facebook]",
                   limits = c("CDU", "CSU", "SPD", "alternativefuerde", "FDP", "linkspa
                   labels = c("alternativefuerde" = "AfD", "linkspartei" = "Linke", "B9
  scale_fill_manual(guide = NULL,
                    values = c("CDU" = "black",
                               "CSU" = "black",
                               "SPD" = "red",
                               "alternativefuerde" = "blue",
                               "FDP" = "yellow",
                               "linkspartei" = "purple",
                               "B90DieGruenen" = "darkgreen"))
```



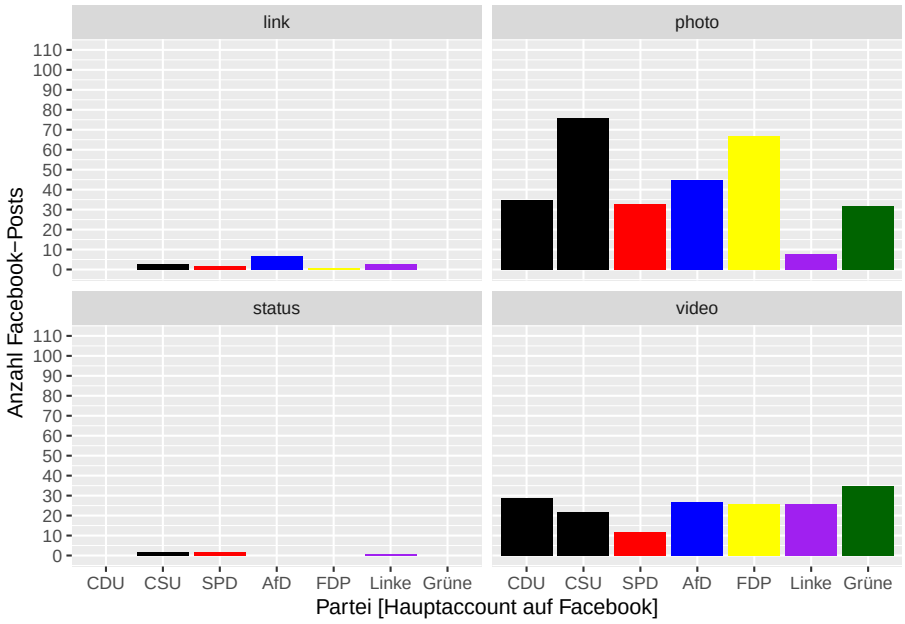
11.2.4 Plots in Facets aufteilen

Sobald wir mehr als drei oder vier Variablen kodieren möchten, wird eine einzelne Grafik oftmals unübersichtlich. Mittels *Facets* können wir den gleichen Plot auf Basis einer oder mehrerer Variablen erstellen. Die zugehörigen Funktionen lauten `facet_wrap()` (für eine *Facet*-Variable) sowie `facet_grid()` (für mehrere *Facet*-Variablen).

Wenn wir beispielsweise für obigen Plot noch den `type`, also die Art des Posts, unterscheiden möchten, könnten wir *Facets* für eben diese Variable über `facet_wrap()` anfordern:

```
facebook_europawahl %>%
  ggplot(aes(x = party, fill = party)) +
  geom_bar() +
  scale_y_continuous(name = "Anzahl Facebook-Posts", limits = c(0, 110), breaks = seq(0, 110, 10)
  scale_x_discrete(name = "Partei [Hauptaccount auf Facebook]",
                   limits = c("CDU", "CSU", "SPD", "alternativefuerde", "FDP", "linkspartei", "B9
                   labels = c("alternativefuerde" = "AfD", "linkspartei" = "Linke", "B90DieGruene
  scale_fill_manual(guide = NULL,
                    values = c("CDU" = "black",
                               "CSU" = "black",
                               "SPD" = "red",
                               "alternativefuerde" = "blue",
```

```
                                "FDP" = "yellow",
                                "linkspartei" = "purple",
                                "B90DieGruenen" = "darkgreen")) +
  facet_wrap(~ type, nrow = 2)
```



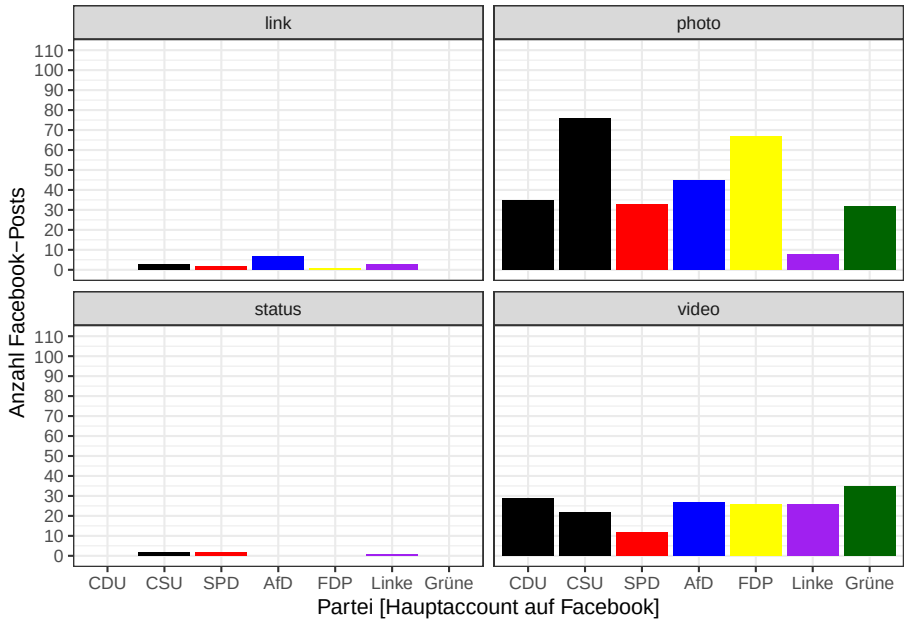
11.2.5 Themes anweden

Alle weiteren, rein ästhetischen Einstellungen werden durch `theme_`-Funktionen ergänzt. Hier bietet `ggplot2` schon einige Vorlagen, um schnell das aussehen des gesamten Plots zu verändern.

Um vom grauen Standardhintergrund der Plots wegzukommen, können wir beispielsweise das `theme_bw()` anwenden:

```
facebook_europawahl %>%
  ggplot(aes(x = party, fill = party)) +
  geom_bar() +
  scale_y_continuous(name = "Anzahl Facebook-Posts", limits = c(0, 110), breaks = seq(0
  scale_x_discrete(name = "Partei [Hauptaccount auf Facebook]",
                   limits = c("CDU", "CSU", "SPD", "alternativefuerde", "FDP", "linkspa
                   labels = c("alternativefuerde" = "AfD", "linkspartei" = "Linke", "B9
  scale_fill_manual(guide = NULL,
                    values = c("CDU" = "black",
```

```
                                "CSU" = "black",
                                "SPD" = "red",
                                "alternativefuerde" = "blue",
                                "FDP" = "yellow",
                                "linkspartei" = "purple",
                                "B90DieGruenen" = "darkgreen")) +
  facet_wrap(~ type, nrow = 2) +
  theme_bw()
```



11.2.6 Häufige Probleme und Hilfestellungen

Wir sehen: in so eine Grafik fließen bisweilen sehr viel Code und viel Tüftelei. Nicht nur gibt es eine Menge an `geom_`-, `scale_`- und zusätzliche Funktionen, diese können zudem über zusätzliche Argumente noch weiter angepasst werden. Die gute Nachricht ist aber, dass für die reine Datenexploration oft auch schon die Grundschritte *Aesthetics* und *Geometrics* ausreichen.

Ein Hauptproblem zu Beginn der Nutzung von `ggplot()` ist es, die Daten überhaupt in das richtige Format, das die Funktion erwartet zu bekommen: `ggplot()` erwartet *Long Data* (siehe Kapitel 10.1.2), wohingegen unsere Datensätze oft zunächst auf *Wide Data* ausgelegt sind.

Möchten wir beispielsweise pro Partei die Verteilung der jeweiligen Facebook-Metriken plotten, haben wir ein Problem: die jeweiligen Werte sind in drei

verschiedenen Variablen gespeichert (`comments_count`, `shares_count` und `reactions_count`); wir können in der Aesthetic `y`, auf der wir in der Regel abhängige numerische Werte abtragen, nur eine dieser drei Variablen kodieren.

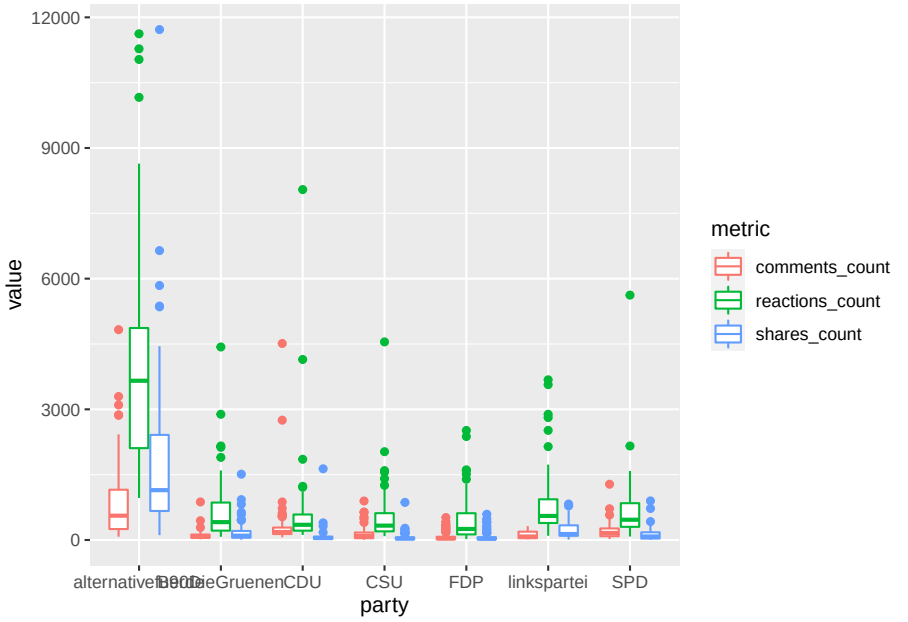
Hier kommt die Long-Tranformation ins Spiel, wie wir sie auch in Übungsaufgabe 10.1 vorgenommen haben – der so transformierte Datensatz hat die Werte der jeweiligen Facebook-Metriken in nur einer Variable gespeichert und betrachtet dafür die Art der Facebook-Metrik als eigene Variable:

```
fb_long <- facebook_europawahl %>%
  select(party, comments_count, shares_count, reactions_count) %>%
  pivot_longer(cols = c(comments_count, shares_count, reactions_count), names_to = "met

fb_long
```

```
## # A tibble: 1,482 x 3
##    party         metric          value
##    <chr>         <chr>           <dbl>
##  1 B90DieGruenen comments_count     70
##  2 B90DieGruenen shares_count       28
##  3 B90DieGruenen reactions_count   215
##  4 FDP           comments_count     16
##  5 FDP           shares_count       9
##  6 FDP           reactions_count   262
##  7 CDU           comments_count    239
##  8 CDU           shares_count      136
##  9 CDU           reactions_count   398
## 10 SPD           comments_count    180
## # ... with 1,472 more rows
```

Und schon können wir die Verteilung der Facebook-Metriken in Abhängigkeit von Partei und Art der Metrik darstellen – etwa indem wir `party` in `x`, `value` in `y` und `metric` in `color` kodieren und dann pro Partei verschiedenenfarbige Boxplots plotten:

```
fb_long %>%
  ggplot(aes(x = party, y = value, color = metric)) +
  geom_boxplot()
```



Oft ist es also sinnvoll, ein Plot-Problem dadurch anzugehen, indem die Struktur des zu plottenden Datensatzes nochmals überdacht wird oder zu plottende Werte bereits vorab mittels `mutate()` angepasst oder mittels `summarize()` nach Gruppen zusammengefasst werden. Die Funktionen aus den Kapiteln 8 und 10 helfen dabei, Datensätze schnell in die richtige Struktur zu bringen.

Im Internet ist zudem Hilfe nicht fern. Besonders hilfreich sind hier sogenannte *Cookbooks*, die "Rezepte" für alle möglichen Visualisierungen bereitstellen. Zwei Empfehlungen:

- R Graphics Cookbook: sehr umfangreiche Ressource, die so ziemlich alles abdeckt
- BBC Visual and Data Journalism cookbook for R graphics: Cookbook der BBC für Infografiken im BBC-Stil

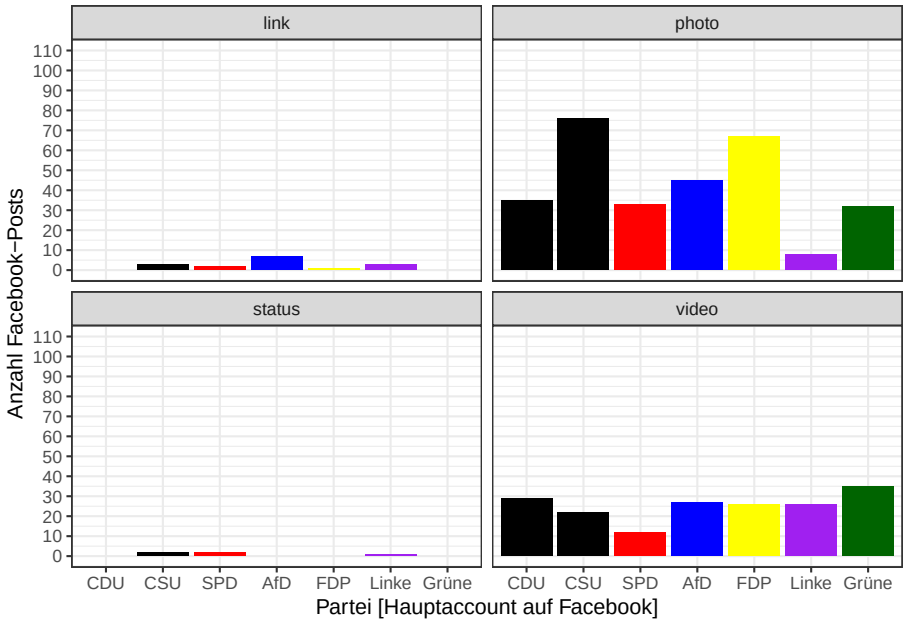
11.2.7 Plots speichern

Nach all der Arbeit möchten wir unsere Grafiken natürlich auch in einem passenden Format speichern. Hierfür bietet sich die Funktion `ggsave()` an, die ähnlich wie `saveRDS()` zum Speichern von R-Objekten funktioniert.

Zunächst weisen wir unsere schöne Visualisierung einem Objekt zu:

```
plot_anzahl <- facebook_europawahl %>%
  ggplot(aes(x = party, fill = party)) +
```

```
  geom_bar() +
  scale_y_continuous(name = "Anzahl Facebook-Posts", limits = c(0, 110), breaks = seq(0
  scale_x_discrete(name = "Partei [Hauptaccount auf Facebook]",
                   limits = c("CDU", "CSU", "SPD", "alternativefuerde", "FDP", "linkspa
                   labels = c("alternativefuerde" = "AfD", "linkspartei" = "Linke", "B9
  scale_fill_manual(guide = NULL,
                    values = c("CDU" = "black",
                               "CSU" = "black",
                               "SPD" = "red",
                               "alternativefuerde" = "blue",
                               "FDP" = "yellow",
                               "linkspartei" = "purple",
                               "B90DieGruenen" = "darkgreen")) +
  facet_wrap(~ type, nrow = 2) +
  theme_bw()

plot_anzahl
```



Nun können wir dieses Objekt mittels `ggsave()` speichern. Das erste Argument gibt hierbei den Speicherort an, das zweite das zu speichernde Plot-Objekt (also genau umgekehrt wie bei `saveRDS()` oder `write_csv()`). Mit weiteren Argumenten können wir die Ausgabedatei anpassen:

- `device`: Ausgabeformat, z. B. `"jpg"`, `"png"` oder `"pdf"`.

- `width`, `height` und `unit`: Breite und Höhe der Datei in einer definierten Maßeinheit für `unit`, entweder `"in"`, `"cm"` oder `"mm"`.
- `dpi`: Pixel-Auflösung (*d*ots *p*er *i*nch). Für Bildschirmausgabe `72`, für Druck `300`.

```
ggsave("figures/plot_anzahl.jpg", plot_anzahl,
       device = "jpg",
       dpi = 300)
```

11.3 Übungsaufgaben

Erstellen Sie für die folgenden Übungsaufgaben eine eigene Skriptdatei oder eine R-Markdown-Datei und speichern diese als `ue11_nachname.R` bzw. `ue11_nachname.Rmd` ab.

Laden Sie den Datensatz `facebook_europawahl.csv`, der schon aus den vorigen Übungen bekannt ist und filtern Sie diesen, sodass lediglich die im Bundestag vertretenen Parteien vorhanden sind.

Übungsaufgabe 11.1. Daten visualisieren I:

Stellen Sie die den Zusammenhang der Variablen `comments_count` und `shares_count` in einem Punktdiagramm dar.

Übungsaufgabe 11.2. Daten visualisieren II:

Fügen Sie dem Diagramm Informationen über die Partei, die für den jeweiligen Post verantwortlich ist (`party`), sowie die Art des Beitrags (`type`) hinzu.

Übungsaufgabe 11.3. Daten visualisieren III (optional):

Machen Sie das Diagramm "publikationsreif", indem Sie Skalen und Beschriftungen anpassen und es nach den eigenen ästhetischen Vorlieben verschönern.

Chapter 12

Arbeiten mit Textdaten

Insbesondere wenn wir zu Medieninhalten forschen, sind wir häufig mit Textdaten konfrontiert. In diesem Kapitel wird daher ein Überblick über die wichtigsten Funktionen zur Arbeit mit `character`-Variablen gegeben sowie das Konzept der *regulären Ausdrücke* eingeführt.

Bevor wir damit beginnnen, nochmals eine kurze Wiederholung zu `character`-Objekten in R sowie ein neues Konzept.

Zeichenketten (auch *Strings* genannt) werden in R (ebenso wie in nahezu allen Programmiersprachen) durch Anführungszeichen definiert:

```
string_objekt <- "Ich bin ein Text"
string_vektor <- c("eins", "zwei", "drei")
```

Dabei ist es unerheblich, ob einfache oder doppelte Anführungszeichen verwendet werden:

```
obst1 <- 'Apfel'
obst2 <- "Banane"
```

Somit können auch Zeichenketten gespeichert werden, die (das jeweils andere) Anführungszeichen enthalten:

```
zitat <- '"I love deadlines. I love the whooshing noise they make as they go by" - Douglas Adams'
```

Was machen wir, wenn beide Arten von Anführungszeichen in einem Textobjekt vorkommen sollen? In diesem Fall helfen uns Maskierungszeichen (*Escape Characters*) weiter, Zeichen, die einer Programmiersprache signalisieren, das nachfolgende Funktionszeichen als einfaches Zeichen ohne spezielle Funktion zu behandeln. In R (wie auch in den meisten anderen Sprachen) wird der Backslash \ als Maskierungszeichen verwendet:

```
maskierter_text <- "In diesem \"Text\" befinden sich weitere Anführungszeichen"
```

In der Konsolenausgabe zeigt R auch Maskierungszeichen an:

```
maskierter_text
```

```
## [1] "In diesem \"Text\" befinden sich weitere Anführungszeichen"
```

Möchten wir den *tatsächlichen* Inhalt eines Textobjekts sehen, können wir die Funktion `writeLines()` verwenden:

```
writeLines(maskierter_text)
```

```
## In diesem "Text" befinden sich weitere Anführungszeichen
```

Im Übrigen bedeutet dies auch, dass wir, wenn ein Backslash in einem String vorkommen soll, diesen durch einen vorangestellten Backslash maskieren müssen – wir signalisieren R also durch das Maskierungszeichen \, dass der nachfolgende \ *nicht* als Maskierungszeichen behandelt werden soll:

```
writeLines("\\")
```

```
## \
```

12.1 Einfache String-Operationen mit `stringr`

Das Tidyverse enthält das Package `stringr`, das auf den Umgang mit Strings spezalisiert ist. Alle relevanten Funktionen beginnen mit dem Suffix `str_`.[1]

Wir laden daher zunächst wieder das Tidyverse-Package.

```
library(tidyverse)
```

Alle `str_`-Funktionen sind vektorisiert, werden also auf jedes Element eines (Text-)Vektors angwendet. Dadurch kann man sie auch gut auf Textvariablen in Datensätzen bzw. Tibbles anwenden, um etwa mit der `mutate()`-Funktion bestehende Variablen zu verändern oder neue Variablen zu erzeugen. Zur Demonstration der Funktionen wird aber der Einfachkeit halber mit einfachen Textvektoren gearbeitet. Das erste Argument der `str_`-Funktionen ist immer ein Textvektor.

[1] Viele der Funktionen aus `stringr` sind unter anderem Namen auch bereits in der Basisversion von R enthalten. Wir nutzen dennoch vorrangig die Funktionen aus `stringr`, da diese neben der einheitlichen Benennung auch eine einheitlichere Syntax aufweisen sowie viele kleine Detailkorrekturen und Hilfsfeatures beinhalten, die man in den Basis-Äquivalenten vermisst.

```
obst <- c("Apfel", "Mango", "Kumquat")
```

12.1.1 Zeichenlänge bestimmen mit `str_length()`

Mit `str_length()` zählen wir die Anzahl an Zeichen in einem String:[2]

```
str_length(obst)
```

```
## [1] 5 5 7
```

12.1.2 Strings zusammenfügen mit `str_c()` und `str_glue()`

Mit `str_c()` lassen sich mehrere einzelne Strings zusammenfügen; dabei kann über das Argument `sep` eine Zeichenkette zum Trennen der Begriffe genutzt werden:[3]

```
str_c("Guten", "Tag", sep = " ")
```

```
## [1] "Guten Tag"
```

Wird ein Vektor mit mehr als einem Element übergeben, werden weitere Strings an jedes Vektorelement angehängt:

```
str_c(obst, "Mus", sep = "-")
```

```
## [1] "Apfel-Mus"   "Mango-Mus"   "Kumquat-Mus"
```

Soll stattdessen ein Vektor mit mehreren Strings in einen einzelnen String umgewandelt werden, muss über das Argument `collapse` eine Trennzeichenkette angegeben werden (wobei auch ein leerer String `""` übergeben werden kann):

```
str_c(obst, collapse = ", ")
```

```
## [1] "Apfel, Mango, Kumquat"
```

Für komplexere String-Verknüpfungen bietet sich die Funktion `str_glue()` an, mit der mittels geschweifter Klammern `{}` Objektnamen oder ganze R-Ausdrücke als Platzhalter definiert werden können:

[2] Die R-Basis-Version dieser Funktion lautet `nchar()`.
[3] In der Basis-Version: `paste()`

```
str_glue("{obst} hat {str_length(obst)} Buchstaben.")
```

```
## Apfel hat 5 Buchstaben.
## Mango hat 5 Buchstaben.
## Kumquat hat 7 Buchstaben.
```

12.1.3 Teile von Strings auswählen mit `str_sub()`

`str_sub()` (von Subset) kann genutzt werden, um einen Teil eines Strings auszuwählen, wobei die Start- und Endposition als zweites und drittes Argument übergeben werden:[4]

```
# Die ersten zwei Buchstaben auswählen
str_sub(obst, 1, 2)
```

```
## [1] "Ap" "Ma" "Ku"
```

Mit negativen Werten wird von hinten gezählt:

```
# Die letzten beiden Buchstaben auswählen
str_sub(obst, -2, -1)
```

```
## [1] "el" "go" "at"
```

Das kann natürlich auch kombiniert werden:

```
# Entferne den Anfangsbuchstaben (wähle alle Zeichen vom zweiten bis zum letzten aus):
str_sub(obst, 2, -1)
```

```
## [1] "pfel"   "ango"   "umquat"
```

12.1.4 Groß- und Kleinschreibung transformieren mit `str_to`-Funktionen

`str_to_lower()` und `str_to_upper()` wandeln Strings komplett in Klein- bzw. Großbuchstaben um:[5]

[4] In der Basis-Version: `strsub()`.
[5] In der Basis-Version: `tolower()` und `toupper()`.

```
str_to_lower(obst)
```

```
## [1] "apfel"   "mango"   "kumquat"
```

```
str_to_upper(obst)
```

```
## [1] "APFEL"   "MANGO"   "KUMQUAT"
```

Daneben gibt es noch `str_to_title()` (alle Anfangsbuchstaben groß) und `str_to_sentence()` (Anfangsbuchstaben des ersten Wortes groß, alles andere klein):

```
str_to_title("in the beginning the Universe was created. this has made a lot of people very angry
```

```
## [1] "In The Beginning The Universe Was Created. This Has Made A Lot Of People Very Angry And E
```

12.1.5 Überschüssigen Whitespace entfernen mit `str_trim()` und `str_squish`

`str_trim()` entfernt alle Leerzeichen am Anfang und am Ende eines Strings. `str_squish()` entfernt auch mehrfache Leerzeichen innerhalb eines Strings:

```
str_trim("   ein unsauberer  String   ")
```

```
## [1] "ein unsauberer  String"
```

```
str_squish("   ein unsauberer  String   ")
```

```
## [1] "ein unsauberer String"
```

12.2 Reguläre Ausdrücke

Gerade bei unbereinigten Daten ist es oft das Ziel, bestimmte Muster in Strings zu erkennen (wird beispielsweise eine bestimmte Person in einem Text genannt?) und/oder zu extrahieren. Hier kommen *reguläre Ausdrücke* (auf Englisch: *regular expressions* oder auch kurz *RegEx*) ins Spiel – Zeichenketten, die Muster in Zeichenketten formal beschreiben.

Reguläre Ausdrücke sehen für Laien oft aus wie Kauderwelsch und benötigen etwas Einübungszeit. Mit den Hilfsfunktionen `str_view()` (erste Übereinstimmung in einem String) und `str_view_all()` (alle Übereinstimmungen in einem String) können wir uns schnell anzeigen lassen, ob ein *RegEx*-Muster in einem String vorkommt oder nicht.

Figure 12.1: Illustration von @allison_horst: https://twitter.com/allison_horst

12.2.1 Exakte Übereinstimmungen

Im einfachsten Fall suchen wir nach einer exakten Zeichenkette – etwa um festzustellen, ob der Name `Trump` in aktuellen Schlagzeilen zur USA auftaucht:

```
schlagzeilen <- c(
  "Was hinter Trumps Obama-Obsession steckt",
  "Trump droht US-Staaten mit Finanzstopp",
  "USA verbieten Einreisen aus Brasilien"
)

str_view(schlagzeilen, "Trump")
```

12.2.2 Anker

Mit den Sonderzeichen `^` und `$` definieren wir, dass die zu suchende Zeichenkette sich am Anfang (`^`) bzw. am Ende (`$`) des Strings befindet. Um etwa nur Schlagzeilen zu finden, die mit `"Trump"` beginnen, suchen wir nach dem Muster `"^Trump"`:

```
str_view(schlagzeilen, "^Trump")
```

Analog findet `n$` nur in der letzten Schlagzeile eine Übereinstimmung, da nur diese auf `"n"` endet:

```
str_view(schlagzeilen, "n$")
```

Wie können wir dann nach dem Vorkommen eines Dollarsymbols suchen? Hier müssen wir – wie auch bei allen folgenden RegEx-Sonderzeichen – wieder auf das Maskierungszeichen \ zurückgreifen. Allerdings ist der Backslash \ in R ja bereits als Maskierungszeichen für Strings im Allgemeinen – und nicht als Maskierungszeichen für RegEx – definiert. Wir müssen den \ daher mit einem weiteren \ maskieren, damit R die RegEx-Zeichenfolge `\$` erkennt (ja, das ist Anfangs sehr verwirrend):

```
str_view("A$AP ROCKY", "\\$")
```

12.2.3 Mehrere Suchbegriffe

Mit dem uns schon bekannten `ODER`-Symbol `|` können wir nach dem Vorkommen mehrerer Zeichenketten suchen:

```
str_view(schlagzeilen, "Trump|USA|Vereinige Staaten")
```

12.2.4 Quantifier

Sogenannte Quantifier können dazu genutzt werden, um festzulegen, wie oft das zuvor angebene Muster in dem String vorkommen muss:

- `*`: 0-mal oder öfter (sinnvoll, um optionale Bestandteile zu definieren)
- `+`: 1-mal oder öfter _ `{n}`: Exakt `n`-mal
- `{n,}`: Mindestens `n`-mal
- `{n,m}`: Mindestens `n`-mal, maximal `m`-mal

```
pflanzen <- c("Maulbeere", "Brennnessel")
str_view(pflanzen, "e+")
```

```
str_view(pflanzen, "n{2}")
```

```
str_view(pflanzen, "n{1,3}")
```

Quantifier beziehen sich standardmäßig auf ein einzelnes, vorangestelltes Zeichen. Soll eine längere Zeichenkette mit einem Quantifier versehen werden, kann diese in runde Klammern `()` gestellt werden:

```
str_view(c("ein leckeres bonbon", "ein langer kassenbon"), "(bon){2}")
```

12.2.5 Spezielle Zeichentypen

Um bestimmte Zeichentypen zu *matchen*, stehen u. a. folgende Zeichen(folgen) zur Verfügung:

- `.`: Alle Zeichen
- `\d`: Alle Ziffern (und `\D` das Gegenteil, also *alles außer Ziffern*)
- `\w`: Alle alphanumerischen Zeichen (Klein- und Großbuchstaben, Ziffern, Unterstrich; `\W` das Gegenteil)
- `\s`: Whitespace (Leerzeichen, Umbrüche; `\S` das Gegenteil)

(Bei den drei letztgenannten muss in R der Backslash maskiert werden, also z. B. `"\\d"`).

```
str_view(c("Schneewittchen und die 7 Zwerge",
           "3 Haselnüsse für Aschenbrödel"),
         "\\d")
```

Zudem können zu matchende Zeichentypen durch eckige Klammern selbst definiert werden – `[abc]` beispielsweise matcht ein `a`, `b` oder `c`.

```
str_view(c("abc", "cde", "xyz"),
         "[abc]")
```

12.2.6 RegEx-Zeichen kombinieren

Natürlich können wir all diese Zeichen kombinieren, um komplexere Muster zu matchen. Mit dem Muster `"[\\w-]+\\s+\\d+[a-z]*"` erfassen wir beispielsweise typische deutsche Straßennamen mitsamt Hausnummern, die in Regel nach dem Muster "Straßenname Hausnummer" aufgebaut sind – das sieht auf den ersten Blick sehr undurchsichtig aus, lässt sich aber wie folgt aufschlüsseln:

- `\w` sucht nach allen alphanumerischen Zeichen; für R müssen wir den Backslash maskieren, schreiben also `\\w`.
- Der Bindestrich `-` ist in diesen Zeichen nicht enthalten, wir fügen diesen also noch manuell hinzu und umschließen beides in eckigen Klammern `[]`. `[\\w-]` sucht also nach allen alphanumerischen Zeichen und dem Bindestrich `-`.
- Wir geben nun an, dass wir diese Zeichen mindestens einmal vorfinden möchten, daher schließt an dieses Suchmuster das `+` an. Damit dürften wir so ziemlich alle deutschen Straßennamen abdecken.

- Typischerweise folgt auf den Straßennamen ein Leerzeichen und dann die Hausnummer. Leerzeichen und andere Whitespace-Zeichen matchen wir mit `\s`, wobei auch hier ein weiterer `\` zum Maskieren benötigt wird. Damit wir auch Fälle erfassen, in denen (aus Versehen) zwei oder mehr Leerzeichen zwischen Straßenname und Hausnummer stehen, schließen wir erneut den Quantifier `+` an – wir suchen also nach mindestens einem Leerzeichen.
- Die Hausnummer besteht in der Regel aus einer oder mehrerer Ziffen; dies matchen wir mittels `\\d+`.
- Manche Hausnummern haben zusätzlich noch einen Kleinbuchstaben, um unterschiedliche Gebäudeeinheiten zu unterscheiden. Mittels `[a-z]` legen wir fest, dass alle Kleinbuchstaben von `a` bis `z` gesucht werden sollen. Dieses Muster ist jedoch optional, da nicht alle Hausnummern darauf enden. Wir fügen also hier ein `*` an, das die vorangestellte Zeichenfolge *0-mal* oder öfter matcht.

```
adressen <- c(
  "Oettingenstraße 67",
  "Geschwister-Scholl-Platz  1",
  "Schellingstraße 3a"
)

str_view(adressen, "[\\w-]+\\s+\\d+[a-z]*")
```

Ja, das sieht auf den ersten Blick aus wie Kauderwelsch und ist zu Beginn nicht sonderlich intuitiv; man gewöhnt sich aber daran. Und: für viele Anwendungsfälle (z. B. URLs oder Twitter-IDs aus einem Text extrahieren) findet man online schnell passende RegEx-Phrasen, die dann nur noch auf R angepasst (Maskierungszeichen!) und validiert werden müssen.

12.3 RegEx und `stringr`

Schauen wir uns nach diesem eher abstrakten Überblick einige praktische Anwendungsbeispiele an, die Funktionen auf dem `stringr`-Package verwenden.

12.3.1 Muster finden mit `str_detect()`

`str_detect()` prüft für einen Textvektor, ob das angegebene Muster darin vorkommt, und gibt dies als logischen Vektor zurück:

```
schlagzeilen <- c(
  "Nach Feier: 140 Personen in Corona-Quarantäne",
```

```
  "Wo die Auflagen gelockert werden",
  "Corona-Lockerungen: Das ist seit Montag anders"
)

str_detect(schlagzeilen, "Corona")
```

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

Das kann z. B. auch dazu genutzt werden, schnell einen Datensatz zu filtern. Nehmen wir beispielsweise den Beispiel-Datensatz `starwars`, der zum Tidyverse-Package gehört und entsprechend mit `starwars` aufgerufen werden kann.

```
starwars
```

```
## # A tibble: 87 x 14
##    name            height  mass hair_color    skin_color  eye_color birth_year sex
##    <chr>            <int> <dbl> <chr>         <chr>       <chr>           <dbl> <chr
##  1 Luke Skywalker     172    77 blond         fair        blue             19   male
##  2 C-3PO              167    75 <NA>          gold        yellow          112   none
##  3 R2-D2               96    32 <NA>          white, blue red              33   none
##  4 Darth Vader        202   136 none          white       yellow           41.9 male
##  5 Leia Organa        150    49 brown         light       brown            19   fema
##  6 Owen Lars          178   120 brown, grey   light       blue             52   male
##  7 Beru Whitesun ~    165    75 brown         light       blue             47   fema
##  8 R5-D4               97    32 <NA>          white, red  red              NA   none
##  9 Biggs Darkligh~    183    84 black         light       brown            24   male
## 10 Obi-Wan Kenobi     182    77 auburn, white fair        blue-gray        57   male
## # ... with 77 more rows
```

Um schnell die Star-Wars-Figuren auszuwählen, deren Name eine Ziffer beinhaltet (C-3PO, R2D2 usw.), können wir `filter()` und `str_detect()` kombinieren:

```
starwars %>%
  filter(str_detect(name, "\\d"))
```

```
## # A tibble: 6 x 14
##   name   height  mass hair_color skin_color  eye_color birth_year sex   gender    ho
##   <chr>   <int> <dbl> <chr>      <chr>       <chr>           <dbl> <chr> <chr>     <c
## 1 C-3PO     167    75 <NA>       gold        yellow            112 none  masculine Ta
## 2 R2-D2      96    32 <NA>       white, blue red                33 none  masculine Na
## 3 R5-D4      97    32 <NA>       white, red  red                NA none  masculine Ta
## 4 IG-88     200   140 none       metal       red                15 none  masculine <N
## 5 R4-P17     96    NA none       silver, red red, blue          NA none  feminine  <N
## 6 BB8        NA    NA none       none        black              NA none  masculine <N
```

12.3.2 Muster zählen mit `str_count()`

`str_count()` funktioniert analog zu `str_detect()`, nur dass kein logischer Vektor, sondern ein numerischer Vektor zurückgegeben wird, in dem gezählt wird, wie häufig das gesuchte Muster in den jeweiligen Strings vorkommt:

```
str_count(c("Apfel", "Banane", "Mango"),
          "n")
```

```
## [1] 0 2 1
```

12.3.3 Muster extrahieren mit `str_extract()`

Neben dem Prüfen, ob ein bestimmtes Muster vorhanden ist, zählt das Extrahieren dieser Muster zu den häufigsten Anwendungsfällen. Das ist die Aufgabe der Funktion `str_extract()`:[6]

```
str_extract(c("Herr Müller", "Frau Meier"), "Herr|Frau")
```

```
## [1] "Herr" "Frau"
```

Für ein etwas komplexeres Beispiel nehmen wir einmal an, wir finden folgenden Datensatz vor:

```
test_ergebnis <- tibble(kandidat = c("A", "B", "C", "D", "E", "F"),
                        punkte = c("1", "2", "0.32", ".555", "-22", "33 Punkte"))
test_ergebnis
```

```
## # A tibble: 6 x 2
##   kandidat punkte
##   <chr>    <chr>
## 1 A        1
## 2 B        2
## 3 C        0.32
## 4 D        .555
## 5 E        -22
## 6 F        33 Punkte
```

[6] `str_extract()` extrahiert dabei immer die erste Übereinstimmung. Sollen alle Übereinstimmungen mit dem Muster aus einem String extrahier werden, kann die Funktion `str_extract_all()` verwendet werden, deren Ausgabe aber entsprechend etwas unhandlicher ist.

Für weitere Analysen wäre es natürlich deutlich angenehmer, wenn wir mit den Punktewerten auch rechnen könnten. Wir können diese mit einer RegEx extrahieren. Dafür müssen wir formalisieren, wie Punktezahlen in diesem Datensatz aufgebaut sein können:

- zunächst steht ein optionales `-` für negative Werte; zur Erinnerung: mit einem Asterisk `*` legen wir fest, dass das vorangegange Zeichen mindestens 0-mal, d.h. optional vorkommen soll. Wir beginnen unsere RegEx-Zeichenfolge daher mit `"-*"`.
- Dann folgt, ebenfalls optional, eine oder mehrere Ziffern. Wir benötigen also das (maskierte) Sonderzeichen `\\d` für Ziffern sowie erneut ein Asterisk `*`; unsere RegEx-Folge lautet nun `"-*\\d*`.
- Nun folgt, erneut optional, ein Punkt `.` als Dezimaltrennzeichen. Da es sich bei dem Punkt um ein RegEx-Sonderzeichen handelt, müssen wir dieses doppelt maskieren[7]: `\\.`. Auch dieser Punkt ist optional, wir hängen also erneut ein `*` an; unsere RegEx-Folge lautet nun `"-*\\d*\\.*`
- Schließlich und zwingend kommt mindestens eine Ziffer in Zahlen vor. Wir benötigen also erneut das Sonderzeichen für Ziffern `\\d` und legen mit dem Sonderzeichen `+` fest, dass dieses mindestens einmal oder öfter vorkommen muss. Unsere finale RegEx-Folge lautet `"-*\\d*\\.*\\d+"`.

Da `str_extract()` immer Text extrahiert, wandeln wir das Ergebnis noch in den Typ `numeric` um:

```
test_ergebnis %>%
  mutate(punkte_numerisch = as.numeric(str_extract(punkte, "-*\\d*\\.*\\d+")))
```

```
## # A tibble: 6 x 3
##   kandidat punkte    punkte_numerisch
##   <chr>    <chr>                <dbl>
## 1 A        1                        1
## 2 B        2                        2
## 3 C        0.32                     0.32
## 4 D        .555                     0.555
## 5 E        -22                    -22
## 6 F        33 Punkte               33
```

12.3.4 Muster ersetzen mit `str_replace` (bzw. `str_replace_all()`):

Der dritte häufige Anwendungsfall ist, dass Muster ersetzt werden sollen. Dafür können `str_replace()` (ersetzt erste Übereinstimmung) und

[7] Einmal, damit RegEx merkt, dass wir den Punkt nicht als RegEx-Sonderzeichen behandeln möchten, und einmal, damit R den Maskierungs-Backslash nicht als R-Maskierungszeichen erkennt.

`str_replace_all()` (ersetzt alle Übereinstimmungen) verwendet werden, wobei zunächst das zu ersetzende Muster, dann das *Replacement* angegeben wird. Wurde beispielsweise das Dezimaltrennzeichen fälschlicherweise als Komma, nicht als Punkt eingelesen:

```
str_replace_all(c("1,2", "2,3", "3,66"), ",", ".")
```

```
## [1] "1.2"  "2.3"  "3.66"
```

Bei mehreren Mustern und zugehörigen Replacements können wir einen benannten Vektor übergeben:

```
str_replace_all(c("Hr Müller", "Fr Meier"),
                c("Hr" = "Herr",
                  "Fr" = "Frau"))
```

```
## [1] "Herr Müller" "Frau Meier"
```

Eine praktische Übersicht über alle relevanten `stringr`-Funktionen sowie RegEx in R bietet dieses Cheatsheat.

12.4 Übungsaufgaben

Erstellen Sie für die folgenden Übungsaufgaben eine eigene Skriptdatei oder eine R-Markdown-Datei und speichern diese als `ue12_nachname.R` bzw. `ue12_nachname.Rmd` ab.

Übungsaufgabe 12.1. Arbeiten mit Textdaten I:

Fügen Sie diesem Datensatz zu einem Experiment mittels `mutate()` eine neue Spalte hinzu, die lediglich die Gruppenkennung (`A`, `B` oder `C`) enthält:

```
experiment <- tibble(experimentalgruppe = c("Gruppe A", "Gruppe B", "Gruppe A", "Gruppe C"))
experiment
```

```
## # A tibble: 4 x 1
##   experimentalgruppe
##   <chr>
## 1 Gruppe A
## 2 Gruppe B
## 3 Gruppe A
## 4 Gruppe C
```

Übungsaufgabe 12.2. Arbeiten mit Textdaten II:

In der Internet Movie Database verfügt jeder Film über eine eindeutige ID, die nach dem Schema `"tt[7 Ziffern]"` aufgebaut ist. Extrahieren Sie diese ID aus den folgenden URLS:

```
imdb_urls <- c(
  "https://www.imdb.com/title/tt6751668/?ref_=hm_fanfav_tt_4_pd_fp1",
  "https://www.imdb.com/title/tt0260991/",
  "www.imdb.com/title/tt7282468/reviews",
  "https://m.imdb.com/title/tt4768776/"
)
```

Übungsaufgabe 12.3. Arbeiten mit Textdaten III:

Käpseles-Aufgabe (optional)

`str_match` funktioniert ähnlich zu `str_extract()`, nur dass wir durch runde Klammern `()` Gruppen in einem RegEx-Muster definieren können, die dann getrennt extrahiert werden.

Lesen Sie sich die Dokumentation zu `str_match()` durch und. Extrahieren Sie dann aus folgendem Vektor getrennt folgende Adressbestandteile:

- Straßenname
- Hausnummer
- Postleitzahl
- Stadt
- Land

```
adressen = c(
    "Platz der Republik 1, D-11011 Berlin",
    "Dr.-Karl-Renner-Ring 3, A-1017 Wien",
    "Bundesplatz 3, CH-3005 Bern"
  )
```

Chapter 13

Iterationen im Tidyverse

Zum Abschluss der Einführung in das Tidyverse setzen wir uns erneut mit Iterationen auseinander, die wir zuvor schon in Kapitel 4.2 kennengelernt haben.

```
library(tidyverse)
```

Zunächst eine kurze Wiederholung: um Code mehrfach auszuführen, können wir `for`-Loops schreiben:

```
for (element in vektor) {
  # Body: Code, der ausgeführt wird
}
```

Um z. B. die Zahlen von 1 bis 5 in die Konsole zu schreiben, iterieren wir über einen Vektor, der eben diese Zahlen enthält:

```
for (i in 1:5) {
  print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

Das ist flexibel einsetzbar, hat aber den einen Nachteil, dass wir häufig über jedes Element eines Objekts iterieren möchten, wir dies aber explizit im Code angeben müssen. Nehmen wir beispielsweise den Beispiel-Datensatz `mtcars`, der lediglich numerische Variablen enthält:

```
mtcars
```

```
##                     mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4          21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag      21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710         22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive     21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout  18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
## Valiant            18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Duster 360         14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
## Merc 240D          24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Merc 230           22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Merc 280           19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
## Merc 280C          17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
## Merc 450SE         16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
## Merc 450SL         17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
## Merc 450SLC        15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
## Cadillac Fleetwood 10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
## Lincoln Continental 10.4  8 460.0 215 3.00 5.424 17.82  0  0    3    4
## Chrysler Imperial  14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
## Fiat 128           32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
## Honda Civic        30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
## Toyota Corolla     33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
## Toyota Corona      21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
## Dodge Challenger   15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
## AMC Javelin        15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
## Camaro Z28         13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
## Pontiac Firebird   19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
## Fiat X1-9          27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
## Porsche 914-2      26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
## Lotus Europa       30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
## Ford Pantera L     15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
## Ferrari Dino       19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
## Maserati Bora      15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
## Volvo 142E         21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

Möchten wir nun den Mittelwert jeder Variablen berechnen, müssen wir in einem `for`-Loop explizit definieren, dass wir über jede Spalte des Vektors iterieren möchten. Eine Möglichkeit wäre die Funktion `seq_along()` zu nutzen, die einen Vektor mit den Spaltenindizes eines Datensatzes erstellt:

```
mittelwerte <- c()

for (i in seq_along(mtcars)) {
  mittelwerte <- c(mittelwerte, mean(mtcars[[i]]))
```

```
}

mittelwerte
```

```
##  [1]  20.090625   6.187500 230.721875 146.687500   3.596563   3.217250  17.848750   0.437500
```

Hier kommen im Tidyverse die sogenannten `map_`-Funktionen ins Spiel. Diese wenden einen Funktion automatisch auf alle Elemente eines Objekts an – bei einem Vektor auf Elemente dieses Vektors, bei einem Dataframe bzw. Tibble auf alle Spalten. Zudem wird mit dem Funktionsaufruf bereits das Ausgabe-Format definiert:

- `map()`: Erzeugt eine Liste
- `map_dbl()`: Erzeugt einen numerischen (`double`) Vektor
- `map_chr()`: Erzeugt einen `character`-Vektor
- `map_lgl()`: Erzeugt einen `logical`-Vektor
- `map_dfr()`: Erzeugt einen Datensatz zeilenweise
- `map_dfc()`: Erzeugt einen Datensatz spaltenweise

Beim Aufruf werden immer zunächst das Objekt, über das iteriert werden soll, und dann die zu verwendende Funktion (ohne Klammern) genannt. Um etwa die Mittelwerte aller Variablen im Datensatz `mtcars` zu berechnen und in einem numerischen Vektor zu speichern, können wir den obigen `for`-Loop-Code abkürzen:

```
mittelwerte <- map_dbl(mtcars, mean)
mittelwerte
```

```
##        mpg        cyl       disp         hp       drat         wt       qsec         vs
##  20.090625   6.187500 230.721875 146.687500   3.596563   3.217250  17.848750   0.437500   0.40
```

Wenn wir als Ausgabeformat einen Datensatz haben wollen, rufen wir einfach die zugehörige `map_`-Funktion auf:

```
map_dfr(mtcars, mean)
```

```
## # A tibble: 1 x 11
##     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1  20.1  6.19  231.  147.  3.60  3.22  17.8 0.438 0.406  3.69  2.81
```

Weitere Funktionsargumente können nach dem zweiten Argument übergeben werden:

```
map_dfr(mtcars, mean, na.rm = TRUE)
```

```
## # A tibble: 1 x 11
##     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1  20.1  6.19  231.  147.  3.60  3.22  17.8 0.438 0.406  3.69  2.81
```

Und da immer zunächst das Objekt, über das iteriert werden soll, genannt wird, können wir `map_`-Funktionen auch einfach in Pipes integrieren:

```
iris %>%
  select(-Species) %>%
  map_dfr(mean)
```

```
## # A tibble: 1 x 4
##   Sepal.Length Sepal.Width Petal.Length Petal.Width
##          <dbl>       <dbl>        <dbl>       <dbl>
## 1         5.84        3.06         3.76        1.20
```

In den kommenden Kapiteln werden wir hierfür einige Anwendungsbeispiele kennenlernen.

Part III

Daten aus dem Web

Chapter 14

Einführung und Terminologie

Die schiere Masse und stete Verfügbarkeit digitaler Kommunikationsdaten – beispielsweise digitale Verhaltensspuren der Nutzer*innen von sozialen Medien oder umfassende Textarchive von Nachrichtenmedien – machen das Web zu einer unerschöpflichen Datenquelle für die Sozialwissenschaften im Allgemeinen und die Kommunikationswissenschaft im Speziellen. Wir werden uns daher nun damit auseinandersetzen, wie wir Zugang zu diesen Daten erhalten und wie wir diese Daten abrufen können. Die beiden wichtigsten Zugangswege dafür sind:

1. *Web Scraping*: Hier rufen wir Webseiten auf und extrahieren die für uns relevanten Daten. Dieser Zugangsweg eignet sich besonders für statische Webseiten wie z. b. Nachrichtenwebsites[1], stößt jedoch auf Probleme bei dynamischen Webinhalten.
2. *(Web-)APIs* (*A*pplication *P*rogramming *I*nterface): Hier nutzen wir Programmierschnittstellen, die für den standardisierten Austausch von Daten entwickelt wurden, also auf standardisierte Anfragen ebenso standardisiert Daten zurückgeben.

Bevor wir diese Zugangswege in R ausprobieren, zunächst eine kurze Einführung in Terminologie und Funktionsweise *des Internets*.

14.1 HTTP, Anfragemethoden und Statuscodes

Der Daten- und Nachrichtenaustausch im Internet wird über Protokolle geregelt, die festlegen, wie die Kommunikation zwischen einem *Client* (z. B. einem Web-

[1] Natürlich sind auch die Inhalte auf diesen Webseiten veränderlich. Statisch meint hier, dass die Inhalte nicht für jede*n Nutzer*in eigens "erzeugt" werden.

browser auf unserem Computer) und einem *Server* abläuft. Das wichtigste ist HTTP (*H*yper*t*ext *T*ransfer *P*rotocol), das uns allen von der URL-Eingabe von Webseiten im Browser bekannt ist und vorrangig für Webseiten verwendet wird.

Die Kommunikation zwischen Client und Server ist dabei eine Abfolge von Anfragen (*Requests*) und Antworten (*Responses*), wobei sowohl Requests und Responses jeweils in einen *Header* (Metainformationen über die Nachricht, z. B. der Inhaltstyp) und einen *Body* (die eigentliche Nachricht) unterteilt werden können. Wir können uns das in etwa wie einen Paketversand vorstellen: der Header ist dann vergleichbar zum Sendeschein, enthält also Informationen über den Inhalt und Zustand des Pakets (z. B. auch, ob dieses korrekt zugestellt werden konnte oder warum dies nicht der Fall war), der Body ist das zugehörige Paket mit dem Inhalt, der uns eigentlich interessiert.

Für Anfragen stehen verschiedene HTTP-Anfragemethoden zur Verfügung, von denen die gebräuchlichste *GET* heißt, mit der ein Client eine auf dem Server liegende Ressource anfordert. Mit der Anfragemethode *POST* überträgt ein Client Daten an einen Server übertragen, zum Beispiel wenn wir ein Formular auf einer Webseite ausfüllen.

Wenn wir etwa in einen Browser die *URL* `https://www.ifkw.uni-muenchen.de/index.html` eingeben, dann schickt unser Browser als Client einen *GET*-Request an den Server `ifkw.uni-muenchen.de/`, die Ressource `index.html` zurückzusenden. War die Anfrage erfolgreich, enthält die Antwort im Header den Statuscode `200 (OK)` (= alles in Ordnung) und im Body eben die Datei `index.html`, die dann von unserem Browser dargestellt wird (siehe auch nächstes Unterkapitel). War die Anfrage nicht erfolgreich, enthält der Header der Antwort ebenso Informationen darüber, was schiefgelaufen ist, etwa den Statuscode `404 (Not found)` (Ressource nicht gefunden, wir weil wir uns beispielsweise bei der URL vertippt haben) oder den Statuscode `403 (Forbidden)`, weil uns die Zugangsberechtigung zu dieser Ressource fehlt.

Die Kommunikation von Browsern mit Webservern mag das naheliegendste Beispiel dafür sein, aber letztlich läuft nahezu jede Kommunikation von Programmen über das Internet so ab, egal welches Programm wir dafür verwenden. Entsprechend können wir dank bestimmter Packages auch mit R solche HTTP-Anfragen stellen und die Antworten verarbeiten – aber dazu in Kürze mehr.

14.2 Webseiten und HTML

Das Grundgerüst von Webseiten ist *HTML* (*H*yper*t*ext *M*arkup *L*anguage), eine Auszeichnungssprache[2], mit der Textelemente (also Zeichen, Wörter oder ganze Abschnitte) mit Eigenschaften versehen und so für die Darstellung als Webseite formatiert bzw. mit zusätzlichen Funktionen (z. B. Links) ausgestattet werden:

[2]Mit Markdown (siehe Kapitel 6.3.2) haben wir bereits eine andere Auszeichnungssprache kennengelernt.

```
<html>
<body>
  <div class="main">
    <h1>Eine Überschrift</h1>
    <p>Ein Absatz.</p>
    <p>Noch ein Absatz, in dem Worte <strong>fettgedruckt</strong> sowie <a href="link.html">verl
  </div>
</body>
</html>
```

Webseiten sind also zunächst einmal Textdokumente, die mittels Auszeichung in HTML von Browsern als Webseiten interpretiert und dargestellt werden. Hinzu kommen Skripte in der Programmiersprache JavaScript, mit denen vor allem interaktive Elemente erzeugt werden, sogenannte Stylesheets in CSS (dazu gleich mehr) und etwaige Multimediadateien, z. B. Bilder, die via HTML eingebunden werden:

```
<body id="top">
<div id="page" class="page page-startseite page-1656414">
<div id="home" class="home kopfbild">
<a href="index.html" title="Institut für Kommunikationswissenschaft und Medienforschung (IFKW)"> <img src="//cms-static.uni-
muenchen.de/default/lmu/img/blank.png" class="home-img" alt="Institut für Kommunikationswissenschaft und Medienforschung (IFKW)" title="Institut für
Kommunikationswissenschaft und Medienforschung (IFKW)"> </a>
</div>
<div class="logo-print"> <img border="0" src="//cms-static.uni-muenchen.de/default/lmu/img/header-print.gif" alt="print"> </div>
<hr class="g-hidden">
<div id="bar" class="mod-bar kopfbild">
            <div id="search" class="m-block m-block-search lmu-popover__wrapper">
                <h6 class="g-area-heading area-heading">
                    Suche
                </h6>
                <form action="https://www.ifkw.uni-muenchen.de/funktionen/suche/index.html" id="cref_iframe">
                    <fieldset class="m-search-wrapper">
<input type="text" name="q" class="m-search-term" placeholder="Google&trade; benutzerdefinierte Suche" onclick="this.value=''">
                        <input value="Suchen" type="submit" name="sa" title="Suchen" class="m-search-button" alt="Suchen">
                    </fieldset>
                    <div class="lmu-popover__content">
                        <a href="http://www.uni-muenchen.de/funktionen/datenschutz/index.html#google" class="lmu-popover_message" title="Hinweise zur
Datenübertragung bei der Google™ Suche">Hinweise zur Datenübertragung bei der Google™ Suche</a>
                    </div>
                </form>
            </div>
    <div id="fn" class="m-block m-block-fn">
        <h6 class="g-area-heading area-heading">Links und Funktionen</h6>
        <ul class="m-list">
                        <li class="m-item m-first"><span class="m-separator"><a href="http://www.uni-muenchen.de" class="m-link" target="_blank" title="www.lmu.de
- Startseite">www.lmu.de</a></span></li>
                            <li class="m-item">
                                <span class="m-separator"><a href="https://www.portal.uni-muenchen.de" class="m-link" title="LMU-Portal">LMU-Portal</a></span>
                            </li>
                            <li class="m-item">
                                <span class="m-separator"><a href="https://www.sozialwissenschaften.uni-muenchen.de/index.html" target="_blank" class="m-link"
title="Sozialwissenschaftliche Fakultät">Sozialwissenschaftliche Fakultät</a></span>
                            </li>
                            <li class="m-item">
                                <span class="m-separator"><a href="funktionen/sitemap2/index.html" class="m-link" title="Sitemap">Sitemap</a></span>
                            </li>
        </ul>
        <div class="g-clear"></div>
    </div>
        <div id="lang" class="m-block m-block-lang">
            <h6 class="g-area-heading area-heading">Sprachumschaltung</h6>
            <ul class="m-list">
                            <li class="m-item m-first">
                                <span class="m-separator"><a href="https://www.en.ifkw.uni-muenchen.de/index.html" class="m-link" title="English">English</a>
</span>
                            </li>
            </ul>
```

Figure 14.1: Die IfKW-Webseite als HTML-Quellcode. . .

Eine Webseite besteht aus vielen verschiedenen HTML-Elementen, die auch ineinander verschachtelt sein können. Ein HTML-Element besteht aus folgenden Bestandteilen:

- ein *Tag*, das das Element begrenzt und durch `<tagname>` geöffnet und durch `</tagname>` geschlossen wird

Figure 14.2: ... und ihre Darstellung im Browser

- optionale *Attribute*, die im eröffnenden Tag durch `schlüssel = "wert"`-Paare angelegt werden
- der *Text* des Elements

Zwei Beispiele:

`<p>Hier steht etwas Text</p>` definiert ein Absatz-Element mit dem `<p>`-Tag (für *P*aragraph), das den Text `Hier steht etwas Text` enthält.

`<p>Hier steht ein <a href="verlinkte-seite.html">Link</a></p>` erzeugt einen Absatz, in dem der Text `Hier steht ein Link` enthalten ist, wobei das Wort Link durch das `<a>`-Tag (für *A*nchor) mit dem Attribut `href="verlinkte-seite.html"` in einen Link umgewandelt wird, der auf die Datei `verlinkte-seite.html` verweist.

14.2.1 HTML-Tags

Es gibt sehr viele verschiedene HTML-Tags. Einige der wichtigsten sind:

Table 14.1: Wichtige HTML-Tags

Tag	Bedeutung
`<head>`	Head der Seite mit Metainformationen (Sprache, Kodierung etc.); wird nicht dargestellt
`<body>`	"Körper" der Seite; enthält den gesamten eigentlichen Inhalt
`<p>`	Absatz
`<a>`	Link; das Linkziel wird über das Attribut `href` festgelegt
`<strong>` / `<b>`	Fettdruck
`<em>` / `<i>`	Kursivsetzung
`<h1>`, `<h2>` etc.	Überschrift der ersten, zweiten usw. Ebene
`<table>`	Tabelle
`<ol>`	Geordnete Liste (Einträge werden nummeriert)
`<ul>`	Ungeordnete Liste (Einträge werden Punkte/Striche/etc. gekennzeichnet)
`<li>`	Listeneintrag
`<div>`	Container (wird zum Abgrenzen und Formatieren ganzer Webseiten-Bereiche genutzt)
`<span>`	Inline-Container (wird zum Abgrenzen und Formatieren von Zeichen und Wörtern im Textfluss genutzt)
`<img>`	Bilddatei; die Zieldatei wird über das Attribut `src` festgelegt

Eine umfassende Übersicht über alle HTML-Tags gibt es z. B. hier.

14.2.2 CSS, Klassen und IDs

Tags definieren die Funktion einzelner HTML-Elemente, ändern aber bis auf einige Voreinstellungen zunächst wenig am Aussehen der einzelnen Elemente. Zwar könnten Elemente auch individuell über das `style`-Attribut *gestyled* werden; in der Praxis ist das aber wenig effizient, da häufig gleiche Elemente auch gleich aussehen sollen (z. B. alle Links auf der Seite dieselbe Farbe, dieselbe Reaktion beim Überfahren des Links etc. haben sollen), und das auch über mehrere HTML-Seiten hinweg .

Hier kommen sogenannte Stylesheets im *CSS*-Format (für *C*ascading *S*tyle*s*heets) ins Spiel, mit denen in einer von den einzelnen HTML-Seiten separaten Datei das Aussehen von HTML-Elementen festgelegt wird. Hierzu werden in einer Stylesheet-Datei für einzelne HTML-Tags, Klassen oder IDs (die über die Attribute `class` bzw. `id` HTML-Tags zugewiesen werden)[3] Regeln definiert, die deren Aussehen bestimmen:

[3]Mehrere HTML-Elemente können dieselbe Klasse haben, die ID hingegen muss für jedes Element einzigartig sein.

```
.blueOnRed {
  color: "blue";
  background-color: "red";
}
```

Hier legen wir für die Klasse `blueOnRed` (der `.` identifiziert Klassen, `#` IDs) fest, dass Elemente mit dieser Klasse einen roten Hintergrund `background-color: "red"` sowie blaue Schrift `color: "blue"` aufweisen sollen. Wir können diese Klasse nun HTML-Elementen mit dem Attribut `class` zuweisen: `<p class="blueOnRed">Dieser Text erscheint auf einer Webseite nun blau auf rotem Hintergrund</p>`. Ein HTML-Element kann auch mehrere Klassen (und eine oder mehrere IDs) gleichzeitig zugewiesen bekommen.

Dies ist nun nicht nur praktisch für Webdesigner, sondern auch für uns, wenn wir mittels Web Scraping nur bestimmte Bestandteile von Webseiten automatisiert erfassen möchten. So ist es z. B. naheliegend, dass Artikelbestandteile auf einer Nachrichtenseite zugehörige Klassen haben, etwa der gesamte Artikel immer in einem `<div class="artikel">`-Element steht, die Überschrift des Artikels in einem `<h2>` innerhalb dieses `<div>`-Elements steht, die Artikel-AutorInnen immer in einem `<p class="author">`-Element stehen usw.

Wir können diese Elemente daher über ihre Klassen, die HTML-Tags oder eine Kombination aus allem identifizieren, ansteuern und speichern. Dies machen wir im nächsten Kapitel.

Chapter 15

Web Scraping

Web Scraping bezeichnet allgemein Verfahren zur automatisierten Extraktion von Inhalten auf Webseiten. In unserem Fall setzen wir hierzu ein Skript in R auf, das die uns interessierenden Elemente auf Webseiten identifziert und in einem R-Objekt speichert. Hierfür benötigen wir das zum Tidyverse gehörige Package `rvest`, das wir auch einzeln über `install.packages("rvest")` installieren können:

```
install.packages("rvest")
```

`rvest` zählt jedoch nicht zu den Kernpackages des Tidyverse und muss daher separat geladen werden:

```
library(tidyverse)
library(rvest)
```

Mittels `rvest` laden wir zunächst den HTML-Quelltext einer Webseite herunter und wählen anschließend die HTML-Elemente aus, die uns interessieren. Wir benötigen also zunächst eine Möglichkeit, schnell für uns interessante HTML-Elemente auf einer Webseite identifizieren zu können.

15.1 HTML-Elemente identifizieren

Keine Angst: wir müssen nicht den gesamten Quellcode einer Webseite durchlesen, um entsprechende HTML-Elemente zu finden. Moderne Browser bieten in der Regel eine *Untersuchen*-Funktion, mit der HTML-Elemente per Klick im Code der Webseite hervorgehoben werden. In *Google Chrome* klicken wir hierzu per Rechtsklick auf das Element, das wir identifizieren möchten, und gehen dann

auf *Untersuchen*. Möchten wir z. B. auf meiner Insitutswebseite meinen Namen *scrapen*, zeigt sich folgendes Bild:[1]

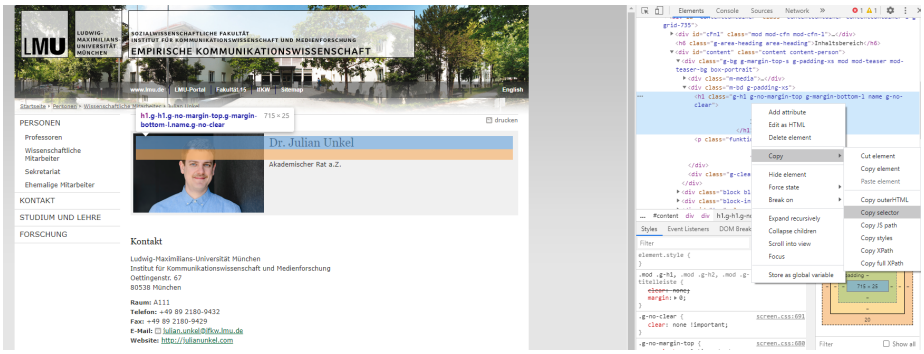


Figure 15.1: Untersuchen-Funktion in Google Chrome

Im rechten Bildschirmbereich (oder je nach Einstellung auch links, oben oder unten) öffnet sich der Quellcode, wobei das gewählte Element grau hervorgehoben ist (`<h1 class="g-h1` usw.). Mit einem Rechtsklick im Codebereich auf ein Element öffnet sich ein weiteres Menü, über das wir via *Copy* - *Copy selector* den zugehörigen *CSS Selector* (also die genaue Kombination aus Tags, Klassen und IDs, die dieses Element identifiziert) kopieren, was in diesem Fall `#content > div.g-bg.g-margin-top-s.g-padding-xs.mod.mod-teaser.mod-teaser-bg.box-portrait > div.m-bd.g-padding-xs > h1` entspricht.[2]

Noch komfortabler gelingt die Identifikation mit dem Tool Selector Gadget, das vom `rvest`-Team zur Verfügung gestellt wird. Hierbei handelt es sich um ein Bookmarklet, das per Klicks die Identifikation von einem oder auch mehreren HTML-Elementen ermöglicht.

Hierzu muss das Tool erst einmal durch Ziehen des Links auf die Lesezeichenleiste des Browsers installiert werden. Im Anschluss kann das Tool durch Klicken auf das Lesezeichen auf jeder Webseite gestartet werden. Ein Klick auf ein Element und das Tool versucht, es zu identifzieren; der zugehörige Selector wird in der Leiste unten angezeigt:

- grün hervorgehoben sind Elemente, die bewusst per Klick ausgewählt wurden
- gelb hervorgehoben sind Elemente, die mit dem aktuellen Selector ebenfalls ausgewählt werden. Per Klick auf diese werden diese deselektiert und

[1] Wir üben mit meinem Profil, nicht weil ich eine ausgeprägte narzisstische Ader habe, sondern weil ich hier relativ sicher sein kann, dass ich die Zustimmung habe, auch auf diese Daten zugreifen zu dürfen.

[2] *Copy selector* erzeugt oft eine sehr lange Kette, die in der Regel jedoch gar nicht nötig ist, um das jeweilige Element eindeutig zu identifizieren. Tatsächlich wären wir hier z. B. auch nur mit der Klasse `.g-h1` schon erfolgreich.

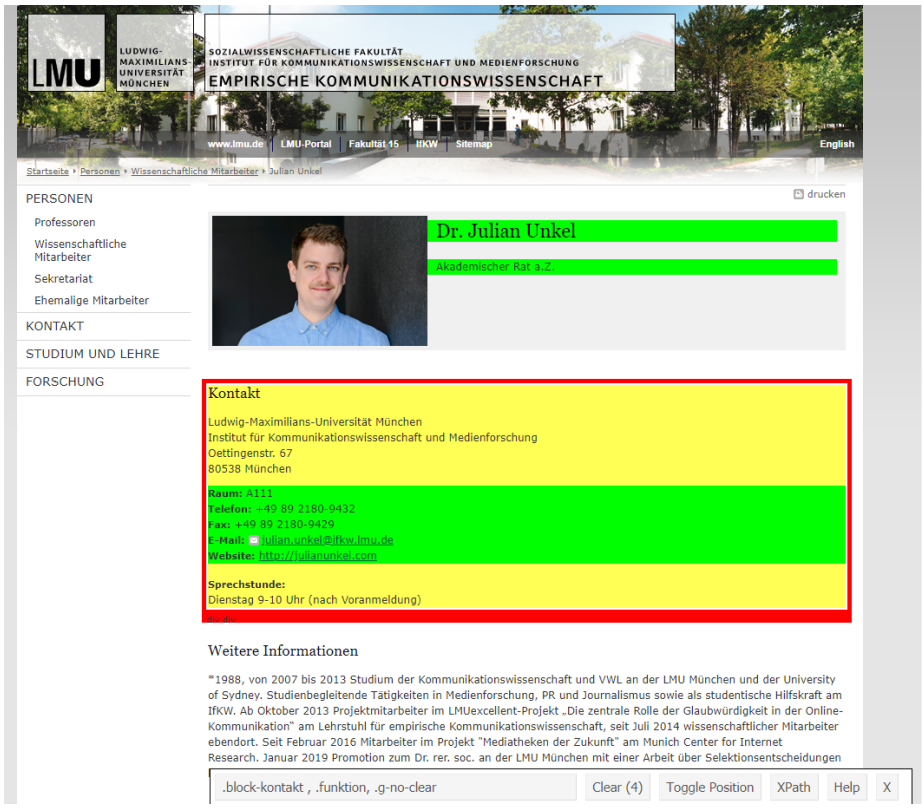


Figure 15.2: Selector Gadget in Aktion

das Tool versucht einen Selector zu finden, das diese Elemente nicht mit umfasst.
- rot hervorgehoben sind Elemente, die gezielt deselektiert wurden

Im Anschluss kann der zugehörige Selector einfach aus der Leiste unten kopiert werden.

15.2 Web Scraping mit `rvest`

Nachdem wir die uns interessierenden HTML-Elemente identifziert haben, können wir uns ans eigentliche Web Scraping machen.

15.2.1 Quellcode einlesen mit `read_html()`

Als erstes laden wir hierzu den Quellcode der betreffenden Webseite herunter. Hierzu stellt uns `rvest` (bzw. ein Package, auf dem `rvest` aufbaut) die Funktion `read_html()` zur Verfügung, der wir einfach die jeweilige URL als String übergeben können:

```
ifkw <- read_html("https://www.ls1.ifkw.uni-muenchen.de/personen/wiss_ma/unkel_julian/i
```

Alle folgenden Funktionen können in einer Pipe verwendet werden.

15.2.2 Elemente auswählen mit `html_nodes()`

Zur Auswahl von HTML-Elementen nutzen wir die Funktion `html_nodes()`, der wir einen CSS-Selector als Argument übergeben. Wir haben z. B. oben gesehen, dass der Name auf den IfKW-Profilseiten in einem Element mit der Klasse `g-h1` steht:

```
ifkw %>%
  html_nodes(".g-h1")
```

```
## {xml_nodeset (1)}
## [1] <h1 class="g-h1 g-no-margin-top g-margin-bottom-l name g-no-clear">\n
```

Dies hat nun das gesamte Element ausgewählt. Bei sehr verschachtelten HTML-Dokumenten können wir die Funktion auch mehrfach hintereinander aufrufen, um uns nach und nach zu dem uns interessierenden Element “vorzutasten”. Hier wählen wir zunächst den Inhaltsbereich der Seite aus (ID: `content`) und wählen innerhalb dieses Bereichs alle Elemente mit dem HTML-Tag `<span>` aus:

```
ifkw %>%
  html_nodes("#content") %>%
  html_nodes("span")
```

```
## {xml_nodeset (12)}
##  [1] <span class="g-label label-raum">Raum:</span>
##  [2] <span class="raum">A111</span>
##  [3] <span class="g-label label-telefon">Telefon:</span>
##  [4] <span class="telefon">+49 89 2180-9432</span>
##  [5] <span class="g-label label-fax">Fax:</span>
##  [6] <span class="fax">+49 89 2180-9429</span>
##  [7] <span class="g-label label-email">E-Mail:</span>
##  [8] <span class="email"><a href="mailto:julian.unkel@ifkw.lmu.de" class="g-link-mail" title="
##  [9] <span class="g-label label-website">Website:</span>
## [10] <span class="website"><a href="http://julianunkel.com">http://julianunkel.com</a></span>
## [11] <span class="g-label label-sprechstunde">Sprechstunde:</span>
## [12] <span class="sprechstunde">Dienstag 9-10 Uhr (nach Voranmeldung)</span>
```

Zum Vergleich: Beschränken wir die Auswahl zuvor nicht mittels der ID `content`, erhalten wir auch alle `<span>`-Elemente, die im Menübereich etc. stehen.

```
ifkw %>%
  html_nodes("span")
```

```
## {xml_nodeset (18)}
##  [1] <span class="m-separator"><a href="http://www.uni-muenchen.de" class="m-link" target="_bl
##  [2] <span class="m-separator"><a href="http://www.portal.lmu.de" target="_blank" class="m-lin
##  [3] <span class="m-separator"><a href="https://www.sozialwissenschaften.uni-muenchen.de/index
##  [4] <span class="m-separator"><a href="https://www.ifkw.uni-muenchen.de/index.html" target="_
##  [5] <span class="m-separator"><a href="../../../funktionen/sitemap2/index.html" class="m-link
##  [6] <span class="m-separator"><a href="https://www.en.ls1.ifkw.uni-muenchen.de/index.html" cl
##  [7] <span class="g-label label-raum">Raum:</span>
##  [8] <span class="raum">A111</span>
##  [9] <span class="g-label label-telefon">Telefon:</span>
## [10] <span class="telefon">+49 89 2180-9432</span>
## [11] <span class="g-label label-fax">Fax:</span>
## [12] <span class="fax">+49 89 2180-9429</span>
## [13] <span class="g-label label-email">E-Mail:</span>
## [14] <span class="email"><a href="mailto:julian.unkel@ifkw.lmu.de" class="g-link-mail" title="
## [15] <span class="g-label label-website">Website:</span>
## [16] <span class="website"><a href="http://julianunkel.com">http://julianunkel.com</a></span>
## [17] <span class="g-label label-sprechstunde">Sprechstunde:</span>
## [18] <span class="sprechstunde">Dienstag 9-10 Uhr (nach Voranmeldung)</span>
```

Wir können zudem mehrere Elemente gleichzeitig auswählen, indem wir die zugehörigen Selektoren durch Kommas `,` trennen. So stehen in den IfKW-Profilen die Kontaktinformationen beispielsweise in HTML-Elementen, die die Klassen `raum`, `telefon`, `fax`, `email` und `website` haben:

```
ifkw %>%
  html_nodes(".raum, .telefon, .fax, .email, .website")
```

```
## {xml_nodeset (5)}
## [1] <span class="raum">A111</span>
## [2] <span class="telefon">+49 89 2180-9432</span>
## [3] <span class="fax">+49 89 2180-9429</span>
## [4] <span class="email"><a href="mailto:julian.unkel@ifkw.lmu.de" class="g-link-mail
## [5] <span class="website"><a href="http://julianunkel.com">http://julianunkel.com</a
```

15.2.3 Text aus HTML-Elementen extrahieren mit `html_text()`

Meistens interessiert uns nicht das gesamte HTML-Element samt Tags und Attributen, sondern lediglich der Text, der in diesem Element steht. Diesen extrahieren wir mit `html_text()`:

```
ifkw %>%
  html_nodes(".g-h1") %>%
  html_text()
```

```
## [1] "\n                    Dr.\n                Julian Unkel\n            "
```

Das ist schon näher am gewünschten Ergebnis, den Namen zu extrahieren – der String enthält zwar noch viel unnötigen Whitespace (`\n` steht für *N*ewline, also einen Zeilenumbruch), aber dagegen haben wir ja auch schon eine Lösung gelernt (siehe Kapitel 12.1.5):

```
ifkw %>%
  html_nodes(".g-h1") %>%
  html_text() %>%
  str_squish()
```

```
## [1] "Dr. Julian Unkel"
```

Das ganze klappt natürlich auch mit mehreren Elementen gleichzeitig. So extrahieren wir den Text aller Kontaktinformationen:

```
ifkw %>%
  html_nodes(".raum, .telefon, .fax, .email, .website") %>%
  html_text()
```

```
## [1] "A111"                    "+49 89 2180-9432"        "+49 89 2180-9429"        "julian.u
```

15.2.4 Attribute aus HTML-Elementen extrahieren mit `html_attr()`

Neben dem Text sind zudem Attribute der selektierten HTML-Elemente für uns von Interesse. Diese können wir mit der Funktion `html_attr()`, wobei wir das uns interessierende Attribut als String übergeben müssen.

Einer der häufigsten Anwendungsfälle ist hierbei die Extraktion von Links aus Webseiten. Aus dem vorhergehenden Kapitel (siehe Kapitel 14.2.1) wissen wir, dass Links in HTML in einem `<a>`-Tag stehen und das Ziel des Links über das Attribut `href` definiert wird. So extrahieren wir beispielsweise alle Link-Ziele, die im Inhalt meines IfKW-Profils stehen:

```
ifkw %>%
  html_nodes("#content") %>% # Inhaltsbereich auswählen
  html_nodes("a") %>%        # Alle Links (<a>-Tag) auswählen
  html_attr("href")          # Attribut "href" extrahieren
```

```
## [1] "mailto:julian.unkel@ifkw.lmu.de" "http://julianunkel.com"          "publikationen.html"
```

Ein weiterer häufiger Anwendungsfall ist das Extrahieren von Bilddaten. Diese werden in HTML durch ein `<img>`-Tag eingebunden, wobei die zugehörige Bilddatei über das Attribut `src` angeben wird. So extrahieren wir beispielsweise den (relativen) Dateipfad aller Bilder auf dieser Seite:

```
ifkw %>%
  html_nodes("img") %>%
  html_attr("src")
```

```
## [1] "//cms-static.uni-muenchen.de/default/lmu/img/blank.png"        "//cms-static.uni-muenchen
## [3] "julian_unkel.png"
```

Gleichzeitig zeigt dies auch schon eine Schwierigkeit beim Web Scraping auf: oft führen Webseiten-Betreiber*innen Schritte durch, die das Web Scraping erschweren. Wenn Sie beispielsweise den ersten Bildpfad (`cms-static.uni-muenchen.de/default/lmu/img/blank.png`) in einen Browser kopieren, werden Sie sehen, dass sich dahinter mitnichten das schöne Institutsbild aus dem Seitenkopf befindet, sondern ein Blanko-Bild, das dann anderweitig durch das Bild im Seitenkopf ersetzt wird.

15.3 Effizientes Scraping

Mit diesen vier Funktionen – `read_html()`, `html_nodes()`, `html_text()` und `html_attr()`[3] – können wir bereits so gut wie alle Web-Scraping-Aufgaben bewältigen. Bisher führen wir unseren Code jedoch nur auf einer einzigen Webseite auf, deren Informationsgehalt wir vermutlich genauso schnell oder schneller von Hand hätten erfassen können. Wirklich effizient scrapen wir daher erst, wenn wir die bisher gelernten Inhalte kombinieren und uns Muster in Webseiten zu Nutze machen, um dieselben Scraping-Schritte auf mehrere Seiten anwenden zu können.

Rufen wir uns nochmals die Schritte in Erinnerung, um die Namens- und Kontaktinformationen von meinen IfKW-Profil zu scrapen:

```
# Quellcode einlesen
ifkw <- read_html("https://www.ls1.ifkw.uni-muenchen.de/personen/wiss_ma/unkel_julian/i

# HTML-Elemente extrahieren
kontakt <- ifkw %>%
  html_nodes(".g-h1, .raum, .telefon, .fax, .email, .website") %>%
  html_text() %>%
  str_squish()

kontakt
```

```
## [1] "Dr. Julian Unkel"         "A111"                     "+49 89 2180-9432"
## [6] "http://julianunkel.com"
```

Wir speichern diese Informationen zur besseren Weiterverarbeitung in einem Tibble:

```
kontakt_tibble <- tibble(
  name = kontakt[1],
  raum = kontakt[2],
  telefon = kontakt[3],
  fax = kontakt[4],
  email = kontakt[5],
  website = kontakt[6]
)

kontakt_tibble
```

[3] Eine fünfte Funktion, `html_table()` soll nicht unerwähnt bleiben. Hierbei handelt es sich um eine *Convenience*-Funktion, die eine gesamte Tabelle extrahiert und automatisch in einen Dataframe umwandelt.

```
## # A tibble: 1 x 6
##   name             raum  telefon          fax              email                    website
##   <chr>            <chr> <chr>            <chr>            <chr>                    <chr>
## 1 Dr. Julian Unkel A111  +49 89 2180-9432 +49 89 2180-9429 julian.unkel@ifkw.lmu.de http://jul
```

Was, wenn wir diese Schritte nun auch auf andere IfKW-Profile anwenden möchten? Es ist zu vermuten, dass dieses Schema auch bei anderen Profilen eingehalten wird. Wir können den Code also in eien Funktion umwandeln, die die jeweilige Profilseite als Argument aufnimmt:

```
scrape_ifkw <- function(url) {
  # Quellcode einlesen
  ifkw <- read_html(url)

  # HTML-Elemente extrahieren
  kontakt <- ifkw %>%
    html_nodes(".g-h1, .raum, .telefon, .fax, .email, .website") %>%
    html_text() %>%
    str_squish()

  # In Tibble umwandeln
  kontakt_tibble <- tibble(
    name = kontakt[1],
    raum = kontakt[2],
    telefon = kontakt[3],
    fax = kontakt[4],
    email = kontakt[5],
    website = kontakt[6]
  )

  # 5 Sekunden warten, um den Server bei vielen Anfragen
  # nicht zu überlasten
  Sys.sleep(5)

  # Tibble zurückgeben
  return(kontakt_tibble)
}
```

Nun benötigen wir nur noch einen Vektor mit den Profilseiten, die wir scrapen möchten:

```
to_scrape <- c(
  "https://www.ls1.ifkw.uni-muenchen.de/personen/wiss_ma/unkel_julian/index.html",
  "https://www.ls1.ifkw.uni-muenchen.de/personen/professoren/brosius_hansbernd/index.html"
)
```

Und schon können wir mithilfe der `map_`-Funktionen (siehe Kapitel @ref(#tidyiteration)) unsere Scrape-Funktion auf alle Profilseiten anwenden. Da unsere Funktion ein Tibble ausgibt, können wir `map_dfr()` verwenden, um die resultierenden Tibbles zeilenweise miteinander zu verbinden:

```
ifkw_kontakte <- map_dfr(to_scrape, scrape_ifkw)
```

Das Resultat:

```
ifkw_kontakte
```

```
## # A tibble: 2 x 6
##   name                          raum  telefon          fax              email
##   <chr>                         <chr> <chr>            <chr>            <chr>
## 1 Dr. Julian Unkel              A111  +49 89 2180-9432 +49 89 2180-9429 julian.unkel@
## 2 Prof. Dr. Hans-Bernd Brosius 102   +49 89 2180-9455 +49 89 2180-9443 hans-bernd.br
```

15.4 Verantwortungsbewusstes Scrapen

Zum Abschluss noch einige "Verhaltensregeln" beim Web Scraping:

- Seitenspezifische Regeln beachten: viele Webseiten legen in einer Datei namens `robots.txt` fest, wie nicht-menschliche "Besucher*innen" die Seite aufrufen dürfen; zudem gibt es oft "Terms of Service" oder ähnliches. Wenn eine Webseite Scraping untersagt, sollte das auch beherzigt werden.
- Sparsam scrapen: Das heißt zum einen, nur das abzurufen, was wirklich benötigt wird. Zum anderen sollten Server auch nicht mit Anfragen überlastet werden; oft ist es schon ausreichend, eine kleine Wartezeit von ein paar Sekunden in entsprechende Scraping-Funktionen einzubauen.
- Vorstellig werden: HTTP-Anfragen enthalten im Header einen sogenannten *User-Agent*, eine Textzeile, die dem Server mitteilt, *wer* gerade eine Anfrage stellt. Bei der Nutzung von Webbrowsern wird hier z. B. der Name und die Versionsnummer des verwendeten Browsers mitgeteilt. Diese Textzeile kann jedoch beliebig angepasst werden, um den Servern weitere Informationen bereitzustellen.

Ein R-Package, das beim verantwortungsbewussten Scrapen viel Hilfestellung leistet, ist `polite`, das hier näher beschrieben wird.

Mit der enthaltenen Funktion `bow()` stellen wir uns dem Server zunächst (optional mit eigenem *User-Agent*) vor; dabei werden auch die seitenspezifischen Regeln in der `robots.txt` abgerufen:

```
library(polite)
session <- bow("https://www.ls1.ifkw.uni-muenchen.de/personen/wiss_ma/unkel_julian/index.html",
               user_agent = "R-Kurs Computational Methods in der politischen Kommunikationsforsch
session
```

```
## <polite session> https://www.ls1.ifkw.uni-muenchen.de/personen/wiss_ma/unkel_julian/index.html
##     User-agent: R-Kurs Computational Methods in der politischen Kommunikationsforschung
##     robots.txt: 1 rules are defined for 1 bots
##    Crawl delay: 5 sec
##   The path is scrapable for this user-agent
```

Wir sehen, dass wir hier prinzipiell scrapen dürfen, zwischen einzelnen *Crawls* jedoch 5 Sekunden warten sollten. Die Funktion `scrape()` ersetzt nun `read_html()` aus `rvest` und berücksichtigt automatisch die zugehörigen Regeln.[4] Anschließend können die gewohnten HTML-Extraktionsfunktionen aus `rvest` genutzt werden:

```
ifkw <- scrape(session)

ifkw %>%
  html_nodes(".g-h1") %>%
  html_text() %>%
  str_squish()
```

```
## [1] "Dr. Julian Unkel"
```

15.5 Übungsaufgaben

Erstellen Sie für die folgenden Übungsaufgaben eine eigene Skriptdatei oder eine R-Markdown-Datei und speichern diese als `ue15_nachname.R` bzw. `ue15_nachname.Rmd` ab.

Übungsaufgabe 15.1. Web Scraping I:

Scrapen Sie von diesem Artikel folgende Informationen:

- Erscheinungsdatum und Uhrzeit ('9. Juni 2020, 9:20 Uhr')
- Kicker ('HSV in der zweiten Liga')

[4] Das heißt auch, dass `scrape()` die Arbeit verweigert, wenn Scrapen nicht erlaubt ist.

- Überschrift (‘Das sind Dinge, die sehr, sehr weh tun"’)
- Lead-Absatz (‘So wird es eng mit dem Aufstieg. . . ’)

Übungsaufgabe 15.2. Web Scraping II:

Im Artikeltext verbirgt sich ein Link zu einer Themenseite von *suedeutsche.de*. Extrahieren Sie den Ziellink.

Übungsaufgabe 15.3. Web Scraping III:

Erstellen Sie eine Funktion, um die Informationen aus Aufgabe 1 von einem beliebigen *sueddeutsche.de*-Artikel zu extrahieren. Testen Sie Ihre Funktion zusätzlich zum obigen Artikel auch anhand von diesem Artikel.

Chapter 16

APIs

APIs (*A*pplication *P*rogramming *I*nterface, zu deutsch Anwendungsprogrammierschnittstelle oder nur Programmierschnittstelle) sind Schnittstellen, mit denen Software-Anwendungen mit anderen Anwendungen kommunizieren und Daten austauschen können. Wenn wir im Webkontext von APIs sprechen, meinen wir damit in der Regel sogenannte *RESTful Web APIs*[1], die auf eine HTTP-Anfrage mit definierten Parametern Daten zurückgeben. Die Webseite ProgrammableWeb bietet einen umfassenden Überblick über solche APIs.

16.1 Grundlagen

Jede API funktioniert anders, nimmt eigene Parameter an, gibt Daten eigens strukturiert zurück und erfordert die Einarbeitung in die jeweilige Dokumentation der API; zugleich ist das Grundprinzip aber gleich:

- wir senden eine HTTP-Anfrage (siehe Kapitel 14.1) an die URL der API, wobei wir mittels Query-Parametern spezifieren, was wir wissen möchten
- die API gibt als Antwort die Daten in einem Textformat (häufig JSON, XML oder CSV) zurück.

Wir setzen uns daher zunächst mit diesen Grundprinzipien auseinander.

16.1.1 URLs, Querys und Parameter

Die Nutzung einer API unterscheidet sich zunächst nicht wesentlich davon, eine URL in einen Webbrowser einzugeben: in beiden Fällen senden wir (bzw. eine

[1] für *Re*presentation *S*tate *T*ransfer. Mehr dazu hier

Software, also z. B. der Webbrowser oder RStudio)[2] eine Anfrage (meistens einen GET-Request) an einen Server und erhalten daraufhin eine Datei zurück – z. B. eine HTML-Datei, die dann vom Browser intepretiert und angezeigt wird.

Server lassen sich so konfigurieren, dass die URL auch Parameter beinhalten kann, die die Anfrage spezifizieren. Sehen wir uns das an einem Alltagsbeispiel an: wenn wir nach "ifkw" googeln, dann sollte die Adresszeile im Browser in etwa so aussehen: `https://www.google.de/search?safe=off&...q=ifkw...`. Wir können diese URL in ihre Bestandteile aufteilen:

- das `https` bezeichnet das *Schema* und gibt in diesem Fall an, dass wir das Netzwerkprotokoll HTTPS (eine sicherere Variante von HTTP) verwenden möchten
- `www.google.de` ist der *Host*, also letztlich der Computer, der die Ressourcen, die wir anfragen möchten, beherbergt
- `/search` gibt den *Pfad* an, unter dem die Ressource zu finden ist
- das `?` schließlich leitet den *Query-String* ein, der benannte Parameter enthält (in der Form `name=wert` und verbunden durch `&`), die vom Server im Rahmen der Anfrage verarbeitet werden

Im Falle unserer Google-Suche sehen wir unter anderem den Parameter `q=ifkw` – das `q` steht in diesem Fall für Query, also Suchanfrage, und ist auf unsere spezifische Suchanfage `ifkw` gesetzt. Wir können diesen Paramter daher nutzen, um auch ohne die Suchmaske direkt über die Adresszeile des Browsers eine Google-Suchanfrage zu starten, indem wir beispielsweise `https://www.google.de/search?q=ifkw` eingeben. Wenn wir direkt die zweite Ergebnisseite anzeigen möchten, können wir den Parameter `start` hinzufügen und auf `10` setzen, Google also mitteilen, dass wir erst beim zehnten Suchresultat beginnen möchten: `https://www.google.de/search?q=ifkw&start=10`.

Google verarbeitet also die Parameter, die wir in der URL angeben, und gibt basierend darauf die entsprechende Ressource zurück – in diesem Fall also eine HTML-Datei mit der zugehörigen Suchergebnisseite. Nicht anders funktionieren auch APIs – wir senden eine Anfrage und definieren über Parameter genauer, was wir erhalten bzw. machen möchten.

16.1.2 JSON

In der Regel geben Web-APIs keine HTML-Dateien zurück, sondern nutzen andere Datenformate. Eines der am häufigsten verwendeten ist *JSON* (*J*ava*S*cript *O*bject *N*otation und gesprochen wie der englische Vorname Jason), ein flexibel einsetzbares, als reine Textdatei speicherbares und zugleich sehr einfach lesbares

[2] In aller Regel nutzen diese Programme wiederum das Programm cURL, das somit das am häufigsten installierte und verwendete Programm der Welt sein dürfte.

Datenformat. Vermutlich reicht bereits die Beispieldatei im Wikipedia-Eintrag, um die Grundzüge zu verstehen:

```
{
  "Herausgeber": "Xema",
  "Nummer": "1234-5678-9012-3456",
  "Deckung": 2e+6,
  "Waehrung": "EURO",
  "Inhaber":
  {
    "Name": "Mustermann",
    "Vorname": "Max",
    "maennlich": true,
    "Hobbys": ["Reiten", "Golfen", "Lesen"],
    "Alter": 42,
    "Kinder": [],
    "Partner": null
  }
}
```

Wir erkennen zum einen die unterschiedlichen Objekttypen (String, Numerisch, Logisch; siehe auch Kapitel 2.2), die ganz ähnlich wie in R definiert werden (Strings durch `""`, Zahlen durch rein numerische Werte, logische Werte durch `true`/`false`); zum anderen sehen wir, dass wir Werte benennen und beliebig tief ineinander verschachteln können.

In R könnten wir die obige Beispieldatei als Liste speichern, die benannte Vektoren ebenso wie weitere Listen enthält – und genau das erledigen dann auch Packages für uns, die JSON-Dateien (bzw. Strings, die wie JSON aussehen) automatisch in R-Listen umwandeln.

16.1.3 Zugangsvoraussetzungen und Rate Limits

Nicht jede API lässt sich von jedem nutzen: wirklich offene APIs sind in der Minderheit, in der Regel ist zumindest ein Account beim jeweiligen Anbieter – also z. B. ein Twitter-Account für die Twitter-API – erforderlich. Viele Plattformen und Anbieter gewähren nur über einen Entwickler-Account, für den man sich extra registrieren, bewerben oder auch zahlen muss, Zugang zu ihrer API. Auch dies wird von Anbieter zu Anbieter unterschiedlich gehandhabt. Tatsächlich ist es so, dass insbesondere Social-Media-Plattformen den Zugang zu ihren APIs für die Wissenschaft in den vergangenen Jahren erschwert haben.[3]

Zudem begrenzen die meisten Anbieter den Zugang zu ihren APIs aus Sicherheitsgründen über sogenanntes *Rate Limiting*. Das bedeutet, dass ein Account

[3] Etwas Lesestoff: After the 'APIcalypse': social media platforms and their fight against critical scholarly research

in einem bestimmten Zeitintervall nur eine bestimmte Anzahl an Anfragen stellen darf (z. B. 15 Anfragen alle 15 Minuten), um eine Überlastung des Servers oder missbräuchlichen Datenabruf zu verhindern. Entsprechend sollte beim Schreiben von API-Anfragen darauf geachtet werden, dass durch diese keine Rate Limits überschritten werden, da sonst lediglich der HTTP-Code `429 Too Many Requests` und eine Fehlermeldung zurückgegeben werden.

16.2 API-Anfragen in R ausführen mit `httr`

Um eigene API-Anfragen in R zu stellen, benötigen wir vorrangig das Package `httr`. Dieses gehört zum erweiterten Tidyverse, sollte also bereits installiert sein, muss aber separat geladen werden. Zudem laden wir das Package `jsonlite`, das den Umgang mit JSON-Dateien in R erleichtert. Auch dieses Package wird über das Tidyverse mitinstalliert, muss aber separat geladen werden:

```
library(tidyverse)
library(httr)
library(jsonlite)
```

Als Beispiel nutzen wir die Pushshift Reddit API, ein privates Projekt, das offenen API-Zugang zu Reddit ermöglicht. Unter obigem Link ist die API beschrieben. Das wichtigste in Kürze:

- Die Stamm-URL der API ist `https://api.pushshift.io`
- Die API bietet zwei *Endpoints*, `/reddit/search/comment` und `/reddit/search/submission`; wir können uns Endpoints einfach als unterschiedliche Pfade vorstellen, die in Kombination mit der Stamm-URL für unterschiedliche Funktionen der API zuständig sind. Um *Submissions* (also Beiträge) auf Reddit abzurufen, nutzen wir also die URL `https://api.pushshift.io/reddit/search/submission`, für Kommentare unter diesen Submissions die URL `https://api.pushshift.io/reddit/search/comment`
- Parameter, die wir für die API verwenden können, unterscheiden sich je Endpoint und sind daher einmal für Kommentare und einmal für Submissions
- Wir erfahren außerdem noch weitere Details, z. B. dass standardmäßig die 25 neuesten Submissions bzw. Kommentare zurückgegeben werden

Mit diesem Wissen können wir unsere erste API-Anfrage schreiben. Hierzu rufen wir die 10 neuesten Beiträge des Subreddits r/politics (einem Subreddit für politische Nachrichten aller Art) ab, die im Beitragstitel das Wort “Corona” enthalten. Aus der API-Dokumentation für Submissions erfahren wir, dass wir

über die Parameter `subreddit`, `title` und `size` das zu durchsuchende Subreddit bzw. Begriffe, die im Titel vorkommen müssen, sowie die Anzahl der Beiträge angeben können.

`httr` umfasst Funktionen für alle typischen HTTP-Anfragetypen. In den meisten Fällen möchten wir Daten abrufen, stellen also eine GET-Anfrage. Hierzu nutzen wir die gleichnamige Funktion `GET()`, für die wir:

- mit dem Argument `url` als String die Stamm-URL angeben, hier also `"https://api.pushshift.io"`
- mit dem Argument `path` als String den Pfad zu unserem gewünschten Endpoint angeben, hier also `"/reddit/search/submission"`
- mit dem Argument `query` eine Liste mit Query-Parametern übergeben können, hier also die Angaben zu `subreddit`, `title` und `size`[4]

Und natürlich sollten wir das Resultat einem R-Objekt zuweisen, um damit weiterarbeiten zu können – ich verwende den Namen `resp` (für Response):

```
resp <- httr::GET(url = "https://api.pushshift.io",
            path = "/reddit/search/submission",
            query = list(
              subreddit = "politics",
              title = "corona",
              size = 10
              ))
```

Ein nächster sinnvoller Schritt ist die Funktion `stop_for_status()`, die den HTTP-Code der Response überprüft und eine Fehlermeldung ausgibt, wenn etwas schiefgelaufen ist – zur Erinnerung, der Code `200` bedeutet "Alles okay", und in diesem Fall bleibt die Funktion ohne sichtbares Ergebis.

```
stop_for_status(resp)
```

Im *Environment*-Bereich von RStudio sehen wir, dass es sich bei unserem neuen Objekt `resp` um eine Liste handelt. Wir sollten uns also zunächst die Struktur dieser Liste mit der `str()`-Funktion ansehen. Das `max.level`-Argument gibt an, dass wir in diesem Fall nur die erste Listenebene betrachten möchten

```
str(resp, max.level = 1)
```

```
## List of 10
##  $ url        : chr "https://api.pushshift.io/reddit/search/submission?subreddit=politics&titl
```

[4] Bei zugangsbeschränkten APIs muss hier häufig ein Anmeldeschlüssel mit übergeben werden.

```
##  $ status_code: int 200
##  $ headers    :List of 15
##   ..- attr(*, "class")= chr [1:2] "insensitive" "list"
##  $ all_headers:List of 1
##  $ cookies    :'data.frame': 1 obs. of  7 variables:
##  $ content    : raw [1:63195] 7b 0a 20 20 ...
##  $ date       : POSIXct[1:1], format: "2020-11-18 16:41:39"
##  $ times      : Named num [1:6] 0 0.000043 0.000045 0.000122 0.640346 ...
##   ..- attr(*, "names")= chr [1:6] "redirect" "namelookup" "connect" "pretransfer" .
##  $ request    :List of 7
##   ..- attr(*, "class")= chr "request"
##  $ handle     :Class 'curl_handle' <externalptr>
##  - attr(*, "class")= chr "response"
```

Die Response enthält also 10 Unterpunkte, darunter die gesamte URL, mit der wir unsere API-Anfrage getätigt haben, den bereits überprüften HTTP-Status-Code, eine Liste mit `headers`-Informationen usw. Über den Eintrag `content-type` in den `headers` können wir z. B. den Inhaltstyp der Antwort ausgeben – in diesem Fall wie erwartet eine JSON-Datei.

```
resp$headers$`content-type`
```

```
## [1] "application/json; charset=UTF-8"
```

Für uns von Interesse ist natürlich der Inhalt, `content`, der Anfrage. Diesen können wir der Funktion `content()` extrahieren, wobei wir zusätzlich angeben, dass der Inhalt als Text ausgeben werden soll.

```
resp_content <- content(resp, "text")
str_sub(resp_content, 1, 200) # Aus Anzeigegründen nur die ersten 200 Zeichen ausgeben
```

```
## [1] "{\n    \"data\": [\n        {\n            \"all_awardings\": [],\n
```

Das Resultat ist ein sehr langer Textstring, der, wie wir bereits wissen, im JSON-Standard ist, mit dem wir aber vorerst nicht viel anfangen können. Hier kommt nun das Package `jsonlite` ins Spiel, mit dessen Funktion `fromJSON` wir JSON-Textdateien in R-Objekte umwandeln können (man nennt dies auch *Parsing*):

```
parsed_content <- fromJSON(resp_content)
```

Auch hierbei handelt es sich wieder um eine Liste, deren Struktur wir mit `str()` untersuchen können:

```
str(parsed_content, max.level = 1)
```

```
## List of 1
##  $ data:'data.frame':    10 obs. of  71 variables:
```

In diesem Fall haben wir nur einen weiteren Eintrag unter `data`, der einen `data.frame` enthält. `10 obs.` deutet darauf hin, dass die Fälle wohl unsere 10 Reddit-Beiträge darstellen. Extrahieren wir also diesen Dataframe aus dem Objekt (und wandeln ihn in ein Tibble um):

```
reddit_tibble <- parsed_content$data %>%
  as_tibble()
```

Mit `names()` können wir uns nun einen Überblick über die enthaltenen Variablen verschaffen:

```
names(reddit_tibble)
```

```
##  [1] "all_awardings"              "allow_live_comments"        "author"
##  [5] "author_flair_richtext"      "author_flair_text"          "author_flair_type"
##  [9] "author_patreon_flair"       "author_premium"             "awarders"
## [13] "contest_mode"               "created_utc"                "domain"
## [17] "gildings"                   "id"                         "is_crosspostable"
## [21] "is_original_content"        "is_reddit_media_domain"     "is_robot_indexable"
## [25] "is_video"                   "link_flair_background_color" "link_flair_css_class"
## [29] "link_flair_text"            "link_flair_text_color"      "link_flair_type"
## [33] "media_only"                 "no_follow"                  "num_comments"
## [37] "over_18"                    "parent_whitelist_status"    "permalink"
## [41] "pwls"                       "removed_by_category"        "retrieved_on"
## [45] "selftext"                   "send_replies"               "spoiler"
## [49] "subreddit"                  "subreddit_id"               "subreddit_subscribers"
## [53] "thumbnail"                  "title"                      "total_awards_received"
## [57] "upvote_ratio"               "url"                        "url_overridden_by_dest"
## [61] "wls"                        "post_hint"                  "preview"
## [65] "thumbnail_width"            "crosspost_parent"           "crosspost_parent_list"
## [69] "media_embed"                "secure_media"               "secure_media_embed"
```

Und tatsächlich, wir haben Informationen über die zehn aktuellsten Beiträge im Subreddit *r/politics*, die den Begriff "Corona" im Titel enthalten, abgerufen und können nun z. B. den Titel des Beitrags und den zugehörigen Link anzeigen:

```
reddit_tibble %>%
  select(title, url)
```

```
## # A tibble: 10 x 2
##    title                                                                url
##    <chr>                                                                <chr>
##  1 Anyone find it strange the number of missing or killed micro biologist~ https://
##  2 What will the newly elected President of the United States do first af~ https://
##  3 kamala harris finally admitted they lied to the public about surgical ~ https://
##  4 kamala harris finally admitted they lied to the public about surgical ~ https://
##  5 German police sympathizes with Corona protesters at demonstration.     https://
##  6 Mark Meadows Test Positive for Corona                                  https://
##  7 Mark Meadows, Trump Chief of Staff test positive for Corona Virus      https://
##  8 Corona Headquarters: The implementation of the traffic ban has started~ https://
##  9 This is how the corona response works basically world wide             https://
## 10 Trump cut funding to programs that would have warned us about the coro~ https://
```

Dies zum allgemeinen Vorgehen. In der Realität würden wir hier natürlich lange noch nicht aufhören. Was wenn wir uns nicht auf zehn Beiträge (oder 500, das Maximum, das die Pushshift Reddit API pro Anfrage vorsieht) beschränken möchten? Wir könnten in diesem Fall beispielsweise das Erstellungsdatum der Beiträge extrahieren, den Minimalwert speichern (also das Erstellungsdatum des ältesten Beitrags) und eine erneute Anfrage starten, dabei jedoch nur Beiträge abrufen, die vor diesem Datum erstellt wurden (also die 500 nächstälteren Beiträge) – und dann, unter Berücksichtigung des Rate Limits, einen Loop schreiben, der diese Schritte so lange wiederholt, bis keine weiteren Beiträge zurückgegeben werden.

Erneut gilt: jede API ist anders aufgebaut und erfordert daher spezifische Einarbeitung. Die Grundschritte sind aber immer nahezugleich: Dokumentation lesen, Anfrage mit `httr` stellen und dann schrittweise vorarbeiten, bis die gewünschten Daten vorhanden sind.

16.3 API-Wrapper nutzen

Die gute Nachricht: in vielen Fällen müssen wir uns nicht die Mühe machen, unsere eigenen Anfragen von Grund auf selbst zu schreiben. Für viele größere APIs gibt es sogenannte Wrapper-Packages, die die gängigen API-Anfragen in simplere Funktionen "verpacken". Um die Twitter-APIs zu nutzen, können wir beispielsweise auf das Package rtweet zurückgreifen.

```
library(rtweet)
```

Um das Package nutzen können, benötigen wir einen Twitter-Account und müssen bei der ersten Verwendung einer Funktion einmalig eine Twitter-App authorisieren (hierzu öffnet sich automatisch ein Browser-Pop-Up), die die Kommunikation zwischen R und Twitter befähigt.

Danach können wir ohne große Zwischenschritte z. B. die fünf aktuellsten Tweets von Donald Trump abrufen:

```
trump_tweets <- get_timeline("realDonaldTrump", n = 5)
```

Das Resultat ist bereits ein Tibble, das mit 90 Variablen sehr viele Informationen über die einzelnen Tweets enthält. Wir sehen z. B., dass Trump offenbar gerne von seinem iPhone aus twittert:

```
trump_tweets %>%
  select(text, source)
```

```
## # A tibble: 1 x 2
##   text                                                  source
##   <chr>                                                 <chr>
## 1 96% Approval Rating in the Republican Party. Thank you! Twitter for iPhone
```

Dank der großartigen R-Community ist die Arbeit mit gängigen APIs also erstaunlich einfach, zumindest sobald man die Zugangskriterien erfüllt. Es gilt also:

- zunächst schauen, ob bereits ein Wrapper-Package für die gewünschte API vorhanden ist
- erst dann mühsam eigene Anfragen schreiben

16.4 Übungsaufgaben

Aufgrund der Vielfalt an APIs gibt es diese Woche keine Übungsaufgaben im gewöhnlichen Sinne. Recherchieren Sie stattdessen in Ihrer Projektgruppe, welche APIs für Ihr Forschungsprojekt relevant sind, wie die Zugangsvoraussetzungen dafür sind und ob es etwaige R-Packages gibt, die Sie verwenden können.

Part IV

Automatisierte Inhaltsanalyse in R

Chapter 17

Einführung und Grundbegriffe

Besonders in den Sozialwissenschaften hat die automatisierte Inhaltsanalyse großer Textmengen in den vergangenen Jahren stark an Bedeutung gewonnen, da mit den zugehörigen Verfahren ganz neue Datenbestände – oder endlich in gebührendem Umfang – analysiert werden können. Zugleich haben sich in den vergangenen Jahren einige Packages als Standardwerkzeug in R empfehlen können und somit einige Prozesse vereinheitlicht.

Zu den beiden relevantesten Packages für die automatisierte Inhaltsanalyse zählen `quanteda` (für *Qu*antitative *An*alysis of *Te*xtual *Da*ta) und `tidytext` (in Anlehnung an das Tidyverse), die insbesondere das Handling von Textdaten sowie die vorbereitenden Schritte (auch als *Preprocessing* bezeichnet) für spezifischere Verfahren deutlich erleichtern. Quanteda hat dabei den größeren Funktionsumfang und wird daher auch unser Primärpackage in den nächsten Kapiteln sein. Tidytext enthält zwar auch Funktionen für die zentralsten Handlings- und Vorbereitungsschritte, zeigt seinen Wert aber vor allem in der Konvertierung von Textdaten in *tidy data* und wird somit für uns insbesondere dann relevant, wenn Ergebnisse z. B. mittels `ggplot2` visualisiert werden sollen.

Zunächst installieren wir beide Packages:

```
install.packages(c("quanteda", "tidytext"))
```

Und natürlich müssen wir diese auch mit dem bekannten `library()`-Befehl laden. Wir laden zudem erneut das Tidyverse:

```
library(tidyverse)
library(tidytext)
library(quanteda)
```

Als Beispiel-Datensatz verwenden wir Tweets von Donald Trump und Joe Biden, die diese dieses Jahr (bis einschließlich 24. Juni) abgesetzt haben, bereinigt um Retweets. Die Daten sind über Moodle als `trump_biden_tweets_2020.csv` verfügbar.

```
tweets <- read_csv("data/trump_biden_tweets_2020.csv")
tweets
```

```
## # A tibble: 4,153 x 7
##       id account  link                    content
##    <dbl> <chr>    <chr>                   <chr>
##  1     1 JoeBiden https://twitter.com/Joe~ "Our final fundraising deadline of 2019 i
##  2     2 JoeBiden https://twitter.com/Joe~ "Every single human being deserves to be
##  3     3 JoeBiden https://twitter.com/Joe~ "With just over one month until the Iowa
##  4     4 JoeBiden https://twitter.com/Joe~ "This election is about the soul of our n
##  5     5 JoeBiden https://twitter.com/Joe~ "Every day that Donald Trump remains in t
##  6     6 JoeBiden https://twitter.com/Joe~ "It was a privilege to work with @JulianC
##  7     7 JoeBiden https://twitter.com/Joe~ "Like Vicky said, we need a president who
##  8     8 JoeBiden https://twitter.com/Joe~ "I'm excited to share that we raised $22
##  9     9 JoeBiden https://twitter.com/Joe~ "If you're a teacher or a firefighter, yo
## 10    10 JoeBiden https://twitter.com/Joe~ "Before the holidays, Jill walked across
## # ... with 4,143 more rows
```

Wie wir sehen, ist die Fallebene der einzelne Tweet. Für jeden Tweet haben wir eine numerische `id`, den `account` (`realDonaldTrump` oder `JoeBiden`), den `link` zum Tweet, den Text des Tweets (`content`), Veröffentlichungsdatum und -uhrzeit (`date`) sowie die Anzahl der `retweets` und `favorites`. Insgesamt liegen uns 4153 Tweets, davon 2654 von Trump und 1499 von Joe Biden, vor.

Wir setzen uns nun zunächst mit einigen Grundbegriffen und -konzepten auseinander, bevor wir in den kommenden Kapiteln unterschiedliche Analyseverfahren an den Daten ausprobieren.

17.1 Korpora und Dokumente

Ziel der Inhaltsanalyse ist die Untersuchung mehrerer Textdokumente, wobei es sich dabei um Bücher, Artikel, Redetranskripte etc., in unserem Fall um Tweets, handeln kann. Die Sammlung aller Dokumente, die wir in unsere Analyse einbeziehen möchten, wird als *Korpus* bezeichnet.

In Quanteda gibt es für Korpora einen spezifischen Objekttypen, den wir mit der Funktion `corpus()` erzeugen können. Wenn wir hierfür einen Dataframe bzw. ein Tibble an Texten verwenden möchten, geben wir mit den Argumenten `docid_field` die Spalte an, in der die ID des Dokuments steht, und identifzieren den Text des jeweiligen Dokuments über das Argument `text_field`:

```
tweets_corpus <- corpus(tweets, docid_field = "id", text_field = "content")
```

Das so erzeugte `corpus`-Objekt enthält für jedes Dokument einen Eintrag. Alle anderen Variablen aus dem Ursprungsdatensatz werden automatisch als sogenannte `docvars` hinterlegt und können jederzeit über die Funktion `docvars()` abgerufen werden.

```
tweets_corpus
```

```
## Corpus consisting of 4,153 documents and 5 docvars.
## 1 :
## "Our final fundraising deadline of 2019 is just hours away an..."
##
## 2 :
## "Every single human being deserves to be treated with dignity..."
##
## 3 :
## "With just over one month until the Iowa Caucus, we need all ..."
##
## 4 :
## "This election is about the soul of our nation - and Donald T..."
##
## 5 :
## "Every day that Donald Trump remains in the White House puts ..."
##
## 6 :
## "It was a privilege to work with @JulianCastro during the Oba..."
##
## [ reached max_ndoc ... 4,147 more documents ]
```

`corpus`-Objekte verfügen über eine eigene `summary()`-Methode, mit der wir uns bereits erste Statistiken über jedes Dokument im Korpus und die zugehörigen Docvars ausgeben lassen können:

```
summary(tweets_corpus, n = 5) %>% # Anzeige auf die ersten 5 Dokumente beschränken
  as_tibble()
```

```
## # A tibble: 5 x 9
```

```
##   Text  Types Tokens Sentences account  link
##   <chr> <int>  <int>     <int> <chr>    <chr>
## 1 1        43     49         3 JoeBiden https://twitter.com/JoeBiden/status/1212180
## 2 2        33     45         4 JoeBiden https://twitter.com/JoeBiden/status/1212442
## 3 3        38     39         2 JoeBiden https://twitter.com/JoeBiden/status/1212524
## 4 4        17     20         1 JoeBiden https://twitter.com/JoeBiden/status/1212540
## 5 5        33     37         2 JoeBiden https://twitter.com/JoeBiden/status/1212556
```

Wir bereiten unseren Textkorpus nun für weitere Analysen vor; die folgenden Schritte werden dabei auch als *Preprocessing* bezeichnet.

17.2 Tokenization, Stopwords und n-Gramme

Unter *Tokenization* versteht man die Aufspaltung eines Textstrings in kleinere Bestandteile. In den meisten Fällen wird als Token das einzelne Wort gewählt, wir können aber beispielsweise Texte auch in Sätze oder einzelne Zeichen aufteilen. Sehen wir uns die obige Ausgabe nochmals an, so sehen wir, dass für jeden Tweet bereits die Anzahl der Tokens (in diesem Fall also Wörter) sowie der Types (einzigartige Wörter) und Sentences (Sätze) angegeben ist.

Die meisten Verfahren, die wir noch kennenlernen werden, arbeiten nach dem sogenannten Bag-of-Words-Modell, womit Texte als – bildlich gesprochen – Wortbeutel betrachtet werden, in denen die einzelnen Wörter und deren Anzahl eine Rolle spielen, nicht jedoch deren syntaktischen und grammatikalischen Zusammenhänge. Für all diese Verfahren müssen Texte daher zunächst in einzelne Wörter tokenisiert werden. In Quanteda erledigen wir dies mit der Funktion `tokens()`, die standardmäßig nach Wörtern tokenisiert:

```
tweet_tokens <- tokens(tweets_corpus)
tweet_tokens
```

```
## Tokens consisting of 4,153 documents and 5 docvars.
## 1 :
##  [1] "Our"         "final"       "fundraising" "deadline"    "of"          "2019"
## [10] "away"        "and"         "we"
## [ ... and 37 more ]
##
## 2 :
##  [1] "Every"    "single"   "human"    "being"    "deserves" "to"       "be"       "
## [ ... and 33 more ]
##
## 3 :
##  [1] "With"   "just"   "over"   "one"    "month"  "until"  "the"    "Iowa"   "Caucus
## [ ... and 27 more ]
```

```
##
## 4 :
##  [1] "This"     "election" "is"       "about"    "the"      "soul"     "of"       "our"      "
## [ ... and 8 more ]
##
## 5 :
##  [1] "Every"   "day"     "that"    "Donald"  "Trump"   "remains" "in"      "the"     "White"
## [ ... and 25 more ]
##
## 6 :
##  [1] "It"             "was"            "a"              "privilege"      "to"             "wor
##  [8] "@JulianCastro"  "during"         "the"            "Obama"          "Administration"
## [ ... and 43 more ]
##
## [ reached max_ndoc ... 4,147 more documents ]
```

Wie wir sehen, wurden die Tweets in einzelne Wörter (und Symbole) aufgeteilt. Der Standard-Tokenizer von Quanteda ist hier insofern komfortabel, als dass Mentions und Hashtags (siehe z. B. Tweet 6) beibehalten werden. Allerdings sind auch noch Bestandteile enthalten, die wir im Sinne des Bag-of-Words-Ansatzes nicht benötigen. Darunter fallen beispielsweise Satzzeichen, Ziffern und Symbole sowie URLs. Wir können diese Bestandteile beim Tokenisieren durch entsprechende `remove_`-Argumente entfernen.

```
tweet_tokens <- tokens(tweets_corpus,
                       remove_punct = TRUE,    # Entfernt Satzzeichen
                       remove_numbers = TRUE, # Entfernt Ziffern
                       remove_symbols = TRUE, # Entfernt Symbole (darunter auch Emojis)
                       remove_url = TRUE)     # Entfernt URLs
tweet_tokens
```

```
## Tokens consisting of 4,153 documents and 5 docvars.
## 1 :
##  [1] "Our"         "final"       "fundraising" "deadline"    "of"          "is"          "just
## [10] "and"         "we"          "need"
## [ ... and 30 more ]
##
## 2 :
##  [1] "Every"    "single"   "human"    "being"    "deserves" "to"       "be"       "treated"  "
## [ ... and 24 more ]
##
## 3 :
##  [1] "With"   "just"   "over"   "one"    "month"  "until"  "the"    "Iowa"   "Caucus" "we"
## [ ... and 23 more ]
##
```

```
## 4 :
##  [1] "This"     "election" "is"       "about"    "the"      "soul"     "of"       "
## [ ... and 5 more ]
##
## 5 :
##  [1] "Every"   "day"     "that"    "Donald"  "Trump"   "remains" "in"      "the"
## [ ... and 22 more ]
##
## 6 :
##  [1] "It"             "was"            "a"              "privilege"      "to"
##  [8] "@JulianCastro"  "during"         "the"            "Obama"          "Administra
## [ ... and 37 more ]
##
## [ reached max_ndoc ... 4,147 more documents ]
```

Unsere so erstellten Tokens lassen sich nun noch weiter verfeinern. Falls Groß-/Kleinschreibung nicht explizit zum Forschungsinteresse gehört, ist es sinnvoll, alle Tokens in Kleinschreibung zu konvertieren, sodass beispielsweise `"trump"`, `"Trump"` und `"TRUMP"` als derselbe Token gezählt werden. Hierfür wenden wir auf die Tokens die Funktion `tokens_tolower()` an:

```
tweet_tokens_LC <- tweet_tokens %>%
  tokens_tolower()

tweet_tokens_LC
```

```
## Tokens consisting of 4,153 documents and 5 docvars.
## 1 :
##  [1] "our"         "final"       "fundraising" "deadline"    "of"          "is"
## [10] "and"         "we"          "need"
## [ ... and 30 more ]
##
## 2 :
##  [1] "every"    "single"   "human"    "being"    "deserves" "to"       "be"       "
## [ ... and 24 more ]
##
## 3 :
##  [1] "with"   "just"   "over"   "one"    "month"  "until"  "the"    "iowa"   "caucus
## [ ... and 23 more ]
##
## 4 :
##  [1] "this"     "election" "is"       "about"    "the"      "soul"     "of"       "
## [ ... and 5 more ]
##
## 5 :
```

```
##  [1] "every"   "day"     "that"    "donald"  "trump"   "remains" "in"      "the"     "white"
## [ ... and 22 more ]
##
## 6 :
##  [1] "it"             "was"            "a"              "privilege"      "to"             "wor
##  [8] "@juliancastro"  "during"         "the"            "obama"          "administration"
## [ ... and 37 more ]
##
## [ reached max_ndoc ... 4,147 more documents ]
```

In der Regel möchten wir mittels automatisierter Verfahren Wörter bzw. Tokens herausarbeiten, die in gewisser Weise distinkt für bestimmte Dokumente bzw. Gruppen von Dokumenten in dem untersuchten Korpus sind. Das heißt auch, dass bestimmte Worttypen keinen Informationsgewinn für uns liefern, da sie vielfach in allen Texten vorkommen, z. B. Artikel, Konjunktionen und Präpositionen. Man bezeichnet diese Wörter auch als *Stopwords*. Quanteda enthält etablierte Stopwords-Sammlungen für unterschiedliche Sprachen, die wir über die Funktion `stopwords()` abrufen können:

```
stopwords("english")
```

```
##   [1] "i"          "me"         "my"         "myself"     "we"         "our"        "ours"
##  [11] "yours"      "yourself"   "yourselves" "he"         "him"        "his"        "himself"
##  [21] "herself"    "it"         "its"        "itself"     "they"       "them"       "their"
##  [31] "which"      "who"        "whom"       "this"       "that"       "these"      "those"
##  [41] "was"        "were"       "be"         "been"       "being"      "have"       "has"
##  [51] "does"       "did"        "doing"      "would"      "should"     "could"      "ought"
##  [61] "she's"      "it's"       "we're"      "they're"    "i've"       "you've"     "we've"
##  [71] "he'd"       "she'd"      "we'd"       "they'd"     "i'll"       "you'll"     "he'll"
##  [81] "isn't"      "aren't"     "wasn't"     "weren't"    "hasn't"     "haven't"    "hadn't"
##  [91] "won't"      "wouldn't"   "shan't"     "shouldn't"  "can't"      "cannot"     "couldn't"
## [101] "who's"      "what's"     "here's"     "there's"    "when's"     "where's"    "why's"
## [111] "the"        "and"        "but"        "if"         "or"         "because"    "as"
## [121] "at"         "by"         "for"        "with"       "about"      "against"    "between"
## [131] "before"     "after"      "above"      "below"      "to"         "from"       "up"
## [141] "on"         "off"        "over"       "under"      "again"      "further"    "then"
## [151] "when"       "where"      "why"        "how"        "all"        "any"        "both"
## [161] "most"       "other"      "some"       "such"       "no"         "nor"        "not"
## [171] "so"         "than"       "too"        "very"       "will"
```

Über die Funktion `tokens_remove()` können wir eigens definierte Tokens aus den erstellten Tokens entfernen, eben beispielsweise Stoppwörter. Zu beachten ist, dass dadurch bestimmte Zusammenhänge in den Texten nicht mehr erkennbar sind; wie oben jedoch angesprochen, folgen die meisten automatischen Verfahren dem Bag-of-Words-Modell, sodass diese Zusammenhänge keine

Berücksichtigung finden würden. Allerdings kann für bestimmte Dokumente jeglicher Inhalt verloren gehen, wie hier im Beispiel der fünfte Tweet zeigt:

```
tweet_tokens_reduced <- tweet_tokens_LC %>%
  tokens_remove(stopwords("english"))
tweet_tokens_reduced
```

```
## Tokens consisting of 4,153 documents and 5 docvars.
## 1 :
##  [1] "final"       "fundraising" "deadline"    "just"        "hours"       "away"
## [10] "donation"    "big"         "small"
## [ ... and 16 more ]
##
## 2 :
##  [1] "every"        "single"       "human"        "deserves"     "treated"      "dig
## [10] "marginalized" "vulnerable"   "least"
## [ ... and 4 more ]
##
## 3 :
##  [1] "just"     "one"      "month"    "iowa"     "caucus"   "need"     "hands"    "c
## [ ... and 5 more ]
##
## 4 :
## [1] "election" "soul"     "nation"   "donald"   "trump"    "poison"   "soul"
##
## 5 :
##  [1] "every"   "day"     "donald"  "trump"   "remains" "white"   "house"   "puts"
## [ ... and 7 more ]
##
## 6 :
##  [1] "privilege"      "work"           "@juliancastro"  "obama"          "administra
##  [8] "talented"       "field"          "candidates"     "led"            "historic"
## [ ... and 11 more ]
##
## [ reached max_ndoc ... 4,147 more documents ]
```

Weitere häufig durchgeführte Preprocessing-Schritte sind *Stemming* oder *Lemmatization*. Beim Stemming werden Wörter um Prefixe und Suffixe bereinigt und somit auf ihren Wortstamm reduziert (z. B. werden aus "Beispiel", "Beispiele" und "[des] Beispiels" jeweils "Beispiel"). Dies wird meist über Algorithmen erreicht, die auf eher heuristischen Regeln basieren, beispielsweise alle "-ing"-Endung etc. abschneiden. So können z. B. Singular- und Plurarlformen desselben Wortes auf einen gemeinsamen Stamm reduziert werden und anschließend als derselbe Token behandelt werden; allerdings scheitern diese Algorithmen häufig an unregelmäßigen Verben oder auch

Eigennamen und die Interpretation einzelner Tokens kann bisweilen erschwert werden. Über die Funktion `tokens_wordstem()` bietet Quanteda verschiedene Stemming-Algorithmen an

```
tweet_tokens_reduced %>%
  tokens_wordstem()
```

```
## Tokens consisting of 4,153 documents and 5 docvars.
## 1 :
##  [1] "final"    "fundrais" "deadlin"  "just"     "hour"     "away"     "need"     "help"     "
## [ ... and 16 more ]
##
## 2 :
##  [1] "everi"     "singl"     "human"     "deserv"    "treat"     "digniti"   "everyon"   "poor
## [12] "least"
## [ ... and 4 more ]
##
## 3 :
##  [1] "just"   "one"    "month"  "iowa"   "caucus" "need"   "hand"   "deck"   "talk"   "folk"
## [ ... and 5 more ]
##
## 4 :
## [1] "elect"  "soul"   "nation" "donald" "trump"  "poison" "soul"
##
## 5 :
##  [1] "everi"  "day"    "donald" "trump"  "remain" "white"  "hous"   "put"    "futur"  "planet"
## [ ... and 7 more ]
##
## 6 :
##  [1] "privileg"      "work"          "@juliancastro" "obama"         "administr"     "true"
##  [9] "field"         "candid"        "led"           "histor"
## [ ... and 11 more ]
##
## [ reached max_ndoc ... 4,147 more documents ]
```

Lemmatization (bzw. Lemmatisierung) ist die anspruchsvollere Variante und führt Tokens auf ihren morphologischen Wortstamm (d.h., die Form, in der das jeweilige Wort im Wörterbuch zu finden ist) zurück (so würden also "ist", "bin", "bist" etc. allesamt auf "sein" zurückgeführt werden). Lemmatization ist daher die deutlich validere Variante, allerdings auch entsprechend aufwändiger und nur durch – oftmals auch selbst erstellte – *Dictionaries* zu bewältigen, in denen für alle relevanten Wörter die jeweilige morphologische Grundform hinterlegt ist und anhand derer dann Tokens ersetzt werden (in Quanteda mit der Funktion `tokens_replace()`). Wir werden diese beiden Preprocessing-Schritte jedoch vorerst nicht anwenden, da sich oftmals auch ohne sie bereits recht gute Ergebnisse erzielen lassen.

Um auch im Bag-of-Words-Ansatz Zusammenhänge zwischen Begriffen abbilden zu können, können weitere *n-Gramme* definiert werden. *n-Gramme* bezeichnen die Anzahl an aufeinanderfolgenden Text-Fragmenten, die beim Tokenisieren berücksichtigt werden sollen. Trennen wir unseren Text also in einzelne Wörter, betrachten wir Unigramme. Wir können jedoch auch angeben, das zusätzlich Bigramme (Abfolgen von zwei Wörtern), Trigramme (Abfolgen von drei Wörtern) etc. berücksichtigt werden sollen. Dies ist sinnvoll, wenn wir annehmen, dass auch Wortkombinationen distinkt für bestimmte Dokumente sind (sich beispielsweise die Trump-Tweets nicht nur durch "MAGA", sondern auch "Crooked Hillary" auszeichnen.).

In Quanteda können wir nach der initalen Tokenisierung auch weitere n-Gramme mit der Funktion `tokens_ngrams` erstellen lassen:

```
tweet_tokens_bigrams <- tweet_tokens_reduced %>%
  tokens_ngrams(n = c(1, 2)) # Erzeuge Uni- und Bigramme
tweet_tokens_bigrams
```

```
## Tokens consisting of 4,153 documents and 5 docvars.
## 1 :
##  [1] "final"       "fundraising" "deadline"    "just"        "hours"       "away"
## [10] "donation"    "big"         "small"
## [ ... and 43 more ]
##
## 2 :
##  [1] "every"        "single"       "human"        "deserves"     "treated"      "dig
## [10] "marginalized" "vulnerable"   "least"
## [ ... and 19 more ]
##
## 3 :
##  [1] "just"     "one"      "month"    "iowa"     "caucus"   "need"     "hands"    "c
## [ ... and 21 more ]
##
## 4 :
##  [1] "election"      "soul"          "nation"        "donald"        "trump"
##  [9] "soul_nation"   "nation_donald" "donald_trump"  "trump_poison"
## [ ... and 1 more ]
##
## 5 :
##  [1] "every"   "day"     "donald"  "trump"   "remains" "white"   "house"   "puts"
## [ ... and 25 more ]
##
## 6 :
##  [1] "privilege"      "work"           "@juliancastro"  "obama"          "administra
##  [8] "talented"       "field"          "candidates"     "led"            "historic"
## [ ... and 33 more ]
```

```
##
## [ reached max_ndoc ... 4,147 more documents ]
```

(Es werden hier immer nur die ersten Elemente bzw. Tokens des jeweiligen Textvektors angezeigt; in Tweet 4 sehen wir aber auch einige der erzeugten Bigramme.)

17.3 Dokument-Feature-Matrizen (DFMs)

Die meisten Verfahren, die wir kennenlernen werden, arbeiten mit sogenannten *Dokument-Feature-Matrizen*, kurz *DFM*, als Input. Hierfür wird eine Matrix erstellt, die in den Zeilen alle Dokumente im Korpus und in Spalten *alle* erzeugten Tokens enthält und in den Zellen dann festhält, wie häufig das jeweilige Token (bzw. Feature) im jeweiligen Dokument vorkommt. Wir erzeugen DFMs in Quanteda durch die Funktion `dfm()`:

```
tweets_dfm <- dfm(tweet_tokens_bigrams)
tweets_dfm
```

```
## Document-feature matrix of: 4,153 documents, 49,341 features (99.9% sparse) and 5 docvars.
##     features
## docs final fundraising deadline just hours away need help every donation
##    1     1           1        1    1     1    1    1    2     1        1
##    2     0           0        0    0     0    0    0    0     1        0
##    3     0           0        0    1     0    0    1    0     0        0
##    4     0           0        0    0     0    0    0    0     0        0
##    5     0           0        0    0     0    0    0    0     1        0
##    6     0           0        0    0     0    0    0    0     0        0
## [ reached max_ndoc ... 4,147 more documents, reached max_nfeat ... 49,331 more features ]
```

Wie wir sehen erzeugt dies eine sehr große Matrix: die 4153 Tweets im Korpus enthalten insgesamt 49341 einzigartige Features (in unserem Fall Uni- und Bigramme). Wir sehen außerdem bereits in dieser abgeschnittenen Ansicht, dass die meisten Zellen eine 0 enthalten, d. h. pro Dokument kommt der Großteil der Features *nicht* vor. Den Anteil der leeren Zellen (bzw. 0-Zellen) wird auch als *Sparsitiy* der DFM bezeichnet – in unserem Fall enthalten 99.9% aller Zellen eine 0.

Diese DFM können wir nutzen, um bereits erste einfache Analysen durchzuführen. Die Funktion `topfeatures()` extrahiert beispielsweise die am häufigsten vorkommenden Features:

```
topfeatures(tweets_dfm)
```

```
##        great        trump    president       people        thank       donald
##          619          512          449          423          380          352
```

Etwas aussagekräftiger wird das Ergebnis, wenn wir mit dem Argument `groups` die Ausgabe gruppieren. Praktischerweise haben wir ja in den Docvars den Account gespeichert, sodass wir nun die Top-Features getrennt für Trump und Biden ausgeben lassen können:

```
topfeatures(tweets_dfm, groups = "account")
```

```
## $JoeBiden
##        trump       donald    president donald_trump         need       nation
##          396          342          307          294          282          188
##
## $realDonaldTrump
##     great     thank    people      news      fake      just       now       big
##       589       309       249       238       196       193       181       168
```

Wir sehen: Joe Biden twittert offenbar vor allem über Donald Trump, wohingegen der sich vor allem wohlbekannter Trumpisms ("great", "big", "fake news") bedient.

Mittels `dfm_select()` können wir die DFM zudem anhand Mustern filtern und z. B. lediglich Mentions (beginnen mit `@`) auswählen, um zu sehen, auf wen sich beiden Kandidaten mit ihren Tweets am häufigsten beziehen:[1]

```
dfm_mentions <- dfm_select(tweets_dfm, "@*")
topfeatures(dfm_mentions, groups = "account")
```

```
## $JoeBiden
##             @nra         @drbiden   @petebuttigieg         @ewarren @realdonaldtrump
##              21               15                7                7                5
##        @nra_pass            @cnn
##               4                4
##
## $realDonaldTrump
##        @foxnews  @foxandfriends        @nytimes          @usdot            @cnn
##             108               40               31               31               30
##  @ingrahamangle @washingtonpost
##              11               11
```

[1] per Default erkennt `dfm_select()` sogenannte "glob-style wildcards", mit denen etwa ein Asterisk `*` als einfacher Platzhalter definiert werden kann. Um RegEx-Muster (siehe Kapitel 12.2) zu nutzen, muss `valuetype="regex"` angegeben werden.

Im kommenden Kapitel werden wir uns basierend auf diesen Grundlagen detaillierter mit Text- und Wortmetriken auseinandersetzen, um Unterschiede und Gemeinsamkeiten in den Tweets der beiden Kandidaten zu analysieren.

17.4 Übungsaufgaben

Erstellen Sie für die folgenden Übungsaufgaben eine eigene Skriptdatei oder eine R-Markdown-Datei und speichern diese als `ue17_nachname.R` bzw. `ue17_nachname.Rmd` ab.

Laden Sie den Datensatz `facebook_europawahl.csv` und filtern Sie lediglich Posts der im Bundestag vertretenen Parteien.

Übungsaufgabe 17.1. Korpus erstellen:

Erstellen Sie mit Quanteda einen Korpus für die Facebook-Posts.

Übungsaufgabe 17.2. Tokenization:

Erstellen Sie Tokens für den Korpus. Dabei sollen:

- alle URLs, Symbole, Satzzeichen und Ziffern entfernt werden
- die Tokens in Kleinschreibung umgewandelt werden
- deutsche Stoppwörter entfernt werden
- Uni-, Bi- und Trigramme erstellt werden

Übungsaufgabe 17.3. DFMs:

Erstellen Sie eine DFM auf Basis der oben erzeugten Tokens. Beantworten Sie anhand der DFM folgende Fragen:

- Was sind die häufigsten Features je Partei?
  - Hat es sich gelohnt, neben Uni- auch Bi- und Trigramme zu betrachten?
  - Fallen Ihnen Probleme auf?
- Was sind die häufigsten Hashtags über alle Posts hinweg und nach Partei getrennt betrachtet?

Chapter 18

Textdeskription und einfache Textvergleiche

Nachdem wir im vergangenen Kapitel die Grundlagen und -begriffe der automatisierten Inhaltsanalyse kennengelernt haben, setzen wir uns nun etwas intensiver mit der deskriptiven Analyse von Texten auseinander und werden auch einige einfache Möglichkeiten betrachten, Texte bzw. Dokumente miteinander zu vergleichen.

Wir arbeiten erneut mit den Tweets von Trump und Biden und führen daher zunächst die uns bereits bekannten Schritte zur Aufbereitung des Tweet-Korpus durch:

```
# Setup
library(tidyverse)
library(tidytext)
library(quanteda)

# Daten einlesen
tweets <- read_csv("data/trump_biden_tweets_2020.csv")

# Korpus erzeugen
tweets_corpus <- corpus(tweets, docid_field = "id", text_field = "content")

# Tokens erzeugen
tweets_tokens <- tokens(tweets_corpus,
                        remove_punct = TRUE,
                        remove_numbers = TRUE,
                        remove_symbols = TRUE,
                        remove_url = TRUE) %>%
```

```
  tokens_tolower() %>%
  tokens_remove(stopwords("english"))

# DFM erzeugen
tweets_dfm <- dfm(tweets_tokens)
```

18.1 Worthäufigkeiten

Wir haben bereits im vergangenen Kapitel gesehen, dass wir anhand der DFM schon simple Worthäufigkeiten auszählen können. Allgemein erhalten wir in Quanteda die absoluten Feature-Häufigkeiten mit `featfreq()`.

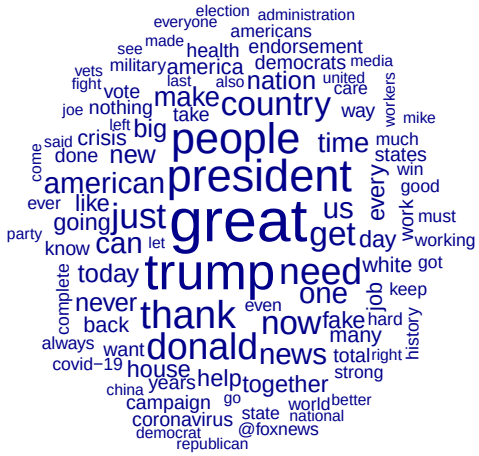
An dieser Stelle sei außerdem auf die Funktion `tidy()` aus dem Tidytext-Package verwiesen, die Output vieler Quanteda-Funktionen automatisch in *tidy data* konvertieren kann und uns somit den weiteren Umgang mit den Daten erleichert:

```
featfreq(tweets_dfm) %>%
  tidy() %>%       # In tidy data konvertieren
  arrange(desc(x)) # Absteigend nach Anzahl sortieren
```

```
## # A tibble: 8,534 x 2
##    names         x
##    <chr>     <dbl>
##  1 great       619
##  2 trump       512
##  3 president   449
##  4 people      423
##  5 thank       380
##  6 donald      352
##  7 need        338
##  8 just        325
##  9 now         308
## 10 country     296
## # ... with 8,524 more rows
```

Quanteda bietet außerdem einige rudimentäre Möglichkeiten, Textdaten zu visualisieren. Erinnert sich noch jemand an den Trend, alles in Wortwolken zu visualisieren? Mit `textplot_wordcloud()` erzeugen wir eine solche:

```
textplot_wordcloud(tweets_dfm, max_words = 100)
```



An dieser Stelle ist es sinnvoll, eine zweite DFM zu erstellen, die nicht auf den einzelnen Dokumenten (= einzelne Tweets), sondern auf den beiden Accounts basiert. Dies führt dazu, dass alle Tweets eines Accounts als ein langes Dokument betrachtet werden, ermöglicht uns aber bereits einfache Vergleiche zwischen den beiden Accounts. Dies erreichen wir im `dfm()`-Befehl mit dem Argument `groups`, wobei wir basierend auf unseren Docvars gruppieren können:

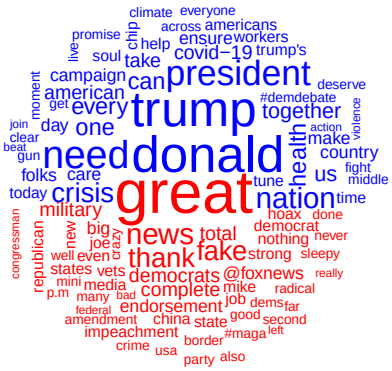
```
tweets_dfm_grouped <- dfm(tweets_tokens, groups = "account")
tweets_dfm_grouped
```

```
## Document-feature matrix of: 2 documents, 8,534 features (36.8% sparse) and 1 docvar.
##                  features
## docs              final fundraising deadline just hours away need help every donation
##   JoeBiden            9           4        4  132     5   46  282  120   151        8
##   realDonaldTrump     5           0        0  193     2   30   56   72    33        0
## [ reached max_nfeat ... 8,524 more features ]
```

Diese DFM hat also nur noch zwei Zeilen (= zwei Dokumente, eines pro Account) und summiert die Feature-Häufigkeiten über alle Tweets, getrennt nach Account, hinweg.

Wir können nun die Wortwolke auch für unsere beiden Accounts getrennt erzeugen lassen, indem wir das Argument `comparison = TRUE` verwenden – mit der alten DFM auf Tweet-Ebene würde hier für jedes Dokument (Tweet) getrennt eine Wolke erzeugt werden, was natürlich kaum sinnvoll darstellbar und interpretierbar wäre.

```
textplot_wordcloud(tweets_dfm_grouped,
                   max_words = 100,
                   comparison = TRUE,
                   color = c("blue", "red"))
```



18.2 Konkordanzen (Keywords in context)

Durch den Bag-of-Word-Approach verlieren wir den Kontext, in dem Begriffe fallen. Daher kann es oft sinnvoll sein, für ausgewählte Schlüsselbegriffe auch die zugehörigen Textstellen samt Kontext auszugeben. Dies wird auch als Konkordanz bezeichnet.

In Quanteda können wir über die Funktion `kwic()` (für *K*ey*w*ords *i*n *c*ontext) solche Konkordanzen ausgeben lassen. Dies kann anhand eines Korpus- oder eines Token-Objekts geschehen; da wir die Tokens bereits um Stoppwörter bereinigt haben, ist es hier sinnvoller den Korpus zu nutzen, damit sich der Kontext im ursprünglichen Satzzusammenhang erschließen lässt. Als zweites Argument benötigen wir noch das Suchmuster, nach dem gesucht werden soll – in der Regel also ein bestimmter Schlüsselbegriff (`kwic()` ist per Default *case-insesitive*, ignoriert also Groß- und Kleinschreibung, sodass wir mit "news" auch "News" und "NEWS" finden):

```
kwic(tweets_corpus, "news") %>%
  as_tibble()
```

```
## # A tibble: 248 x 7
##    docname  from    to pre                                      keyword post
##    <chr>   <int> <int> <chr>                                    <chr>   <chr>
##  1 36         10    10 off on commenting on the                 news    tonight until we know mo
##  2 150         2     2 Great                                    news    out of Utah . No
##  3 282        21    21 already - but I've got                   news    for them : We're not
##  4 446         5     5 I'm heartbroken at the                   news    of yet another mass shoo
##  5 674        28    28 live-tweeting her appearances on cable   news    . We need a president
##  6 882         3     3 This morning's                           news    of another rise in unemp
##  7 943         2     2 Today's                                  news    that more than 22 millio
##  8 1113       15    15 numbers we see in the                    news    are more than just stati
##  9 1268       17    17 toll we see in the                       news    is so much more than
## 10 1435        7     7 , I've got some big                      news    : Next week , I'm
## # ... with 238 more rows
```

Zu den Informationen, die wir erhalten, zählen die Dokument-ID, die Textstelle (in Tokens) in diesem Dokument, ab dem unser Begriff auftritt, sowie der vorherige und nachfolgende Satzkontext (per Default bis zu 5 Wörter, kann mit dem `window`-Argument angepasst werden).

Das Suchmuster kann einen RegEx-Ausdruck beinhalten, um beispielsweise schnell nach allen Hashtags zu suchen:

```
kwic(tweets_corpus, "#*") %>%
  as_tibble()
```

```
## # A tibble: 374 x 7
##    docname  from    to pre                                keyword    post                 patt
##    <chr>   <int> <int> <chr>                              <chr>      <chr>                <fct
##  1 74          9     9 step on stage for tonight's        #DemDebate "in Iowa . Make sure" #*
##  2 75         42    42 be Commander in Chief .            #DemDebate ""                   #*
##  3 77         49    49 an increasingly dangerous world .  #DemDebate ""                   #*
##  4 78         47    47 advance our common security .      #DemDebate ""                   #*
##  5 79         18    18 of the American people .           #DemDebate ""                   #*
##  6 80         53    53 Kim regime's bad behavior .        #DemDebate ""                   #*
##  7 81         39    39 don't share our values .           #DemDebate ""                   #*
##  8 82         38    38 system easier to navigate .        #DemDebate ""                   #*
##  9 83         12    12 prices with drug companies .       #DemDebate ""                   #*
## 10 86         41    41 affordable for every parent .      #DemDebate ""                   #*
## # ... with 364 more rows
```

18.3 Kollokationen

Als Kollokation wird das gemeinsame Auftreten von zwei oder mehr Wörtern bezeichnet. Wir können uns solche Kollokationen mit `textstat_collocations()` ausgeben lassen, wobei auch hier entweder ein Korpus- oder ein Token-Objekt angegeben werden muss. Hier zunächst mit dem Korpus:

```
textstat_collocations(tweets_corpus) %>%
  as_tibble() %>%
  arrange(desc(count))
```

```
## # A tibble: 11,805 x 6
##    collocation  count count_nested length lambda     z
##    <chr>        <int>        <int>  <dbl>  <dbl> <dbl>
##  1 of the         434            0      2  1.75  31.2
##  2 in the         396            0      2  1.91  32.2
##  3 thank you      309            0      2  8.11  34.8
##  4 donald trump   302            0      2  8.17  47.7
##  5 will be        254            0      2  4.33  49.9
##  6 to the         251            0      2  0.469  6.96
##  7 for the        245            0      2  1.67  22.9
##  8 on the         229            0      2  2.05  26.2
##  9 is a           225            0      2  2.36  30.7
## 10 we need        223            0      2  5.04  42.1
## # ... with 11,795 more rows
```

Das sagt uns also noch nicht sonderlich viel über die Dokumente aus, da die häufigsten Kollokationen aus Stoppwörtern ("of the", "in the" etc.) bestehen. Wir können dies umgehen, indem wir die bereits um diese Wörter bereinigten Tokens übergeben:

```
textstat_collocations(tweets_tokens) %>%
  as_tibble() %>%
  arrange(desc(count))
```

```
## # A tibble: 6,231 x 6
##    collocation       count count_nested length lambda     z
##    <chr>             <int>        <int>  <dbl>  <dbl> <dbl>
##  1 donald trump        302            0      2   7.50  44.4
##  2 fake news           151            0      2   7.67  38.4
##  3 white house         145            0      2   9.28  30.7
##  4 complete total      104            0      2   8.93  31.2
##  5 total endorsement   103            0      2   8.42  32.9
##  6 united states        87            0      2   8.63  28.7
```

```
##  7 american people      79            0      2   4.37  29.8
##  8 health care          73            0      2   7.12  32.4
##  9 president trump      65            0      2   3.26  22.7
## 10 military vets        60            0      2   7.64  28.8
## # ... with 6,221 more rows
```

Eine andere Möglichkeit besteht darin, nicht nach der absoluten Häufigkeit, sondern nach dem ebenfalls berechneten Lambda-Koeffizienten zu sortieren. Dieser fällt, vereinfacht gesagt, umso höher aus, je wahrscheinlicher exakt diese Kombination aus Wörtern ist (so gehören etwa sowohl "donald trump" als auch "president trump" zu den absolut am häufigsten vorhandenen Kollokationen; diese weisen aber ein geringeres Lambda auf, da "trump" sowohl mit "donald" als auch mit "president" auftritt und entsprechend die Wahrscheinlichkeit geringer ist als bei Wortpaaren, die nahezu immer in dieser Kombination in diesem Korpus auftreten, z. B. "oval office"). Mit dem `min_count`-Argument können wir festlegen, dass die jeweilige Kollokation mindestens x mal im Korpus vorkommen muss:

```
textstat_collocations(tweets_corpus, min_count = 10) %>%
  as_tibble() %>%
  arrange(desc(lambda))
```

```
## # A tibble: 1,200 x 6
##    collocation     count count_nested length lambda     z
##    <chr>           <int>        <int>  <dbl>  <dbl> <dbl>
##  1 town hall          28            0      2   15.1  9.19
##  2 swine flu          10            0      2   14.1  8.48
##  3 prime minister     11            0      2   13.3  8.66
##  4 THANK YOU          50            0      2   13.3 14.2
##  5 witch hunt         21            0      2   13.1  8.89
##  6 approval rating    26            0      2   12.8 13.3
##  7 george floyd       14            0      2   12.8  8.63
##  8 FAKE NEWS          17            0      2   12.2 14.1
##  9 oval office        12            0      2   11.8  8.05
## 10 KEEP AMERICA       13            0      2   11.6  7.97
## # ... with 1,190 more rows
```

Zwar werden in der Regel Kollokationen von zwei Wörtern untersucht, mit dem `size`-Argument können wir aber auch das gemeinsame Auftreten von mehr als zwei Wörtern ausgeben lassen:

```
textstat_collocations(tweets_tokens, size = 4) %>%
  as_tibble() %>%
  arrange(desc(count))
```

```
## # A tibble: 2,249 x 6
##    collocation                     count count_nested length lambda      z
##    <chr>                           <int>        <int>  <dbl>  <dbl>  <dbl>
##  1 approval rating republican party    20            0      4   9.43  2.28
##  2 radical left nothing democrats      19            0      4   2.43  0.671
##  3 donald trump white house            15            0      4   6.15  1.62
##  4 end gun violence epidemic           15            0      4   1.82  0.441
##  5 white house news conference         14            0      4   1.51  0.414
##  6 get donald trump white              12            0      4   1.25  0.347
##  7 help keep momentum going            12            0      4  -1.44 -0.436
##  8 rating republican party thank       12            0      4  -1.87 -0.462
##  9 complete total endorsement vote     10            0      4  -3.91 -1.00
## 10 military vets second amendment       9            0      4   8.45  2.30
## # ... with 2,239 more rows
```

18.4 Kookkurenzen

Während bei Kollokationen zwei oder mehr Wörter genau in dieser Wortfolge gemeinsam auftreten müssen, untersucht man mittels Kookkurenzen das gemeinsame Auftreten von Wörtern (oder anderen lexikalischen Einheiten) innerhalb einer höher geordneten Einheit, z. B. in einem Dokument. Hierfür wird zunächst eine Co-occurence-Matrix aufgestellt, die für jeden Token prüft, wie oft dieser mit jeweils allen anderen Tokens im Korpus gemeinsam in einem Dokument auftritt. Das Ergebnis ist also eine Matrix, die genausoviele Zeilen wie Spalten (= alle Tokens im Korpus) aufweist. Wir können diese Matrix mit der Funktion `fcm()` (für *f*eature *c*o-occurence *m*atrix) erstellen:

```
tweets_com <- fcm(tweets_tokens)
tweets_com
```

```
## Feature co-occurrence matrix of: 8,534 by 8,534 features.
##              features
## features      final fundraising deadline just hours away need help every donation
##   final           0           2        2    7     1    3    6    7     8        3
##   fundraising     0           0        3    3     1    1    2    4     2        2
##   deadline        0           0        0    2     1    1    2    5     2        2
##   just            0           0        0   14     2   22   39   50    37        6
##   hours           0           0        0    0     0    1    4    4     2        1
##   away            0           0        0    0     0    8   13   10     8        3
##   need            0           0        0    0     0    0   40   73    43        6
##   help            0           0        0    0     0    0    0   21    28        9
##   every           0           0        0    0     0    0    0    0    14        6
##   donation        0           0        0    0     0    0    0    0     0        0
## [ reached max_feat ... 8,524 more features, reached max_nfeat ... 8,524 more feature
```

Auch hier können wir wieder die `tidy()`-Funktion aus dem Tidytext-Package nutzen, um schnell die häufigsten Kookkurenzen zu erhalten – zu beachten ist hier, dass die Reihenfolge der Wörter keine Rolle spielt:

```
tweets_com %>%
  tidy() %>%
  arrange(desc(count))
```

```
## # A tibble: 370,365 x 3
##    document    term      count
##    <chr>       <chr>     <dbl>
##  1 donald      trump       329
##  2 news        fake        191
##  3 white       house       154
##  4 trump       president   153
##  5 endorsement complete    113
##  6 need        president   110
##  7 total       complete    108
##  8 endorsement total       107
##  9 care        health       98
## 10 united      states       95
## # ... with 370,355 more rows
```

18.5 Textkomplexität

um die Komplexität von Texten zu quantifizieren, gibt es mehrere Herangehensweisen, wobei insbesondere die folgenden beiden weit verbreitet sind:

- *Lesbarkeit*: Hier wird quantifiziert, wie einfach ein Text lesbar ist. Das wohl bekannteste Lesbarkeitsmaß ist der Flesch Reading Ease (FRE), bei dem die durchschnittliche Satzlänge in Wörtern und die durchschnittliche Silbenzahl pro Wort miteinander verrechnet werden. In Quanteda können zahlreiche Lesbarkeitsmaße mit der Funktion `textstat_readability()` berechnet werden – für kurze Tweets sind solche Berechnungen aber weniger sinnvoll, weshalb dies hier ausgespart wird.
- *Lexikalische Diversität*: Hier wird quantifiziert, wie vielfältig (*'lexically rich'*) ein Text ist. Das wohl bekannteste Maß ist das *Type-Token-Ratio (TTR)*, wobei einfach die Anzahl an Types, also einzigartigen Tokens, durch die Anzahl an Token geteilt wird. Ein hohes TTR steht demnach für einen großen Wortschatz, wohingegen ein geringes TTR dafür spricht, dass sich viele Wörter häufig wiederholen. Allerdings ist zu beachten, dass das TTR von der Textlänge beeinflusst wird, da es naturgemäß immer schwieriger wird, keine Wörter mehrfach zu verwenden, je länger ein Text

ist. Mit der Funktion `textstat_lexdiv()` lassen sich neben dem TTR (Default-Maß) daher auch noch einige andere Maße berechnen.

```
textstat_lexdiv(tweets_dfm_grouped)
```

```
##          document       TTR
## 1        JoeBiden 0.1736629
## 2 realDonaldTrump 0.1801794
```

18.6 Keyness

Während die bisherigen Auswertungen und Maße auch zur Beschreibung von einzelnen Texten bzw. Dokumenten oder gesamten Korpora verwendet werden können, lernen wir nun ein erstes Vergleichsmaß kennen. Mit *Keyness* wird quantifiziert, wie distinkt ein Begriff für einen Text im Vergleich zu allen anderen Texten im Korpus ist. Es geht also nicht nur darum, dass ein Wort häufig in einem Text vorkommt, sondern zugleich eher selten in den Vergleichstexten ist und somit besonders gut geeignet ist, um den Zieltext zu identifzieren. Wörter mit hoher Keyness können entsprechend als *Keyword* für diesen Text bezeichnet werden.

Keyness-Maße werden berechnet, indem die Worthäufigkeiten im Zieltext mit den erwarteten Worthäufigkeiten im Vergleichskorpus in einem statistischen Test (z. B. Chi²-Test oder Likelihood-Ratio-Test) verglichen werden. In Quanteda können wir die Keyness mit der Funktion `textstat_keyness()` berechnen, wobei eine DFM als erstes Argument sowie mit dem Argument `target` das Zieldokument angegeben wird (alle anderen Dokumente dienen dann jeweils als Vergleichsdokumente). Per Default wird der Chi²-Test genutzt, andere Testverfahren können über das `measure`-Argument angefordert werden.

Um besonders distinkte Begriffe für Joe Biden bzw. Donald Trump auszuwerten, müssen wir wieder die gruppierte DFM nutzen, sodass alle Tweets eines Accounts als "ein" Dokument gezählt werden:

```
textstat_keyness(tweets_dfm_grouped, target = "JoeBiden") %>%
  as_tibble()
```

```
## # A tibble: 8,534 x 5
##    feature    chi2     p n_target n_reference
##    <chr>     <dbl> <dbl>    <dbl>       <dbl>
##  1 donald     452.     0      342          10
##  2 trump      274.     0      396         116
##  3 need       246.     0      282          56
##  4 crisis     196.     0      157           8
```

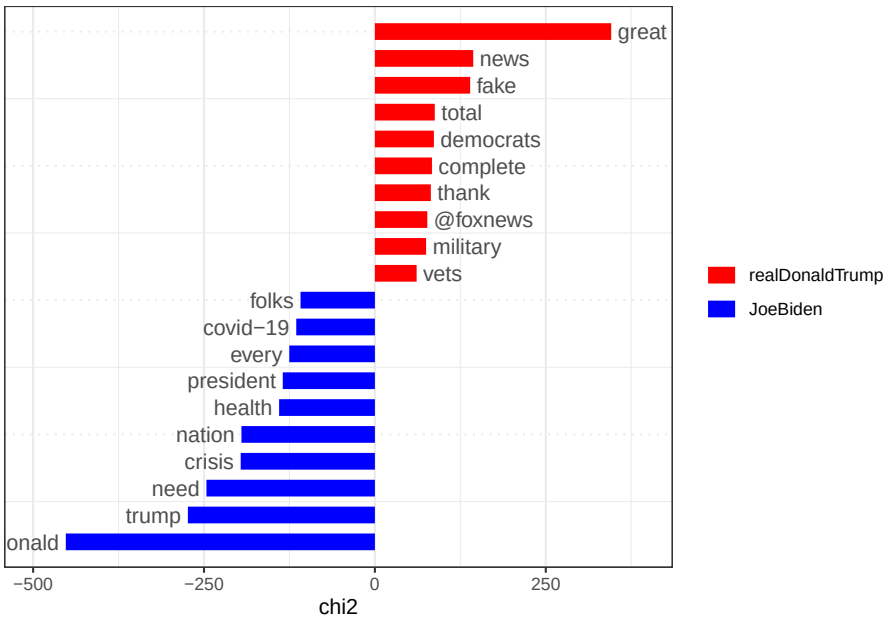
```
##  5 nation     195.     0      188          24
##  6 health     140.     0      127          13
##  7 president  135.     0      307         142
##  8 every      125.     0      151          33
##  9 covid-19   115.     0       99           8
## 10 folks      109.     0       77           0
## # ... with 8,524 more rows
```

```
textstat_keyness(tweets_dfm_grouped, target = "realDonaldTrump") %>%
  as_tibble()
```

```
## # A tibble: 8,534 x 5
##    feature    chi2        p n_target n_reference
##    <chr>     <dbl>    <dbl>    <dbl>       <dbl>
##  1 great     346.  0.            589          30
##  2 news      144.  0.            238          10
##  3 fake      139.  0.            196           0
##  4 total      87.6 0.            138           4
##  5 democrats  86.3 0.            143           6
##  6 complete   83.5 0.            125           2
##  7 thank      81.9 0.            309          71
##  8 @foxnews   76.7 0.            108           0
##  9 military   74.9 0.            120           4
## 10 vets       61.0 5.55e-15       86           0
## # ... with 8,524 more rows
```

Ein mittels `textstat_keyness()` erzeugtes Objekt kann zudem der Funktion `textplot_keyness()` übergeben werden, um die Keywords auch grafisch darzustellen:

```
textstat_keyness(tweets_dfm_grouped, target = "realDonaldTrump") %>%
  textplot_keyness(n = 10, color = c("red", "blue"))
```



18.7 Übungsaufgaben

Erstellen Sie für die folgende Übungsaufgabe eine eigene Skriptdatei oder eine R-Markdown-Datei und speichern diese als `ue18_nachname.R` bzw. `ue18_nachname.Rmd` ab.

Laden Sie den Datensatz `facebook_europawahl.csv` und filtern Sie lediglich Posts der im Bundestag vertretenen Parteien.

Übungsaufgabe 18.1. Textdeskription:

Führen Sie selbstständig eine Textdeskription der Facebook-Posts (und die dazu notwendigen Vorbereitungsschritte) durch. Welche Verfahren bieten sich dafür an? Welche Probleme fallen Ihnen dabei auf?

Betrachten Sie abschließend Keywords für mindestens drei der im Datensatz vorhandenen Parteien. Beschreiben Sie zudem Möglichkeiten, wie man die Ergebnisse (noch) aussagekräftiger gestalten könnte.

Chapter 19

Diktionärbasierte Ansätze

Diktionärbasierte Ansätze verfolgen das Ziel, latente Konstrukte in (Teilmengen von) Textkorpora durch das (vermehrte) Auftreten bestimmter, dem jeweiligen Konstrukt zugeordneter Begriffe zu messen. Kurz gesagt, wird ein Textkorpus anhand einer Liste von Begriffen (Diktionär oder Lexikon), die ein oder mehrere Konstrukte (z. B. die emotionale Polarität, den Populismusgrad etc.) abbilden sollen, ausgezählt – je höher die Anzahl bzw. der Anteil der einem Konstrukt zugeordneten Begriffe, desto stärker ist das jeweilige Konstrukt ausgeprägt.

In den einfachsten (aber auch häufigsten) Fällen handelt es sich bei den verwendeten Diktionären um aus Unigrammen zusammengestellte *kategoriale Lexika*, die also einzelne Wörter einem bestimmten Konstrukt zuordnen (z. B. "Volk" zum Konstrukt "Populismus"), wobei alle enthaltenen Wörter gleich gewichtet werden. Entsprechend handelt es sich um ein relativ simples Verfahren, das Kontext und Grammatik ausblendet; die Klassifikation einzelner, kurzer Text-Dokumente ist daher recht fehleranfällig, etwa wenn eine Negation nicht als solche erkannt wird (taucht beispielsweise in unserem Lexikon zum Konstrukt "Zufriedenheit" das Wort "glücklich" auf, werden "Ich bin glücklich" als auch "Ich bin nicht glücklich" den gleichen Zufriedenheits-Score erhalten). Gute, etablierte Lexika erzielen jedoch trotz dieser Einschränkungen in der Regel und gerade bei größeren Textmengen zufriedenstellende Ergebnisse. Zudem können und sollten die im Folgenden vorgestellten Techniken für die jeweilige Umsetzung noch verfeinert und verbessert werden, beispielsweise durch Gewichtung, der Berücksichtigung von Bigrammen (um beispielsweise Negationen zu erfassen) etc.

Wir werden uns die technische Umsetzung diktionärbasierter Ansätze in diesem Kapitel erneut anhand des aus den vorherigen Kapiteln bekannten Tweet-Korpus ansehen, wobei wir uns die Grundlagen zunächst anhand eines eigens erstellten Diktionärs aneignen, bevor wir mit der Sentiment-Analyse einen häufigen Anwendungsfall betrachten. Abschließend wird noch auf die Verwendung gewichteter Lexika eingegangen.

Zur Vorbereitung laden wir die uns schon bekannten Packages sowie den Tweet-Datensatz und erzeugen ein Korpus-Objekt. Beim Laden des Tweet-Datensatzes wird zudem eine neue Variable `day` erzeugt, die das Datum (ohne Uhrzeit) festhält – da wir für jeden Tweet das exakte Erscheinungsdatum in der Form `YYYY/MM/DD hh::mm::ss` in der Variable `date` vorliegen haben, können wir hierzu einfach die ersten zehn Zeichen dieser Variable extrahieren. Diese neue Variable erleichtert uns im Folgenden die Auswertung auf Tagesebene.

```
# Setup
library(tidyverse)
library(tidytext)
library(quanteda)

# Daten einlesen
tweets <- read_csv("data/trump_biden_tweets_2020.csv") %>%
  mutate(day = str_sub(date, 1, 10))

# Korpus erzeugen
tweets_corpus <- corpus(tweets, docid_field = "id", text_field = "content")
```

19.1 Grundlagen

Donald Trump ist nicht zuletzt für seinen, nunja, *markanten* Redestil bekannt, der sich neben dem unverwechselbaren Sprachfluss auch durch die häufige Verwendung bestimmter Wörter, Floskeln und Neologismen ("bigly") auszeichnet. Wir möchten nun prüfen, ob sich auch in seinen Tweets vermehrt diese *Trumpisms* finden. Um dieses Konstrukt zu messen, erstellen wir also zunächst ein Wörterbuch, das häufig von Trump verwendete Wörter sammelt. Praktischerweise wird uns diese Arbeit durch diesen Artikel in The Atlantic abgenommen. Wir erstellen zunächst einen Textvektor, der die Begriffe enthält (bereinigt um einige uneindeutige Begriffe sowie alle Nicht-Unigramme). Um den Aufwand möglichst gering zu halten, nutzen wir die Funktion `str_split()` aus dem uns bekannten `stringr`-Package (siehe Kapitel 12.1), mit der wir einen Textstring schnell anhand einer Trennzeichenkette (in diesem Fall `" / "`) in einzelne Bestandteile aufteilen können. Das Resultat ist ein `character`-Vektor mit insgesamt 90 Trumpisms.

```
trump_words <- str_split("amazing / beautiful / best / big league / brilliant / elegant
                         pattern = " / ",
                         simplify = TRUE)
```

Um dieses Begriffsliste in Quanteda verwenden zu können, müssen wir ein Dictionary-Objekt erstellen. Hierzu gibt es die Quanteda-Funktion

`dictionary()`, der wir eine Liste mit den enthaltenen Konstrukten übergeben, wobei der Name eines jeden Listeneintrags ein Konstrukt abdeckt, dem die entsprechenden Begriffe zugeordnet werden. In unserem Fall haben wir nur ein Konstrukt, `trumpisms`:

```
trumpisms_dictionary <- dictionary(list(trumpisms = trump_words))
trumpisms_dictionary
```

```
## Dictionary object with 1 key entry.
## - [trumpisms]:
##   - amazing, beautiful, best, big league, brilliant, elegant, fabulous, fantastic, fine, good,
```

Die Auszählung erweist sich denkbar einfach und geschieht innerhalb der uns bereits bekannten `dfm()`-Funktion. Geben wir mit dem Argument `dictionary` ein Diktionär an, werden die enthaltenen Kategorien ausgezählt und als einzelne Spalten in der resultierenden DFM angegeben; alle nicht im Diktionär enhaltenen Begriffe werden schlichtweg ignoriert. Wir können uns an dieser Stelle also auch die bisher durchgeführte, schrittweise Tokenisierung sparen – `dfm()` funktioniert auch mit einem Korpus-Objekt, konvertiert alle Texte in Kleinschreibung und tokenisiert diese automatisch, Stoppwörter etc. sind in unserem Diktionär nicht enhalten und werden daher eh ignoriert.

```
dfm(tweets_corpus, dictionary = trumpisms_dictionary, groups = "account")
```

```
## Document-feature matrix of: 2 documents, 1 feature (0.0% sparse) and 1 docvar.
##                  features
## docs              trumpisms
##   JoeBiden              442
##   realDonaldTrump      2233
```

Das Resultat ist eine sehr überschaubare 2x1-DFM: Joe Biden nutzt in seinen Tweets insgesamt 442 Trumpisms, Donald Trump hingegen 2233. Es sieht also danach aus, dass die gewählten Begriffe tatsächlich recht typisch für Trumps Sprach- und auch Tweet-Stil sind.

Allerdings ist die absolute Häufigkeitsauszählung an dieser Stelle nur bedingt für einen Vergleich geeignet, da Trump auch insgesamt deutlich mehr twittert – von Trump liegen uns 2654, von Biden lediglich 1499 Tweets für das erste Halbjahr 2020 vor. Hier ist es also sinnvoll, die DFM vor der Auszählung zu gewichten. Dies erledigen wir mit der Funktion `dfm_weight()`, wobei unterschiedliche Gewichtungsvarianten mit dem Argument `scheme` (siehe Funktionsdokumentation) gewählt werden können. Mit `scheme = "prop"` gewichten wir jedes Feature nach der Gesamtzahl aller Features in einem Dokument.

Für die folgende Analyse erstellen wir also zunächst eine DFM und gruppieren dabei nach Account, sodass alle Tweets eines Accounts ein einziges Dokument

darstellen. Anschließend gewichten wir proportional und wenden dann unser Diktionär erneut mit dem `dfm()`-Befehl auf die gewichtete DFM an:

```
dfm(tweets_corpus, groups = "account") %>%
  dfm_weight(scheme = "prop") %>%
  dfm(dictionary = trumpisms_dictionary)
```

```
## Document-feature matrix of: 2 documents, 1 feature (0.0% sparse) and 1 docvar.
##                  features
## docs                trumpisms
##   JoeBiden        0.007872052
##   realDonaldTrump 0.028331451
```

Das Ergebnis lässt sich einfach interpretieren: rund 2,8 Prozent aller Wörter in Trumps Tweets sind eines der 90 Trumpisms in unserem Wörterbuch, bei Joe Biden sind dies nur rund 0,8 Prozent – auch proportional verwendet Trump also deutlich mehr Trumpisms als Biden.

Häufig werden diktionärsbasierte Ansätze genutzt, um Veränderungen im (Zeit-)Verlauf aufzuzeigen (z. B.: Wie hat sich etwa der Populismusgehalt in den Reden im Bundestag über die Zeit geändert?). Auch dies ist in Quanteda sehr simpel umzusetzen – wir ziehen einfach eine weitere Variable, die die Zeitebene repräsentiert, beim Gruppieren hinzu. So erhalten wir schnell eine DFM, in der für jeden Account und jeden Tag der Anteil der Trumpisms angegeben ist.

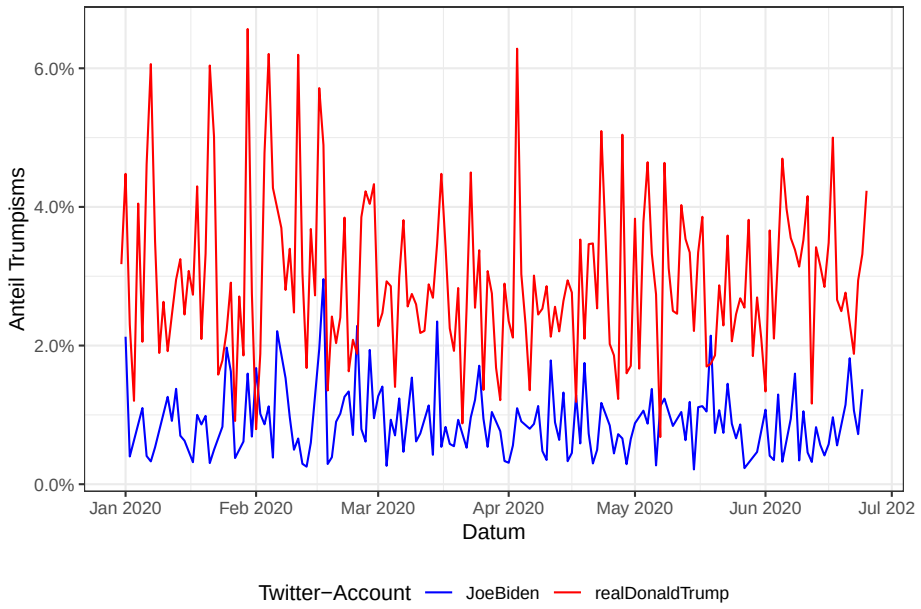
```
trumpisms_per_day <- dfm(tweets_corpus, groups = c("account", "day")) %>%
  dfm_weight(scheme = "prop") %>%
  dfm(dictionary = trumpisms_dictionary)
trumpisms_per_day
```

```
## Document-feature matrix of: 354 documents, 1 feature (6.5% sparse) and 2 docvars.
##                              features
## docs                          trumpisms
##   realDonaldTrump.2019/12/31 0.03174603
##   JoeBiden.2020/01/01        0.02127660
##   realDonaldTrump.2020/01/01 0.04477612
##   JoeBiden.2020/01/02        0.00400000
##   realDonaldTrump.2020/01/02 0.02302632
##   JoeBiden.2020/01/03        0
## [ reached max_ndoc ... 348 more documents ]
```

Um dies grafisch darzustellen, konvertieren wir die DFM mit der `tidy()`-Funktion aus dem `tidytext`-Package in ein Tibble und können dann auf das uns bekannte Visualisierungspackage `ggplot2` (siehe Kapitel 11.2) zurückgreifen. Vorab konvertieren wir die Variable `day` noch von einer `character`-

in eine `date`-Variable; ich nutze zum Verschönern des Plots zudem ein paar Funktionen aus dem `ggplot2`-Package, die uns noch nicht bekannt sind – im Idealfall erschließt sich deren Funktion bereits aus dem Namen und Verwendungszusammenhang.

```
trumpisms_per_day %>%
  tidy() %>%
  separate(document, c("account", "day"), sep = "\\.") %>%
  mutate(day = as.Date(day)) %>%
  ggplot(aes(x = day, y = count, color = account, group = account)) +
  geom_line() +
  scale_color_manual(values = c("blue", "red")) +
  scale_x_date(date_breaks = "1 month", date_labels = "%b %Y") +
  scale_y_continuous(labels = scales::percent) +
  theme_bw() +
  theme(legend.position = "bottom") +
  labs(x = "Datum", y = "Anteil Trumpisms", color = "Twitter-Account")
```



Zwar ist Trump fast immer *trumpiger* als Biden unterwegs, bei beiden variiert der Anteil der Trumpisms aber deutlich. In einem nächsten Schritt könnten wir nun also versuchen, die Ausschläge nach oben auf bestimmte Ereignisse am jeweiligen Datum zurückzuführen.

19.2 Beispiel-Anwendung: Sentiment-Analyse

Da der wissenschaftliche Gehalt des vorherigen Beispiels eher gering ist, wenden wir uns nun einem gebräuchlicherem Anwendungsfall zu: Sentiment-Analysen, also die Bestimmung der (emotionalen) Valenz bzw. Polarität von Texten, greifen sehr häufig auf Diktionäre zurück. Ein bekanntes und weitverbreitetes Sentiment-Dictionary für englischsprachige Texte ist das Sentiment Lexicon von Bing Liu, das aus zwei Wortlisten, einmal positive Begriffe, einmal negative Begriffe, besteht. Es wird nicht zuletzt häufig für Social-Media-Sentiment-Analysen herangezogen, da es auch weitverbreitete *misspellings* enthält. Insgesamt sind knapp 6800 Wörter (2006 positive, 4783 negative) enthalten.

Das *Sentiment Lexicon* kann unter obigem Link kostenfrei heruntergeladen werden und muss zunächst entpackt werden.[1] Beide Wortlisten sind jeweils als Textdatei (`positive-words.txt` bzw. `negative-words.txt`) vorhanden, wobei – nach einem Einführungstext – jedes Wort in einer eigenen Zeile steht.

Um die Wortlisten zu verwenden, nutzen wir die Basis-Funktion `scan()`, mit der Textdateien in einen Vektor eingelesen werden können. Neben dem Dateipfad geben wir mit dem Argument `what` den Objekttypen des Ziel-Vektors an, in unserem Fall also `character()`. `scan()` trennt automatisch bei Whitespace und Zeilenumbrüchen, sodass wir in diesem Fall kein weiteres Trennzeichen definieren müssen. Mit dem Argument `skip` geben wir zudem an, dass die ersten 30 bzw. 31 Zeilen übersprungen werden sollen, da diese in den jeweiligen Dateien den Einführungstext beinhalten (Tipp: öffnet man die Textdateien in RStudio, werden diese mit Zeilennummerierung angezeigt).

```
positive_words <- scan("data/positive-words.txt", what = character(), skip = 30)
negative_words <- scan("data/negative-words.txt", what = character(), skip = 31)
```

Nun erstellen wir auch schon das entsprechende Dictionary-Objekt. Nachdem wir vorher nur eine Kategorie (Trumpisms) definiert haben, sind nun zwei Kategorien nötig – eine für positives Sentiment, eine für negatives –, die entsprechend die jeweilige Wortliste zugeordnet bekommen. Prinzipiell ist die Anzahl der Kategorien in `dictionary()` nicht beschränkt und auch verschachtelte Kategorien sind möglich:

```
sentiment_dictionary <- dictionary(list(
  positive = positive_words,
  negative = negative_words
))
sentiment_dictionary
```

[1] Hierfür ist ein Kompressionsprogramm, das mit `.rar`-Dateien umgehen kann, etwa WinRar oder 7Zip, nötig.

```
## Dictionary object with 2 key entries.
## - [positive]:
##   - a+, abound, abounds, abundance, abundant, accessable, accessible, acclaim, acclaimed, accl
## - [negative]:
##   - 2-faced, 2-faces, abnormal, abolish, abominable, abominably, abominate, abomination, abort
```

Der Rest erfolgt wie gehabt. Zunächst zählen wir wieder die absoluten Häufigkeiten, gruppiert nach Account, aus:

```
dfm(tweets_corpus, dictionary = sentiment_dictionary, groups = "account")
```

```
## Document-feature matrix of: 2 documents, 2 features (0.0% sparse) and 1 docvar.
##                  features
## docs              positive negative
##   JoeBiden            2483     1467
##   realDonaldTrump     3806     2512
```

Joe Biden verwendet also ca. 1,6-mal so viele positive Wörter wie negative Wörter, Donald Trump 1,5-mal so viele. Wir können dieses Verhältnis auch darstellen, indem wir die DFM nachträglich gewichten:

```
dfm(tweets_corpus, dictionary = sentiment_dictionary, groups = "account") %>%
  dfm_weight(scheme = "prop")
```

```
## Document-feature matrix of: 2 documents, 2 features (0.0% sparse) and 1 docvar.
##                  features
## docs               positive  negative
##   JoeBiden        0.6286076 0.3713924
##   realDonaldTrump 0.6024058 0.3975942
```

In beiden Fällen lassen also rund 60% aller in den Tweets verwendeten *und* im Sentiment Lexicon enthaltenen Begriffe auf positives Sentiment schließen. Zu beachten ist, dass es durchaus eine Rolle spielt, wann wir die DFM gewichten; führen wir die Gewichtung, wie oben, vor der Anwendung des Diktionärs durch, erhalten wir den Anteil, den positive bzw. negative Begriffe am gesamten Text aller Tweets der beiden Kandidaten ausmachen.

```
dfm(tweets_corpus, groups = "account") %>%
  dfm_weight(scheme = "prop") %>%
  dfm(dictionary = sentiment_dictionary)
```

```
## Document-feature matrix of: 2 documents, 2 features (0.0% sparse) and 1 docvar.
##                  features
```

```
## docs                positive   negative
##   JoeBiden        0.04422241 0.02618081
##   realDonaldTrump 0.04828907 0.03187130
```

Wir sehen: Trumps Tweets enthalten anteilig sowohl mehr positive als auch mehr negative Begriffe; dies deutet darauf hin, dass Trump insgesamt eine emotionalere Sprache verwendet (natürlich sind auch einige beliebte Trumpisms wie "great" und "sad" im Sentiment Lexicon enthalten).

Betrachten wir erneut den Zeitverlauf. Auch hier ergeben sich kaum Änderungen gegenüber dem Vorgehen bei den Trumpisms – wir gruppieren erneut zusätzlich nach `day`, gewichten aber dieses Mal erst im Anschluss an die Diktionär-Auszählung, um das Verhältnis von positivem zu negativem Sentiment zu erhalten.

```
sentiment_ratio_per_day <- dfm(tweets_corpus, groups = c("account", "day"), dictionary
  dfm_weight(scheme = "prop")
sentiment_ratio_per_day
```

```
## Document-feature matrix of: 354 documents, 2 features (0.565% sparse) and 2 docvars
##                             features
## docs                         positive  negative
##   realDonaldTrump.2019/12/31 0.5000000 0.5000000
##   JoeBiden.2020/01/01        0.5714286 0.4285714
##   realDonaldTrump.2020/01/01 0.8750000 0.1250000
##   JoeBiden.2020/01/02        0.8333333 0.1666667
##   realDonaldTrump.2020/01/02 0.3636364 0.6363636
##   JoeBiden.2020/01/03        0.3500000 0.6500000
## [ reached max_ndoc ... 348 more documents ]
```

Zur grafischen Darstellung konviertieren wir das Ergebnis wieder in *tidy data*. Zu beachten ist, dass `tidy()` direkt in *long data* konvertiert, die Werte für positiv/negativ (bzw. allgemeiner gesprochen: die Kategorien des Diktionärs) stehen nun nicht mehr in eigenen Spalten, sondern sind Ausprägungen der Variablen `term`, wobei der jeweilige Wert in der Variable `count` steht.

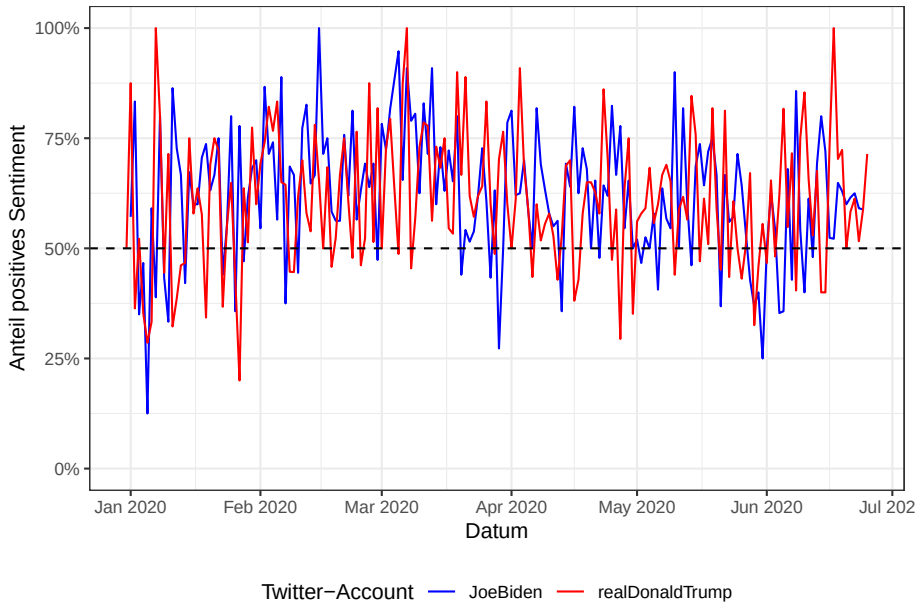
```
sentiment_ratio_per_day %>%
  tidy()
```

```
## # A tibble: 704 x 3
##    document                   term     count
##    <chr>                      <chr>    <dbl>
##  1 realDonaldTrump.2019/12/31 positive 0.5
##  2 JoeBiden.2020/01/01        positive 0.571
```

```
##  3 realDonaldTrump.2020/01/01 positive 0.875
##  4 JoeBiden.2020/01/02        positive 0.833
##  5 realDonaldTrump.2020/01/02 positive 0.364
##  6 JoeBiden.2020/01/03        positive 0.35
##  7 realDonaldTrump.2020/01/03 positive 0.522
##  8 JoeBiden.2020/01/04        positive 0.467
##  9 realDonaldTrump.2020/01/04 positive 0.353
## 10 JoeBiden.2020/01/05        positive 0.125
## # ... with 694 more rows
```

Wie oben können wir nun einen Plot erzeugen. Da sich aus dem Anteil des positiven Sentiments automatisch Anteil des negativen Sentiments (da sich beide Werte zu 1 bzw. 100% aufsummieren) ergibt, ist es hier wenig sinnvoll, beide Werte zu plotten; wir filtern daher nur eine der beiden Kategorien an. Ich zeichne außerdem eine horizontale Linie bei 50% ein, um den Übergang von mehrheitlich positivem zu mehrheitlich negativen Sentiment zu kennzeichnen.

```
sentiment_ratio_per_day %>%
  tidy() %>%
  separate(document, c("account", "day"), sep = "\\.") %>%
  mutate(day = as.Date(day)) %>%
  filter(term == "positive") %>%
  ggplot(aes(x = day, y = count, color = account, group = account)) +
  geom_line() +
  geom_hline(aes(yintercept = 0.5), linetype = "dashed") +
  scale_color_manual(values = c("blue", "red")) +
  scale_x_date(date_breaks = "1 month", date_labels = "%b %Y") +
  scale_y_continuous(labels = scales::percent, limits = c(0,1)) +
  theme_bw() +
  theme(legend.position = "bottom") +
  labs(x = "Datum", y = "Anteil positives Sentiment", color = "Twitter-Account")
```



Wir sehen, dass beide Kandidaten mehrheitlich *positiv* twittern, es aber durchaus auch negative Ausschläge gibt; auch hier könnten wir nun im nächsten Schritt versuchen, diese Ausschläge auf bestimmte Ereignisse zurückzuführen.

19.3 Gewichtete Lexika

Bisher haben wir mit kategorialen Lexika gearbeitet, die alle Begriffe einer Kategorie gleichermaßen gewichten. Für manche Anwendungen ist das aber eine zu simplifizierende Annahme, da man davon ausgehen kann, dass bestimmte Begriffe stärker mit einem Konstrukt in Verbindung stehen als andere; so dürfte beispielsweise das Wort "hate" auf einen stärkeren negativen Affekt hinweisen als das Wort "dislike". In gewichteten Lexika wird dies versucht zu berücksichtigen, indem jeder Begriff einen numerischen Wert zugeteilt bekommt, der die Stärke der Assoziation mit dem übergeordneten Konstrukt ausdrückt.

Quanteda bietet aktuell (noch) keine einfache Möglichkeit, mit gewichteten Lexika zu arbeiten. Dies ist daher ein guter Zeitpunkt, uns den bisher bekannten Workflow in einem anderen Package, dem `tidytext`-Package anzusehen. Praktischerweise enthält `tidytext` eines der bekanntesten gewichteten Lexika, das AFINN von Finn Årup Nielsen, mit dem ebenfalls positives bzw. negatives Sentiment erfasst werden soll. Wir können uns dieses Lexikon mit der Funktion `get_sentiments("afinn")` anzeigen lassen:[2]

[2]Falls eine Fehlermeldung auftritt, muss zunächst noch das `textdata`-Package installiert werden: `install.packages("textdata")` und das Lexikon anschließend heruntergeladen wer-

```
get_sentiments("afinn")
```

```
## # A tibble: 2,477 x 2
##    word       value
##    <chr>      <dbl>
##  1 abandon       -2
##  2 abandoned     -2
##  3 abandons      -2
##  4 abducted      -2
##  5 abduction     -2
##  6 abductions    -2
##  7 abhor         -3
##  8 abhorred      -3
##  9 abhorrent     -3
## 10 abhors        -3
## # ... with 2,467 more rows
```

Wie wir sehen, ist das Lexikon als Tibble mit zwei Spalten hinterlegt: `word` enthält die einzelnen Begriffe (2477 an der Zahl, also deutlich weniger umfangreich als das Sentiment Lexicon), `value` den zugehörigen numerischen Wert, wobei das Sentiment von `-5` (stark negativ) bis `+5` (stark positiv) skaliert ist.

Der Textkorpus wird in Tidytext durch ein Tibble, in dem je Zeile ein Dokument hinterlegt ist, repräsentiert – in unserem Fall also durch das Objekt `tweets`, das wir ganz zu Beginn des Kapitels eingelesen haben. Die Tokenisierung erfolgt durch die Funktion `unnest_tokens()`, wobei der Name der neu zu erstellenden Token-Variable (in diesem Fall `word`) und der Name der Textvariable im Ausgangsdatensatz (in diesem Fall `content`) angegeben werden muss. Das Resultat ist ein Tibble, in dem nun jede Zeile für ein Wort eines Dokuments steht – aus ursprünglich 4153 Zeilen (= Dokumente bzw. Tweets) sind nun 119.956 Zeilen (= einzelne Wörter) geworden:

```
tidy_tweets <- tweets %>%
  unnest_tokens(word, content) %>%
  select(id, account, day, word)   # Auswahl der für uns relevanten Variablen
tidy_tweets
```

```
## # A tibble: 119,956 x 4
##       id account  day        word
##    <dbl> <chr>    <chr>      <chr>
##  1     1 JoeBiden 2020/01/01 our
##  2     1 JoeBiden 2020/01/01 final
##  3     1 JoeBiden 2020/01/01 fundraising
```

den.

```
##  4     1 JoeBiden 2020/01/01 deadline
##  5     1 JoeBiden 2020/01/01 of
##  6     1 JoeBiden 2020/01/01 2019
##  7     1 JoeBiden 2020/01/01 is
##  8     1 JoeBiden 2020/01/01 just
##  9     1 JoeBiden 2020/01/01 hours
## 10     1 JoeBiden 2020/01/01 away
## # ... with 119,946 more rows
```

Im Gegensatz zu Quanteda müssen wir für die Sentiment-Analyse keine DFM erstellen.[3] Sowohl unsere Tokens als auch das AFINN-Dictionary liegen uns als Tibbles vor. Diese sollen nun miteinander verbunden werden, sodass wir die Werte aus dem AFINN-Dictionary mit dem Inhalt der Tweets verknüpfen können. Zwei Tibbles miteinander verbinden – das ruft nach Join-Operationen (siehe Kapitel 10.2.2.2).

In diesem Fall ist ein `inner_join()` angebracht. Wir gleichen also für jede Zeile im Tokens-Tibble ab, ob sich ein passender Eintrag dafür im AFINN-Dictionary findet; falls ja, wird der zugehörige Wert angefügt, falls nein, fliegt die Zeile aus dem Datensatz. Das Resultat ist ein verbundenes Tibble mit nun 10.132 Zeilen – unter den rund 120,000 Tokens insgesamt sind also 10.132, für die eine entsprechende emotionale Polarität im 2.477 Einträge umfassenden AFINN-Dictionary gefunden wurde.

```
tidy_sentiments <- tidy_tweets %>%
  inner_join(get_sentiments("afinn"))
```

```
## Joining, by = "word"
```

```
tidy_sentiments
```

```
## # A tibble: 10,132 x 5
##       id account  day        word       value
##    <dbl> <chr>    <chr>      <chr>      <dbl>
##  1     1 JoeBiden 2020/01/01 help           2
##  2     1 JoeBiden 2020/01/01 big            1
##  3     1 JoeBiden 2020/01/01 helping        2
##  4     1 JoeBiden 2020/01/01 help           2
##  5     1 JoeBiden 2020/01/01 reach          1
##  6     2 JoeBiden 2020/01/01 poor          -2
##  7     2 JoeBiden 2020/01/01 powerless     -2
##  8     2 JoeBiden 2020/01/01 vulnerable    -2
##  9     2 JoeBiden 2020/01/01 faith          1
## 10     3 JoeBiden 2020/01/02 join           1
## # ... with 10,122 more rows
```

[3] Dies geht in Tidytext mit der Funktion `cast_dfm()`.

Von hier an können wir Auswertungen mit den uns bekannten Tidyverse-Funktionen vornehmen. Um etwa pro Account und Tag das durchschnittliche Sentiment der Tweets zu berechnen, gruppieren wir zunächst mittels `group_by()` nach `account` und `day` und berechnen anschließend den Mittelwert des Sentiments mit `summarise()`.[4]

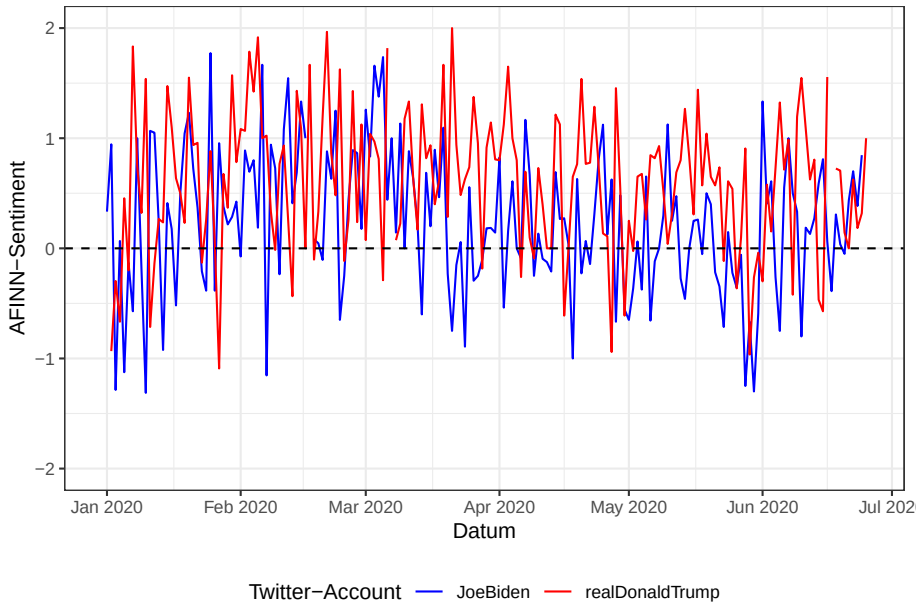
```
afinn_per_day <- tidy_sentiments %>%
  group_by(account, day) %>%
  summarise(mean_sentiment = mean(value), .groups = "drop")

afinn_per_day
```

```
## # A tibble: 354 x 3
##    account  day        mean_sentiment
##    <chr>    <chr>               <dbl>
##  1 JoeBiden 2020/01/01         0.333
##  2 JoeBiden 2020/01/02         0.944
##  3 JoeBiden 2020/01/03        -1.29
##  4 JoeBiden 2020/01/04         0.0667
##  5 JoeBiden 2020/01/05        -1.12
##  6 JoeBiden 2020/01/06        -0.133
##  7 JoeBiden 2020/01/07        -0.571
##  8 JoeBiden 2020/01/08         1
##  9 JoeBiden 2020/01/09        -0.192
## 10 JoeBiden 2020/01/10        -1.31
## # ... with 344 more rows
```

Auch hier bietet sich natürlich wieder eine grafische Darstellung an:

```
afinn_per_day %>%
  mutate(day = as.Date(day)) %>%
  ggplot(aes(x = day, y = mean_sentiment, color = account, group = account)) +
  geom_line() +
  geom_hline(aes(yintercept = 0), linetype = "dashed") +
  scale_color_manual(values = c("blue", "red")) +
  scale_x_date(date_breaks = "1 month", date_labels = "%b %Y") +
  scale_y_continuous(limits = c(-2,2)) +
  theme_bw() +
  theme(legend.position = "bottom") +
  labs(x = "Datum", y = "AFINN-Sentiment", color = "Twitter-Account")
```

[4] Das uns noch unbekannte Argument `.groups` ist ein neues Argument ab der `dplyr`-Version `1.0.0`, das es ermöglicht, vorhandene Gruppierungen nach dem Zusammenfassen mit dem Wert `"drop"` auch ohne anschließendes `ungroup()` aufzuheben oder mit dem Wert `"keep"` beizubehalten.



Das berechnete Sentiment der Tweets mittels AFINN-Dictionary ist also durchaus mit der vorherigen Sentiment-Analyse, basierend auf Bing Lius Sentiment Lexicon, vergleichbar: auch hier überwiegt bei beiden Kandidaten positives Sentiment, die Ausschläge nach unten kommen zu denselben Zeitpunkten, wenn auch die absoluten Wertausprägungen natürlich unterschiedlich (und auch unterschiedlich skaliert) sind.

19.4 Übungsaufgaben

Erstellen Sie für die folgenden Übungsaufgaben eine eigene Skriptdatei oder eine R-Markdown-Datei und speichern diese als `ue19_nachname.R` bzw. `ue19_nachname.Rmd` ab.

Laden Sie den Datensatz `facebook_europawahl.csv` und filtern Sie lediglich Posts der im Bundestag vertretenen Parteien.

Für diese Übungsaufgaben wechseln wir also die Sprache. Entsprechend benötigen wir auch ein neues Lexikon: Der SentimentWortSchatz (kurz SentiWS) von Robert Remus, Uwe Quasthoff und Gerhard Heyer enthält die positive bzw. negative Polarität (skaliert von `-1` bis `1`) von rund 3.500 deutschen Wörtern + zugehörige Flexionsformen, zusammen also rund 34.000 Wörter. Die aktuelle Version (v2.0) lässt sich auf der oben verlinkten Seite kostenfrei herunterladen.

Positive und negative Wörter liegen in zwei Textdateien ab, deren Import nicht ganz trivial ist. Mit folgendem Code (Dateipfad natürlich eventuell anpassen)

werden beide Wörterbücher geladen und in einem Tibble `sentiws` vereint. Natürlich schadet es nicht, die einzelnen Schritte selbst nachzuvollziehen.

```
sentiws_pos <- read_delim("data/SentiWS_v2.0_Positive.txt", col_names = c("word", "value", "flect
  mutate(sentiment = "positive")
sentiws_neg <- read_delim("data/SentiWS_v2.0_Negative.txt", col_names = c("word", "value", "flect
  mutate(sentiment = "negative")

sentiws <- sentiws_pos %>%
  bind_rows(sentiws_neg) %>%
  separate(word, c("word", "type"), sep = "\\|") %>%
  mutate(word = str_c(word, flections, sep = ",")) %>%
  select(-flections, -type) %>%
  separate_rows(word, sep = ",") %>%
  na.omit()
```

Übungsaufgabe 19.1. Sentiment-Analyse mit Quanteda:

Erstellen Sie ein Quanteda-Dictionary mit den Kategorien `positiv` und `negativ` aus den SentiWS-Lexikon. Gibt es Unterschiede zwischen den Parteien hinsichtlich des Sentiments ihrer Posts zur Europawahl?

- Um die beiden Wortlisten aus dem Tibble als Vektoren zu extrahieren, können Sie mit den Tidyverse-Funktionen `filter()` und `pull()` arbeiten.
- (Wann) Ist es sinnvoll, die Wortlisten in Kleinschreibung zu konvertieren?

Übungsaufgabe 19.2. Sentiment-Analyse mit Tidytext:

Berechnen Sie das durchschnittliche Sentiment pro Partei auf Basis der Polaritäts-Werte (`value`) in SentiWS. Unterscheiden sich die Ergebnisse von der obigen Variante, bei der nur die Kategorien verwendet werden?

Chapter 20

Textklassifikation durch überwachtes maschinelles Lernen

In diesem Kapitel wenden wir uns der Textklassifikation zu, also der automatisierten Einteilung von Textdokumenten in Klassen, die uns interessierende Kategorien repräsentieren. Zugleich ist dieses Kapitel unser Einstieg in das maschinelle Lernen, bei dem mittels Algorithmen und statistischen Modellen Muster in Daten "erlernt" werden, um z. B. auf Basis dieser Muster Fälle zu gruppieren oder zuzuordnen.

Genauer gesagt steigen wir in Verfahren des *überwachten* maschinellen Lernens (*supervised machine learning*) ein. Dies bedeutet, dass wir einen bereits annotierten Datensatz verwenden, um Modelle zu trainieren; wir geben also vor, welche Klassen in den Daten existieren und versuchen Muster zu finden, die die Zuordnung zu diesen Klassen ermöglichen. Anschließend kann das trainierte Modell auf einen nicht annotierten Datensatz angewendet werden, um dort die Klassenzugehörigkeit vorherzusagen. Bei Verfahren des *unüberwachten* maschinellen Lernens (*unsupervised machine learning*) sind diese Klassen hingegen nicht vorgeben und die Modelle sollen die Daten selbst einteilen.

Im Kontext der automatisierten Inhaltsanalyse wird überwachtes maschinelles Lernen vor allem dann eingesetzt, wenn ein Teildatensatz eines großen Textkorpus in einer manuellen Inhaltsanalyse annotiert wurde (z. B. ob ein Text populistisch ist oder nicht, ob er Hassrede enthält, oder ob er zu einem von zuvor festgelegten Themen gehört); auf Basis dieses Teildatensatzes wird nun ein Modell trainiert, das anschließend den gesamten Datensatz möglichst reliabel codieren kann. Zudem können die trainierten Modelle genutzt werden, um ein Verständnis über die erlernten Muster zu erhalten – welche *Features* (bei Textdaten also vor allem N-Gramme) tragen auf welche Art dazu bei, dass

das Modell zu einem bestimmten Klassfikationsergebnis kommt, welche Wörter sind also beispielsweise besonders gut dazu geeignet, den Populismusgehalt von Texten vorherzusagen.

Im Folgenden trainieren wir Modelle, die Anhand des bekannten Tweet-Datensatzes die Account-Zugehörigkeit vorhersagen sollen. Der Datensatz ist bereits annotiert, da wir für jeden Tweet auch den Account angeben haben. Zudem handelt es sich um die einfachst mögliche Klassfikation, da nur zwei verschiedene (Account-)Klassen im Datensatz vorkommen: Donald Trump ("realDonaldTrump") oder Joe Biden ("JoeBiden"). Natürlich ist es aber auch möglich, Modelle zur Klassifikation in mehr als zwei Klassen zu trainieren.

Wir beginnen wie immer damit, unsere bekannten Packages sowie den Datensatz zu laden. Während eine Zeit lang Verfahren zur Textklassifikation in Quanteda von Haus aus enthalten waren, sind diese inzwischen in ein eigenes Package, `quanteda.textmodels`, ausgelagert. Wir müssen dieses also vorab noch einmalig mit `install.packages("quanteda.textmodels")` installieren.

```
library(quanteda)
library(quanteda.textmodels)
library(tidyverse)


tweets <- read_csv("data/trump_biden_tweets_2020.csv")
```

Nochmals ein kurzer Überblick über unseren Datensatz – zentral sind hier die Variablen `account` und `content`. Wir werden also versuchen, ein Modell zu trainieren, das auf Basis von `content`, dem Inhalt der Tweets, den `account` hinreichend zuverlässig vorhersagen kann.

```
tweets
```

```
## # A tibble: 4,153 x 7
##       id account  link                    content
##    <dbl> <chr>    <chr>                   <chr>
##  1     1 JoeBiden https://twitter.com/Joe~ "Our final fundraising deadline of 2019 i
##  2     2 JoeBiden https://twitter.com/Joe~ "Every single human being deserves to be
##  3     3 JoeBiden https://twitter.com/Joe~ "With just over one month until the Iowa
##  4     4 JoeBiden https://twitter.com/Joe~ "This election is about the soul of our n
##  5     5 JoeBiden https://twitter.com/Joe~ "Every day that Donald Trump remains in t
##  6     6 JoeBiden https://twitter.com/Joe~ "It was a privilege to work with @Julian0
##  7     7 JoeBiden https://twitter.com/Joe~ "Like Vicky said, we need a president who
##  8     8 JoeBiden https://twitter.com/Joe~ "I'm excited to share that we raised $22
##  9     9 JoeBiden https://twitter.com/Joe~ "If you're a teacher or a firefighter, yo
## 10    10 JoeBiden https://twitter.com/Joe~ "Before the holidays, Jill walked across
## # ... with 4,143 more rows
```

Auch die nächsten Schritte sind uns bereits bekannt – wir erstellen eine Korpus, Tokens und abschließend eine DFM. Wir fügen mit dem Befehl `docvars(tweet_corpus, "tweet_id") <- docid(tweet_corpus)` außerdem unserem Korpus eine zusätzliche Docvar `"tweet_id"` hinzu, in der wir die ID des jeweiligen Tweets festhalten. Dies erleichert uns nachher die Auswahl einzelner Dokumente im Korpus.

```
# Korpus

tweet_corpus <- corpus(tweets, docid_field = "id", text_field = "content")
docvars(tweet_corpus, "tweet_id") <- docid(tweet_corpus)

# Tokens

tweet_tokens <- tokens(tweet_corpus,
                       remove_punct = TRUE,
                       remove_numbers = TRUE,
                       remove_symbols = TRUE,
                       remove_url = TRUE) %>%
  tokens_tolower() %>%
  tokens_remove(stopwords("english"))


# DFM

tweet_dfm <- dfm(tweet_tokens)
```

Im Übrigen: Wenn wir bisher von *Features* gesprochen haben, dann haben wir den Begriff weitestgehend synonym mit N-Grammen, bzw. im Falle von Unigrammen auf Wortebene, mit Wörtern verwendet. Das ist bei der automatisierten Inhaltsanalyse auch die häufigste Art von Feature, tatsächlich kann ein Feature aber jede beliebige Eigenschaft eines Dokuments sein; wir könnten unsere aktuell rein aus Wörtern bestehende Feature-Liste also problemlos um weitere Texteigenschaften (z. B. Länge in Zeichen, Anzahl Substantive, Anzahl Großbuchstaben etc.) oder weitere Dokumenteigenschaften (z. B. Anzahl der Retweets) erweitern.

20.1 Training- und Test-Datensätze

Eines der häufigsten Probleme beim Einsatz von überwachtem maschinellen Lernen ist *Overfitting*: das Modell wird sehr gut an den annotierten Datensatz angepasst, performt aber bei neuen Datensätzen schlecht, da es beispielsweise nicht zwischen allgemeingültigen und datensatzspezifischen Mustern unterscheiden kann. In der Regel sollten wir beim überwachten maschinellen Lernen

ein Klassifikationsmodell daher nicht nur trainieren, sondern auch dessen Güte testen.

Hierzu wird der annotierte Datensatz einem *Train-Test-Split* unterzogen: man teilt den Datensatz in zwei Teildatensätze, einen *Trainings*-Datensatz sowie einen *Test*-Datensatz auf. Das Modell wird anschließend nur anhand des Trainings-Datensatzes trainiert. Zum Überprüfen der Klassifikationsgüte werden dann die Klassen des Test-Datensatzes mit dem trainierten Modell vorhergesagt und anschließend mit den tatsächlichen, annotierten Klassen verglichen. So sehen wir, ob unser trainiertes Modell auch außerhalb des Trainings-Datensatzes zufriedenstellend performt.

Das richtige Verhältnis von Trainings- zu Test-Datensatz ist eine Wissenschaft für sich. Als Faustregel gilt jedoch, dass man meistens mit einem 80/20-Split (80% der Fälle in den Trainings-, 20% der Fälle in den Test-Datensatz) ganz ordentlich fährt. Im Folgenden wählen wir daher zunächst 80% der Tweet-IDs zufällig aus unserem Ursprungsdatensatz aus.

- Mit der Tidyverse-Funktion `slice_sample()` wählen wir zufällige Zeilen eines Datensatzes aus. Mit dem Argument `prop` legen wir fest, welchen Anteil der Zeilen wir auswählen möchten, in unserem Fall also 80% bzw. `.8`.[1]
- Anschließend extrahieren wir mit der `pull()`-Funktion die Variable `id` als Vektor
- Zuvor wird mit der Funktion `set.seed()` ein Seed für den Zufallsgenerator festgelegt. Das führt dazu, dass die nachfolgende Zufallsauswahl reproduzierbar wird – wenn Sie zu Hause den Code ausfällen, werden also *zufällig* exakt die gleichen IDs ausgewählt wir hier im Beispiel. Führen Sie den Code ohne `set.seed()` aus, erhalten Sie andere zufällig ausgewählte IDs. Die Zahl `667` als Seed ist dabei völlig willkürlich gewählt; geben Sie eine andere Zahl als Seed ein, lässt sich eine andere Zufallsauswahl reproduzieren.

```
set.seed(667)
train_ids <- slice_sample(tweets, prop = .8) %>%
  pull(id)
```

Das Resultat ist ein Vektor `train_ids`, der zufällig ausgewählte Tweet-IDs in unserem Datensatz enthält. Er umfasst 3322 Elemente (zur Erinnerung: der Ausgangsdatensatz enthält 4143 Zeilen; `3322 / 4143 = 0.802`).

```
head(train_ids)
length(train_ids)
```

[1] Alternativ lässt sich mit dem Argument `n` eine genaue Anzahl an auszuwählenden Zeilen angeben

```
## [1] 2605 2926 1353  505 6272 3347
## [1] 3322
```

Mit der Quanteda-Funktion `dfm_subset()` können wir unsere DFM `tweet_dfm` nun anhand dieser IDs in einen Trainings- und einen Testdatensatz aufteilen. Hierzu wählen wir für die Trainings-DFM alle Tweet-IDs aus, die in unserem neu erzeugten Vektor `train_ids` enthalten sind, und für Test-DFM alle Tweet-IDs, die darin *nicht* enthalten sind:

```
train_dfm <- dfm_subset(tweet_dfm, tweet_id %in% train_ids)
test_dfm <- dfm_subset(tweet_dfm, !tweet_id %in% train_ids)
```

Wie wir sehen, enthält die Trainings-DFM `train_dfm` wie gewünscht 3322 Zeilen:

```
train_dfm
```

```
## Document-feature matrix of: 3,322 documents, 8,534 features (99.8% sparse) and 6 docvars.
##     features
## docs final fundraising deadline just hours away need help every donation
##    1     1           1        1    1     1    1    1    2     1        1
##    2     0           0        0    0     0    0    0    0     1        0
##    3     0           0        0    1     0    0    1    0     0        0
##    4     0           0        0    0     0    0    0    0     0        0
##    6     0           0        0    0     0    0    0    0     0        0
##    7     0           0        0    0     0    0    1    0     0        0
## [ reached max_ndoc ... 3,316 more documents, reached max_nfeat ... 8,524 more features ]
```

Die Test-DFM `test_dfm` enthält die verbleibenden 831 Tweets:

```
test_dfm
```

```
## Document-feature matrix of: 831 documents, 8,534 features (99.8% sparse) and 6 docvars.
##     features
## docs final fundraising deadline just hours away need help every donation
##   5     0           0        0    0     0    0    0    0     1        0
##   8     0           0        0    0     0    0    0    0     0        0
##   20    0           0        0    0     0    0    0    0     0        0
##   24    0           0        0    0     0    0    0    0     0        0
##   26    0           0        0    0     0    1    0    0     0        0
##   30    0           0        0    0     0    0    1    0     0        0
## [ reached max_ndoc ... 825 more documents, reached max_nfeat ... 8,524 more features ]
```

Die Anzahl der Features ist bei beiden DFMs weiterhin gleich; das ist wichtig, da wir ein Modell, das mit bestimmten Features trainiert wurde, nur auf Daten anwenden können, die ebenfalls Informationen zu exakt diesen Features enthalten.

20.2 Naive Bayes-Klassifikation

Wir sind nun bereit, unser erstes Klassifikationsmodell zu trainieren. Hierzu nutzen wir einen sogenannten Naïve Bayes classifier. Der Name ergibt sich zum einen aus Satz von Bayes, den sich das Klassifikationsmodell zu Nutze macht, zum anderen von der "naiven" Annahme, dass alle Features voneinander unabhängig sind. Dass dies bei Textdaten in der Regel nicht der Fall ist, haben wir bereits daran gesehen, dass die Auftrittswahrscheinlichkeit von bestimmten Begriffen (z. B. "oval") durchaus vom Auftreten anderer Begriffe (z. B. "office") beeinflusst wird; dennoch liefert der naive Bayes-Klassifikator häufig gute Ergebnisse, die mit anderen, deutlich komplexeren Klassifikationsmodellen mithalten können. Zugleich hat naive Bayes-Klassifikation die Vorteile, dass sie nicht sehr rechenaufwändig ist und oft auch bereits mit geringen Datenmengen ganz passable Ergebnisse liefert.

Ich erspare an dieser Stelle detaillierte Formeln, aber grob gesagt funktioniert der Klassifikator wie folgt:

- für jede Klasse wird eine Grundwahrscheinlichkeit (*Prior*) angenommen; in der Regel nutzt man hierfür die relative Häufigkeit der Klassen im Korpus. In unserem Fall liegt die Grundwahrscheinlichkeit der Klasse `realDonaldTrump` über der Grundwahrscheinlichkeit der Klasse `JoeBiden`, da wir mehr Tweets von Trump als von Biden im Datensatz haben.
- nun wird für jedes Feature und jede Klasse eine Wahrscheinlichkeit zur Klassenzugehörig berechnet. In unserem Fall wird also für jedes Wort, das in unserem Korpus vorkommt, die Wahrscheinlichkeit berechnet, dass es in Tweets von Trump bzw. Biden vorkommt.[2]
- schließlich wird für jedes Dokument (hier also jeden Tweet) und für jede Klasse die Zugehörigkeitswahrscheinlichkeit berechnet. Hierzu werden je Klasse die Wortwahscheinlichkeiten miteinander und schließlich mit der Grundwahrscheinlichkeit multipliziert; der Klassifikator entscheidet sich sodann je Dokument für diejenige Klasse, die die höchste Zugehörigkeitswahrscheinlichkeit erhält.

Wir *fitten* ein naives Bayes-Klassifikations-Modell in Quanteda mit der Funktion `textmodel_nb()`. Als Input benötigen wir als erstes Argument eine DFM, anhand das Modell trainiert werden soll (also unsere `train_dfm`), und als zweites Argument die annotierten Klassenzugehörigkeiten, die in unserem Fall in der Docvar `account`, die sich auch über die `$`-Notation aufrufen lässt. Mit dem

[2] Das könnten wir, viel Zeit vorausgesetzt, problemlos von Hand machen: wir zählen, wie häufig das jeweilige Wort in den Dokumenten einer Klasse vorkommt, und teilen dies durch die Anzahl aller Wörter in der jeweiligen Klasse. Anschließend wird noch eine Korrektur vorgenommen, um 0-Wahrscheinlichkeiten (wenn ein Wort in einer Klasse gar nicht vorkommt) zu vermeiden.

Argument `prior` geben wir außerdem die Grundwahrscheinlichkeit an; Default-Wert ist hier `uniform`, also die gleiche Wahrscheinlichkeit für alle Klassen (bei zwei Klassen als jeweils 50%), aber wie oben angegeben, nutzen wir hier die relative Dokumenthäufigkeit mit dem Wert `"docfreq"`.

```
tweets_nbc <- textmodel_nb(train_dfm, train_dfm$account, prior = "docfreq")
```

Das Resultat ist ein `textmodel_nb`-Objekt, das sämtliche Informationen über unser trainiertes Modell enthält. Wir können uns die einzelnen "Bestandteile" mit `str()` anzeigen lassen. Besonders relevant ist für uns der Eintrag `param`, der die Parameterschätzer des Modells – also die Klassenzugehörigkeitswahrscheinlichkeiten für jedes Feature – enthält.

```
str(tweets_nbc, max.level = 1)
```

```
## List of 7
##  $ call        : language textmodel_nb.dfm(x = train_dfm, y = train_dfm$account, prior = "docf
##  $ x           :Formal class 'dfm' [package "quanteda"] with 8 slots
##  $ y           : Factor w/ 2 levels "JoeBiden","realDonaldTrump": 1 1 1 1 1 1 1 1 1 1 ...
##  $ distribution: chr "multinomial"
##  $ smooth      : num 1
##  $ priors      : Named num [1:2] 0.363 0.637
##   ..- attr(*, "names")= chr [1:2] "JoeBiden" "realDonaldTrump"
##  $ param       : num [1:2, 1:8534] 0.000208 0.000162 0.000139 0.000027 0.000139 ...
##   ..- attr(*, "dimnames")=List of 2
##  - attr(*, "class")= chr [1:3] "textmodel_nb" "textmodel" "list"
```

20.2.1 Vorhersagen und Klassifikationsgüte

Wie gut funtkioniert unser Modell nun? Wir haben bewusst einen Teil der Tweets als `test_dfm` zurückgehalten und nicht in das Modelltraining einbezogen. Wir können nun die `predict()`-Funktion nutzen, um Vorhersagen auf Basis unseres Klassifikationsmodells zu treffen. Hier benötigen wir als erstes Argument das Modell, mit dem wir unsere Vorhersagen treffen wollen, und geben mit dem Argument `newdata` einen Datensatz an, auf den wir unser Modell nun anwenden möchten.[3]

```
predicted_account <- predict(tweets_nbc, newdata = test_dfm)
```

Das Resultat ist ein Vektor mit den vorhersagten Klassen, in unserem Fall also Twitter-Accounts, für jeden Fall im Test-Datensatz:

[3] Die `predict()`-Funktion ist eine Funktion aus der Basisversion von R, die auf unterschiedlichste Modellobjekte angewendet werden kann.

```
head(predicted_account)
```

```
##        5        8       20       24       26       30
## JoeBiden JoeBiden JoeBiden JoeBiden JoeBiden JoeBiden
## Levels: JoeBiden realDonaldTrump
```

Um diesen auszuzählen, nutzen wir wir die `table()`-Funktion:

```
table(predicted_account)
```

```
## predicted_account
##        JoeBiden realDonaldTrump
##             307             524
```

Unser Modell klassifiziert 307 der 831 Tweets in `test_dfm` als von Joe Biden stammend, 524 werden Donald Trump zugeschrieben.

Wie verhält sich das zu den *tatsächlichen* Accounts?

```
true_account <- test_dfm$account
table(true_account)
```

```
## true_account
##        JoeBiden realDonaldTrump
##             294             537
```

Hier gibt es also Unterschiede und somit auch Fehlklassifikationen. Weitaus aussagekräftiger wird die Tabelle, wenn wir *wahre* Accounts und *vorhergesagte* Accounts gegeneinander abtragen. Eine solche Darstellung wird auch Confusion Matrix genannt:

```
conf_tab <- table(predicted_account, true_account)
conf_tab
```

```
##                  true_account
## predicted_account JoeBiden realDonaldTrump
##   JoeBiden             280              27
##   realDonaldTrump       14             510
```

Wir sehen: von den Tweets, die von Joe Biden stammen, wurden 280 auch Joe Biden zugeordnet , 14 hingegen wurden als von Donald Trump stammend klassifiziert. Ebenso wurden 510 Tweets von Donald Trump korrekt klassifiziert, wohingegen 27 fälschlicherweise Joe Biden zugeordnet wurden.

Aus den Verhältnissen in der Confusion Matrix lassen sich unterschiedliche Kennwerte zur Beurteilung der Klassifikationsgüte berechnen. Das `caret`-Package beinhaltet eine Funktion `confusionMatrix()`, die uns diese Kennwerte zusätzlich berechnet. Hierzu installieren wir zunächst das Package:

```
install.packages("caret")
```

Anschließend wird die erweiterte `confusionMatrix()` angefordert:

```
caret::confusionMatrix(conf_tab)
```

```
## Confusion Matrix and Statistics
##
##                  true_account
## predicted_account JoeBiden realDonaldTrump
##   JoeBiden             280              27
##   realDonaldTrump       14             510
##
##                Accuracy : 0.9507
##                  95% CI : (0.9337, 0.9644)
##     No Information Rate : 0.6462
##     P-Value [Acc > NIR] : < 2e-16
##
##                   Kappa : 0.8932
##
##  Mcnemar's Test P-Value : 0.06092
##
##             Sensitivity : 0.9524
##             Specificity : 0.9497
##          Pos Pred Value : 0.9121
##          Neg Pred Value : 0.9733
##              Prevalence : 0.3538
##          Detection Rate : 0.3369
##    Detection Prevalence : 0.3694
##       Balanced Accuracy : 0.9511
##
##        'Positive' Class : JoeBiden
##
```

Die Maße wurden vor allem im Kontext diagnostischer Klassifikation, z. B. in der Medizin (z. B. krank/nicht krank), entwickelt, daher arbeitet die Terminologie mit den Begriffen positiv (Merkmal vorhanden) vs. negativ (Merkmal nicht vorhanden). Im Falle binärer Klassifikation wird also eine Klasse als positiv bezeichnet, die andere als negativ. In diesem Falle gibt uns die Tabelle an, dass "JoeBiden" als positive Klasse dient.

Von besonderer Bedeutung sind für uns folgende Werte:

- *Accuracy*: wie hoch ist der Anteil korrekt klassifizierter Fälle insgesamt. In unserem Fall werden rund 95% aller Tweets korrekt klassifiziert (`(280 + 510) / 831 = 0.9507`).
- *Sensitivity* (auch *Recall*): wie hoch ist der Anteil als positiv klassifizierter Fälle an allen *tatsächlich* positiven Fällen. In unserem Fall werden ebenfalls rund 95% aller Tweets, die tatsächlich von Biden stammen, auch als "JoeBiden" klassifiziert (`280 / (280 + 14) = 0.9524`).
- *Specificity*: wie hoch ist der Anteil als negativ klassifizierter Fälle an allen *tatsächlich* negativen Fällen. In unserem Fall werden erneut rund 95% aller Tweets, die tatsächlich von Trump stammen, auch als "realDonaldTrump" klassifiziert (`280 / (280 + 14) = 0.9524`).

- *Positive Predictive Value* (PPV; auch *Precision*): wie hoch ist der Anteil positiver Fälle an allen als positiv *klassifizierten* Fällen. In unserem Fall stammen rund 91% aller Tweets, die als von Biden stammend klassifiziert wurden, auch tatsächlich von Biden (`280 / (280 + 27) = 0.9121`).
- *Negative Predictive Value* (NPV): wie hoch ist der Anteil negativer Fälle an allen als negativ *klassifizierten* Fällen. In unserem Fall stammen rund 97% aller Tweets, die als von Trump stammend klassifiziert wurden, auch tatsächlich von Trump (`510 / (510 + 14) = 0.9733`).

Insgesamt klassifiziert unser Classifier also rund 19 von 20 Tweets als korrekt; zugleich gibt es aber durchaus Klassenunterschiede, da fast alle Tweets, die als von Trump stammend klassifiziert werden, auch tatsächlich von Trump sind, wohingegen in etwa jeder zehnte Tweet, der als von Biden stammend klassifiziert wird, ebenfalls von Trump stammt.

20.2.2 Parameter extrahieren

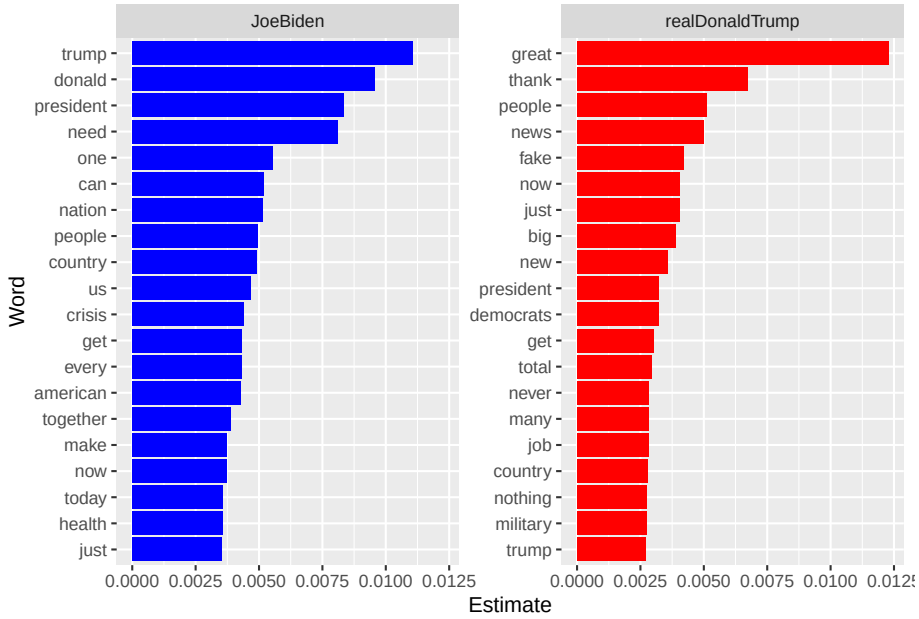
Welche Features sind nun besonders wichtig für die Klassifikation? Hierzu können wir auf die berechneten Parameter des Modells zurückgreifen, die für jedes Feature die Zugehörigkeitswahrscheinlichkeiten pro Klasse (also in unserem Fall für beide Twitter-Accounts) umfassen. Diese sind als Matrix `"param"` im Modellobjekt hinterlegt und können leicht über die `$`-Notation abgerufen werden.

```
tweets_nbc$param[, 1:5] # Anzeige der ersten fünf Spalten
```

```
##                        final  fundraising     deadline        just        hours
## JoeBiden        0.0002081382 0.0001387588 0.0001387588 0.003538349 2.081382e-04
## realDonaldTrump 0.0001622016 0.0000270336 0.0000270336 0.004028007 5.406721e-05
```

Wir können also die Features mit dem höchsten Wert pro Klasse heraussuchen, um so ein Gefühl dafür zu bekommen, welche Features besonders wichtig für die jeweiligen Klassen sind. Hier bietet sich eine grafische Darstellung an:

```
tweets_nbc$param %>%
  t() %>%  # Transponiert die Matrix, vertauscht also Zeilen und Spalten
  as_tibble(rownames = "Word") %>%
  pivot_longer(c("JoeBiden", "realDonaldTrump"), names_to = "Account", values_to = "Estimate") %>
  group_by(Account) %>%
  top_n(20, Estimate) %>%
  mutate(Word = tidytext::reorder_within(Word, Estimate, Account)) %>%
  ggplot(aes(x = Word, y = Estimate, fill = Account)) +
  facet_wrap(~ Account, scales = "free_y") +
  geom_col(show.legend = FALSE) +
  scale_fill_manual(values = c("blue", "red")) +
  tidytext::scale_x_reordered() +
  coord_flip()
```



Die Ergebnisse sind vergleichbar mit den wichtigsten Begriffen, die wir über andere Verfahren, beispielsweise Keyness-Analysen erhalten haben (siehe Kapitel 18.6). Zu beachten ist, dass bei naiver Bayes-Klassifikation Features für beide Klassen eine hohe Bedeutung haben können, da die Parameter schlichtweg anhand der relativen Häufigkeit berechnet werden, in diesem Fall z. B. “trump”. Andere Klassifikationsverfahren suchen hier stärker nach Features, die besonders distinkt für die jeweiligen Klassen sind.

20.2.3 Klassifikation unannotierter Texte

Da unser Modell eine hohe Klassifikationsgüte aufweist, können wir es nun auch guten Gewissens bei unannotierten Daten einsetzen. Natürlich wissen im Falle von Tweets stets, von wem dieser Tweet stammt, weshalb die Vorhersage unannotierter Daten in diesem Beispiel eher eine Spielerei ist. In Anwendungsfällen, in denen wir einen Teil-Datensatz manuell annotiert haben, um dann den Klassifikator auf den gesamten Datensatz anzuwenden, wäre dies aber das eigentliche Ziel. Wir gehen daher die dafür notwendigen Schritte anhand des Tweet-Beispiels durch.

Hierzu benötigen wir zunächst natürlich unannotierte Tweets, also ohne Account-Zuordnung. Bei den folgenden beiden aktuellen Tweets dürfte es uns leicht fallen, zu erahnen, ob sie von Biden oder Trump stammen. Ob unser Klassifikator das auch schafft?

```
new_tweets <- c("Big Senate Race in Alabama on Tuesday. Vote for @TTuberville, he is a
                "I want every single American to know: If you're sick, struggling, or
```

Unser Klassifikationsmodell benötigt als Input eine DFM; entsprechend konvertieren wir unsere neuen Text-Dokumente ebenfalls in eine DFM:

```
new_dfm <- dfm(new_tweets)
new_dfm
```

```
## Document-feature matrix of: 2 documents, 66 features (44.7% sparse).
##        features
## docs    big senate race in alabama on tuesday . vote for
##   text1   1      1    1  2       1  1       1 3    1   1
##   text2   0      0    0  1       0  0       0 3    0   0
## [ reached max_nfeat ... 56 more features ]
```

Allerdings enthält diese DFM natürlich andere (und deutlich weniger) Features als die DFM, anhand das Modell trainiert wurde. Nutzen wir nun die `predict()`-Funktion, erzeugt dies eine Fehlermeldung, da das Modell alle ihm bekannten Features erwartet:

```
predict(tweets_nbc, newdata = new_dfm)
```

```
## Error in force_conformance(newdata, colnames(object$param), force): newdata's featu
```

Um dieses Problem zu beheben, nutzen wir die Quanteda-Funktion `dfm_match()`, mit der wir eine bestehende DFM an eine vorgegebene Struktur anpassen können. Mit dem Argument `features` geben wir an, welche Features unsere neue DFM beinhalten soll – nämlich alle, die in der ursprünglichen Trainings-DFM enthalten sind:

```
new_dfm_matched <- dfm_match(new_dfm, features = featnames(train_dfm))
new_dfm_matched
```

```
## Document-feature matrix of: 2 documents, 8,534 features (99.8% sparse).
##        features
## docs    final fundraising deadline just hours away need help every donation
##   text1     0           0        0    0     0    0    0    0     0        0
##   text2     0           0        0    0     0    0    0    0     1        0
## [ reached max_nfeat ... 8,524 more features ]
```

Unsere neue DFM enthält nun alle Features der ursprünglichen DFM – Wörter, die in unseren beiden neuen Tweets nicht vorkommen, haben eine 0 erhalten, Wörter in unseren neuen Tweets, die nicht in der alten DFM enthalten waren, wurden gelöscht.

Sehen wir uns nun an, zu welchem Ergebnis unser Klassifikator kommt:

```
predict(tweets_nbc, newdata = new_dfm_matched)
```

```
##           text1           text2
## realDonaldTrump        JoeBiden
## Levels: JoeBiden realDonaldTrump
```

Tatsächlich wurden beide Tweets korrekt klassifiziert. Unser Modell hat also offensichtlich "erlernt", auch unannotierte Tweets von Trump und Biden zu unterscheiden.

Falls Sie das erneut mit anderen Tweets der beiden Kandidaten ausprobieren möchten – hier ist das Verfahren in eine Funktion verpackt, die Sie mit `trump_or_biden()` aufrufen und anschließend den Text eines beliebigen Tweets der beiden Kandidaten einfügen können, um den Urheber vorherzusagen:

```
trump_or_biden <- function(model = tweets_nbc) {
  text <- rstudioapi::showPrompt("Tweet-Text", "Text des Tweets:")
  new_dfm <- dfm(text)
  new_dfm_matched <- dfm_match(new_dfm, features = featnames(model))
  prediction <- predict(model, newdata = new_dfm_matched) %>%
    as.character()
  print(paste("Dieser Tweet stammt wahrscheinlich von", prediction))
}
```

20.3 Ausblick

Dies war ein erster Einblick in Textklassifikation durch überwachtes maschinelles Lernen. Unser Klassifikator funktioniert hier sehr gut, es handelt sich aber auch

um ein eher einfaches Klassifikationsproblem, da wir nur binär zwischen zwei Klassen unterscheiden und diese auch sehr verschieden sind, da beide Kandidaten einen jeweils distinkten Tweet-Stil pflegen. Im Forschungsalltag ist in der Regel deutlich mehr Aufwand erforderlich, um zufriedenstellende Ergebnisse zu erhalten.

Textklassifikation ist – neben den Ausgangsdaten – inbesondere von zwei Schritten abhängig:

1. **Preprocessing des Textmaterials**: Durch eine sorgsame Aufbereitung und Bereinigung des Textmaterials, also Schritte wie Stemming und Lemmatisierung, die Verwendung von Bi-Grammen, Tri-Grammen etc. sowie die Entfernung von Stoppwörtern (siehe Kapitel 17.2) lässt sich die Klassifikationsgüte oft deutlich verbessern. Zusätzlich können Transformationen der DFM, z. B. durch tf-idf, helfen, die Performance von Klassifikationsmodellen zu steigern. Auch unbalancierte Klassen stellen Klassifikatoren oft vor Probleme, die z. B. durch Over- und Undersampling angegangen werden können.
2. **Wahl des Klassifikationsmodells**: Wir haben mit einem denkbar simplen Klassifikationsmodell, dem naiven Bayes-Klassifikator, gearbeitet. Oftmals müssen komplexere Modelle und Algorithmen wie (logistische) Regressionen, Support Vector Machines oder Random Forests, kombiniert mit Verfahren der Regularization, Kreuzvalidierung, Techniken der automatisierten Feature-Auswahl und/oder der Kombination von verschiedenen Klassifikationsalgorithmen durch Boosting, ausprobiert werden. Je nach zur Verfügung stehender Rechenleistung und Umfang des Datenmaterials kann die Berechnung eines einzelnen solchen Modells einige Stunden bis mehrere Tage beanspruchen.

Sollte sich aus den Forschungsprojekten die Notwendigkeit von Textklassifikation ergeben, werden wir uns in der zweiten Hälfte des Masterprojekts eingehender mit solch komplexeren Verfahren beschäftigen. Zudem folgen noch Verfahren des unüberwachten maschinellen Lernens, insbesondere Topic Modeling, sowie die manuelle Validierung von automatisierten Inhaltsanalysen.

Nun aber erst einmal: schöne Semesterferien :)

Figure 20.1: Illustration von @allison_horst: https://twitter.com/allison_horst

Chapter 21

Topic Modeling

Topic Modeling ist das aktuell wohl am häufigsten eingesetzte Verfahren (bzw. genauer: Gruppe von Verfahren) der automatisierten Inhaltsanalyse in der kommunikationswissenschaftlichen Forschung. Im Gegensatz zur vorab beschriebenen Textklassifikation (siehe Kapitel 20) handelt es sich um Verfahren des *unüberwachten maschinellen Lernens* – also Verfahren, die ohne Vorabwissen in Form annotierter Klassen und mit oft nur minimalem (aber bedeutsamen) Input des Forschenden *selbstständig* Muster in Dokumenten erkennen. Topic Modeling eignet sich daher insbesondere zur Exploration und Deskription großer Textmengen.

In diesem Kapitel setzen wir uns zunächst mit den Grundlagen dieser Verfahrensgruppe auseinander und setzen dann ein eigenes Topic Model in R um. Im Vergleich zu den bisher besprochenen Verfahren sind Topic-Modeling-Verfahren sowohl mathematisch als auch interpretatorisch deutlich komplexer; wir müssen zudem auf einige Konzepte und Techniken zurückgreifen, die im Kurs schon einige Wochen zurückliegen. Es ist daher sehr sinnvoll, alle beschriebenen Schritte selbst am eigenen Rechner nachzuvollziehen und längere Pipes nach und nach auszuführen.

21.1 Grundlagen

Wer schon einmal in einer (manuellen) Inhaltsanalyse Themen in einem Textkorpus untersucht hat, weiß, wie schwierig es sein kann, den Begriff *Thema* zu definieren und operationalisieren und eine trennscharfe und eindeutige Codierung vorzunehmen. Handelt es sich bei einem Artikel zu den Corona-Maßnahmen bei Bundesligaspielen um einen Artikel aus dem Themenbereich Gesundheit, Innenpolitik oder Sport? Entsprechend finden sich in Codebüchern zum Thema oft lange Codieranweisungen mit vielen (Negativ-)Beispielen und

es steht in der Regel eine lange Schulung der Codierer*innen an, bevor die Codierung auch nur annähernd reliabel verläuft.

Topic Modeling bietet hier eine reizvolle Alternative: Themen werden strikt auf Basis von Worthäufigkeiten in den einzelnen Dokumenten vermeintlich objektiv berechnet, ganz ohne subjektive Einschätzungen und damit einhergehenden etwaigen Verzerrungen. Wie wir aber sehen werden, ist die Sache bei weitem nicht so *straightforward* – und menschlicher Input und Intepretation letztlich ebenso relevant wie bei der manuellen Themencodierung.

Wie oben bereits angeprochen, handelt es sich bei Topic Modeling um eine Gruppe von Verfahren, die ähnlichen Grundprinzipien folgen, sich aber in der genauen mathematischen Ausführung unterscheiden. Die bekanntesten dieser Verfahren sind *LDA* (Latent Dirichlet Allocation) sowie die darauf aufbauenden *CTM* (Correlated Topic Models) und *STM* (Structural Topic Models). All diesen Verfahren sind wesentliche Annahmen und Schritte gemein:

- Ein Textkorpus besteht aus D Dokumenten (z. B. Artikel oder Posts, wobei die einzelnen Dokumente als $d_1, d_2, ...$ bezeichnet werden) und V Wörtern bzw. Terms (d.h. alle Wörter, die im gesamten Korpus vorkommen, wobei die einzelnen Wörter als $w_1, w_2, ...$ bezeichnet werden). Dabei wird dem *Bag-of-Words*-Modell (siehe Kapitel 17.2) gefolgt, das heißt es zählt lediglich die Worthäufigkeit je Dokument, die syntaktischen und grammatikalischen Zusammenhänge zwischen einzelnen Wörtern werden ignoriert.
- Es wird nun angenommen, dass latente Themen K zu unterschiedlichen Anteilen in den Dokumenten D vorkommen und alle Wörter V mit unterschiedlicher Wahrscheinlichkeit zu den K Themen gehören. K muss dabei vorab vom Forschenden festgelegt werden.
- Ziel der Verfahren ist die Berechnung zweier Matrizen $D \times K$ und $V \times K$. Die erste Matrix $D \times K$ enthält für jedes einzelne Dokument d und jedes einzelne Thema k die Wahrscheinlichkeit, dass das Thema in diesem Dokument vorkommt. Analog enthält $V \times K$ für jedes einzelne Wort w und jedes einzelne Thema k die Wahrscheinlichkeit, dass das jeweilige Wort in diesem Thema vorkommt.
- Mit Hilfe dieser Matrizen können die Themen dann beschrieben und interpretiert werden. So können aus $V \times K$ die wichtigsten Wörter je Thema (d.h., die Wörter mit der höchsten konditionalen Wahrscheinlichkeit, zu einem bestimmten Thema k zu gehören) abgelesen werden; mittels $D \times K$ können Themen Dokumenten und umgekehrt zugeordnet werden, z. B. in dem für jedes Dokument d das Thema k mit der höchsten konditionalen Wahrscheinlichkeit identifiziert wird.
- Zur Berechnung dieser Matrizen wird sozusagen der umgekehrte Weg gegangen und die Erzeugung der Dokumente als statistischer Prozess beschrieben: ein Dokument wird demnach erzeugt, in dem zufällig Themen aus der zum Dokument zugehörigen Themenverteilung und Wörter aus der den Themen zugehörigen Wortverteilungen gezogen

werden. Hierzu wird das Topic Model zunächst mit zufälligen Themen- und Wortverteilungen initialisiert und dann in einem iterativen, algorithmischen Verfahren nach und nach adaptiert, bis es möglichst gut zu den Daten (dem Textkorpus) passt (d.h. die gemeinsame Likelihood der Themen- und Wortverteilungen maximiert wird).[1]

Daraus ergeben sich einige Konsequenzen für die Interpretation und Unterschiede zum gewöhnlichen Vorgehen bei einer manuellen Themenanalyse:

- Der im Topic Modeling verwendete *algorithmische* Themenbegriff unterscheidet sich von dem, was wir im intuitiven, alltäglichen Begriffsverständnis meinen, wenn wir von "Themen" besprechen (wobei wir dieses alltägliche Begriffsverständnis auch nur sehr schwer operational definieren können) und beschreibt letztlich semantische Wortgruppierungen. Das *können* je nach Textkorpus und verwendetem Verfahren *Themen* sein, die wir klassisch als Themen der Berichterstattung beschreiben würden, also z. B. Wortgruppierungen, die auf Berichterstattung zu Sport, zu Politik oder zu bestimmten Nachrichtenereignissen verweisen, aber eben auch andere Arten von Wortgruppierungen, die der Algorithmus im Textkorpus identifziert und die z. B. auf Handlungsstränge, wiederkehrende sprachliche Stilmittel etc. verweisen.
- Wo es bei der manuellen Themencodierung in der Regel darum geht, Artikel Themen eindeutig und trennscharf zuzuweisen, gehen Topic-Modeling-Verfahren von sogenannter *Mixed Membership* aus, d.h. Dokumente können in wechselnden Anteilen zu verschiedenen Themen gehören. Um Dokumenten Themen und umgekehrt zuzuordnen, müssen also manuell Entscheidungen getroffen werden, z. B. indem jedes Dokument das Thema bzw. die Themen zugeordnet bekommt, das für das jeweilige Dokument die höchste Wahrscheinlichkeit aufweist bzw. die über einem bestimmten Cutoff-Wert (z. B. 30%, 50%) liegen.
- Topic Modeling führt *immer* zu der vorgegebenen Anzahl an Themen. Ob es sich dabei auch um sinnvoll interpretierbare Themen handelt, muss manuell erörtert werden.

Wenn dies bis hierher sehr abstrakt klang, keine Sorge: Wir werden uns all diese Schritte nun an einem konkreten Beispiel genauer ansehen.

[1] Die einzelnen Topic-Modeling-Verfahren unterscheiden sich u.a. in den verwendeten Wahrscheinlichkeitsverteilungen. So wird bei der LDA die namensgebende Dirichlet-Verteilung verwendet, die mit einer Unabhängigkeitsannahme einhergeht; entsprechend sind die Themen in der LDA unabhängig voneinander, die Wahrscheinlichkeit eines Themas beeinflusst also nicht die Wahrscheinlichkeit der anderen Themen. Beim CTM hingegen wird die Logit-Normalverteilung verwendet, die korrelierte Themenverteilungen erlaubt. Entsprechend können sich hier Themenwahrscheinlichkeiten gegenseitig beeinflussen, z. B. indem das Thema "Sport" mit einer höheren Wahrscheinlichkeit für das Thema "Gesundheit" einhergeht als das Thema "Außenhandel".

21.2 Topic Modeling mit `stm`

Für das Beispiel berechnen wir ein *Structural Topic Model* mit dem Package `stm`. Falls noch nicht geschehen, muss dieses wie gewohnt installiert werden:

```
install.packages("stm")
```

Neben diesem Package benötigen wir außerdem einige bereits bekannte Packages: das `tidyverse` zum allgemeinen Datenhandling und für Grafiken sowie `tidytext` und `quanteda` für die Arbeit mit Textdaten:

```
library(tidyverse)
library(tidytext)
library(quanteda)
library(stm)
```

Im Folgenden replizieren wir Teile der Analyse des Papers Whose ideas are worth spreading? The representation of women and ethnic groups in TED talks von Carsten Schwemmer und Sebastian Jungkunz, in dem Zusammenhänge von Ethnie und Geschlecht der Sprecher*innen aller TED Talks zwischen 2006 und 2017 mit den Themen der Talks untersucht werden. Der Datensatz für die Analyse wurde dankenswerterweise im Harvard Dataverse öffentlich zugänglich gemacht – wir benötigen lediglich die Datei `ted_main_dataset.tab`, die die Transkripte aller Talks enthält.[2]

21.2.1 Vorbereitung: Daten laden und Preprocessing

Die Dateiendung `.tsv` dürfte vielen noch unbekannt sein, es handelt sich jedoch um ein Dateiformat, das dem bereits bekannten CSV-Format sehr ähnlich ist – nur werden die Werte nicht durch Kommas, sondern durch Tabstopps voneinander getrennt. Auch für dieses Dateiformat gibt es eine passende `read_`-Funktion:

```
ted_talks <- read_tsv("data/ted_main_dataset.tsv")
```

Wir nehmen zudem einige kurze Modifikationen an dem Originaldatensatz vor:

- Wir entfernen alle Talks, zu denen keine Speaker-Daten vorliegen, um dieselbe Datenbasis wie das Paper zu haben.
- Wir erzeugen eine numerische `id` für jeden Talk.

[2] Beim Download als "Original File Format" sollte die Datei mit der Endung `.tsv` heruntergeladen werden.

- Wir wandeln die Variable `date` von einem Character- zu einem Datumsobjekt um. Hierzu nutzen wir die Funktion `ymd()` aus dem `lubridate`-Package; da im Originaldatensatz lediglich Jahr und Monat, nicht aber Tag, des jeweiligen Talks festgehalten ist (z. B. `2006-06`), geben wir mit dem Argument `truncated = 1` an, dass auf ein Datumsbestandteil (hier also der Tag) verzichtet werden kann. Das Datum wird dann automatisch auf den Monatsersten gesetzt.
- Schließlich entfernen wir der Übersicht halber alle Variablen außer dem Datum des Talks (`date`), dem `title` des Talks und dem Transkript des Talks `text`.

```
ted_talks <- ted_talks %>%
  filter(!is.na(speaker_image_nr_faces)) %>%
  mutate(id = 1:n(),
         date = lubridate::ymd(date, truncated = 1)) %>%
  select(id, date, title, text)

ted_talks
```

```
## # A tibble: 2,333 x 4
##       id date       title                              text
##    <int> <date>     <chr>                              <chr>
##  1     1 2006-06-01 Do schools kill creativity?        Good morning. How are you? It's been grea
##  2     2 2006-06-01 Averting the climate crisis        Thank you so much, Chris. And it's truly
##  3     3 2006-06-01 The best stats you've ever seen    About 10 years ago, I took on the task to
##  4     4 2006-06-01 Why we do what we do               Thank you. I have to tell you I'm both ch
##  5     5 2006-06-01 Simplicity sells                   Hello voice mail, my old friend. I've cal
##  6     6 2006-06-01 Greening the ghetto                If you're here today - and I'm very happy
##  7     7 2006-07-01 My wish: A global day of film      I can't help but this wish: to think abou
##  8     8 2006-07-01 Behind the design of Seattle's ~ I'm going to present three projects in ra
##  9     9 2006-07-01 My wish: Help me stop pandemics    I'm the luckiest guy in the world. I got
## 10    10 2006-07-01 Let's teach religion - all reli~ It's wonderful to be back. I love this wo
## # ... with 2,323 more rows
```

Insgesamt haben wir also 2333 Talks vorliegen. Als nächstes folgen die schon bekannten Schritte zur Textvorbereitung. Zunächst überführen wir unseren Datensatz in ein Quanteda-Corpus-Objekt:

```
ted_corpus <- corpus(ted_talks, text_field = "text")
```

Als nächstes erzeugen wir eine Document-Feature-Matrix und führen dabei auch einige Preprocessing-Schritte wie Konvertierung in Kleinschreibung, das Entfernen von Stopwords, Ziffern, Satzzeichen, Symbolen und URLs sowie Stemming durch – das Argument `verbose = TRUE` sorgt dafür, dass wir etwas zusätzlichen Output für die einzelnen Preprocessing-Schritte erhalten:

```
ted_dfm <- dfm(ted_corpus,
               stem = TRUE,
               tolower = TRUE,
               remove_punct = TRUE,
               remove_url = FALSE,
               remove_numbers = TRUE,
               remove_symbols = TRUE,
               remove = stopwords('english'),
               verbose = TRUE)
```

```
## Creating a dfm from a corpus input...
```

```
##  ...lowercasing
```

```
##  ...found 2,333 documents, 68,271 features
```

```
##  ...removed 175 features
##  ...stemming types (English)
##  ...complete, elapsed time: 5.74 seconds.
## Finished constructing a 2,333 x 44,692 sparse dfm.
```

Das Ergebnis ist eine sehr große DFM mit über Einhundertmillionen Zellen (2333 Dokumente mal 44692 Features). Da Topic Modeling an sich schon sehr rechenaufwändig ist, kann eine solche DFM so manchen Heimrechner in die Knie zwingen. Um die Berechnung zu vereinfachen und zu beschleunigen, lohnt es sich daher die DFM zu reduzieren. Hierzu können wir die Funktion `dfm_trim()` verwenden, mit der wir Features ausschließen können, die besonders häufig oder selten vorkommen und somit entweder zu generisch oder zu spezifisch für eine sinnvolle Interpretation sein könnten.

Zu beachten ist, dass diese Reduktion der DFM einen großen Einfluss auf das Ergebnis des Topic Modelings haben kann. Die verwendeten Werte sollten also wohlüberlegt sein und im Idealfall mit einigen alternative Berechnungen mit anderen Werten verglichen werden. Für unser Beispiel folgen wir der Analyse aus dem Paper und schließen alle Wörter bzw. Features aus, die in mehr als der Hälfte oder in weniger als einem Prozent aller Talks vorkommen:

```
ted_dfm <- dfm_trim(ted_dfm,
                    max_docfreq = 0.50,
                    min_docfreq = 0.01,
                    docfreq_type = 'prop')

ted_dfm
```

```
## Document-feature matrix of: 2,333 documents, 4,803 features (93.4% sparse) and 3 docvars.
##        features
## docs    morn blown away whole leav theme run confer relev extraordinari
##   text1    1     1    2     7    2     1   4      2     1             4
##   text2    0     1    1     0    0     0   2      1     0             0
##   text3    0     0    2     1    0     0   1      0     1             0
##   text4    0     0    0     1    2     0   1      0     0             0
##   text5    1     0    1     2    2     0   1      0     0             0
##   text6    1     0    0     0    1     0   1      1     0             0
## [ reached max_ndoc ... 2,327 more documents, reached max_nfeat ... 4,793 more features ]
```

Die resultierende DFM umfasst "nur" noch rund 11 Millionen Zellen, da wir die Feature-Zahl auf rund 4800 reduziert haben. Zu sehen ist außerdem, dass das Stemming erfolgreich war.

Das `stm`-Package arbeitet mit einem etwas anderen Dateiformat als Quanteda. Praktischerweise gibt es aber in Quanteda die Funktion `convert`, mit der Quanteda-Objekte für andere gängige Textanalyse-Packages umgewandelt werden können:

```
stm_dfm <- convert(ted_dfm, to = "stm")

str(stm_dfm, max.level = 1)
```

```
## List of 3
##  $ documents:List of 2333
##  $ vocab    : chr [1:4803] "10th" "15-year-old" "15th" "17th" ...
##  $ meta     :'data.frame':   2333 obs. of  3 variables:
```

21.2.2 Ein erstes Topic Model

Wir können nun unser erstes Modell berechnen. Der zentrale Input-Parameter ist wie oben beschrieben K, die Anzahl der Themen. Zu Demonstrationszwecken wählen wir – vollkommen willkürlich – 20, möchten also 20 Themen berechnen lassen; wir setzen uns gleich damit auseinander, wie man K sinnvoller bestimmt, aber dafür lohnt es sich, schon etwas Erfahrung mit der Modellierung und Modell-Kennwerten zu haben.

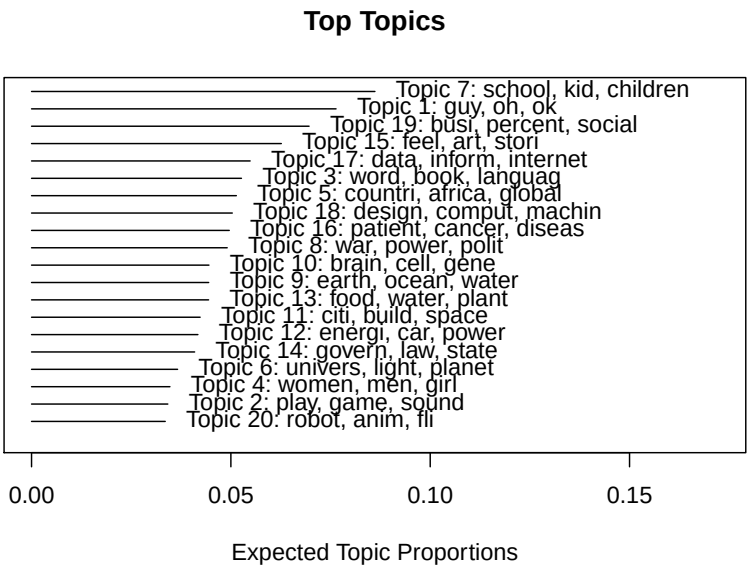
Die Modellierung erfolgt über die Funktion `stm()`. Zentrale Input-Paramter sind `documents` und `vocab`, die jeweils unter diesem Namen in unserem neu erzeugten `stm_dfm`-Objekt enhalten sind, sowie `K`, mit der wir die Anzahl der Themen bestimmen. Mit `verbose = FALSE` lasse ich für die Darstellung im Kurs den Zusatzoutput während der Berechnung ausblenden. Lassen Sie sich diesen aber gerne anzeigen, wenn Sie das Modell an Ihrem eigenen Rechner berechnen (also `verbose = TRUE`, was auch die Default-Einstellung ist), um so

die Berechnungsschritte verfolgen zu – nicht zuletzt, da die Berechnung durchaus ein paar Minuten dauern kann:[3]

```
first_model <- stm(documents = stm_dfm$documents,
                   vocab = stm_dfm$vocab,
                   K = 20,
                   verbose = FALSE)
```

Das erzeugte Objekt enthält alle relevanten Modellparameter, die wir uns in Kürze noch genauer ansehen. Wir können uns Themenverteilung und wichtigste Begriffe direkt plotten lassen:

```
plot(first_model)
```



Auch verfügt das Objekt über eine eigene `summary()`-Funktion mit den wichtigsten Wörtern je Thema, die Sie gerne am eigenen Rechner ausprobieren können (`summary(first_model)`), deren umfassender Output hier aber das Format sprengen würde.

[3] Die Modellspezifikation bietet noch deutlich mehr Einstellungsmöglichkeiten. Insbesondere lassen wir hier einen großen Vorteil von STMs gegenüber anderen Topic-Modeling-Verfahren außer Acht: wir könnten auch Kovariaten modellieren, um z. B. den Einfluss anderer Variablen im Datensatz – in der Originalstudie werden hier Ethnie und Geschlecht des Speakers sowie das Datum verwendet – auf die Prävalenz von einzelnen Themen zu untersuchen. Aus Gründen der Einfachkeit belassen wir es aber vorerst bei einem simplen Topic Model ohne Kovariaten.

Stattdessen wenden wir uns zunächst der Frage zu, anhand welcher Metriken wir ein Topic Model beurteilen können. Eine der zentralen Metriken nennt sich hierbei *Semantic Coherence*, die, vereinfacht gesagt, angibt, wie häufig Wörter, die eine hohe Wahrscheinlichkeit für ein bestimmtes Thema aufweisen, auch gemeinsam in einem Dokument (*Kookkurrenz*, siehe Kapitel 18.4) auftreten – je höher der Wert, desto häufiger ist dies der Fall. Es hat sich gezeigt, dass es menschlichen Codierer*innen mit steigender Semantic Coherence einfacher fällt, die generierten Themen auch sinnvoll zu interpretieren. Mit der Funktion `semanticCoherence()` erhalten wir diesen Wert für jedes einzelne Thema:

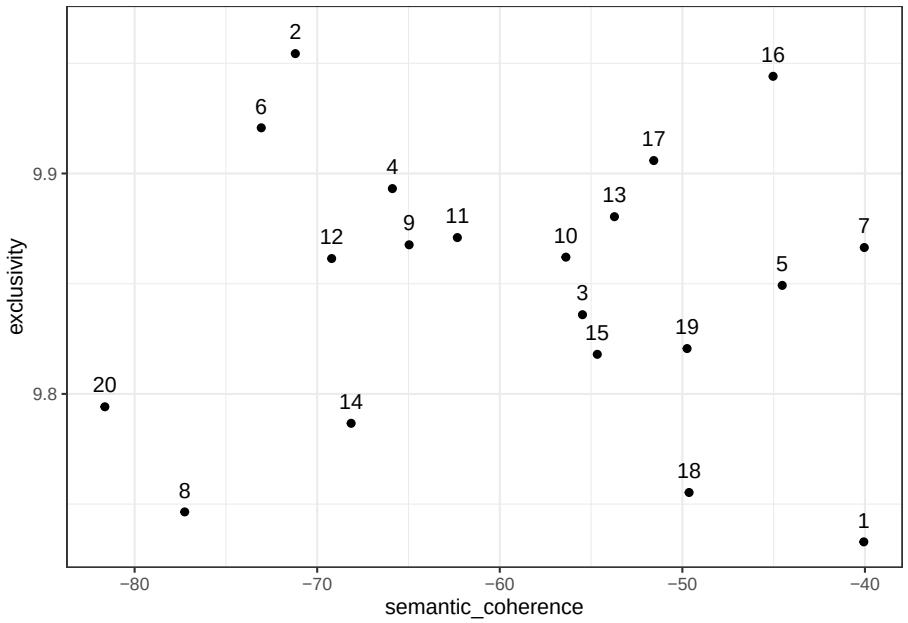
```
semanticCoherence(first_model, stm_dfm$documents)
```

```
##  [1] -40.06270 -71.19646 -55.47152 -65.88169 -44.53030 -73.06287 -40.03670 -77.25560 -64.96361
## [14] -68.14111 -54.66015 -45.03171 -51.56690 -49.63193 -49.74282 -81.62830
```

Der absolute Wert lässt sich hierbei kaum interpretieren, stattdessen bietet sich der Vergleich zwischen unterschiedlichen Themen (und vor allem: unterschiedlichen Themenmodellen an). Wir würden hier also erwarten, dass sich z. B. Thema 7 (mit dem Maximalwert von `-40.03670`) leichter interpretieren lässt als Thema 20 (mit dem Minimalwert von `-81.62830`).

Zugleich ist es vergleichsweise simpel, die Semantic Coherence zu steigern, indem eine relativ geringe Themenanzahl spezifiziert wird. Daher ist es sinnvoll, Semantic Coherence gemeinsam mit einer zweiten Messgröße zu betrachten, der *Exclusivity*. Diese gibt an, wie exklusiv die Wörter, die eine hohe Wahrscheinlichkeit für ein bestimmtes Thema aufweisen, für dieses Thema sind, zugleich also bei allen anderen Themen eine möglichst geringe Wahrscheinlichkeit aufweisen. Diese können wir mit der Funktion `exclusivity()` anfordern, wobei es sinnvoll ist, Semantic Coherence und Exclusivity gegeneinander zu plotten, um einen schnellen Vergleich zu ermöglichen:

```
tibble(
  topic = 1:20,
  exclusivity = exclusivity(first_model),
  semantic_coherence = semanticCoherence(first_model, stm_dfm$documents)
  ) %>%
  ggplot(aes(semantic_coherence, exclusivity, label = topic)) +
  geom_point() +
  geom_text(nudge_y = .01) +
  theme_bw()
```



"Gute" Themen finden wir rechts oben in der Grafik (z. B. Thema 16), diese weisen - im Vergleich zu den anderen Themen - sowohl eine hohe Semantic Coherence als auch eine hohe Exklusivität auf. Thema 1 weist zwar eine hohe Semantic Coherence auf (Wörter, die eine hohe Wahrscheinlichkeit für das Thema aufweisen, treten also auch vergleichsweise häufig gemeinsam in einem Dokument auf), hat aber zugleich eine geringe Exclusivity (Wörter, die eine hohe Wahrscheinlichkeit für das Thema aufweisen, weisen tendeziell also auch bei anderen Themen eine hohe Wahrscheinlichkeit auf). Problematisch zu interpretieren könnten entsprechend vor allem die Themen 8, 14 und 20 werden.

21.2.3 Modellvergleiche und Bestimmung von K

Nachdem wir nun einige Metriken zur Modellbeurteilung kennengelernt haben, wenden wir uns der vielleicht gewichtigsten Entscheidung bei der Berechnung von Topic Models zu: der Bestimmung einer geeigneten Themenanzahl K. Anstatt wie oben eine willkürliche Themenanzahl zu verwenden, ist es sinnvoll, bereits für das erste berechnete Modell eine begründete Wahl zu treffen:

- Ist der Untersuchungsgegenstand bzw. der Textkorpus schon bekannt (wurde z. B. an einem Teilkorpus bereits eine manuelle Inhaltsanalyse durchgeführt), kann man sich daran orientieren. Auch vergleichbare Studien bieten Anhaltspunkte – untersucht man z. B. die Berichterstattung zu einer bestimmten Wahl, hat vielleicht schon einmal jemand eine Inhaltsanalyse zu einer der vorherigen Wahlen durchgeführt und dort ebenfalls Themen ausgewertet.

- Ss existieren einige Faustregeln, die aber allenfalls grobe Anhaltspunkte darstellen. So empfehlen beispielsweise die Package-Autoren von `stm` 3-10 Themen für kleine Korpora mit sehr spezifischen Untersuchungsgegenständen (z. B. offene Antworten in einer Befragung von wenigen Hundert Personen), 5-50 Themen für Korpora mit einigen Hundert bis einigen Tausend Dokumenten, und 60-100 Themen für Korpora mit einigen Zehn- bis Hundertausend Dokumenten sowie 100 Themen für noch größere Korpora.
- Ist man völlig blank, kann man die `stm()` mit dem Argument `K = 0` ausführen; es wird dann ein Algorithmus genutzt, der eine "geeignete" Themenzahl bestimmt. Diese ist aber keinesfalls mit der "wahren" oder "besten" Themenanzahl gleichzusetzen, sondern versucht lediglich, einige Modellanpassungswerte zu maximieren.

In jedem Fall muss das Themenmodell auch manuell interpretiert werden und auf seine Sinnhaftigkeit geprüft werden. Ist eine genaue Themenanzahl vorab nicht festgelegt – was in den allermeisten Anwendungsfällen zutreffen dürfte – , sollten mehrere Modelle mit unterschiedlicher Themenanzahl gerechnet und verglichen werden (sowohl auf Basis von Kennwerten als auch manuell über die Intepretation von Themen).

R macht es uns zum Glück einfach, mehrere Modelle auf einen Schlag zu berechnen. Hierzu nutzen wir Funktionen zur Iteration und die Fähigkeit von Tibbles, so gut wie jedes Objekt – also auch Topic-Modelle – in Listen verpackt als Werte speichern zu können. Das mag auf den ersten Blick nun etwas unüblich wirken, aber erklärt sich schnell:

- Wir erzeugen zunächst ein Tibble, das in einer Variablen `K` alle unterschiedlichen Werte von K enthält, die wir nutzen möchten. In unserem Fall berechen wir insgesamt vier Modelle mit 20, 30, 40 und 50 Themen.
- Nun erzeugen wir mittels `mutate()` eine neue Variable `model`, in der die zugehörigen Modelle berechnet und gespeichert werden sollen. Hierzu nutzen wir die Funktion `map()` (siehe Kapitel 13), mit der wir über einen Vektor iterieren und die einzelnen Vektorwerte als Argument in einer Funktion verwenden können. Wir iterieren also über `K` und setzen den jeweiligen Wert von `K` an der entsprechend Stelle der `stm()`-Funktion, symbolisiert durch den `.`, ein.

Das Resultat ist ein Tibble, das all unsere Modelle enthält. Achtung: die folgende Berechnung kann, je nach vorhandener Hardware, eine ganze Zeit dauern[4] – machen Sie also ruhig einen Spaziergang o.ä., während der Rechner arbeitet.[5]

[4] auf meinem relativ modernen Heimrechner in etwa eine Viertelstunde.

[5] Eine Möglichkeit, die Berechnungszeit zu verkürzen, ist – entsprechend leistungsfähige Hardware, d.h. vor allem schnelle Mehrkernprozessoren und viel Arbeitsspeicher, vorausgesetzt – die Modelle nicht nacheinander, sondern parallel über mehrere Prozessorkerne verteilt berechnen zu lassen. Das wird natürlich umso relevanter, je größer die Textkorpora werden und je mehr unterschiedliche Modelle berechnet werden sollen. Eine genaue Erklärung würde hier zu weit führen, aber wer mag, darf sich gerne mit dem Package `furrr` auseinandersetzen.

```
many_models <- tibble(K = c(20, 30, 40, 50)) %>%
  mutate(model = map(K, ~ stm(stm_dfm$documents,
                              stm_dfm$vocab,
                              K = .,
                              verbose = FALSE)))

many_models
```

```
## # A tibble: 4 x 2
##       K model
##   <dbl> <list>
## 1    20 <STM>
## 2    30 <STM>
## 3    40 <STM>
## 4    50 <STM>
```

Das praktische ist nun, dass wir ähnlich auch über die Modelle iterieren können, um beispielsweise schnell für alle Modelle Semantic Coherence und Exclusivity zu berechnen:

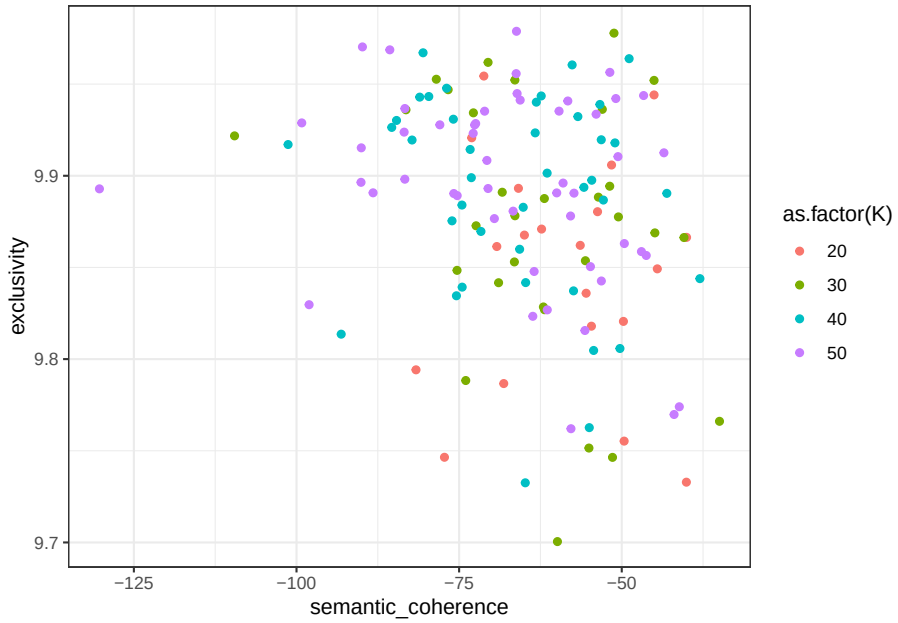
```
model_scores <- many_models %>%
  mutate(exclusivity = map(model, exclusivity),
         semantic_coherence = map(model, semanticCoherence, stm_dfm$documents)) %>%
  select(K, exclusivity, semantic_coherence)

model_scores
```

```
## # A tibble: 4 x 3
##       K exclusivity semantic_coherence
##   <dbl> <list>      <list>
## 1    20 <dbl [20]>  <dbl [20]>
## 2    30 <dbl [30]>  <dbl [30]>
## 3    40 <dbl [40]>  <dbl [40]>
## 4    50 <dbl [50]>  <dbl [50]>
```

Die jeweiligen Modell-Kennwerte stehen nun jeweils als Listen verpackt in den Zellen - für das Modell mit `K = 20` entsprechend 20 Werte für Semantic Coherence und Exclusivity, für das Modell mit `K = 30` 30 Werte etc.

Zum Modellvergleich müssen wir diese Werte nun mit der Funktion `unnest()` aus den Listen "entpacken" und können diese anschließend plotten:

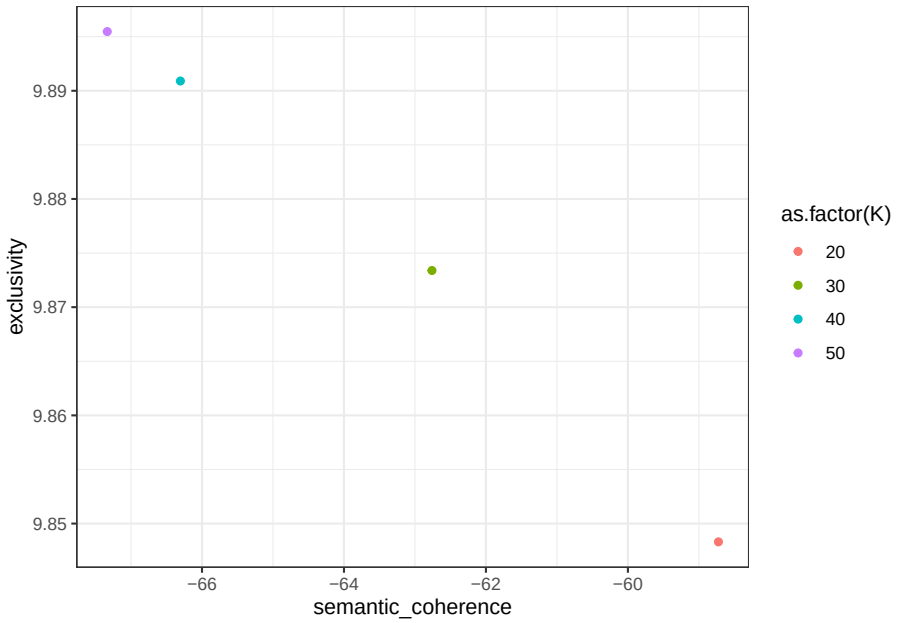
```
model_scores %>%
  unnest(c(exclusivity, semantic_coherence)) %>%
  ggplot(aes(x = semantic_coherence, y = exclusivity, color = as.factor(K))) +
  geom_point() +
  theme_bw()
```



Auf den ersten Blick unterscheiden sich die Modelle nicht sonderlich – lediglich für das Modell mit `K = 50` können wir ein klares Ausreißer-Thema mit deutlich geringerer Semantic Coherence erkennen; auch scheint es bei allen Modellen einige Themen mit verhältnismäßig geringerer Exclusivity zu geben.

Um die Modelle schneller miteinander vergleichen zu können, berechnen wir den Mittelwert der jeweiligen Kennwerte:

```
model_scores %>%
  unnest(c(exclusivity, semantic_coherence)) %>%
  group_by(K) %>%
  summarize(exclusivity = mean(exclusivity),
            semantic_coherence = mean(semantic_coherence)) %>%
  ggplot(aes(x = semantic_coherence, y = exclusivity, color = as.factor(K))) +
  geom_point() +
  theme_bw()
```



Hier zeigt sich nun recht deutlich der typische Tradeoff von Semantic Coherence und Exclusivity – das Modell mit der höchsten Semantic Coherence hat die geringste Exclusivity (`K = 20`), umgekehrt weisen sowohl `K = 40` und `K = 50` die höchste Exclusivity und geringste Semantic Coherence auf, unterscheiden sich von einander aber kaum auf den beiden Dimensionen. Als Mittelweg empfiehlt sich das Modell mit `K = 30`, das wir nun – ebenso wie die Autoren der Originalstudie – auswählen werden.

Hierzu "ziehen" wir das Modell aus dem Tibble heraus – wir filtern zunächst die entsprechende Zeile an und extrahieren den Zelleninhalt der Spalte `model` dann mittels `pull()`. Da das Modell noch in einer Liste verpackt ist, extrahieren wir danach noch das (erste und einzige) Listenelement mit `[[1]]` (der vorangestellte Punkt `.` bedeutet in etwa so viel wie "setze hier das aktuelle Objekt in der Pipe ein" – das wirkt zunächst auch etwas unüblich, ist aber die einzige Möglichkeit, Listenelemente direkt in Pipes zu anzusteuern).

```
final_model <- many_models %>%
  filter(K == 30) %>%
  pull(model) %>%
  .[[1]]

final_model
```

```
## A topic model with 30 topics, 2333 documents and a 4803 word dictionary.
```

21.2.4 Modellinterpretation

Wenden wir uns nun der Modellinterpretation zu. Wie bereits oben geschildert, sind zur Beschreibung und Interpretation eines Topic Models die Wortwahrscheinlichkeiten je Thema und die Themenwahrscheinlichkeiten je Dokument zentral. Aus ersteren können wir die gefundenen Themen inhaltlich interpretieren, zweitere geben uns Auskunft über die Prävalenz von Themen.

Einen ersten Überblick gibt uns die Funktion `labelTopics()` aus dem `stm`-Package, das uns die wichtigsten Wörter je Thema (Default-Wert: 7) anhand von vier Metriken angibt. Um die Übersichtlichkeit zu wahren, werden in der Kursansicht nur die ersten fünf Themen angezeigt; führen Sie den Code zu Hause aus, sollten Sie eine lange Ausgabe mit allen 30 Themen erhalten:

```
terms <- labelTopics(final_model)
terms
```

```
## Topic 1 Top Words:
##       Highest Prob: guy, stuff, oh, ok, sort, yeah, mayb
##       FREX: guy, oh, ok, stuff, hey, card, everybodi
##       Lift: da, dude, gotta, comedi, ass, fuck, hey
##       Score: da, guy, ok, stuff, oh, card, yeah
## Topic 2 Top Words:
##       Highest Prob: play, game, music, sound, hear, video, listen
##       FREX: music, game, song, play, musician, player, sound
##       Lift: jersey, orchestra, music, musician, piano, violin, vocal
##       Score: music, jersey, game, song, play, player, piano
## Topic 3 Top Words:
##       Highest Prob: ca, ted, yeah, chris, poem, la, mr
##       FREX: ca, la, poem, chris, ted, anderson, mr
##       Lift: ca, la, anderson, poem, chris, poetri, mr
##       Score: la, ca, poem, poetri, anderson, chris, ted
## Topic 4 Top Words:
##       Highest Prob: women, men, girl, woman, black, sex, boy
##       FREX: women, men, sexual, gender, gay, sex, girl
##       Lift: ski, gay, women, lesbian, feminist, gender, men
##       Score: women, ski, men, girl, gay, gender, sex
## Topic 5 Top Words:
##       Highest Prob: countri, africa, global, china, india, develop, percent
##       FREX: china, africa, india, aid, african, countri, incom
##       Lift: curtain, ghana, sub-saharan, china, capita, gdp, poorest
##       Score: curtain, africa, countri, india, china, african, economi
```

Zunächst zu den Metriken: `Highest Prob` bezieht sich auf die oben angesprochene Wortwahrscheinlichkeit je Thema (bezeichnet als β), angegeben sind also die sieben Wörter, die das höchste β je Thema erhalten haben. Bei den anderen

drei Metriken handelt es sich um alternative Berechnungen der bedeutsamsten Wörter je Thema; so ist `FREX` (für *Frequency-Exclusivity*) die Worthäufigkeit und -exklusivität ins Verhältnis, versucht also diejenigen Wörter zu identifzieren, die für ein bestimmtes Thema besonders distinkt sind, da sie sowohl häufig im betrachteten Thema als auch in anderen Themen selten vorkommen (ein Wort kann nämlich auch bei verschiedenen Themen ein hohes β aufweisen). Auch `Score` und `Lift` nehmen zusätzliche Gewichtungen vor. Im Idealfall stützt sich die Interpretation daher auf mehrere bzw. alle vier Metriken.

Inhaltlich sehen wir sowohl Themen, die sich relativ eindeutig einem *thematischen* Überbegriff zuordnen lassen – Talks, die Thema 2 enthalten, beschäftigen sich augenscheinlich mit Musik, bei Thema 4 mit Gender- und Sexualitätsfragen, bei Thema 5 mit Armut, vorrangig in Entwicklungsländern im globalen Süden – wie auch Themen, die offenbar gängige Sprachmuster aufgreifen (Thema 1 und 3, die in der Originalpublikation als "Miscellaneous" und "Stopwords" bezeichnet werden). Zu beachten ist außerdem, dass wir die *gestemmten* Wörter sehen – das erschwert die Interpretation bei bestimmten Wörtern wie `"ca"` und `"la"` etwas, sodass wir hier in den ungestemmten Originaldaten nachsehen könnten, auf was sich diese Wortfragmente beziehen.

Zwar können wir die gesamten Wahrscheinlichkeitsmatrizen auch direkt aus dem Modellobjekt erhalten, der Output wird jedoch besser weiterverarbeitbar, wenn wir wieder die bereits bekannte `tidy()`-Funktion aus dem `tidytext`-Package nutzen. Standardmäßig gibt uns diese die Wortwahrscheinlichkeiten je Thema β aus:

```
terms_probs <- tidy(final_model)
terms_probs
```

```
## # A tibble: 144,090 x 3
##    topic term      beta
##    <int> <chr>    <dbl>
##  1     1 10th  7.67e-15
##  2     2 10th  3.81e- 5
##  3     3 10th  4.56e-58
##  4     4 10th  2.45e-47
##  5     5 10th  4.70e-43
##  6     6 10th  1.53e- 7
##  7     7 10th  1.75e- 4
##  8     8 10th  5.03e-42
##  9     9 10th  6.31e- 5
## 10    10 10th  3.38e-25
## # ... with 144,080 more rows
```

Das Resultat ist eine lange Tabelle, in der für jedes der 4803 Wörter in unserer DFM und jedes Thema ein Wahrscheinlichkeitswert zwischen 0 und 1 angegeben wird. Alle β-Werte je Thema summieren sich zu 1 auf:

```
terms_probs %>%
  group_by(topic) %>%
  summarise(sum_beta = sum(beta))
```

```
## # A tibble: 30 x 2
##    topic sum_beta
##    <int>    <dbl>
##  1     1     1
##  2     2     1.00
##  3     3     1
##  4     4     1.00
##  5     5     1
##  6     6     1
##  7     7     1
##  8     8     1
##  9     9     1.00
## 10    10     1
## # ... with 20 more rows
```

Ebenfalls mit der `tidy()`-Funktion können wir die Themenwahrscheinlichkeiten je Dokument (bezeichnet als γ) abrufen. Hierfür müssen wir lediglich mit dem Argument `matrix = "gamma"` angeben, dass wir nun eben die γ-Werte abrufen möchten. Mit dem Argument `document_names` können wir zudem einen Vektor angeben, der die Namen der Dokumente enthält – hier bietet sich beispielsweise der Titel der Vorträge an.

```
doc_probs <- tidy(final_model, matrix = "gamma", document_names = stm_dfm$meta$title)
doc_probs
```

```
## # A tibble: 69,990 x 3
##    document                                          topic   gamma
##    <chr>                                             <int>   <dbl>
##  1 Do schools kill creativity?                           1 0.0810
##  2 Averting the climate crisis                           1 0.0964
##  3 The best stats you've ever seen                       1 0.00131
##  4 Why we do what we do                                  1 0.167
##  5 Simplicity sells                                      1 0.237
##  6 Greening the ghetto                                   1 0.0196
##  7 My wish: A global day of film                         1 0.0558
##  8 Behind the design of Seattle's library                1 0.0287
##  9 My wish: Help me stop pandemics                       1 0.0433
## 10 Let's teach religion - all religion - in schools      1 0.0112
## # ... with 69,980 more rows
```

Auch hier erhalten wir nun eine lange Tabelle, in der für jedes Dokument (in unserem Fall für jeden Vortrag) für alle identifizierten Themen eine Themenwahrscheinlichkeit angegeben ist. Auch die γ-Werte summieren sich je Dokument zu 1:

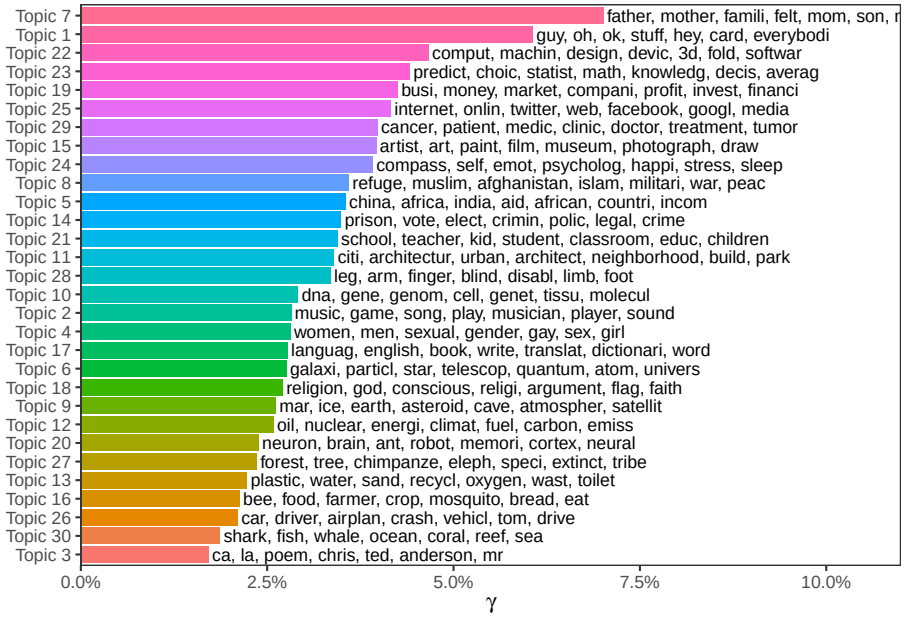
```
doc_probs %>%
  group_by(document) %>%
  summarise(sum_gamma = sum(gamma))
```

```
## # A tibble: 2,333 x 2
##    document                                          sum_gamma
##    <chr>                                                 <dbl>
##  1 "\"(Nothing But) Flowers\" with string quartet"       1.00
##  2 "\"Black Men Ski\""                                   1
##  3 "\"Clonie\""                                          1
##  4 "\"High School Training Ground\""                     1.00
##  5 "\"Kiteflyer's Hill\""                                1.00
##  6 "\"La Vie en Rose\""                                  1
##  7 "\"Love Is a Loaded Pistol\""                         1.
##  8 "\"Mother of Pearl,\" \"If I Had You\""               1.00
##  9 "\"Peace on Earth\""                                  1.
## 10 "\"Redemption Song\""                                 1.
## # ... with 2,323 more rows
```

Natürlich können wir die Daten nun auch miteinander verbinden und beispielsweise Topic-Prävalenz und bedeutsamste Wörter gemeinsam plotten:

```
top_terms <- tibble(topic = terms$topicnums,
                    prob = apply(terms$prob, 1, paste, collapse = ", "),
                    frex = apply(terms$frex, 1, paste, collapse = ", "))

gamma_by_topic <- doc_probs %>%
  group_by(topic) %>%
  summarise(gamma = mean(gamma)) %>%
  arrange(desc(gamma)) %>%
  left_join(top_terms, by = "topic") %>%
  mutate(topic = paste0("Topic ", topic),
         topic = reorder(topic, gamma))

gamma_by_topic %>%
  ggplot(aes(topic, gamma, label = frex, fill = topic)) +
  geom_col(show.legend = FALSE) +
  geom_text(hjust = 0, nudge_y = 0.0005, size = 3) +
  coord_flip() +
  scale_y_continuous(expand = c(0, 0), limits = c(0, 0.11), labels = scales::percent) +
```

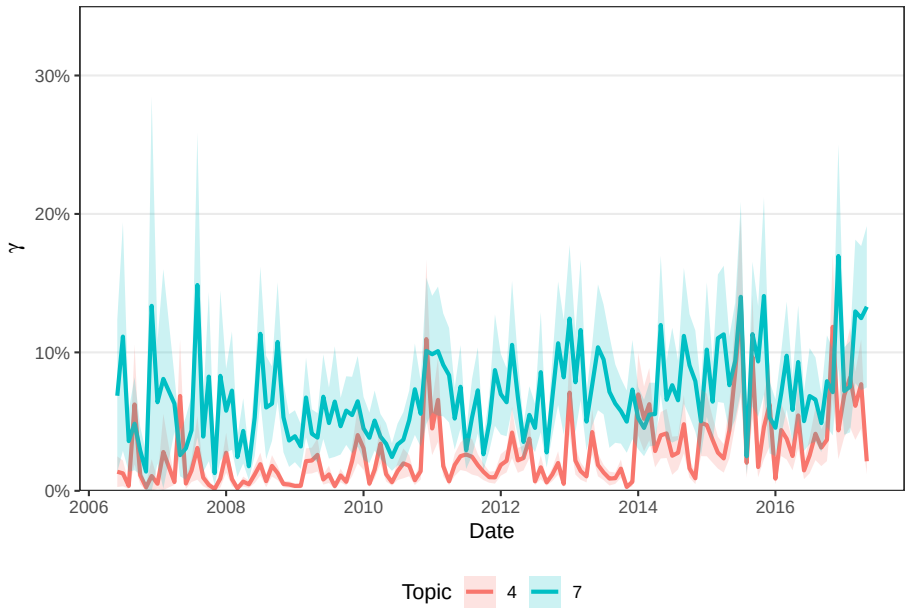
```
  theme_bw() +
  theme(panel.grid.minor = element_blank(),
        panel.grid.major = element_blank()) +
  labs(x = NULL, y = expression(gamma))
```



Auch können wir andere Informationen, die wie über unsere Dokumente haben, nun hinzuziehen und beispielsweise uns die Prävalenz einzelner Topics im Zeitverlauf ansehen:

```
doc_probs %>%
  left_join(ted_talks, by = c("document" = "title")) %>%
  group_by(topic, date) %>%
  summarise(n = n(),
            gamma = mean(gamma),
            .groups = "drop") %>%
  mutate(ci_ll = gamma - qnorm(0.975) * gamma/sqrt(n),
         ci_ul = gamma + qnorm(0.975) * gamma/sqrt(n),
         ci_ll = if_else(ci_ll < 0, 0, ci_ll),
         topic = as_factor(topic)) %>%
  filter(topic %in% c(7, 4)) %>%
  ggplot(aes(x = date, y = gamma, ymin = ci_ll, ymax = ci_ul, color = topic, fill = topic)) +
  geom_line(size = 1) +
  geom_ribbon(alpha = .2, linetype = 0) +
  theme_bw() +
  theme(panel.grid.minor = element_blank(),
```

```
        panel.grid.major.x = element_blank(),
        legend.position = "bottom") +
  scale_y_continuous(expand = c(0, 0), limits = c(0, 0.35), labels = scales::percent) +
  labs(x = "Date", y = expression(gamma), color = "Topic", fill = "Topic")
```



21.3 Übungsaufgaben

Erstellen Sie für die folgende Übungsaufgabe eine eigene Skriptdatei oder eine R-Markdown-Datei und speichern diese als `ue21_nachname.R` bzw. `ue21_nachname.Rmd` ab.

Für die Übungsaufgabe verwenden wir einen Korpus aus Artikeln des Guardian. Dieser ist in einem Zusatzpaket zu Quanteda enthalten, das einige Beispielkorpora enthält: `quanteda.corpora`. Wir können dieses Package mit folgendem Befehl installieren:

```
remotes::install_github("quanteda/quanteda.corpora")
```

Anschließend lässt sich der gewünschte Korpus über den `download()`-Befehl des Pakets herunterladen:

```
guardian_corpus <- quanteda.corpora::download("data_corpus_guardian")

guardian_corpus
```

```
## Corpus consisting of 6,000 documents and 9 docvars.
## text136751 :
## "London masterclass on climate change | Do you want to unders..."
##
## text118588 :
## "As colourful fish were swimming past him off the Greek coast..."
##
## text45146 :
## "FTSE 100 | -101.35 | 6708.35 | FTSE All Share | -58.11 | 360..."
##
## text93623 :
## "Australia's education minister, Christopher Pyne, has vowed ..."
##
## text136585 :
## "block-time published-time 3.05pm GMT | The former leader of ..."
##
## text65682 :
## "Darren Wilson will be unable to return to work as a police o..."
##
## [ reached max_ndoc ... 5,994 more documents ]
```

Wie wir sehen, handelt es sich bereits um ein Korpus-Objekt, dieser erste Konvertierungsschritt entfällt also. Enthalten sind 6,000 Artikel als Volltext.

Übungsaufgabe 21.1. Topic Modeling:

Rechnen Sie ein Topic Model mit 20 Themen. Führen Sie daher zunächst die notwendigen Preprocessing-Schritte durch.

Interpretieren Sie das vorgeschlagene Themenmodell anhand der Funktion `labelTopics()`. Können Sie die einzelnen Themen sinnvoll benennen? Gibt es Problemfälle?

Bonus: Wie verteilen sich die Themen über den Korpus? Besteht die Möglichkeit, sich die Themenprävalenz auch im Zeitverlauf anzusehen?

Chapter 22

Keyword Assisted Topic Models

Wie bereits im vorherigen Kapitel 21 geschrieben, handelt es sich bei Topic Modeling um eine ganze Klasse an ähnlichen, aber in Details auch recht unterschiedlichen Verfahren zum Ergründen von 'Themen' in Textkorpora. Eine noch sehr neue, aber besonders vielversprechende Weiterentwicklung des wegweisenden LDA-Ansatzes nennt sich *Keyword Assisted Topic Models*.

Das besondere an diesem Verfahren ist, dass es gewissermaßen die Unterscheidung zwischen überwachten und unüberwachtem maschinellen Lernen auflöst. Es können zum einen, wie auch bei anderen Topic-Modeling-Verfahren, Themen ganz automatisch in einem Textkorpus ermittelt werden; es können jedoch auch zudem – und darauf verweist der Name – auch vorab bereits Themen anhand von einigen Schlüsselwörtern definiert werden.

22.1 Keyword Assisted Topic Models in R mit `keyATM`

Durch das Package `keyATM` ist das Verfahren bereits in R implementiert. Wir installieren also zunächst das Package:

```
install.packages("keyATM")
```

Wie gehabt laden wir unsere wichtigsten Packages:[1]

[1] wir verzichten an dieser Stelle auf das `tidytext`-Package, da dieses (noch) keine `tidy()`-Funktion für Themenmodelle aufweist, die mit `keyATM` gerechnet werden. Das dürfte sich jedoch bald ändern.

```
library(tidyverse)
library(quanteda)
library(keyATM)
```

Da das Anwendungsbeispiel in der offiziellen Dokumentation von `keyATM` – Themen in der "Inaugural Address", also der Antrittsrede bei der Amtseinführung des Präsidenten der Vereinigten Staaten – so schön in die Zeit passt, sehen wir uns die Funktionsweise des Packages ebenfalls an diesem Beispiel an. Das hat den zusätzlichen Vorteil, dass ein entsprechender Textkorpus bereits zum Umfang von Quanteda gehört – wir können einen entsprechenden Korpus mit den 58 Antrittsreden aller 45 US-Präsidenten bis einschließlich Donald Trump also direkt über das Objekt `data_corpus_inaugural` verwenden, ohne Daten selbst einlesen zu müssen:

```
data_corpus_inaugural
```

```
## Corpus consisting of 58 documents and 4 docvars.
## 1789-Washington :
## "Fellow-Citizens of the Senate and of the House of Representa..."
##
## 1793-Washington :
## "Fellow citizens, I am again called upon by the voice of my c..."
##
## 1797-Adams :
## "When it was first perceived, in early times, that no middle ..."
##
## 1801-Jefferson :
## "Friends and Fellow Citizens: Called upon to undertake the du..."
##
## 1805-Jefferson :
## "Proceeding, fellow citizens, to that qualification which the..."
##
## 1809-Madison :
## "Unwilling to depart from examples of the most revered author..."
##
## [ reached max_ndoc ... 52 more documents ]
```

22.1.1 Preprocessing

Wie gehabt setzen wir alle Preprocesing-Schritte mit Quanteda um: Wir erstellen eine DFM und entfernen dabei Satzzeichen, Zahlen sowie Symbole und URLs (die in den Reden vermutlich nicht vorkommen dürften, aber da die Text-Daten von einer Website gescraped wurden, ist es dennoch sinnvoll, sicher

zu gehen); außerdem entfernen wir Stoppwörter sowie einige zusätzliche häufig vorkommenden Wörter:

```
inaug_dfm <- dfm(data_corpus_inaugural,
                 remove_punct = TRUE,
                 remove_url = FALSE,
                 remove_numbers = TRUE,
                 remove_symbols = TRUE,
                 remove = c(stopwords('english'),
                            "may", "shall", "can", "must", "upon", "with", "without"),
                 verbose = TRUE)
```

```
## Creating a dfm from a corpus input...

##  ...lowercasing

##  ...found 58 documents, 9,277 features

##  ...removed 143 features
##  ...complete, elapsed time: 0.35 seconds.
## Finished constructing a 58 x 9,135 sparse dfm.
```

Wie auch bei den anderen Modellen und Verfahren, die wir uns bisher angesehen haben, ist es erneut sinnvoll, die DFM um Wörter mit geringem Informationsgehalt (also Wörter, die sehr selten oder sehr häufig vorkommen) zu 'trimmen', um die Rechenzeit zu senken und Interpretierbarkeit zu erhöhen. Bisher haben wir anhand prozentualer Anteile gemacht; in einem vergleichsweise kleinen Korpus mit nur 58 Dokumenten können wir dies aber auch anhand absoluter Häufigkeiten machen:

```
trimmed_dfm <- dfm_trim(inaug_dfm,
                        min_termfreq = 5,
                        min_docfreq = 2,
                        termfreq_type = "count",
                        docfreq_type = "count")
```

Entfernt wurden also alle Wörter, die nicht mindestens 5 mal insgesamt und nicht in mindestens 2 verschiedenen Reden vorkommen. Mit `termfreq_type = "count"` und `docfreq_type = "count"` legen wir fest, dass diese Zahlen nun als absolute Häufigkeiten (anstatt wie bisher `prop`, also prozentualer Anteile) interpretiert werden sollen (hierbei handelt es sich auch um die Standardeinstellung, allerdings ist es sinnvoll, dies dennoch im Code zu explizieren, damit andere schneller erfassen können, was hier geschieht).

Abschließend müssen wir die DFM in ein Format konvertieren, mit dem `keyATM` umgehen kann. Bisher haben wir dies mit der Funktion `convert()` von Quanteda erledigt; da das keyATM-Package noch sehr neu ist, bietet Quanteda aber noch keine Konvertierungsmöglichkeit. Allerdings bietet keyATM selbst eine Konvertierungsfunktion namens `keyATM_read()`:

```
keyATM_docs <- keyATM_read(texts = trimmed_dfm)
```

```
## Using quanteda dfm.
```

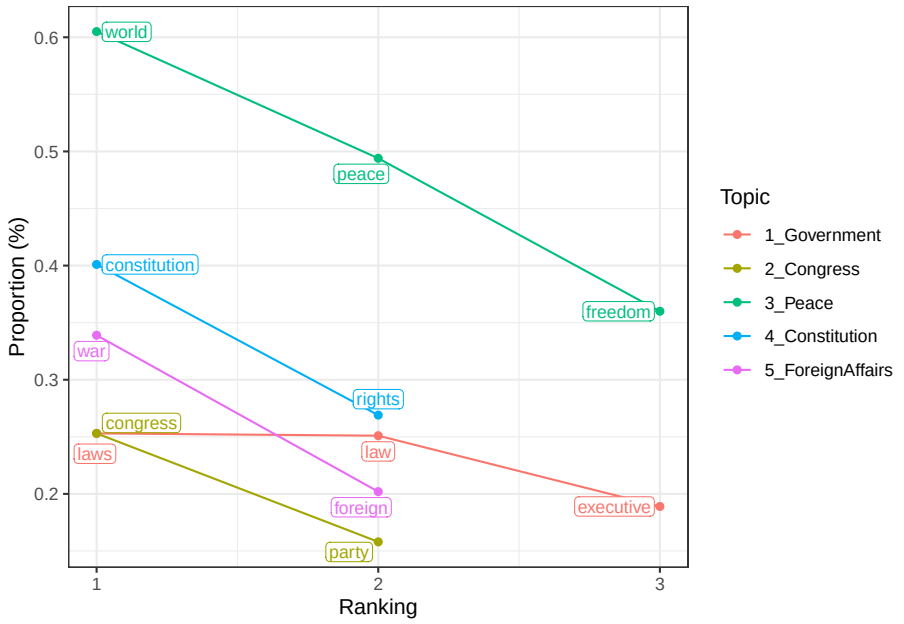
22.1.2 A-priori-Themen und zugehörige Schlüsselwörter definieren

Die große Neuerung ist wie bereits erwähnt, dass wir nun bereits vorab einige Themen und zugehörige Schlüsselwörter definieren können. Bei den Antrittsreden würden wir erwarten, dass immer wieder echte Dauerbrenner wie Regierungs- und Kongressbezüge, Verweise auf die Verfassung, aber auch auf Friedensbemühungen und Außenpolitik vorkommen. Wir definieren diese als Liste, wobei wir der Themenbezeichnung jeweils einen Vektor an Schlüsselwörtern zuorden (diese Vektoren können auch unterschiedlich lang sein, also eine unterschiedliche Anzahl an Schlüsselwörtern aufweisen):

```
keywords <- list("Government" = c("laws", "law", "executive"),
                 "Congress" = c("congress", "party"),
                 "Peace" = c("peace", "world", "freedom"),
                 "Constitution" = c("constitution", "rights"),
                 "ForeignAffairs" = c("foreign", "war"))
```

Das Package bietet uns zudem eine Funktion, mit der wir vorab überprüfen können, ob es sich um sinnvolle Schlüsselwörter handelt. Mit `visualize_keywords()` erzeugen wir eine Grafik, die den prozentualen Anteil aller vorab definierten Schlüsselwörter ausgibt. Die Autoren des Packages empfehlen, dass jedes Schlüsselwort einen Anteil von über 0,1% aufweisen sollte – dies ist hier bei allen definierten Schlüsselwörtern der Fall:

```
visualize_keywords(keyATM_docs, keywords)
```



Außerdem würde die Funktion eine Warnung ausgeben, sollte ein Schlüsselwort gar nicht im Korpus vorkommen – auch dies ist hier nicht der Fall, wir können also gut mit diesen Schlüsselwörtern arbeiten. Wichtig ist: ob diese Wörter die vorab definierten Themen auch inhaltlich sinnvoll beschreiben, ist eine inhaltliche, menschliche Beurteilung, die keine Statistik ersetzen kann.

22.1.3 Modell berechnen und interpretieren

Nun können wir das Modell mit der Funktion `keyATM()` berechnen. Hier sind folgende Argumente relevant:

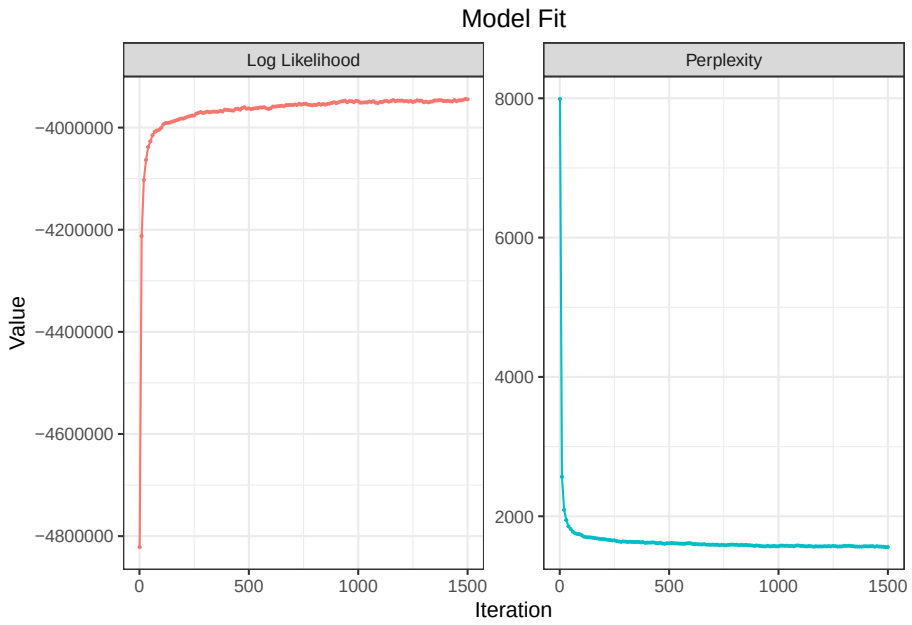
- `docs`: unsere konvertierte DFM, anhand der das Modell berechnet werden soll.
- `no_keyword_topics`: die Anzahl an weiteren Topics, die das Modell *zusätzlich* zu den vorab definierten Themen beinhalten soll.
- `keywords`: die Liste der vorab definierten Topics mit zugehörigen Schlüsselwörtern.
- `model`: hier geben wir `"base"` an, um das Standard-keyATM zu rechnen. Weitere Modellvarianten können, ebenso wie bei STM, beispielsweise auch Kovariaten enthalten, mit denen die Prävalenz der Themen geschätzt werden kann. Dies könnte im Beispiel dazu genutzt werden, um die Prävalenz bestimmter Themen etwa durch den Zeitverlauf oder die Parteizugehörigkeit der Präsidenten zu erklären. Wir blenden dies der Einfachheit halber an dieser Stelle aus, die offizielle Dokumentation verfügt jedoch über lange, gute Erklärungen der weiteren Modellvarianten.

- `options`: hier können wir weitere Optionen festlegen; da die initiale Themenlösung zufällig erzeugt wird, können wir mit der Angabe eines Seeds sichern, dass unser Modell exakt replizierbar ist.

```
inaug_model <- keyATM(docs = keyATM_docs,
                      no_keyword_topics = 5,
                      keywords = keywords,
                      model = "base",
                      options = list(seed = 667))
```

Zunächst sollten wir den Modelfit überprüfen. keyATM bietet hierfür die Funktion `plot_modelfit()`, die zwei Kennwerte – Log-Likelihood und Perplexity – im Verlauf der Modelliterationen darstellt. Bei einem Modell, das gut zu den Daten passt, sollte sich im Verlauf der Iterationen die Log-Likelihood auf einem höheren, die Perplexity auf einem geringeren Wert einpendeln. Beides ist hier der Fall (erneut gilt jedoch: ob das Modell auch inhaltlich sinnvoll ist, muss manuell und inhaltlich interpretiert werden):

```
plot_modelfit(inaug_model)
```



Die inhaltliche Interpretation des Modells erfolgt vor allem anhand der Funktionen `top_words()` und `top_docs()`. Erstere stellt uns die – per Standardeinstellung 10 – wichtigsten Wörter je Thema dar:

```
top_words(inaug_model, n = 5)
```

```
##           1_Government 2_Congress          3_Peace          4_Constitution 5_ForeignAffairs
## 1      laws [<U+2713>]    country              us              government           states  p
## 2         congress [2]       best          nation                  people           public
## 3       law [<U+2713>]       well peace [<U+2713>] constitution [<U+2713>]            union
## 4 executive [<U+2713>]       much             new                   power           united
## 5               policy  necessary          people       rights [<U+2713>]        interests pr
##    Other_5
## 1    great
## 2  war [5]
## 3     made
## 4 national
## 5    never
```

Bei einigen Wörtern fallen Ihnen hinter den Wörtern zusätzliche Zeichenketten auf. Die Zeichenkette `"<U+2713>"` sollte eigentlich ein Häkchensymbol darstellen (es handelt sich bei der kryptischen Nummernfolge um den zugehörigen Unicode); führen Sie den Code *nicht* an einem deutschen Windows-Rechner aus, sollter dies auch bereits funktionieren. In jedem Fall signalisiert Ihnen das, dass es sich um ein vorab definiertes Schlüsselwort handelt. Das vorab definierte Thema 3, Peace, enthält beispielsweise als drittwichtigstes Schlüsselwort "peace", ebenfalls gehören jedoch auch neue Wörter wie "us", "nation" etc. dazu. Eine Zahl in eckigen Klammern signalisiert hingegen, dass das betreffende vorab definierte Schlüsselwort auch einem anderen Thema zugeordnet wurde. Das Schlüsselwort "war" im neuen Thema "Other_5" entstammt beispielsweise dem vorab definierten Thema 5, "ForeignAffairs".

`top_docs()` wiederum reiht uns pro Thema die wichtigsten Dokumente auf, also diejenigen Dokumente, in denen das jeweilige Thema den größten Anteil hat.

```
top_docs(inaug_model, n = 5)
```

```
##   1_Government 2_Congress 3_Peace 4_Constitution 5_ForeignAffairs Other_1 Other_2 Other_3 Othe
## 1           31         16      46              14                15      23      51      36
## 2           19         21      53               3                 6      13      44      41
## 3           26          3      45               2                 8      11      46      34
## 4           23         22      52              10                 9      15      54      33
## 5           24         18      47               1                10      36      47      38
```

Thema 1, "Government", ist also am stärksten in Antrittrede 31 enthalten, gefolgt von Antrittsrede 19 usw.

Ansonsten gestaltet sich die Interpretation analog zu den im vorigen Kapitel berechneten Themenmodellen mit STM. Zentral sind erneut die Matrizen mit

den Themenwahrscheinlichkeiten je Dokument, bei keyATM als θ bezeichnet, und die Wortwahrscheinlichkeiten je Thema, hier als ϕ bezeichnet, die wir über das Modellobjekt mit `inaug_model$theta` bzw. `inaug_model$phi` abrufen können.

Mit etwas Umformung erhalten wir beispielsweise wieder einen Datensatz, der uns pro Thema die 7 wichtigste Wörter in einem String verbindet:

```
top_terms <- inaug_model$phi %>%
  t() %>%
  as_tibble(rownames = "word") %>%
  pivot_longer(-word, names_to = "topic", values_to = "phi") %>%
  group_by(topic) %>%
  top_n(7, phi) %>%
  arrange(topic, desc(phi)) %>%
  group_by(topic) %>%
  summarise(top_words = paste(word, collapse = ", "), .groups = "drop")

top_terms
```

```
## # A tibble: 10 x 2
##    topic            top_words
##    <chr>            <chr>
##  1 1_Government     laws, congress, law, executive, policy, administration, states
##  2 2_Congress       country, best, well, much, necessary, party, ever
##  3 3_Peace          us, nation, peace, new, people, freedom, america
##  4 4_Constitution   government, people, constitution, power, rights, one, duty
##  5 5_ForeignAffairs states, public, union, united, interests, foreign, powers
##  6 Other_1          political, duties, free, federal, prosperity, men, far
##  7 Other_2          hope, free, faith, good, yet, strength, come
##  8 Other_3          american, government, world, future, progress, justice, purpose
##  9 Other_4          every, now, citizens, us, spirit, just, power
## 10 Other_5          great, war, made, national, never, many, support
```

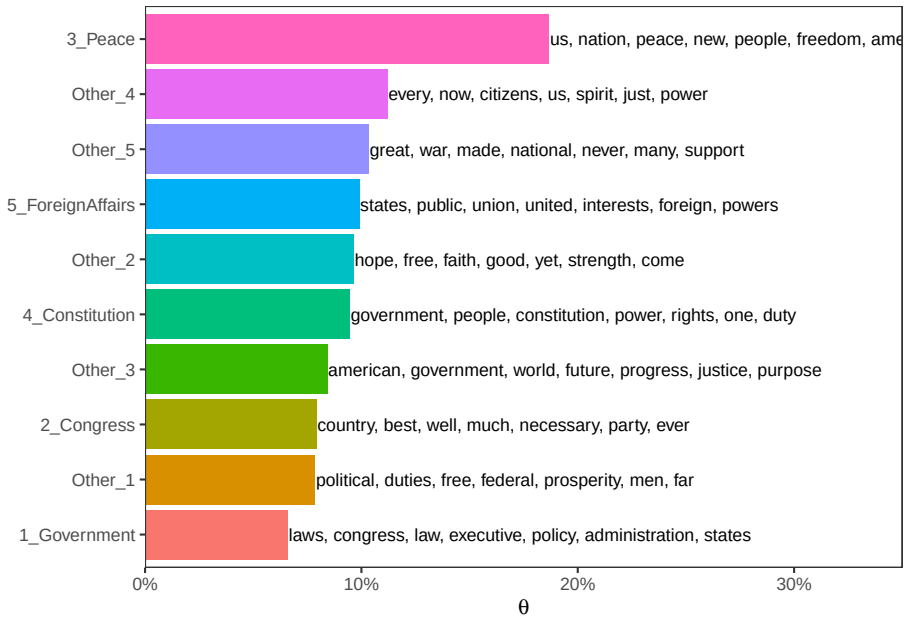
Ebenso können wir die Themen mit ihrem durchschnittlichen Anteil in den Dokumenten nach Wichtigkeit sortieren:

```
top_topics <- inaug_model$theta %>%
  as_tibble(rownames = "speech") %>%
  pivot_longer(-speech, names_to = "topic", values_to = "theta") %>%
  group_by(topic) %>%
  summarise(mean_theta = mean(theta), .groups = "drop") %>%
  arrange(desc(mean_theta))

top_topics
```

```
## # A tibble: 10 x 2
##    topic            mean_theta
##    <chr>                 <dbl>
##  1 3_Peace              0.187
##  2 Other_4              0.112
##  3 Other_5              0.103
##  4 5_ForeignAffairs     0.0990
##  5 Other_2              0.0965
##  6 4_Constitution       0.0946
##  7 Other_3              0.0842
##  8 2_Congress           0.0792
##  9 Other_1              0.0785
## 10 1_Government         0.0660
```

Und wenn wir beides verbinden, erhalten wir wieder die bereits bekannte Grafik, die uns Themenprävalenz und wichtigste Wörter zusammen darstelle:

```
top_topics %>%
  left_join(top_terms, by = "topic") %>%
  mutate(topic = reorder(topic, mean_theta)) %>%
  ggplot(aes(topic, mean_theta, label = top_words, fill = topic)) +
  geom_col(show.legend = FALSE) +
  geom_text(hjust = 0, nudge_y = 0.0005, size = 3) +
  coord_flip() +
  scale_y_continuous(expand = c(0, 0), limits = c(0, 0.35), labels = scales::percent) +
  theme_bw() +
  theme(panel.grid.minor = element_blank(),
        panel.grid.major = element_blank()) +
  labs(x = NULL, y = expression(theta))
```



22.2 Übungsaufgaben

Erstellen Sie für die folgende Übungsaufgabe eine eigene Skriptdatei oder eine R-Markdown-Datei und speichern diese als `ue22_nachname.R` bzw. `ue22_nachname.Rmd` ab.

Für die Übungsaufgabe verwenden wir erneut den Korpus aus Artikeln des Guardian (siehe Übungsaufgabe **??**.

```
guardian_corpus <- quanteda.corpora::download("data_corpus_guardian")
```

Übungsaufgabe 22.1. Keyword Assisted Topic Models:

Rechnen Sie ein Keyword Assisted Topic Model mit mindestens 3 vorab definierten Themen und 20 Themen insgesamt. Orientieren Sie sich bei den vorab definierten Themen an den Ergebnissen der vorherigen Übungsaufgabe.

Chapter 23

Validierung automatisierter Inhaltsanalysen

In ihrem wegweisenden Artikel Text as Data: The Promise and Pitfalls of Automatic Content Analysis Methods for Political Texts bringen die Politikwissenschaftler Justin Grimmer und Brandon M. Stewart eines der Grundprinzipien einer jeden guten, automatisierten Inhaltsanyalyse mit drei Worten[1] auf den Punkt: "Validate, Validate, Validate".

> Automated text analysis methods can substantially reduce the costs and time of analyzing massive collections of political texts. When applied to any one problem, however, the output of the models may be misleading or simply wrong. [...] What should be avoided, then, is the blind use of any method without a validation step (Grimmer & Stewart, 2013, S. 5).

Computationale Verfahren der automatisierten Inhaltsanalyse kommen in der Regel immer zu einem Ergebnis: ein Klassifikationsmodell klassifiziert alle Dokumente, ein Diktinär spuckt für jedes Dokument ein Ergebnis aus, ein Topic Model findet immer die vorgegebene Anzahl an Themen. Ob es sich dabei auch um inhaltlich sinnvolle Ergebnisse handelt, kann und muss durch manuelle Validierungen festgestellt werden.

Wie auch bei manuellen Erhebungs- und Analyseverfahren ist die Bestimmung der Validität einer automatisierten Messung komplex und mit Schwierigkeiten behaftet, da es gerade bei den bei uns so häufig untersuchten Konstrukten wie Themen und emotionaler Valenz oder der Kategorisierung von Texten nach inhaltlichen Kategorien letztlich keine abschließende Möglichkeit gibt, die Validität einer Untersuchung 100%ig zu bestimmen. Je nach verwendetem Verfahren

[1] aber nur einem Feature

werden daher unterschiedliche Optionen angewandt und/oder kombiniert, um sich der Bestimmung der Validität zumindest anzunähern:

- ähnlich wie bei den Intercoderreliabilitätstests der manuellen Inhaltsanalyse können die automatisierten Codierungen (eines Teilsamples) mit manuellen Codierungen verglichen werden; die manuellen Codierungen gelten hierbei meist als Goldstandard[2], es wird also berechnet, wie gut (=reliabel) die automatisierten Codierungen die manuellen Codierungen replizieren können (Reliabilität als notwendige Voraussetzung von Validität).
- für bestimmte Verfahren können statistische Kennwerte herangezogen werden, die angeben, wie gut die automatisierten Codierungen zu den Daten passen (Validität im Sinne von Kriteriumsvalidität).
- sinnhafte Beziehungen zwischen den automatisierten Codierungen und anderen Variablen des Textkorpus können überprüft werden (Validität im Sinne von Konstruktvalidität). Haben wir beispielsweise in einem Topic Model ein "Terrorismus"-Thema identifiziert, so würden wir auch erwarten, dass dieses Thema häufiger auftritt, wenn das Veröffentlichungsdatum der jeweiligen Artikel kurz nach Terroranschlägen liegt.

Im Folgenden wird ein Überblick über konkrete Validierungsmöglichkeiten bei unterschiedlichen Verfahrensklassen gegeben. Wir werden uns außerdem mit einem Package auseinandersetzen, das Verfahren zur Validierung insbesondere von diktionärbasierten Ansätzen und Themenmodellen in R implementiert: `oolong`.

```
install.packages("oolong")
```

23.1 Validierung von Textklassifikationen

Textklassifikationen bzw. Verfahren des überwachten maschinellen Lernens allgemein haben den Vorteil, dass – zumindest wenn ein eigenes Klassifikationsmodell erstellt wird und nicht auf eine Out-of-the-Box-Lösung zurückgegriffen wird – bereits ein manuell codierter oder anderweitig annotierter (Teil-)Datensatz vorliegt. In Kapitel 20.1 haben wir bereits die Unterteilung in Trainings- und Testdatensätze kennengelernt und uns anhand dieser eine *Confusion Matrix* (siehe Kapitel 20.2.1) ausgeben lassen, auf Basis dieser dann wiederum statistische Messgrößen zur Beurteilung der Klassifikationsgüte berechnet werden können.

Grundsätzlich ist es ratsam, Klassifikationsmodelle immer anhand eines Teils des codierten Materials zu validieren, das *nicht* für die Berechnung des Modells

[2]dies unterstellt allerdings, dass die manuelle Codierung bereits valide ist, was, wie oben angeführt, auch nicht unbedingt gegeben sein muss.

verwendet wurde. In der Praxis wird häufig auch eine Dreiteilung des Ursprungsmaterials vorgenommen:

- Trainings-Datensatz: der Datensatz, anhand dem das Modell bzw. die Modelle trainiert werden. Um ein geeignetes Modell zu finden, müssen in der Regel mehrere Modelle berechnet werden (z. B. unterschiedliche Klassifikations-Algorithmen, Optimierung von Hyperparametern).
- Validierungs-Datensatz: der Datensatz, anhand dem die Performance der trainierten Modelle bewertet wird. Das Modell mit der besten Performance wird dann zum finalen Modell auserkoren.
- Test-Datensatz: der Datensatz, an dem die Performance des finalen Modells bewertet wird; der Testdatensatz war also weder an der Berechnung noch an der Validierung der Zwischenmodelle beteiligt.

Zudem können einige Modellklassen der Textklassifikation auf Kreuzvalidierungsverfahren schon bei der Berechnung der Modelle zurückgreifen. Hierbei wird der annotierte Textkorpus in k Teildatensätze aufgeteilt (bei einer 10-fachen Kreuzvalidierung also in 10 Teildatensätze), wobei das Modell dann automatisiert k mal jeweils auf Basis von $k - 1$ Teildatensätzen trainiert und anhand des verbleiben Teildatensatzes validiert wird.

23.2 Validierung von diktionärsbasierten Ansätzen

Diktionärsbasierte Ansätze resultierten in der Regel in metrischen Werten je Dokument, die etwa den Anteil der im Diktionär enthaltenen Begriffe an allen Wörtern jeweiligen Dokument oder, bei gewichteten Lexika, den Mittelwert der numerischen Gewichtung der einzelnen Wörter im Dokument. Bei einem Vergleich mit manuellen Codierungen kann also entweder diese ebenfalls eine gewichtete Abstufung des untersuchten Konstrukts berücksichtigen, oder es wird mit Cutoff-Werten gearbeitet, ab denen ein Dokument als einer Diktionärkategorie zugehörig gewertet wird (z. B. ob ein Dokument als 'negativ' gilt, wenn der Anteil der negativen Wörter im Dokument einen Anteil von soundsoviel Prozent übersteigt).

Mittels `oolong` wird die erste Variante umgesetzt. In einem sogenannten *Gold-Standard-Test* wird zunächst ein zufälliges Sample des gesamten Textmaterials von manuellen Codern (im Idealfall mindestens zweien) eingestuft und anschließend mit den automatisierten Einstufungen verglichen.

Kehren wir hierzu noch einmals zum bekannten Tweet-Datensatz von Donald Trump und Joe Biden zurück (siehe Kapitel 19):

```
library(tidyverse)
library(quanteda)
library(tidytext)

tweets <- read_csv("data/trump_biden_tweets_2020.csv") %>%
  mutate(day = str_sub(date, 1, 10))

tweets
```

```
## # A tibble: 4,153 x 8
##       id account  link                    content
##    <dbl> <chr>    <chr>                   <chr>
##  1     1 JoeBiden https://twitter.com/Jo~ "Our final fundraising deadline of 2019 is
##  2     2 JoeBiden https://twitter.com/Jo~ "Every single human being deserves to be t
##  3     3 JoeBiden https://twitter.com/Jo~ "With just over one month until the Iowa C
##  4     4 JoeBiden https://twitter.com/Jo~ "This election is about the soul of our na
##  5     5 JoeBiden https://twitter.com/Jo~ "Every day that Donald Trump remains in th
##  6     6 JoeBiden https://twitter.com/Jo~ "It was a privilege to work with @JulianCa
##  7     7 JoeBiden https://twitter.com/Jo~ "Like Vicky said, we need a president who
##  8     8 JoeBiden https://twitter.com/Jo~ "I'm excited to share that we raised $22.7
##  9     9 JoeBiden https://twitter.com/Jo~ "If you're a teacher or a firefighter, you
## 10    10 JoeBiden https://twitter.com/Jo~ "Before the holidays, Jill walked across t
## # ... with 4,143 more rows
```

Wir möchten die positive und negative Valenz dieser Tweets nun durch das AFINN-Dictionary bewerten lassen (siehe Kapitel 19.3) und diese automatisierte Einstufung auch manuell validieren. All dies kann direkt in R und RStudio mittels `oolong` erfolgen:

```
library(oolong)
```

Mit der Funktion `create_oolong()` können wir ein Test-Objekt erzeugen. Hierzu übergeben wir die Texte der Tweets als Argument `input_corpus` und benennen das Attribut, das wir codieren möchten, mit dem Argument `construct`. Oolong sieht vor, dass dieses Konstrukt mit einem passenden Adjektiv bezeichnet wird, hier würde sich z. B. "positive" anbieten, da höhere AFINN-Werte einem positiveren Tweet entsprechen:

```
gs_test <- create_oolong(input_corpus = tweets$content, construct = "positive")
gs_test
```

```
## An oolong test object (gold standard generation) with 41 cases, 0 coded.
## Use the method $do_gold_standard_test() to generate gold standard.
## Use the method $lock() to finalize this object and see the results.
```

Wir sehen, dass das neue Objekt ein Testobjekt ist, in dem automatisch 1% des Textkorpus – in unserem Fall also 41 Tweets – zufällig ausgewählt wurden. Als nächstes weist uns das Package an, einen Gold-Standard-Test mit der Methode `$do_gold_standard_test()` durchzuführen.

```
gs_test$do_gold_standard_test()
```

Es öffnet sich nun in RStudio eine Codiermaske, in der wir diese 41 Tweets auf einer fünfstufigen Skala bewerten können.[3]

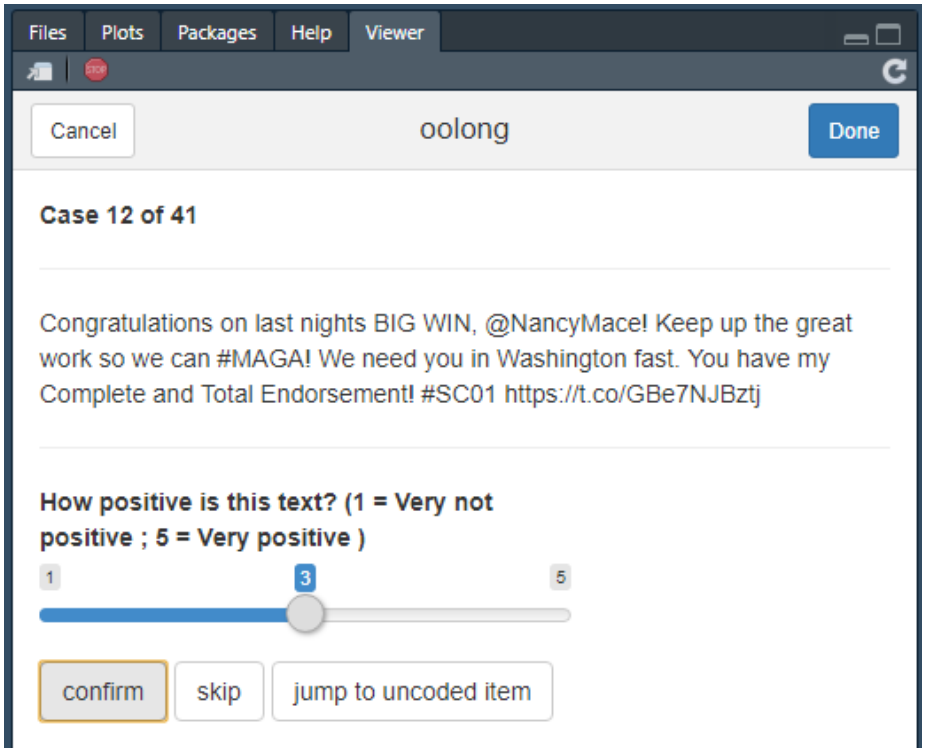


Figure 23.1: Gold-Standard-Test in `oolong`

Nach der Codierung sperren wir das Test-Objekt mit der Methode `$lock`, damit diese nicht mehr verändert werden können:

[3]Sinnvoll wäre es, diese Bewertung von mindestens zwei Codern durchführen zu lassen, für das Beispiel nutzen wir aber nur einen Durchgang. Soll die Codierung von mehreren Codern durchgeführt werden, muss das ursprüngliche Testobjekt (in diesem Fall `gs_test` mittels der Funktion `clone_oolong()` entsprechend oft kopiert werden).

```
gs_test$lock()
```

Nun können wir anhand des Test-Objekts die automatisierte Codierung vornehmen. Mit der Methode `$turn_gold()` erzeugen wir automatisch einen Quanteda-Korpus aus den zu codierenden Tweets:

```
gs_corpus <- gs_test$turn_gold()
gs_corpus
```

```
## Corpus consisting of 41 documents and 1 docvar.
## text1 :
## "....He is Strong on Crime, the Border, and Second Amendment...."
##
## text2 :
## "After last night, we are one step closer to restoring decenc..."
##
## text3 :
## "Congratulations to @serenawilliams on another big win. She i..."
##
## text4 :
## "We need a president who demonstrates the leadership to addre..."
##
## text5 :
## "Mini Mike, you're easy! https://t.co/rxFiqSB9RQ https://t.co..."
##
## text6 :
## "....very often FAKE NEWS. Lamestream Media should be forced ..."
##
## [ reached max_ndoc ... 35 more documents ]
## Access the answer from the coding with quanteda::docvars(obj, 'answer')
```

Nun können wir wie bereits bekannt den AFINN-Score berechnen:[4]

```
tweets_afinn <- gs_corpus %>%
  convert("data.frame") %>% # In Dataframe konvertieren
  unnest_tokens(word, text, token = "tweets") %>% # Text in Token splitten
  left_join(get_sentiments("afinn")) %>%  # AFINN-Scores hinzufügen
  mutate(value = if_else(is.na(value), 0, value)) %>% # fehlende Werte (= Wort nicht i
  group_by(doc_id) %>% # Nach Texten gruppieren
  summarise(afinn = mean(value), .groups = "drop") # Durchschnittlichen AFINN-Score be

tweets_afinn
```

[4]Eine alternative Berechnungsweise, die ohne `tidytext` auskommt, findet sich in der offiziellen Dokumentation von Oolong.

```
## # A tibble: 41 x 2
##    doc_id   afinn
##    <chr>    <dbl>
##  1 text1   0.1
##  2 text10 -0.0217
##  3 text11 -0.0513
##  4 text12  0.387
##  5 text13  0
##  6 text14  0
##  7 text15  0.0952
##  8 text16  0
##  9 text17  0.111
## 10 text18  0.2
## # ... with 31 more rows
```

Nun extrahieren wir die AFINN-Scores als Vektor, um diese anschließend mit den manuellen Codierungen zu vergleichen:

```
afinn_scores <- tweets_afinn %>%
  pull(afinn)
```

Der Vergleich erfolgt über die Funktion `summarize_oolong()`:

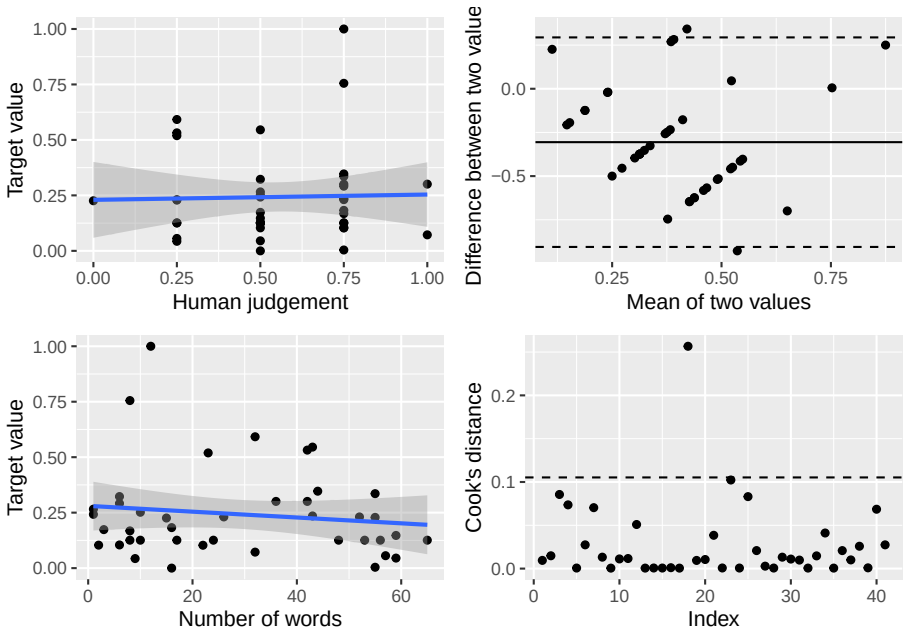
```
results_gs_test <- summarize_oolong(gs_test, target_value = afinn_scores)
```

Die Resultate können am einfachsten über eine mitgelieferte Visualisierung betrachtet werden, die in vier Subgrafiken unterteilt ist:

- links oben ist die Korrelation zwischen den manuellen Codierungen und den automatischen Codierungen. Hier möchten wir natürlich eine möglichst hohe Korrelation erreichen; im Beispiel ist das offensichtlich nicht der Fall (das kann aber auch an meiner sehr schludrigen Codierung liegen).
- rechts oben ist ein sogenannter Bland-Altmann-Plot, der den Mittelwert zweier Messungen (hier also manueller und automatischer Codierung eines Tweets) gegen deren Differenz abträgt. Hier sollten im Idealfall keinen Muster erkennbar sein (damit keine systematischen Unterschiede zwischen manueller und automatischer Codierung vorliegen) und keine allzu großen Schwankungen vorliegen (damit manuelle und automatische Codierung im Mittel nicht zu stark voneinander abweichen).
- links unten wird die Korrelation zwischen der Anzahl der Wörter und der automatisierten Codierung abgetragen. Hier sollte im Idealfall ebenfalls keine Korrelation erkennbar sein, damit unser Dictionary nicht systematisch durch die Textlänge beeinflusst wird.

- rechts unten wird schließlich der Cook-Abstand zur Beurteilung von besonders einflussreichen Fällen ausgegeben. Hier sollte im Idealfall kein Wert über der gestrichelten Linie liegen.

```
plot(results_gs_test)
```



In diesem Falle würden wir insgesamt also auf keine sonderlich valide Codierung schließen – wobei fraglich ist, ob dies an der manuellen oder der automatisierten Codierung liegt.

23.3 Validierung von Themenmodellen

Neben den uns bereits bekannten Möglichkeiten zur statistischen und interpretativen Validierung von Themenmodellen (siehe Kapitel 21), bringt Oolong zwei weitere manuelle Valdierungsverfahren mit, die jeweils auf das Paper Reading Tea Leaves: How Humans Interpret Topic Models[5] zurückgehen: *Word Intrusion Tests* und *Topic Intrusion Tests*.

Für die Beispiele verwenden wir das bereits bekannte Ted-Talks-Themenmodell mit 30 Themen, das wir im Rahmen des Kapitels zu Themenmodellen (siehe Kapitel 21) erstellt haben. Falls Sie das Modell nicht gespeichert haben, können Sie es mit folgendem Code erneut berechnen:

[5]das wiederum der Namensgeber für das Package ist

```
library(stm)

ted_talks <- read_tsv("data/ted_main_dataset.tsv")

ted_talks <- ted_talks %>%
  filter(!is.na(speaker_image_nr_faces)) %>%
  mutate(id = 1:n(),
         date = lubridate::ymd(date, truncated = 1)) %>%
  select(id, date, title, text)

ted_corpus <- corpus(ted_talks, text_field = "text")

ted_dfm <- dfm(ted_corpus,
               stem = TRUE,
               tolower = TRUE,
               remove_punct = TRUE,
               remove_url = FALSE,
               remove_numbers = TRUE,
               remove_symbols = TRUE,
               remove = stopwords('english'))

stm_dfm <- convert(ted_dfm, to = "stm")

ted_model <- stm(documents = stm_dfm$documents,
                 vocab = stm_dfm$vocab,
                 K = 30)
```

23.3.1 Word Intrusion Tests

Im Word Intrusion Test werden manuelle Codern zu jedem Topic (in zufälliger Reihenfolge) sechs zufällig ausgewählte Wörter vorgelegt, von denen alle bis auf eines eine hohe Themenwahrscheinlichkeit aufweisen; das letzte Wort, das sogenannte *Intruder Word*, weist eine geringe Themenwahrscheinlichkeit für dieses Thema auf, allerdings eine hohe Themenwahrscheinlichkeit bei mindestens einem anderen Thema. Das Ziel ist es nun, die Intruder Words manuell zu identifizieren – gelingt dies gut, so handelt es sich um eine brauchbare Themenlösung. Das Ergebnis wird als *Precision* bezeichnet und gibt den Anteil der korrekt identifizierten Intruder Words an allen Intrusion Tests an.

In Oolong setzen wir Word Intrusion Tests ganz ähnlich zum Vorgehen bei Gold-Standard-Tests mit `create_oolong()` um, nur dass nun anstatt eines Textkorpus mit dem Argument `input_model` ein Themenmodell übergeben wird:

```
wi_test <- create_oolong(input_model = ted_model)
wi_test
```

```
## An oolong test object with k = 30, 0 coded.
## Use the method $do_word_intrusion_test() to do word intrusion test.
## Use the method $lock() to finalize this object and see the results.
```

Wie angegeben, können wir mit der Methode `$do_word_intrusion_test()` nun den Word Intrusion Test umsetzen. Dies erfolgt erneut direkt in RStudio:

```
wi_test$do_word_intrusion_test()
```

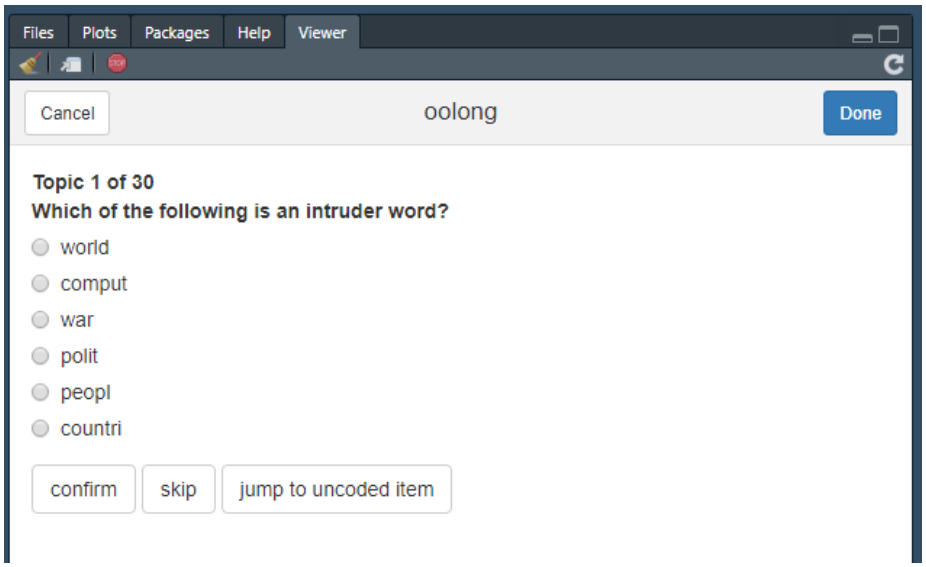


Figure 23.2: Word Intrusion Test in Oolong

Sind wir mit der manuellen Codierung durch, müssen wir das Oolong-Objekt erneut sperren, um uns die Ergebnisse anzeigen zu können:

```
wi_test$lock
```

Anschließend wird uns das Ergebnis direkt in der Konsole angezeigt:

```
wi_test
```

```
## An oolong test object with k = 30, 30 coded.
## 70%  precision
```

70% der Intruder Words wurden korrekt identifiziert – das ist nicht schlecht, aber sicher verbesserungswürdig.

23.3.2 Topic Intrusion Tests

Beim Topic Intrusion Tests wird überprüft, ob die Zuordnung von Themen zu Dokumenten über die Dokumentwahrscheinlichkeiten auch manuell nachvollziehbar ist. Hierzu werden zufällig Dokumente ausgewählt und vier Topics angezeigt: die drei Topics mit der höchsten Dokumentwahrscheinlichkeit und ein zufällig ausgewähltes *Intruder Topic*, das eine geringe Dokumentwahrscheinlichkeit aufweist. Erneut ist das Ziel, das Intruder Topic zu identifizieren (hierzu werden zu allen Topics die wichtigsten Wörter angezeigt). Das Ergebnis wird erneut als *Topic Log Odds* (TLO) ausgewiesen, wobei ein Wert, der möglichst nahe an 0 liegt erreicht werden sollte.

Auch Topic Intrusion Tests werden in Oolong über die Funktion `create_oolong()` erzeugt, nur dass dieses Mal sowohl ein `input_model` als auch ein `input_corpus` angegeben wird. Für ersteres verwenden wir erneut das Themenmodell, für letzteres kann die DFM genutzt werden, auf deren Basis das Themenmodell erzeugt wurde.

```
ti_test <- create_oolong(input_model = ted_model, input_corpus = ted_talks$text)
ti_test
```

```
## An oolong test object with k = 30, 0 coded.
## Use the method $do_word_intrusion_test() to do word intrusion test.
## With 23 cases of topic intrusion test. 0 coded.
## Use the method $do_topic_intrusion_test() to do topic intrusion test.
## Use the method $lock() to finalize this object and see the results.
```

Wie angegeben, starten wir den Test mit `do_topic_intrusion_test()`:

```
ti_test$do_topic_intrusion_test()
```

Und wieder einmal sperren wir das Objekt:

```
ti_test$lock
```

... und lassen uns das Ergebnis ausgeben:

```
ti_test
```

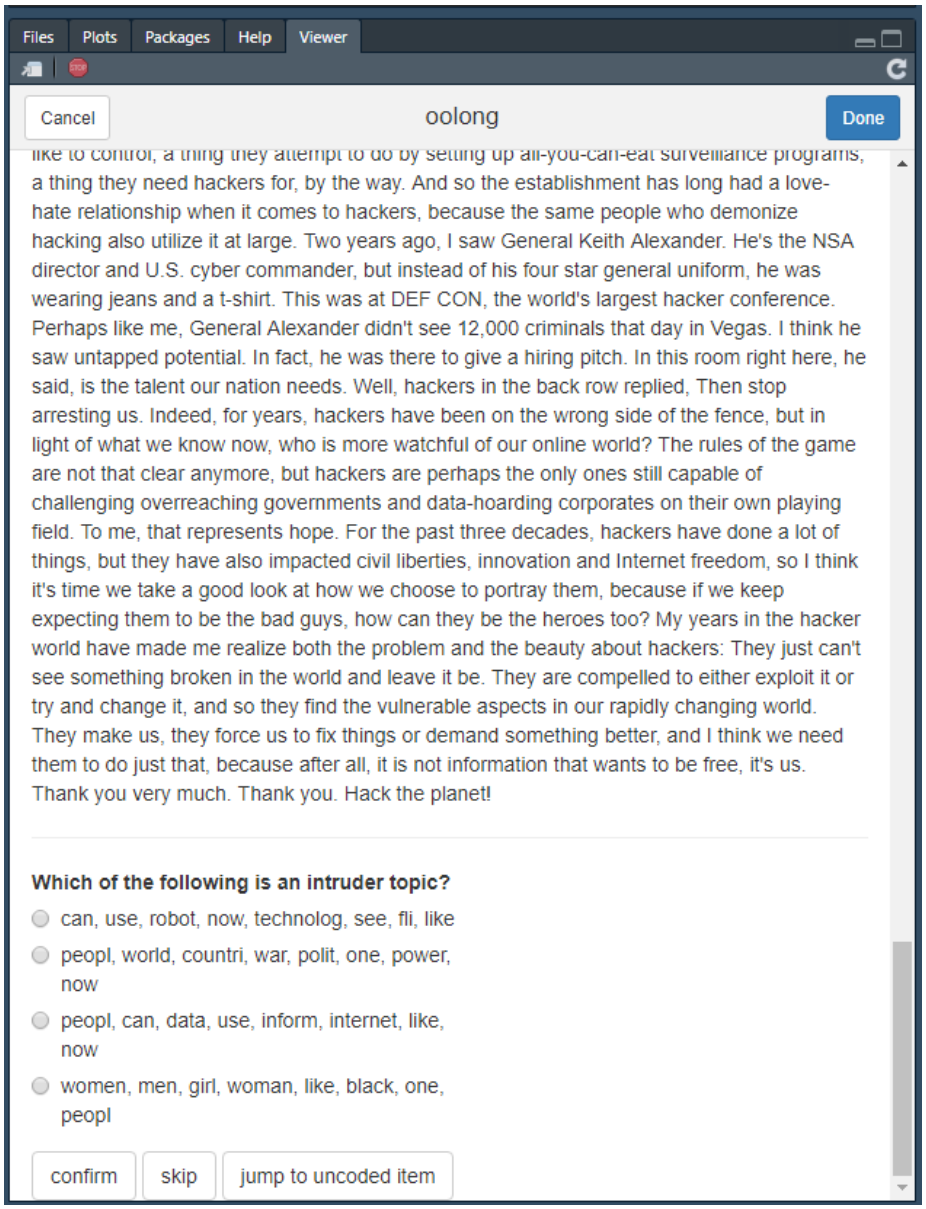


Figure 23.3: Topic Intrusion Test in Oolong

```
## An oolong test object with k = 30, 0 coded.
## 0%  precision
## With 23 cases of topic intrusion test. 23 coded.
## TLO: -4.413
```

Hier haben wir einen Wert, der weit von 0 entfernt ist, und entsprechend als schlecht beurteilt werden würde.

23.3.3 Vorgehen bei mehreren Codern

Wie auch schon bei Gold-Standard-Test ist es sinnvoll, diese Tests mit mehreren manuellen Codern durchzuführen. Hier kann analog zum Gold-Standard-Test das initial erzeugte Oolong-Objekt mit `clone_oolong()` kopiert werden, sodass zwei Coder beide Tests hintereinander durchführen können:

```
# Oolong-Objekte erstellen
ti_test_coder1 <- create_oolong(input_model = ted_model, input_corpus = ted_talks$text)
ti_test_coder2 <- clone_oolong(ti_test_coder1)

# Codieren
ti_test_coder1$do_word_intrusion_test()
ti_test_coder1$do_topic_intrusion_test()
ti_test_coder2$do_word_intrusion_test()
ti_test_coder2$do_topic_intrusion_test()

# Sperren
ti_test_coder1$lock
ti_test_coder2$lock
```

Anschließend kann erneut die Funktion `summarize_oolong()` verwendet werden, um die Ergbenisse mehrerer manueller Codierungen einzubeziehen:

```
summarize_oolong(ti_test_coder1, ti_test_coder2)
```

23.4 Übungsaufgaben

Abschließend gibt es keine Übungsaufgaben – für die Projekte sollten Sie aber in allen Fällen auf die dargestellten Validierungsmethoden zurückgreifen. Viel Erfolg!

Appendix A

Lösungen der Übungsaufgaben

Kapitel 2: Objekte und Datenstrukturen

Lösung zur Übungsaufgabe 2.1:

Am sinnvollsten ist eine Liste `list()`, da diese heterogene Objekttypen beinhalten kann. Ein Dataframe lohnt sich bei nur einem Fall eher nicht.

```
myself <- list(
  name = "Julian", # Texte als character
  year = 1988L, # Jahr als numeric - oder noch präziser als Integer
  from_bavaria = FALSE # Binäre Entscheidung als logical
)
```

Auch wenn wir hier alle Werte z. B. als Text repräsentieren könnten, ist es immer sinnvoll, den Objekttypen zu verwenden, der am besten zu den Werten passt – numerische (`year`) und logische Objekte (`from_bavaria`) ermöglichen uns mehr Rechenoptionen, einfacheres Filtern von Datensätzen etc.

Lösung zur Übungsaufgabe 2.2:

```
values <- c(1.2, 1.3, 0.8, 0.7, 0.7, 1.5, 1.1, 1.0, 1.1, 1.2, 1.1)
average <- mean(values)
above_average <- values > average
sum(above_average) / length(values)
```

```
## [1] 0.6363636
```

1. In der ersten Zeile ordnen wir `values` einen numerischen Vektor aus einigen Zahlen zu
2. In der zweiten Zeile berechnen wir den Mittelwert von `values` und weisen diesen `average` zu.
3. `values > average` prüft nun für jeden Wert in `values`, ob dieser größer als der Mittelwert (gespeichert in `average` ist). Dies erzeugt einen `logical`-Vektor, den wir `above_average` zuweisen.
4. `sum(above_average)` zählt, wie viele `TRUE`-Werte in dem Vektor sind. Das ist darauf zurückzuführen, dass `TRUE` die numerische Entsprechung `1`, `FALSE` die numerische Entsprechung `0` hat; `sum()` wandelt den logischen Vektor automatisch in einen numerischen um. Wir teilen dies durch die Anzahl der Werte in `values` und bekommen als Ergebnis, dass 63.6 % der Werte in `values` über dem Mittelwert liegen. (Etwas schneller hätten wir dieses Ergebnis auch bekommen, wenn wir die letzte Zeile durch `mean(above_average)` ersetzen.)

Lösung zur Übungsaufgabe 2.3:

```
str(mtcars)
```

```
## 'data.frame':    32 obs. of  11 variables:
##  $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
##  $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
##  $ disp: num  160 160 108 258 360 ...
##  $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
##  $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
##  $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
##  $ qsec: num  16.5 17 18.6 19.4 17 ...
##  $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
##  $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
##  $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
##  $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

`mtcars` enthält 11 Variablen, allesamt `numeric`, und 32 Fälle.

```
mean(mtcars$cyl)
```

```
## [1] 6.1875
```

Im Durchschnitt haben die Fahrzeuge ca. 6.2 Zylinder.

Um einen Teildatensatz `cars_short`, der lediglich die Variablen `mpg` und `hp` enthält, zu erstellen, führen viele Wege zum Ziel, z. B.:

```
cars_short <- mtcars[, c("mpg", "hp")]
cars_short <- mtcars[c("mpg", "hp")] # Steht nur eine Angabe in den eckigen Klammern, interpretie
cars_short <- data.frame(mtcars$mpg, mtcars$hp)
```

Kapitel 3: Funktionen

Lösung zur Übungsaufgabe 3.1:

Um die gewünschte Sequenz zu erzeugen, benötigen wir die Argumente `from` (Startwert), `to` (Endwert) und `by` (Zunahmewert).

```
seq(from = 0, to = 100, by = 5)
```

```
##  [1]   0   5  10  15  20  25  30  35  40  45  50  55  60  65  70  75  80  85  90  95 100
```

Da es sich, wie wir der Funktionsdokumentation entnehmen können, dabei um die ersten drei Funktionsargumente handelt, können wir diese auch unbenannt übergeben:

```
seq(0, 100, 5)
```

```
##  [1]   0   5  10  15  20  25  30  35  40  45  50  55  60  65  70  75  80  85  90  95 100
```

Lösung zur Übungsaufgabe 3.2:

Unsere Funktion benötigt lediglich ein Argument, die Temperatur in Grad Fahrenheit (als numerischen Wert), und soll diese in Grad Celsius mit der Formel řC = (řF − 32)Ö5/9 umwandeln:

```
fahrenheit_to_celsius <- function(fahrenheit) {
  celsius <- (fahrenheit - 32) * 5/9
  return(celsius)
}

# Unsere neue Funktion kann sogar mehrere Temperaturwerte auf einmal umrechnen
fahrenheit_to_celsius(c(0, 50, 80, 100))
```

```
## [1] -17.77778  10.00000  26.66667  37.77778
```

Lösung zur Übungsaufgabe 3.3:

Für das erste zusätzliche Feature, der Anzahl der fehlenden Werte, müssen wir `descriptives_vector` lediglich ein Element hinzufügen (das wir z. B. `Missing` nennen), in dem eben diese Anzahl festgehalten wird. Mit der Funktion `is.na()` prüfen wir jeden Wert eines Vektors darauf, ob es sich um einen fehlenden Wert `NA` handelt, mit der Summenfunktion `sum()` können wir diese addieren. Wir ändern `descriptives_vector` daher wie folgt:

```
  descriptives_vector <- c(
    n = length(x),
    Missing = sum(is.na(x)), # Hier zählen wir die fehlenden Werte
    M = mean(x, na.rm = na.rm),
    SD = sd(x, na.rm = na.rm),
    Minimum = min(x, na.rm = na.rm),
    Maximum = max(x, na.rm = na.rm),
    Median = median(x, na.rm = na.rm)
  )
```

Für das zweite zusätzliche Feature, Rundung auf eine gewünsche Anzahl an Nachkommastelle, benötigen wir die `round()`-Funktion, mit dem wir `descriptives_vector` abschließend runden, und ein zusätzliches Argument, mit dem die gewünschte Anzahl an Nachkommastellen übergeben werden kann. Da dieses Argument bei der `round()`-Funktion `digits` heißt, nennen wir es aus Konsistenzgründen auch in unserer Funktion so. Um standardmäßig auf zwei Nachkommastellen zu runden, geben wir dem Argument den Default-Wert `2`. Der vollständige Funktionscode sieht also wie folgt aus:

```
descriptives <- function(x, na.rm = FALSE, digits = 2) { # Zusätzliches Argument digit

  # Vektor mit Variablenbeschreibung erstellen
  descriptives_vector <- c(
    n = length(x),
    Missing = sum(is.na(x)), # Hier zählen wir die fehlenden Werte
    M = mean(x, na.rm = na.rm),
    SD = sd(x, na.rm = na.rm),
    Minimum = min(x, na.rm = na.rm),
    Maximum = max(x, na.rm = na.rm),
    Median = median(x, na.rm = na.rm)
  )

  # Vektor runden
  descriptives_vector <- round(descriptives_vector, digits = digits)

  return(descriptives_vector)
}
```

Wir haben nun eine flexibel einsetzbare Funktion, um schnell relevante Kennwerte einer numerischen Variablen zu erhalten:

```
descriptives(iris$Sepal.Length)
```

```
##       n Missing       M      SD Minimum Maximum  Median
##  150.00    0.00    5.84    0.83    4.30    7.90    5.80
```

```
descriptives(mtcars$cyl, digits = 1)
```

```
##       n Missing       M      SD Minimum Maximum  Median
##    32.0     0.0     6.2     1.8     4.0     8.0     6.0
```

Kapitel 4: Kontrollstrukturen

Lösung zur Übungsaufgabe 4.1:

Erneut gibt es verschiedene Möglichkeiten, den Entscheidungsbaum abzubilden. Wenn wir uns pro `if ()` bzw. `else ()` oder `else if ()` auf das Prüfen einer Bedingung beschränken wollen, benötigen wir einen verschachtelten Baum, also eine Bedinung in einer Bedingung:

```
if (news_channel != "Internet") {             # Prüfen, ob news_channel NICHT "Internet" ist...
  news_category <- "Offline"                  # dann news_category "Offline" zuweisen
} else {                                      # Falls das nicht der Fall ist, also news_channel "
  if (news_website == "Twitter") {            # dann prüfen wir ob news_category "Twitter" ist
    news_category <- "SNS"                    # falls das so ist, weisen wir news_category "SNS"
  } else if (news_website == "Facebook") {    # analog verfahren wir mit Facebook
    news_category <- "SNS"                    #
  } else if (news_website == "Instagram") {   # analog mit Instagram
    news_category <- "SNS"                    #
  } else {                                    # falls das alles nicht zutrifft
    news_category <- "Online: Sonstige"       # weisen wir "Online: Sonstige" zu
  }
}
```

Das erzeugt allerdings einen ziemlich langen Entscheidungsbaum und einige Redundanzen, da wir für `"Twitter"`, `"Facebook"` und `"Instagram"` jeweils dieselbe Aktion, `news_category <- "SNS"` ausführen. Wir können diese Bedingungen also auch verknüpfen:

```
if (news_channel != "Internet") {             # Prüfen, ob news_channel NICHT "Interne
  news_category <- "Offline"                  # dann news_category "Offline" zuweisen
} else {                                      # Falls das nicht der Fall ist, also new
  if (news_website == "Twitter" | news_website == "Facebook" | news_website == "Instagr
    news_category <- "SNS"                    # Alle Bedingungen mit ODER verbunden, d
  } else {                                    # falls das nicht zutrifft
    news_category <- "Online: Sonstige"       # weisen wir "Online: Sonstige" zu
  }                                           # und haben uns einige Zeilen gespart
}
```

Tatsächlich können wir auch die Verschachtelung aufheben, da nach `if (news_channel != "Internet")` folgt, dass bei allen anschließenden `else if()`-Bedingungen `news_channel == "Internet"` ist:

```
if (news_channel != "Internet") {
  news_category <- "Offline"
} else if (news_website == "Twitter" | news_website == "Facebook" | news_website == "In
  news_category <- "SNS"
} else {
  news_category <- "Online: Sonstige"
}
```

Und noch kürzer wir der Entscheidungsbaum, wenn wir den `%in%`-Operator verwenden:

```
SNS <- c("Twitter", "Facebook", "Instagram")

if (news_channel != "Internet") {
  news_category <- "Offline"
} else if (news_website %in% SNS) {
  news_category <- "SNS"
} else {
  news_category <- "Online: Sonstige"
}
```

Lösung zur Übungsaufgabe 4.2:

Beim ersten Platzhalter müssen wir einen `for`-Loop, wie in Kapitel 4.2.2 beschrieben, einfügen und uns für einen Namen für das Iterator-Objekt entscheiden. Da wir über den Vektor `variables` loopen, bietet sich der Singular `variable` an (aber natürlich funktioniert auch jeder andere Objektname). Diesen müssen wir dann bei den folgenden Platzhaltern ergänzen:

```r
numeric_summary <- function(data) {

  # Alle Variablennamen in Vektor speichern
  variables <- names(data)

  # Leere Liste für Ausgabe vorbereiten
  summary_list <- list()

  # Über alle Variablen iterieren
  for (variable in variables) { # Wir loopen über variables
    variable_vector <- data[[variable]] # Und arbeiten nun immer mit dem Iterator-Objekt variable

    if (is.numeric(variable_vector)) { # Prüfen ob die Variable numerisch ist

      # Mittelwert und Standardabweichung dieser Variablen der summary_list hinzufügen
      summary_list[[variable]] <- c(
        M = mean(variable_vector),
        SD = sd(variable_vector)
      )
    }

  }

  # Summary List ausgeben
  return(summary_list)
}
```

Diese Funktion erzeugt uns nun auf einen Schlag eine Kurzzusammenfassung anhand von Mittelwert und Standardabweichung *aller* numerischen Variablen in einem Datensatz:

```r
numeric_summary(iris)
```

```
## $Sepal.Length
##         M        SD
## 5.8433333 0.8280661
##
## $Sepal.Width
##         M        SD
## 3.0573333 0.4358663
##
## $Petal.Length
##        M       SD
## 3.758000 1.765298
##
```

```
## $Petal.Width
##         M        SD
## 1.1993333 0.7622377
```

```
numeric_summary(mtcars)
```

```
## $mpg
##         M        SD
## 20.090625  6.026948
##
## $cyl
##        M       SD
## 6.187500 1.785922
##
## $disp
##        M       SD
## 230.7219 123.9387
##
## $hp
##         M        SD
## 146.68750  68.56287
##
## $drat
##         M        SD
## 3.5965625 0.5346787
##
## $wt
##         M        SD
## 3.2172500 0.9784574
##
## $qsec
##         M        SD
## 17.848750  1.786943
##
## $vs
##         M        SD
## 0.4375000 0.5040161
##
## $am
##         M        SD
## 0.4062500 0.4989909
##
## $gear
##         M        SD
## 3.6875000 0.7378041
```

```
##
## $carb
##      M     SD
## 2.8125 1.6152
```

Kapitel 8: Daten laden, modifizieren und speichern

Lösung zur Übungsaufgabe 8.1:

Wir benötigen die `read_csv()`-Funktion, da alle Werte durch Kommas getrennt sind. Falls der Datensatz im Hauptverzeichnis des Projektordners liegt, genügt die Angabe von `"facebook_europawahl.csv"` als Argument:

```
facebook_europawahl <- read_csv("facebook_europawahl.csv")
```

Liegt der Datensatz in einem Unterordner, muss der Dateipfad entsprechend als Argument angepasst werden, z. b. `"data/facebook_europawahl.csv"`.

```
##
## -- Column specification --------------------------------------------------------
## cols(
##   id = col_double(),
##   URL = col_character(),
##   party = col_character(),
##   timestamp = col_datetime(format = ""),
##   type = col_character(),
##   message = col_character(),
##   link = col_character(),
##   comments_count = col_double(),
##   shares_count = col_double(),
##   reactions_count = col_double(),
##   like_count = col_double(),
##   love_count = col_double(),
##   wow_count = col_double(),
##   haha_count = col_double(),
##   sad_count = col_double(),
##   angry_count = col_double()
## )
```

Lösung zur Übungsaufgabe 8.2:

Um den Datensatz zu filtern, benötigen wir zunächst die Schreibweisen der Partei-Accounts. Hierzu bietet es sich an, die `party`-Variable auszuzählen:

```
count(facebook_europawahl, party)
```

```
## # A tibble: 14 x 2
##    party                              n
##    <chr>                          <int>
##  1 alternativefuerde                 79
##  2 B90DieGruenen                     67
##  3 CDU                               64
##  4 CSU                              103
##  5 DiePARTEI                         96
##  6 FamilienParteiDeutschlands        30
##  7 FDP                               94
##  8 freie.waehler.bundesvereinigung   30
##  9 linkspartei                       38
## 10 oedp.de                           71
## 11 Piratenpartei                     73
## 12 SPD                               49
## 13 tierschutzpartei                  33
## 14 VoltDeutschland                   75
```

Ebenfalls optional, aber hilfreich ist es, die entsprechenden Parteiseiten in einem Vektor zu speichern:

```
bt_parteien <- c("alternativefuerde", "B90DieGruenen", "CDU", "CSU", "FDP", "linkspart
```

Nun filtern wir den Datensatz zunächst nach Parteien:

```
df_bt_parteien <- filter(facebook_europawahl, party %in% bt_parteien)
```

Dann wählen wir nur die gewünschten Variablen aus:

```
df_reduziert <- select(df_bt_parteien, party, timestamp, type,
                       comments_count, shares_count, reactions_count)
```

Und schließlich erzeugen wir die neue Variable `total_count`:

```
df_mit_tc <- mutate(df_reduziert,
                    total_count = sum(c(comments_count, shares_count, reactions_count)

df_mit_tc
```

```
## # A tibble: 494 x 7
##    party             timestamp           type  comments_count shares_count reactions_count tot
##    <chr>             <dttm>              <chr>          <dbl>        <dbl>           <dbl>
##  1 B90DieGruenen     2019-04-28 06:00:01 video             70           28             215
##  2 FDP               2019-04-28 11:49:59 photo             16            9             262
##  3 CDU               2019-04-28 09:12:19 video            239          136             398
##  4 SPD               2019-04-28 13:06:09 photo            180           54             699
##  5 CSU               2019-04-28 08:21:00 photo            174           61             458
##  6 alternativefuerde 2019-04-28 14:55:00 link            1163         1499            3944
##  7 FDP               2019-04-28 06:18:18 photo             47          110             622
##  8 FDP               2019-04-28 14:03:00 video            358           89             463
##  9 FDP               2019-04-28 10:40:57 photo             14           19             226
## 10 CSU               2019-04-28 12:35:00 photo            133           20             330
## # ... with 484 more rows
```

Warum addieren wir nicht einfach alle Spalten ohne die Summenfunktion?

```
df_mit_total2 <- mutate(df_reduziert, total_count = comments_count + shares_count + reactions_cou

df_mit_total2
```

```
## # A tibble: 494 x 7
##    party             timestamp           type  comments_count shares_count reactions_count tot
##    <chr>             <dttm>              <chr>          <dbl>        <dbl>           <dbl>
##  1 B90DieGruenen     2019-04-28 06:00:01 video             70           28             215
##  2 FDP               2019-04-28 11:49:59 photo             16            9             262
##  3 CDU               2019-04-28 09:12:19 video            239          136             398
##  4 SPD               2019-04-28 13:06:09 photo            180           54             699
##  5 CSU               2019-04-28 08:21:00 photo            174           61             458
##  6 alternativefuerde 2019-04-28 14:55:00 link            1163         1499            3944
##  7 FDP               2019-04-28 06:18:18 photo             47          110             622
##  8 FDP               2019-04-28 14:03:00 video            358           89             463
##  9 FDP               2019-04-28 10:40:57 photo             14           19             226
## 10 CSU               2019-04-28 12:35:00 photo            133           20             330
## # ... with 484 more rows
```

Dies führt augenscheinlich zunächst zum gleichen Ergebnis, hat aber ein Problem: kommen in einer der drei Facebook-Metriken fehlende Werte in Form von `NA` vor, ist das Ergebnis in `total_count` ebenfalls `NA`. Der Summenfunktion `sum()` können wir mit dem Argument `na.rm = TRUE` mitteilen, dass fehlende Werte nicht berücksichtigt werden sollen:

```
# Fehlende Werte zählen:
sum(is.na(df_mit_tc$total_count))
sum(is.na(df_mit_total2$total_count))
```

```
## [1] 0
## [1] 5
```

Wählen wir diese +-Variante, haben wir also 5 fehlende Werte in unserem `total_count`, bei der ersten Variante keine.

Nun können wir den Datensatz abspeichern:

```
write_csv(df_mit_tc, "data/df_reduziert.csv")
saveRDS(df_mit_tc, "data/df_reduziert.rds")
```

Lösung zur Übungsaufgabe 8.3:

Zunächst wählen wir nur die Woche vor der Wahl aus. Hierzu können wir die `timestamp`-Variable anfiltern – Text, der wie ein Datum aussieht, wir dabei automatisch in ein Datum konvertiert:

```
df_woche_vor_wahl <- filter(facebook_europawahl, timestamp >= "2019-05-20")
```

Nun gruppieren wir den Datensatz nach `party`:

```
df_group_by_party <- group_by(df_woche_vor_wahl, party)
```

Und schließlich berechnen wir mit `summarize()` die gewünschten Kennwerte:

```
summarize(df_group_by_party,
          M_comments = mean(comments_count, na.rm = TRUE),
          SD_comments = sd(comments_count, na.rm = TRUE),
          M_shares = mean(shares_count, na.rm = TRUE),
          SD_shares = sd(shares_count, na.rm = TRUE),
          M_reactions = mean(reactions_count, na.rm = TRUE),
          SD_reactions = sd(reactions_count, na.rm = TRUE))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## # A tibble: 14 x 7
##    party                          M_comments SD_comments M_shares SD_shares M_react
##    <chr>                               <dbl>       <dbl>    <dbl>     <dbl>   <
##  1 alternativefuerde                   1210.       1250.    1819.     2491.      46
##  2 B90DieGruenen                        106.         67.0     211.      244.       7
##  3 CDU                                  508.        897.      113.      293.       8
##  4 CSU                                  182.        148.       58.5      45.3      5
```

```
##  5 DiePARTEI                          92.5       137.      147.     175.        1874.
##  6 FamilienParteiDeutschlands          0.714       1.50     14        9.78        14.9
##  7 FDP                               106.        108.      111.     107.         802.
##  8 freie.waehler.bundesvereinigung    10.7        12.3      25.7     20.3         86
##  9 linkspartei                       161.        100.      217.     176.        1176.
## 10 oedp.de                             8.30       11.7      30.3     24.6        141.
## 11 Piratenpartei                      15.3        25.9      44.2     50.1        163.
## 12 SPD                              240.        277.      137.     120.         658.
## 13 tierschutzpartei                   82         110.      288      378.         999.
## 14 VoltDeutschland                    27.7        70.8      54.7     88.7        316.
```

Kapitel 9: Der Pipe-Operator %>%

Lösung zur Übungsaufgabe 9.1:

Dank Pipes können wir uns bei Übungsaufgabe 8.2 die ganzen Zwischendatensätze sparen. Es ist jedoch sinnvoll, vor dem Speichern ein Datensatz-Objekt zuzuweisen, da wir dieses auf zweierlei Arten speichern möchten. Auch nach dem erstmaligen Laden bietet es sich an, den Originaldatensatz zunächst als Objekt zuzuweisen:

```
facebook_europawahl <- read_csv("data/facebook_europawahl.csv")

bt_parteien <- c("alternativefuerde", "B90DieGruenen", "CDU", "CSU", "FDP", "linkspartei", "SPD")

df_reduziert <- facebook_europawahl %>%
  filter(party %in% bt_parteien) %>%
  select(party, timestamp, type, comments_count, shares_count, reactions_count) %>%
  mutate(total_count = sum(c(comments_count, shares_count, reactions_count), na.rm = TRUE))

df_reduziert
```

```
## # A tibble: 494 x 7
##    party             timestamp           type  comments_count shares_count reactions_count tot
##    <chr>             <dttm>              <chr>          <dbl>        <dbl>           <dbl>
##  1 B90DieGruenen     2019-04-28 06:00:01 video             70           28             215
##  2 FDP               2019-04-28 11:49:59 photo             16            9             262
##  3 CDU               2019-04-28 09:12:19 video            239          136             398
##  4 SPD               2019-04-28 13:06:09 photo            180           54             699
##  5 CSU               2019-04-28 08:21:00 photo            174           61             458
##  6 alternativefuerde 2019-04-28 14:55:00 link            1163         1499            3944
##  7 FDP               2019-04-28 06:18:18 photo             47          110             622
##  8 FDP               2019-04-28 14:03:00 video            358           89             463
##  9 FDP               2019-04-28 10:40:57 photo             14           19             226
```

```
## 10 CSU               2019-04-28 12:35:00 photo            133           20
## # ... with 484 more rows
```

Anschließend können wir den Datensatz wieder speichern:

```
write_csv(df_ungrouped, "data/df_reduziert.csv")
saveRDS(df_ungrouped, "data/df_reduziert.rds")
```

Die Schritte aus Übungsaufgabe 8.3 können wir ebenfalls in eine Pipe verpacken – da wir den Datensatz nicht speichern bzw. weiter mit diesem arbeiten, ist auch keine Objektzuweisung erforderlich:

```
facebook_europawahl %>%
  filter(timestamp >= "2019-05-20") %>%
  group_by(party) %>%
  summarize(M_comments = mean(comments_count, na.rm = TRUE),
            SD_comments = sd(comments_count, na.rm = TRUE),
            M_shares = mean(shares_count, na.rm = TRUE),
            SD_shares = sd(shares_count, na.rm = TRUE),
            M_reactions = mean(reactions_count, na.rm = TRUE),
            SD_reactions = sd(reactions_count, na.rm = TRUE))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## # A tibble: 14 x 7
##    party                          M_comments SD_comments M_shares SD_shares M_react
##    <chr>                               <dbl>       <dbl>    <dbl>     <dbl>    <
##  1 alternativefuerde                  1210.        1250.     1819.    2491.       4
##  2 B90DieGruenen                       106.          67.0     211.     244.       
##  3 CDU                                 508.         897.      113.     293.       
##  4 CSU                                 182.         148.       58.5     45.3      
##  5 DiePARTEI                            92.5        137.      147.     175.       1
##  6 FamilienParteiDeutschlands            0.714        1.50     14        9.78     
##  7 FDP                                 106.         108.      111.     107.       
##  8 freie.waehler.bundesvereinigung      10.7         12.3      25.7     20.3      
##  9 linkspartei                         161.         100.      217.     176.       1
## 10 oedp.de                               8.30        11.7      30.3     24.6      
## 11 Piratenpartei                        15.3         25.9      44.2     50.1      
## 12 SPD                                 240.         277.      137.     120.       
## 13 tierschutzpartei                     82          110.      288      378.       
## 14 VoltDeutschland                      27.7         70.8      54.7     88.7      
```

Kapitel 10: Daten umstrukturieren und zusammenfügen

Lösung zur Übungsaufgabe 10.1:

Da wir den Datensatz vom Wide- ins Long-Format transformieren, benötigen wir die Funktion `pivot_longer()`:

```
facebook_europawahl <- read_csv("data/facebook_europawahl.csv")

facebook_europawahl %>%
  select(id, party, timestamp, comments_count, shares_count, reactions_count) %>%
  pivot_longer(c(comments_count, shares_count, reactions_count), names_to = "metric")
```

```
## # A tibble: 2,706 x 5
##       id party            timestamp           metric          value
##    <dbl> <chr>            <dttm>              <chr>           <dbl>
##  1     1 oedp.de          2019-04-28 09:00:00 comments_count      0
##  2     1 oedp.de          2019-04-28 09:00:00 shares_count        4
##  3     1 oedp.de          2019-04-28 09:00:00 reactions_count     9
##  4     2 tierschutzpartei 2019-04-28 13:57:00 comments_count     17
##  5     2 tierschutzpartei 2019-04-28 13:57:00 shares_count      130
##  6     2 tierschutzpartei 2019-04-28 13:57:00 reactions_count   395
##  7     3 B90DieGruenen    2019-04-28 06:00:01 comments_count     70
##  8     3 B90DieGruenen    2019-04-28 06:00:01 shares_count       28
##  9     3 B90DieGruenen    2019-04-28 06:00:01 reactions_count   215
## 10     4 FDP              2019-04-28 11:49:59 comments_count     16
## # ... with 2,696 more rows
```

Lösung zur Übungsaufgabe 10.2:

Wir laden zunächst den zusätzlichen Datensatz:

```
facebook_codings <- read_csv("data/facebook_codings.csv")

facebook_codings
```

```
## # A tibble: 902 x 23
##       id topic100 topic200 topic310 topic320 topic331 topic332 topic330 topic341 topic342 topi
##    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <
##  1    34        0        1        0        0        1        1        1        0        0
##  2    62        1        0        0        0        0        0        1        0        0
##  3   122        0        1        0        0        0        0        1        0        0
##  4   178        0        0        0        0        0        0        1        0        1
```

```
##  5   300        1        0        0        0        0        0        1        0
##  6   303        0        1        0        0        0        0        1        0
##  7   419        0        1        0        0        0        0        1        0
##  8   421        1        0        0        0        0        0        1        0
##  9   429        0        0        0        0        0        0        1        0
## 10   448        1        0        0        0        0        0        1        0
## # ... with 892 more rows, and 8 more variables: topic380 <dbl>, topic391 <dbl>, topi
## #   topic300 <dbl>, topic998 <dbl>, topic999 <dbl>
```

Wie wir sehen, ist die `id`-Variable anders sortiert als im Datensatz `facebook_europawahl`. Wollen wir die Datensätze mittels `bind_cols()` zusammenfügen, müssten wir `facebook_codings` vorab mittels `arrange()` entsprechend `facebook_europawahl` sortieren.

Sicherer und genauso simpel ist allerdings `left_join()`:

```
joined_df <- facebook_europawahl %>%
  left_join(facebook_codings, by = "id")

joined_df
```

```
## # A tibble: 902 x 38
##       id URL   party timestamp           type  message link  comments_count shares_
##    <dbl> <chr> <chr> <dttm>              <chr> <chr>   <chr>          <dbl>        <
##  1     1 http~ oedp~ 2019-04-28 09:00:00 video "Guido~ http~             0
##  2     2 http~ tier~ 2019-04-28 13:57:00 photo "Aus u~ http~            17
##  3     3 http~ B90D~ 2019-04-28 06:00:01 video "Beim ~ http~            70
##  4     4 http~ FDP   2019-04-28 11:49:59 photo "Unser~ http~            16
##  5     5 http~ tier~ 2019-04-28 08:24:15 link  "Eine ~ http~             6
##  6     6 http~ CDU   2019-04-28 09:12:19 video "Freih~ http~           239
##  7     7 http~ SPD   2019-04-28 13:06:09 photo "Katar~ http~           180
##  8     8 http~ Pira~ 2019-04-28 17:36:30 video "Unser~ http~             0
##  9     9 http~ DieP~ 2019-04-28 07:44:28 link  "Der a~ http~            35
## 10    10 http~ CSU   2019-04-28 08:21:00 photo "#Klar~ http~           174
## # ... with 892 more rows, and 25 more variables: haha_count <dbl>, sad_count <dbl>,
## #   topic310 <dbl>, topic320 <dbl>, topic331 <dbl>, topic332 <dbl>, topic330 <dbl>,
## #   topic340 <dbl>, topic350 <dbl>, topic360 <dbl>, topic370 <dbl>, topic380 <dbl>,
## #   topic400 <dbl>, topic300 <dbl>, topic998 <dbl>, topic999 <dbl>
```

Wir sehen, dass der neue Datensätze weiterhin 902 Zeilen hat, aber nun alle 38 Variablen aus beiden Datensätzen umfasst. Zur Sicherheit sollten wir überprüfen, ob doppelte Werte in der `id`-Variablen vorkommen, um auszuschließen, dass Fälle bei der Join-Operation verdoppelt wurden. Dafür können wir die `distinct()`-Funktion nutzen, die nur einzigartige Werte der angegebenen Variablen ausgibt:

```
joined_df %>%
  distinct(id)
```

```
## # A tibble: 902 x 1
##       id
##    <dbl>
##  1     1
##  2     2
##  3     3
##  4     4
##  5     5
##  6     6
##  7     7
##  8     8
##  9     9
## 10    10
## # ... with 892 more rows
```

902 einzigartige Werte in `id` – es wurden also keine Fälle verdoppelt oder sind weggefallen.

Kapitel 11: Daten visualisieren

Für alle Aufgaben benötigen wir das Tidyverse und den Datensatz `facebook_europawahl.csv`. Zudem filtern wir in diesem nur die im Bundestag vertretenen Parteien an:

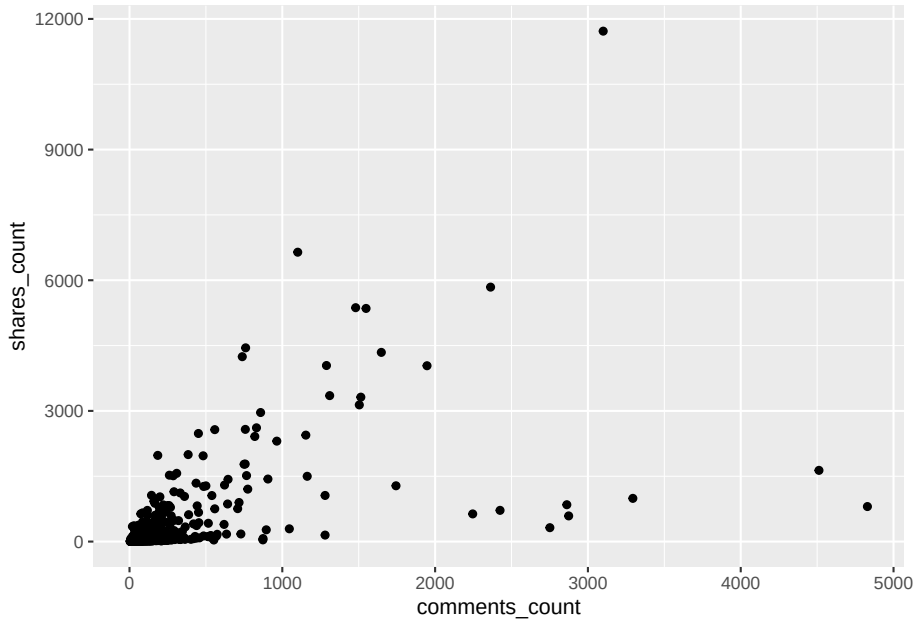
```
library(tidyverse)

bt_parteien <- c("alternativefuerde", "B90DieGruenen", "CDU", "CSU", "FDP", "linkspartei", "SPD")

facebook_europawahl <- read_csv("data/facebook_europawahl.csv") %>%
  filter(party %in% bt_parteien)
```

Lösung zur Übungsaufgabe 11.1:

Als *Aesthetics* weisen wir die Kommentarzahl (`comments_count`) der `x`-Achse, die Anzahl an Shares (`shares_count`) der `y`-Achse zu (oder andersum). Für Punktediagramme benötigen wie das *Geometric* `geom_point()`:

```
facebook_europawahl %>%
  ggplot(aes(x = comments_count, y = shares_count)) +
  geom_point()
```

```
## Warning: Removed 5 rows containing missing values (geom_point).
```

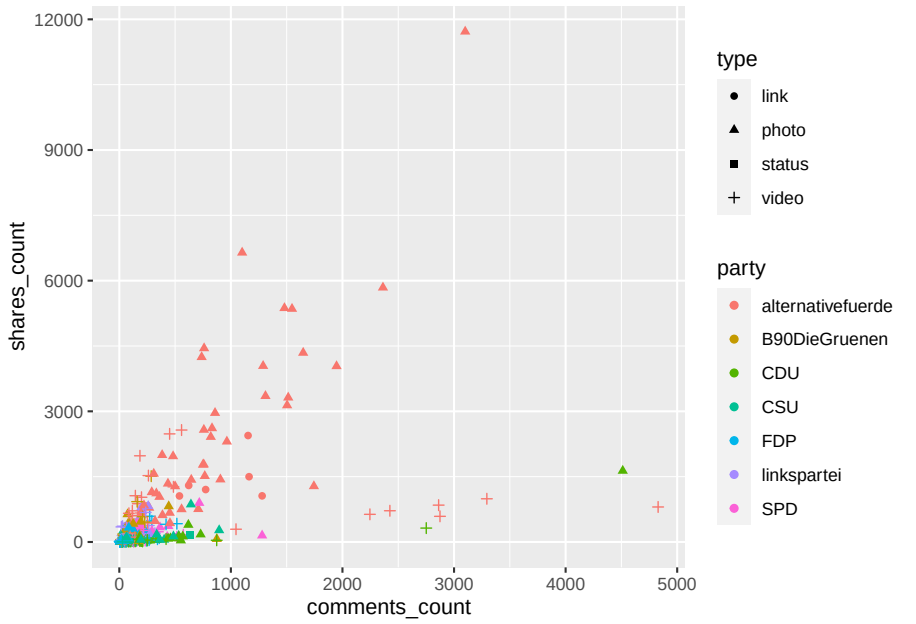


An der Warnmeldung sehen wir im Übrigen, dass 5 Posts nicht abgebildet werden – hierbei handelt es sich um `NA`-Werte.

Lösung zur Übungsaufgabe 11.2:

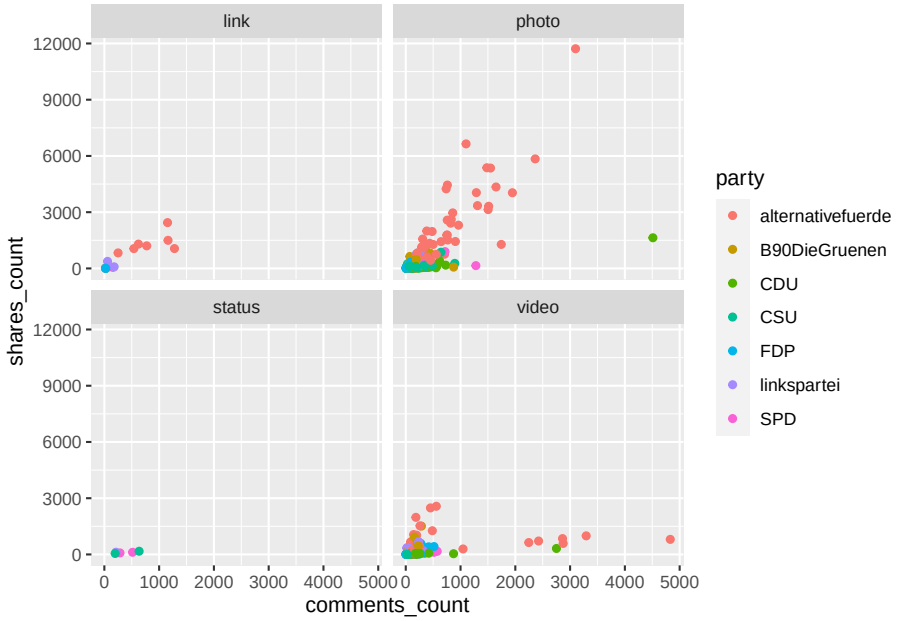
Eine Möglichkeit, sowohl Partei (`party`) als auch Typ des Posts (`type`) eine Aesthetic zuzuweisen – z. B. `color` (Punkt- bzw. Linienfarbe) für die Partei, `shape` (Punktform) für die den Typ des Beitrags:

```
facebook_europawahl %>%
  ggplot(aes(x = comments_count, y = shares_count, color = party, shape = type)) +
  geom_point()
```



Eine andere Möglichkeit besteht darin, zusätzlich mit Facets zu arbeiten:

```
facebook_europawahl %>%
  ggplot(aes(x = comments_count, y = shares_count, color = party)) +
  geom_point() +
  facet_wrap(~type)
```



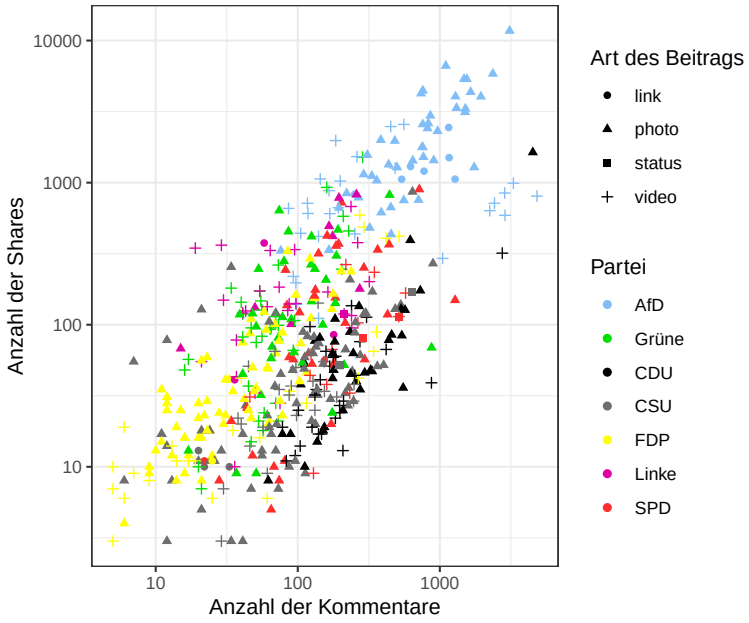
Lösung zur Übungsaufgabe 11.3:

Einige Möglichkeiten zur Verbesserung und Verschönerung des Plots:

- Verwendung eines *Themes*
- Achsen-/Skalenbeschriftungen
- Verwendung der tatsächlichen Parteifarben
- Gleiche Skalierung von x- und y-Achse (da gleiche zugrundeliegende Einheit); der bisher noch unbekannte Befehl `coord_fixed()` sorgt dafür, dass Einheiten auf der x- und y-Achse gleich dargestellt werden:

```
facebook_europawahl %>%
  ggplot(aes(x = comments_count, y = shares_count, color = party, shape = type)) +
  geom_point() +
  scale_y_log10(name = "Anzahl der Shares", ) +
  scale_x_log10(name = "Anzahl der Kommentare",) +
  scale_color_manual(name = "Partei",
                     values = c("CDU" = "#000000",
                                "CSU" = "#6E6E6E",
                                "SPD" = "#FE2E2E",
                                "alternativefuerde" = "#81BEF7",
                                "FDP" = "#FFFF00",
                                "linkspartei" = "#DF01A5",
                                "B90DieGruenen" = "#01DF01"),
                     labels = c("alternativefuerde" = "AfD",
```

```
                                "linkspartei" = "Linke",
                                "B90DieGruenen" = "Grüne")) +
  scale_shape_discrete(name = "Art des Beitrags") +
  theme_bw() +
  coord_fixed()
```



Kapitel 12: Arbeiten mit Textdaten

Lösung zur Übungsaufgabe 12.1:

```
experiment <- tibble(experimentalgruppe = c("Gruppe A", "Gruppe B", "Gruppe A", "Gruppe C"))
```

Ziel war es, lediglich die Gruppenkennung in einer neuen Spalte hinzuzufügen. Dafür gibt es viele verschiedene Möglichkeiten, z. B.:

- `str_sub()`: Lediglich das letzte Zeichen als Substring auswählen

```
experiment %>%
  mutate(gruppe_kurz = str_sub(experimentalgruppe, -1, -1))
```

```
## # A tibble: 4 x 2
```

```
##   experimentalgruppe gruppe_kurz
##   <chr>              <chr>
## 1 Gruppe A           A
## 2 Gruppe B           B
## 3 Gruppe A           A
## 4 Gruppe C           C
```

- `str_replace()`: `"Gruppe "` durch einen leeren String `""` ersetzen:

```
experiment %>%
  mutate(gruppe_kurz = str_replace(experimentalgruppe, "Gruppe ", ""))
```

```
## # A tibble: 4 x 2
##   experimentalgruppe gruppe_kurz
##   <chr>              <chr>
## 1 Gruppe A           A
## 2 Gruppe B           B
## 3 Gruppe A           A
## 4 Gruppe C           C
```

Lösung zur Übungsaufgabe 12.2:

```
imdb_urls <- c(
  "https://www.imdb.com/title/tt6751668/?ref_=hm_fanfav_tt_4_pd_fp1",
  "https://www.imdb.com/title/tt0260991/",
  "www.imdb.com/title/tt7282468/reviews",
  "https://m.imdb.com/title/tt4768776/"
)
```

Zur Extraktion der IDs bietet sich `str_extract()` an mit RegEx an. Mit dem RegEx-String `"tt\\d{7}"` matchen wir jegliche IMDb-ID, die immer dem Schema `"tt"`, gefolgt von 7 Ziffern folgen:

```
imdb_urls %>%
  str_extract("tt\\d{7}")
```

```
## [1] "tt6751668" "tt0260991" "tt7282468" "tt4768776"
```

Lösung zur Übungsaufgabe 12.3:

```
adressen = c(
    "Platz der Republik 1, D-11011 Berlin",
    "Dr.-Karl-Renner-Ring 3, A-1017 Wien",
    "Bundesplatz 3, CH-3005 Bern"
  )
```

Sinnvoll ist es, nach und nach die einzelnen Adress-Bestandteile auszuwählen.

- Der Straßenname ist dabei der komplizierteste Part, da er aus Groß- und Kleinbuchstaben, Bindestrichen, Leerzeichen und Punkten bestehen kann. Eine Möglichkeit ist es daher, all diese Zeichen als eigene Übereinstimmungsgruppe zu definieren: `[A-Za-z-\\s\\.]+`. Da keine Ziffern im Straßennamen vorkommen, können wir das jedoch abkürzen, indem wir für den Straßennamen alles matchen, was *keine* Ziffer ist: `\\D+`. Durch Klammern können wir angegeben, dass dies der erste Bestandteil der Adresse ist, den wir extrahieren möchten: `"(\\D+)"`.
- Es folgt (in diesem Beispiel) stets Whitespace und die Hausnummer, was wir mit `\\s\\d+` matchen können. Da wir das Leerzeichen nicht extrahieren möchten, ziehen wir die nächsten Klammern lediglich um das `\\d+`: `"(\\D+)\\s(\\d+)"`.
- Es folgen nun ein Komma, Whitespace und ein oder zwei Großbuchstaben für den Ländercode; letztere können wir beispielsweise mit `[A-Z]{1,2}` matchen. Erneut wollen wir nur die 1-2 Großbuchstaben extrahieren: `"(\\D+)\\s(\\d+),\\s([A-Z]{1,2})"`.
- Nun kommt ein Bindestrich und die 4-5-stellige Postleitzahl: `"(\\D+)\\s(\\d+),\\s([A-Z]{1,2})-(\\d{4,5}`
- Schließlich folgt noch ein Whitespace und die Stadt, die wir z. B. schnell mittels `\\D+` (alles außer Ziffern) matchen können: `"(\\D+)\\s(\\d+),\\s([A-Z]{1,2})-(\\d{4,5})\\s(\\D+)"`

Diesen String können wir nun `str_match()` übergeben:

```
adr_string <- "(\\D+)\\s(\\d+),\\s([A-Z]{1,2})-(\\d{4,5})\\s(\\D+)"

adressen %>%
  str_match(adr_string)
```

```
##      [,1]                                    [,2]                    [,3] [,4] [,5]    [,6]
## [1,] "Platz der Republik 1, D-11011 Berlin" "Platz der Republik"    "1"  "D"  "11011" "Berlin"
## [2,] "Dr.-Karl-Renner-Ring 3, A-1017 Wien"  "Dr.-Karl-Renner-Ring" "3"  "A"  "1017"  "Wien"
## [3,] "Bundesplatz 3, CH-3005 Bern"          "Bundesplatz"          "3"  "CH" "3005"  "Bern"
```

Als Resultat erhalten wir eine Matrix, in der in der ersten Spalten der komplette gematchte String sowie in den folgenden Spalten die einzelnen gematchten Bestandteile, definiert durch runde Klammern `()` stehen.

Kapitel 15: Web Scraping

Für die Lösungen wird das Package `polite` in Kombination mit `rvest` verwendet. Die Extraktion der HTML-Elemente unterscheidet sich jedoch nicht, wenn nur `rvest` verwendet wird.

```
library(polite)
library(rvest)
```

Lösung zur Übungsaufgabe 15.1:

Wir besuchen zunächst den Artikel und finden über SelectorGadget oder die Untersuchen-Funktion heraus, dass

- der Artikel in der CSS-Klasse `.sz-article` steht
- die gesuchten Inhalte in den CSS-Klassen `.css-11lvjqt` (Datum und Uhrzeit), `.css-1keap3i` (Kicker), `.css-1kuo4az` (Überschrift) und `.css-1psf6fc` (Lead) stehen.

Nun stellen wir uns mit `bow()` dem Server vor (wenn nur `rvest` genutzt wird, wird dieser Schritt übersprungen):

```
url <- "https://www.sueddeutsche.de/sport/hsv-kiel-hecking-ausgleich-1.4931360"
sz1 <- bow(url)
```

Und scrapen die Seite (analog zu `read_html()` in `rvest`):

```
html_content <- scrape(sz1)
```

Nun extrahieren wir die gewünschten Elemente:

```
html_content %>%
  html_nodes(".css-11lvjqt, .css-1keap3i, .css-1kuo4az, .css-1psf6fc") %>%
  html_text() %>%
  str_squish()
```

```
## [1] "9. Juni 2020, 9:20 Uhr"
## [2] "HSV in der zweiten Liga"
## [3] "\"Das sind Dinge, die sehr, sehr weh tun\""
## [4] "So wird es eng mit dem Aufstieg: In einer wilden Schlussphase kassiert der Haml
```

Lösung zur Übungsaufgabe 15.2:

Ein Problem ist hier, dass viele Links auf der Seite stehen, wir aber nur einen abgreifen möchten. Dies können wir erreichen, indem wir zunächst lediglich den Artikel selbst über die Klasse `.sz-article` auswählen, dann nur die Textabsätze mit dem HTML-Tag `p` und schließlich Links mit dem HTML-Tag `a`:

```
html_content %>%
  html_nodes(".sz-article") %>%
  html_nodes("p") %>%
  html_nodes("a") %>%
  html_attr("href")
```

```
## character(0)
```

Lösung zur Übungsaufgabe 15.3:

Wir wandeln den obigen Code in eine Funktion um:

```
scrape_sz <- function(url) {
  # Vorstellen
  sz <- bow(url)

  # Scrapen
  html_content <- scrape(sz)

  # Interessierende HTML-Elemente extrahieren
  info <- html_content %>%
    html_nodes(".css-11lvjqt, .css-1keap3i, .css-1kuo4az, .css-1psf6fc") %>%
    html_text() %>%
    str_squish()

  # In Tibble umwandeln
  info_tibble <- tibble(
    release = info[1],
    kicker = info[2],
    headline = info[3],
    lead = info[4]
  )

  # Tibble zurückgeben
  return(info_tibble)
}
```

Und testen dies am zweiten Artikel:

```
scrape_sz("https://www.sueddeutsche.de/sport/stuttgart-hsv-2-bundesliga-castro-1.4921867")
```

```
## # A tibble: 1 x 4
##   release           kicker     headline                    lead
##   <chr>             <chr>      <chr>                       <chr>
## 1 28. Mai 2020, 22~ 2. Bundes~ Castro schockt den HSV in ~ Der VfB Stuttgart liegt gegen den H
```

Kapitel 17: Automatisierte Inhaltsanalyse: Einführung und Grundbegriffe

Für alle Aufgaben benötigen wir Quanteda und müssen den Facebook-Datensatz wie gewohnt filtern:

```
library(tidyverse)
library(quanteda)

bt_parteien <- c("alternativefuerde", "B90DieGruenen", "CDU", "CSU", "FDP", "linkspart

facebook_europawahl <- read_csv("data/facebook_europawahl.csv") %>%
  filter(party %in% bt_parteien)
```

Lösung zur Übungsaufgabe 17.1:

Wir erstellen das Korpus-Objekt mit der `corpus()`-Funktion.

```
fb_corpus <- corpus(facebook_europawahl, docid_field = "id", text_field = "message")
```

```
## Warning: NA is replaced by empty string
```

Lösung zur Übungsaufgabe 17.2:

Die einzelnen Schritte zur Tokenisierung können wir in eine Pipe packen:

```
fb_tokens <- tokens(fb_corpus,                  # Erzeuge Tokens
                    remove_punct = TRUE,
                    remove_numbers = TRUE,
                    remove_symbols = TRUE,
                    remove_url = TRUE) %>%
  tokens_tolower() %>%                          # Kleinschreibung
  tokens_remove(stopwords("german")) %>%        # Deutsche Stoppwörter entfernen
  tokens_ngrams(n = c(1, 2, 3))                 # Erzeuge Uni-, Bi- und Trigramme
```

Lösung zur Übungsaufgabe 17.3:

Zunächst erstellen wir die DFM:

```
fb_dfm <- dfm(fb_tokens)
```

Die Top-Features pro Partei:

```
topfeatures(fb_dfm, groups = "party")
```

```
## $alternativefuerde
##         afd     unseren        dass deutschland      finden  europawahl  kandidaten          ma
##          85          53          48          37          37          33          32           3
##
## $B90DieGruenen
##      europa      wählen         mai        grün   gruene.de       teile       innen klimaschut
##          24          23          18          18          17          17          16           1
##
## $CDU
##               #unsereuropa                  #26maicdu                     europa
##                         46                         38                         23
##   <U+0001F1EA><U+0001F1FA>                   annegret          kramp-karrenbauer annegret_kram
##                         16                         16                         16
##
## $CSU
##       manfred         weber manfred_weber        europa    europawahl        markus          sö
##            41            41            41            35            33            24
##         heute
##            19
##
## $FDP
##                          #chancennutzen                                  europa               <U
##                                      75                                      59
##                                 #ep2019                 #chancennutzen_#ep2019
##                                      51                                      48
##                #ep2019_#europawahl2019 #chancennutzen_#ep2019_#europawahl2019
##                                      32                                      32
##        europa_<U+0001F1EA><U+0001F1FA>
##                                      22
##
## $linkspartei
##            europa             heute            martin        schirdewan martin_schirdewan
##                19                11                 8                 8                 8
##         riexinger   bernd_riexinger             özlem
##                 7                 7                 7
##
## $SPD
##               europa #europaistdieantwort                 mehr             katarina
##                   33                   24                   15                   13
##      katarina_barley                dafür      soziales_europa                 geht
##                   12                   11                   10                    9
```

Das sich darunter fast keine Trigramme (außer eine Hashtag-Kominbation

bei der FDP) befinden, ziehen wir daraus vorerst keinen Mehrwert. Die Zeichenkette `\U0001f1ea\U0001f1fa` verweist auf Fehler bei der Bereinigung der Texte – hierbei handelt es sich um den Code für das Europaflaggen-Emoji, der eigentlich durch `remove_symbols = TRUE` hätte entfernt werden sollen. Auch einige URL-Bestandteile wurden nicht korrekt entfernt, ebenso gibt es ein paar falsche Worttrennungen, z. B. wenn ein Gendersternchen enthalten war. Hier sollte also manuell nachgebessert werden.

Um die Hashtags zu analysieren, erstellen wir eine DFM, die nur diese enthält:

```
dfm_hashtags <- dfm_select(fb_dfm, "#*") # Wählt alle Hashtags aus
```

Wir können uns nun wieder mittels `topfeatures()` die häufigsten Hashtags ausgeben lassen:

```
topfeatures(dfm_hashtags) # allgemein
```

```
##                          #chancennutzen                                   #ep2019
##                                      75                                         51
##                           #unsereuropa                                  #26maicdu
##                                      47                                         38
##               #ep2019_#europawahl2019 #chancennutzen_#ep2019_#europawahl2019
##                                      32                                         32
##                                 #europa
##                                      23
```

```
topfeatures(dfm_hashtags, groups = "party") # getrennt nach Partei
```

```
## $alternativefuerde
##               #greding       #greding_bayern #greding_bayern_heute          #grundre
##                      1                     1                      1
## #chancennutzen_#bpt19          #unsereuropa    #unsereuropa_steht              #ep
##                      0                     0                      0
##
## $B90DieGruenen
##                           #europawahl                                  #europa
##                                    11                                       10
##                          #zusammenhalt                        #europawahl_mai
##                                      3                                        3
## #europa_einzigartiges_friedensprojekt           #klimaschutz_#zusammenhalt
##                                      3                                        2
##                      #europawahl_teile
##                                      1
##
```

```
## $CDU
##              #unsereuropa                  #26maicdu    #unsereuropa_#26maicdu #unsereuropa_<U+
##                        46                         38                        11
##                   #europa             #thepowerofwe                      #cdu    #26maicdu_#un
##                         7                          6                         5
##
## $CSU
##                           #klartext                  #klartext_unseres #klartext_unseres_spit
##                                  16                                  8
##                  #klartext_bayerns #klartext_bayerns_ministerpräsident
##                                  6                                  6
##                          #wahlarena                       #miasanbayern
##                                  4                                  3
##                            #tvduell
##                                  1
##
## $FDP
##                           #chancennutzen                                #ep2019
##                                       75                                    51
##                          #europawahl2019               #ep2019_#europawahl2019 #chancennutzen_#
##                                       36                                    32
##                                   #live                                 #bpt19
##                                       6                                     4
##                           #teameurope
##                                       3
##
## $linkspartei
##      #1europafueralle          #grundrechte        #chancennutzen                #bpt19 #chanc
##                     1                     0                     0                     0
##    #unsereuropa_steht               #ep2019       #europawahl2019   #grundrechte_stehen
##                     0                     0                     0                     0
##
## $SPD
##      #europaistdieantwort                   #europa              #grundgesetz            #sozi
##                        24                         6                         3
##                    #rezo                      #evp                #evp_sagen            #euro
##                         2                         1                         1
```

Kapitel 18: Textdeskription und einfache Textvergleiche

Auch hier benötigen wir Quanteda und müssen den Facebook-Datensatz wie gewohnt filtern. Zudem laden wir auch Tidytext:

```
library(tidyverse)
library(tidytext)
library(quanteda)

bt_parteien <- c("alternativefuerde", "B90DieGruenen", "CDU", "CSU", "FDP", "linkspart

facebook_europawahl <- read_csv("data/facebook_europawahl.csv") %>%
  filter(party %in% bt_parteien)
```

Lösung zur Übungsaufgabe 18.1:

Wir erzeugen zunächst wie gehabt Korpus-, Tokens- und DFM-Objekte:

```
fb_corpus <- corpus(facebook_europawahl, docid_field = "id", text_field = "message")
```

```
## Warning: NA is replaced by empty string
```

```
fb_tokens <- tokens(fb_corpus,
                    remove_punct = TRUE,
                    remove_numbers = TRUE,
                    remove_symbols = TRUE,
                    remove_url = TRUE) %>%
  tokens_tolower() %>%
  tokens_remove(stopwords("german"))

fb_dfm <- dfm(fb_tokens)
```

Zunächst ein Blick auf die einfachen Worthäufigkeiten:

```
featfreq(fb_dfm) %>%
  tidy() %>%
  arrange(desc(x))
```

```
## # A tibble: 6,896 x 2
##    names                      x
##    <chr>                  <dbl>
##  1 "europa"                 208
##  2 "dass"                   101
##  3 "europawahl"              96
##  4 "mehr"                    92
##  5 "heute"                   92
##  6 "afd"                     90
##  7 "\U0001f1ea\U0001f1fa"    87
```

```
## 8 "wählen"                    79
## 9 "#chancennutzen"            75
## 10 "unseren"                  72
## # ... with 6,886 more rows
```

Wenig überraschend fallen Begriffe wie “Europa”, “Deutschland” und “Europawahl” sehr häufig. Es zeigen sich aber auch bereits ein paar Probleme, die wir bereits von der vorherigen Übung kennen, z. B. dass Stoppwörter wie “dass” weiterhin im Datensatz verbleiben. Auch das Europaflaggen-Emoji wird offenbar sehr häufig verwendet.

Wir können uns die wichtigsten Begriffe auch als Wordcloud anzeigen lassen:

```
textplot_wordcloud(fb_dfm, max_words = 100)
```

```
## Warning in graphics::strwidth(word[i], cex = size[i]): conversion failure on 'ðŸ‡ªðŸ‡º' in 'mb

## Warning in graphics::strwidth(word[i], cex = size[i]): conversion failure on 'ðŸ‡ªðŸ‡º' in 'mb

## Warning in graphics::strwidth(word[i], cex = size[i]): conversion failure on 'ðŸ‡ªðŸ‡º' in 'mb

## Warning in graphics::strwidth(word[i], cex = size[i]): conversion failure on 'ðŸ‡ªðŸ‡º' in 'mb

## Warning in graphics::strwidth(word[i], cex = size[i]): conversion failure on 'ðŸ‡ªðŸ‡º' in 'mb

## Warning in graphics::strwidth(word[i], cex = size[i]): conversion failure on 'ðŸ‡ªðŸ‡º' in 'mb

## Warning in graphics::strwidth(word[i], cex = size[i]): conversion failure on 'ðŸ‡ªðŸ‡º' in 'mb

## Warning in graphics::strwidth(word[i], cex = size[i]): conversion failure on 'ðŸ‡ªðŸ‡º' in 'mb

## Warning in text.default(x1, y1, word[i], cex = (1 + adjust) * size[i], offset = 0, : conversio
## substituted for <f0>

## Warning in text.default(x1, y1, word[i], cex = (1 + adjust) * size[i], offset = 0, : conversio
## substituted for <9f>

## Warning in text.default(x1, y1, word[i], cex = (1 + adjust) * size[i], offset = 0, : conversio
## substituted for <87>

## Warning in text.default(x1, y1, word[i], cex = (1 + adjust) * size[i], offset = 0, : conversio
## substituted for <aa>
```

```
## Warning in text.default(x1, y1, word[i], cex = (1 + adjust) * size[i], offset = 0,
## substituted for <f0>

## Warning in text.default(x1, y1, word[i], cex = (1 + adjust) * size[i], offset = 0,
## substituted for <9f>

## Warning in text.default(x1, y1, word[i], cex = (1 + adjust) * size[i], offset = 0,
## substituted for <87>

## Warning in text.default(x1, y1, word[i], cex = (1 + adjust) * size[i], offset = 0,
## substituted for <ba>

## Warning in text.default(x1, y1, word[i], cex = (1 + adjust) * size[i], offset = 0,

## Warning in text.default(x1, y1, word[i], cex = (1 + adjust) * size[i], offset = 0,
```

Kollokationen verweisen vor allem auf die Namen der Kandidat*innen sowie beliebte Hashtag-Kombinationen:

```
textstat_collocations(fb_tokens) %>%
  as_tibble() %>%
  arrange(desc(count))
```

```
## # A tibble: 1,007 x 6
##    collocation                   count count_nested length lambda     z
##    <chr>                         <int>        <int>  <dbl>  <dbl> <dbl>
##  1 "manfred weber"                  55            0      2  13.2   8.68
##  2 "#chancennutzen #ep2019"         48            0      2   9.60 15.8
##  3 "#ep2019 #europawahl2019"        32            0      2   9.47 16.1
##  4 "unseren kandidaten"             30            0      2   8.13 13.9
##  5 "europa \U0001f1ea\U0001f1fa"    27            0      2   3.77 15.5
##  6 "finden infos"                   27            0      2   8.97 14.5
##  7 "infos europawahl"               26            0      2   7.54 13.0
##  8 "europawahl unseren"             24            0      2   4.75 17.3
##  9 "markus söder"                   23            0      2  10.8  11.4
## 10 "europawahl manfred"             17            0      2   4.58 14.7
## # ... with 997 more rows
```

Kookkurenzen zeigen, dass wohl vor allem die AfD thematisiert wurde (ob durch sich selbst oder andere Parteien, ergeht hieraus nicht). Auch hier sehen wir, dass noch URL-Bestandteile (`"https"`) im Datensatz verbleiben; diese sollten also noch manuell gefiltert werden.

```
fcm(fb_tokens) %>%
  tidy() %>%
  arrange(desc(count))
```

```
## # A tibble: 472,434 x 3
##    document               term       count
##    <chr>                  <chr>      <dbl>
##  1 "afd"                  afd          109
##  2 "europa"               europa        96
##  3 "afd"                  unseren       85
##  4 "\U0001f1ea\U0001f1fa" europa        70
##  5 "unseren"              kandidaten    60
##  6 "afd"                  finden        59
##  7 "afd"                  europawahl    59
##  8 "#chancennutzen"       europa        58
##  9 "ab"                   afd           58
## 10 "manfred"              weber         58
## # ... with 472,424 more rows
```

Für Keyness-Analysen benötigen wir zunächst eine nach Parteien gruppierte DFM:

```
fb_dfm_grouped <- dfm(fb_tokens, groups = "party")
```

Wir können uns nun die Keywords je Partei ausgeben lassen – im Beispiel für SPD und Grüne:

```
textstat_keyness(fb_dfm_grouped, target = "SPD") %>%
  as_tibble()
```

```
## # A tibble: 6,896 x 5
##    feature               chi2        p n_target n_reference
##    <chr>                <dbl>    <dbl>    <dbl>       <dbl>
##  1 #europaistdieantwort 385.  0.             24           0
##  2 katarina             171.  0.             13           2
##  3 barley               143.  0.             12           3
##  4 soziales             130.  0.             13           6
##  5 bullmann             100.  0.              7           0
##  6 udo                  100.  0.              7           0
##  7 andrea                83.7 0.              6           0
##  8 nahles                70.2 0.              6           1
##  9 sozialen              53.3 2.90e-13        7           5
## 10 europa                41.6 1.15e-10       33         175
## # ... with 6,886 more rows
```

```
textstat_keyness(fb_dfm_grouped, target = "B90DieGruenen") %>%
  as_tibble()
```

```
## # A tibble: 6,896 x 5
##    feature       chi2     p n_target n_reference
##    <chr>        <dbl> <dbl>    <dbl>       <dbl>
##  1 grün         219.      0       18           0
##  2 gruene.de    206.      0       17           0
##  3 freund       167.      0       14           0
##  4 innen        158.      0       16           3
##  5 zusammenhalt 154.      0       14           1
##  6 teile        152.      0       17           5
##  7 ska          116.      0       10           0
##  8 #europawahl   88.8     0       11           4
##  9 robert        81.0     0        9           2
## 10 habeck        78.3     0        8           1
## # ... with 6,886 more rows
```

Neben wenigen inhaltlichen Begriffe (`"soziales"`, `"zusammenhalt"`) werden die Listen dominiert durch Eigennamen; um tatsächliche inhaltliche Keywords zu bestimmen, würde es sich daher lohnen, Namen von Kandidat*innen, Parteien etc. aus der DFM zu löschen.

Kapitel 19: Diktionärbasierte Ansätze

Erneut hier benötigen wir Quanteda und müssen den Facebook-Datensatz wie gewohnt filtern. Zudem laden wir auch Tidytext:

```
library(tidyverse)
library(tidytext)
library(quanteda)

bt_parteien <- c("alternativefuerde", "B90DieGruenen", "CDU", "CSU", "FDP", "linkspartei", "SPD")

facebook_europawahl <- read_csv("data/facebook_europawahl.csv") %>%
  filter(party %in% bt_parteien)
```

Wir laden außerdem, wie in der Aufgabenstellung angegeben, die Dictionaries:

```
sentiws_pos <- read_delim("data/SentiWS_v2.0_Positive.txt", col_names = c("word", "value", "flect
  mutate(sentiment = "positive")
sentiws_neg <- read_delim("data/SentiWS_v2.0_Negative.txt", col_names = c("word", "value", "flect
  mutate(sentiment = "negative")

sentiws <- sentiws_pos %>%
  bind_rows(sentiws_neg) %>%
  separate(word, c("word", "type"), sep = "\\|") %>%
  mutate(word = str_c(word, flections, sep = ",")) %>%
  select(-flections, -type) %>%
  separate_rows(word, sep = ",") %>%
  na.omit()
```

Außerdem erzeugen wir einen Korpus der Facebook-Posts:

```
fb_corpus <- corpus(facebook_europawahl, docid_field = "id", text_field = "message")
```

```
## Warning: NA is replaced by empty string
```

Lösung zur Übungsaufgabe 19.1:

Wir extrahieren zunächst die positiven und negativen Begriffe als Vektoren mittels `filter()` und `pull()`:

```
senti_pos <- sentiws %>%
  filter(sentiment == "positive") %>%
  pull(word, sentiment)
```

```
senti_neg <- sentiws %>%
  filter(sentiment == "negative") %>%
  pull(word, sentiment)
```

Mit beiden Vektoren können wir nun ein Quanteda-Dictionary mittels `dictionary()` erstellen. `dictionary()` konvertiert automatisch in Kleinschreibung – soll dies nicht geschehen, kann das mit dem Argument `tolower = FALSE` angepasst werden. Für unsere Zwecke ist eine Konvertierung in Kleinschreibung aber sinnvoll, da wir auch für die DFM in der Regel alle Wörter in Kleinschreibung umwandeln.

```
sentiment_dictionary <- dictionary(list(
  positiv = senti_pos,
  negativ = senti_neg
))

sentiment_dictionary
```

```
## Dictionary object with 2 key entries.
## - [positiv]:
##   - abmachung, abmachungen, abschluß, abschluss, abschlusse, abschlusses, abschlüsse
## - [negativ]:
##   - abbau, abbaus, abbaues, abbauen, abbaue, abbauten, abbruch, abbruches, abbrüche
```

Um die absoluten Häufigkeiten auszuzählen, genügt der `dfm()`-Befehl mit Angabe unseres Dictionaries und einer Gruppierung nach `party`:

```
dfm(fb_corpus, dictionary = sentiment_dictionary, groups = "party")
```

```
## Document-feature matrix of: 7 documents, 2 features (0.0% sparse) and 1 docvar.
##                    features
## docs                positiv negativ
##   alternativefuerde     637     467
##   B90DieGruenen         119      26
##   CDU                   224      28
##   CSU                   197      47
##   FDP                   444      90
##   linkspartei            88      42
## [ reached max_ndoc ... 1 more document ]
```

Das Verhältnis von positivem zu negativem Sentiment erhalten wir durch anschließende Gewichtung mit `dfm_weight()`:

```
dfm(fb_corpus, dictionary = sentiment_dictionary, groups = "party") %>%
  dfm_weight(scheme = "prop")
```

```
## Document-feature matrix of: 7 documents, 2 features (0.0% sparse) and 1 docvar.
##                    features
## docs                  positiv   negativ
##   alternativefuerde 0.5769928 0.4230072
##   B90DieGruenen     0.8206897 0.1793103
##   CDU               0.8888889 0.1111111
##   CSU               0.8073770 0.1926230
##   FDP               0.8314607 0.1685393
##   linkspartei       0.6769231 0.3230769
## [ reached max_ndoc ... 1 more document ]
```

Die AfD verzeichnet also den größten Anteil negativen Sentiments, gefolgt von der Linkspartei. Alle anderen Parteien kommunizieren sehr positiv.

Interessieren wir uns für den Anteil, den positive und negative Begriffe am Gesamt-Wortschatz der Parteien-Posts ausmachen, gewichten wir die DFM vor der Anwendung des Dictionaries:

```
dfm(fb_corpus, groups = "party") %>%
  dfm_weight(scheme = "prop") %>%
  dfm(dictionary = sentiment_dictionary)
```

```
## Document-feature matrix of: 7 documents, 2 features (0.0% sparse) and 1 docvar.
##                    features
## docs                   positiv    negativ
##   alternativefuerde 0.03748571 0.02828571
##   B90DieGruenen     0.04621212 0.01060606
##   CDU               0.08509848 0.01412114
##   CSU               0.06289111 0.01752190
##   FDP               0.07307567 0.01607665
##   linkspartei       0.04972973 0.02540541
## [ reached max_ndoc ... 1 more document ]
```

Lösung zur Übungsaufgabe 19.2:

Um den SentiWS als gewichtetes Lexikon zu nutzen und die Polaritätswerte zu berücksichtigen, benutzen wir Tidytext. Zunächst Tokenisieren wir unseren Datensatz. Auch dabei wird direkt in Kleinschreibung konvertiert:

```
tidy_facebook <- facebook_europawahl %>%
  unnest_tokens(word, message) %>%
```

```
  select(id, party, word)

tidy_facebook
```

```
## # A tibble: 32,031 x 3
##       id party         word
##    <dbl> <chr>         <chr>
##  1     3 B90DieGruenen beim
##  2     3 B90DieGruenen wahlkampf
##  3     3 B90DieGruenen camp
##  4     3 B90DieGruenen in
##  5     3 B90DieGruenen berlin
##  6     3 B90DieGruenen waren
##  7     3 B90DieGruenen gestern
##  8     3 B90DieGruenen hunderte
##  9     3 B90DieGruenen freiwillige
## 10     3 B90DieGruenen die
## # ... with 32,021 more rows
```

Per `inner_join()` können wir nun die Sentimentwerte anfügen:

```
tidy_sentiments <- tidy_facebook %>%
  inner_join(sentiws)
```

```
## Joining, by = "word"
```

```
tidy_sentiments
```

```
## # A tibble: 1,969 x 5
##       id party         word           value sentiment
##    <dbl> <chr>         <chr>          <dbl> <chr>
##  1     3 B90DieGruenen freiwillige   0.004  positive
##  2     3 B90DieGruenen mobilisieren  0.004  positive
##  3     3 B90DieGruenen freiwilligen  0.004  positive
##  4     3 B90DieGruenen freiwillige   0.004  positive
##  5     3 B90DieGruenen gewinnen      0.004  positive
##  6     4 FDP           neuer         0.004  positive
##  7     6 CDU           gute          0.372  positive
##  8     7 SPD           gut           0.372  positive
##  9     7 SPD           zusammenhält  0.0834 positive
## 10    10 CSU           langsam      -0.0167 negative
## # ... with 1,959 more rows
```

Schließlich gruppieren wir per `group_by()` nach `party` und berechnen den Mittelwert des Sentiments:

```
tidy_sentiments %>%
  group_by(party) %>%
  summarise(mean_sentiment = mean(value), .groups = "drop")
```

```
## # A tibble: 7 x 2
##   party             mean_sentiment
##   <chr>                      <dbl>
## 1 alternativefuerde        -0.0277
## 2 B90DieGruenen             0.0530
## 3 CDU                       0.0643
## 4 CSU                       0.0125
## 5 FDP                       0.0341
## 6 linkspartei               0.0425
## 7 SPD                       0.0925
```

Auch hier weist die AfD das negativste Sentiment auf. Allerdings erscheint die Kommunikation der Linkspartei auf diesem Wege deulich positiver als zuvor.

Kapitel 21: Topic Modeling

Wie in der Aufgabenstellung geschrieben, laden wir die Daten dieses Mal aus dem `quanteda.corpora`-Package. Wir laden außerdem die bereits bekannten Packages zum `tidyverse` zum Datenhandling, `quanteda` zur Vorbereitung der Textdaten, sowie `stm` für das Topic Modeling.

```
library(tidyverse)
library(stm)
library(quanteda)

guardian_corpus <- quanteda.corpora::download("data_corpus_guardian")

guardian_corpus
```

Lösung zur Übungsaufgabe 21.1:

Die Preprocessing-Schritte führen wir wie gehabt mit Quanteda durch. Wir erzeugen zunächst eine DFM mit `dfm()` und können dabei irrelevante Token wie Satzzeichen, Zahlen, Symbole und Stoppwörter entfernen. Um die Berechnung zu erleichtern, trimmen wir die DFM zusätzlich um besonders seltene und häufige Wörter mit `dfm_trim()` (ich habe mich hier für alle Wörter die in mehr

als 50% der Artikel sowie in weniger als 2% der Artikel vorkommen entschieden). Schließlich muss die DFM noch mit `convert()` in ein Format konvertiert werden, das das `stm`-Package erwartet.

```
# DFM Erzeugen
guardian_dfm <- dfm(guardian_corpus,
                    remove_punct = TRUE,
                    remove_numbers = TRUE,
                    remove_symbols = TRUE,
                    remove_url = TRUE,
                    remove = stopwords("english"))


# Trimmen
trimmed_dfm <- dfm_trim(guardian_dfm,
                        min_docfreq = 0.02,
                        max_docfreq = 0.50,
                        docfreq_type = "prop")


# Konvertieren
stm_dfm <- convert(trimmed_dfm, to = "stm")
```

Nun berechnen wir das Modell mit `K = 20` Themen. Da die Berechnung eine Zeit dauern kann, ist es sinnvoll, das Modellobjekt im Anschluss abzuspeichern.

```
guardian_model <- stm(stm_dfm$documents, stm_dfm$vocab, K = 20)
saveRDS(guardian_model, "data/stm_guardian_model.rds")
```

Anschließend können wir uns eine Übersicht der wichtigsten Wörter je Thema mittels `labelTopics()` ausgeben lassen:

```
labelTopics(guardian_model)
```

```
## Topic 1 Top Words:
##       Highest Prob: year, game, friday, music, team, best, christmas
##       FREX: music, game, christmas, stores, football, tv, store
##       Lift: sorry, music, football, players, game, christmas, stores
##       Score: sorry, game, stores, music, store, players, retailers
## Topic 2 Top Words:
##       Highest Prob: party, labour, vote, election, leader, mps, cameron
##       FREX: corbyn, labour, mps, party, tory, mp, conservative
##       Lift: corbyn, electorate, ukip, tory, tories, miliband, lib
##       Score: electorate, labour, corbyn, party, vote, election, ukip
```

```
## Topic 3 Top Words:
##       Highest Prob: growth, year, market, economy, prices, markets, uk
##       FREX: prices, growth, markets, economy, quarter, market, inflation
##       Lift: ftse, forecasts, output, pound, monetary, economists, recession
##       Score: ftse, growth, prices, eurozone, markets, inflation, oil
## Topic 4 Top Words:
##       Highest Prob: us, security, military, syria, war, foreign, attacks
##       FREX: syria, military, isis, russian, de, syrian, islamic
##       Lift: de, afghanistan, syria, fighters, russian, troops, isis
##       Score: de, syria, isis, syrian, military, russian, islamic
## Topic 5 Top Words:
##       Highest Prob: climate, energy, countries, change, world, global, china
##       FREX: climate, energy, gas, emissions, environmental, china, development
##       Lift: climate, india, environmental, carbon, emissions, energy, gas
##       Score: india, climate, emissions, energy, china, carbon, countries
## Topic 6 Top Words:
##       Highest Prob: government, minister, australia, monday, australian, bill, law
##       FREX: monday, australian, australia, labor, bill, legislation, federal
##       Lift: monday, australians, australian, malcolm, labor, australia, abbott
##       Score: monday, australian, labor, australia, minister, senate, abbott
## Topic 7 Top Words:
##       Highest Prob: health, better, care, nhs, services, staff, doctors
##       FREX: better, health, nhs, doctors, care, patients, mental
##       Lift: patients, better, doctors, cancer, health, nhs, mental
##       Score: better, nhs, health, patients, doctors, care, mental
## Topic 8 Top Words:
##       Highest Prob: says, can, london, get, money, now, years
##       FREX: games, buy, space, london, products, says, small
##       Lift: games, design, space, waste, product, products, buy
##       Score: games, says, products, london, business, buy, product
## Topic 9 Top Words:
##       Highest Prob: city, many, country, video, south, rights, years
##       FREX: video, church, city, park, gay, religious, protests
##       Lift: video, protests, church, religious, gay, protesters, park
##       Score: video, muslim, church, protesters, religious, gay, city
## Topic 10 Top Words:
##       Highest Prob: like, think, just, can, us, even, going
##       FREX: really, think, thing, things, like, something, feel
##       Lift: watching, film, stuff, book, feels, thinking, books
##       Score: watching, film, like, think, really, story, bbc
## Topic 11 Top Words:
##       Highest Prob: children, school, food, water, local, child, education
##       FREX: children, water, school, food, students, schools, girls
##       Lift: storm, girls, students, children, water, schools, school
##       Score: storm, children, water, school, food, girls, schools
## Topic 12 Top Words:
```

```
##       Highest Prob: block-time, published-time, says, updated-timeupdated, photograp
##       FREX: block-time, published-time, updated-timeupdated, photograph, pic.twitter
##       Lift: block-time, updated-timeupdated, published-time, today's, pic.twitter.co
##       Score: block-time, published-time, today's, updated-timeupdated, pic.twitter.c
## Topic 13 Top Words:
##       Highest Prob: eu, uk, european, europe, britain, cameron, british
##       FREX: eu, europe, european, refugees, britain, brexit, migrants
##       Lift: cars, eu, refugees, migrants, migration, europe, italy
##       Score: cars, eu, european, brexit, refugees, greece, referendum
## Topic 14 Top Words:
##       Highest Prob: information, data, online, media, users, using, internet
##       FREX: internet, users, google, apple, information, facebook, app
##       Lift: weekly, user, google, privacy, app, users, internet
##       Score: weekly, google, users, app, apple, data, facebook
## Topic 15 Top Words:
##       Highest Prob: told, family, two, back, home, time, man
##       FREX: late, son, man, father, died, friends, went
##       Lift: late, son, sister, daughter, brother, father, bus
##       Score: late, family, died, mother, daughter, man, son
## Topic 16 Top Words:
##       Highest Prob: tax, government, pay, budget, cuts, spending, workers
##       FREX: tax, cuts, housing, budget, income, osborne, spending
##       Lift: innovation, tax, councils, income, osborne's, budget, cuts
##       Score: innovation, tax, osborne, budget, cuts, income, housing
## Topic 17 Top Words:
##       Highest Prob: trump, clinton, republican, sanders, campaign, obama, donald
##       FREX: clinton, republican, sanders, trump, cruz, hillary, trump's
##       Lift: bernie, hillary, republican, sanders, trump's, clinton, cruz
##       Score: bernie, trump, clinton, sanders, cruz, republican, hillary
## Topic 18 Top Words:
##       Highest Prob: police, court, officers, case, investigation, told, evidence
##       FREX: police, officers, arrested, prison, investigation, trial, criminal
##       Lift: custody, suspect, jury, prosecution, prosecutors, officers, police
##       Score: suspect, police, officers, court, arrested, investigation, prison
## Topic 19 Top Words:
##       Highest Prob: company, bank, business, companies, executive, chief, financial
##       FREX: company, bank, banks, banking, executive, shares, shareholders
##       Lift: investments, executives, shareholders, directors, banking, company's, pr
##       Score: investments, bank, shareholders, company, shares, customers, companies
## Topic 20 Top Words:
##       Highest Prob: women, report, found, research, year, number, years
##       FREX: women, study, research, drug, drugs, report, female
##       Lift: equivalent, researchers, gender, study, drugs, drug, women
##       Score: equivalent, women, drug, report, drugs, violence, research
```