

# R für die sozioökonomische Forschung

Aktuelle Version: 0.7.2 (September 24, 2020)

Dr. Claudius Gräßner

Dieses Skript begleitet die Lehrveranstaltung  
‘Methoden der Sozioökonomie’ von Prof. Jakob  
Kapeller und Dr. Claudius Gräßner im Master  
‘Sozioökonomie’ an der Universität Duisburg-Essen  
im Wintersemester 2019/20.  
Feedback über Moodle oder [Github](#) ist sehr  
willkommen.

Wintersemester 2019/2020

# Contents

<b>Willkommen</b>	<b>3</b>
Danksagung . . . . .	3
Lizenz . . . . .	3
Änderungshistorie . . . . .	4
<b>1 Vorbemerkungen</b>	<b>5</b>
1.1 Warum R? . . . . .	5
1.2 Besonderheiten von R . . . . .	6
<b>I Grundlagen in R</b>	<b>9</b>
<b>2 Einrichtung</b>	<b>11</b>
2.1 Installation von R und R-Studio . . . . .	11
2.2 Die R Studio Oberfläche . . . . .	11
2.3 Einrichtung eines R Projekts . . . . .	12
2.4 Optional: Schritt 4 und das here-Paket . . . . .	16
2.5 Abschließende Bemerkungen . . . . .	18
<b>3 Erste Schritte in R</b>	<b>19</b>
3.1 Befehle in R an den Computer übermitteln . . . . .	19
3.2 Objekte, Funktionen und Zuweisungen . . . . .	20
3.3 Grundlegende Objekte in R . . . . .	22
3.4 Pakete . . . . .	43
<b>4 Datenkunde und Datenaufbereitung</b>	<b>47</b>
Verwendete Pakete . . . . .	48
4.1 Arten von Daten . . . . .	49
4.2 Datenakquise . . . . .	53
4.3 Daten einlesen und schreiben . . . . .	56
4.4 Verarbeitung von Daten ('data wrangling') . . . . .	62
4.5 Abschließende Bemerkungen zum Umgang mit Daten innerhalb eines Forschungsprojekts . . . . .	85

4.6 Anmerkungen zu Paketen . . . . .	86
<b>5 Visualisierung von Daten</b>	<b>89</b>
Verwendete Pakete . . . . .	89
Einleitung . . . . .	89
5.1 Optional: Theoretische Grundlagen . . . . .	90
5.2 Grundlegende Elemente von <code>ggplot2</code> -Grafiken . . . . .	94
5.3 Arten von Datenvisualisierung . . . . .	111
5.4 Beispiele aus der Praxis und fortgeschrittene Themen . . . . .	122
5.5 Typische Fehler in der Datenvisualisierung vermeiden . . . . .	129
5.6 Lügen mit grafischer Statistik . . . . .	137
5.7 Links und weiterführende Literatur . . . . .	142
<b>II Mathematische Grundlagen</b>	<b>143</b>
<b>6 Formale Methoden der Sozioökonomie</b>	<b>145</b>
Verwendete Pakete . . . . .	145
6.1 Änderungsraten und die Rolle des Logarithmus . . . . .	146
6.2 Grundlagen der Differentialrechnung . . . . .	149
6.3 Lineare Algebra . . . . .	161
6.4 Analyse von Verteilungen . . . . .	173
<b>7 Grundlagen der Wahrscheinlichkeitstheorie</b>	<b>193</b>
Verwendete Pakete . . . . .	193
7.1 Einleitung: Wahrscheinlichkeitstheorie und Statistik . . . . .	193
7.2 Grundbegriffe der Wahrscheinlichkeitstheorie . . . . .	194
7.3 Diskrete Wahrscheinlichkeitsmodelle . . . . .	197
7.4 Stetige Wahrscheinlichkeitsmodelle . . . . .	206
7.5 Zusammenfassung Wahrscheinlichkeitsmodelle für einzelne ZV . . . . .	212
7.6 Analyse mehrerer Zufallsvariablen: gemeinsame und marginale Verteilungen . . . . .	212
<b>III Grundlagen der Regressionsanalyse in R</b>	<b>219</b>

# Willkommen

Das folgdene Skript ist als eine erste Einführung in die Programmiersprache R ([R Core Team, 2018](#)) und ihrer Anwendung im Bereich der quantitativen sozioökonomischen Forschung gedacht. Ursprünglich war es als Begleitung für die Lehrveranstaltung “Wissenschaftstheorie und Einführung in die Methoden der Sozioökonomie” im Master “Sozioökonomie” an der Universität Duisburg-Essen konzipiert, es soll jedoch zu einer eigenständigen Einführung in R weiterentwickelt werden. Dabei richtet es sich zunächst an Menschen mit keinen oder geringen Vorkenntnissen in R. Einzelne Kapitel, insbesondere die zur Datenaufbereitung und -visualisierung könnten aber auch für fortgeschrittene Studierende interessant sein.

Insgesamt ist das Projekt noch in der Anfangsphase und somit unbedingt auf das Feedback von Nutzer\*innen angewiesen. Ich bin Ihnen daher für jegliches Feedback sehr dankbar. Am besten Sie verwenden für Ihr Feedback den [Issue-Tracker auf Github](#). Dort ist auch der Quellcode des Skripts verfügbar. Sie können mir das Feedback aber auch gerne per Email zukommen lassen. Verwenden Sie dafür im Zweifel das [Kontaktformular](#) auf meiner Homepage. Vielen Dank!

Ein Hinweis zu den unterschiedlichen Versionen: das Skript ist aktuell in einer HTML und einer PDF-Variante verfügbar. Ich empfehle Ihnen die PDF-Variante zu verwenden, das diese Version die final lektorierte Version ist und bestimmte Formatierungen für die HMTL-Version (noch) nicht funktionieren. Entsprechend ist diese Variante leicht unvollständig. Sie können die PDF auf der Homepage des Skripts <https://graebnerc.github.io/RforSocioEcon/> herunterladen indem Sie auf das PDF-Icon oben links (neben dem i) klicken. Alternativ können Sie auch [diesem Link](#) folgen. Aktualisieren Sie vor dem Download aber Ihr Browserfenster um sicherzugehen, dass Sie die aktuellste Version herunterladen.

## Danksagung

Ich möchte mich bei Jakob Kapeller und Anika Radkowitsch für das regelmäßige Feedback und die guten Hinweise bedanken. Bei Birte Strunk möchte ich mich für das hervorragende Lektorat und das Beisteuern vieler guter Ideen bedanken. Am *work-in-progress*-Charakter des Skripts haben alle natürlich keine Mitschuld.

## Lizenz



Das gesamte Skript ist unter der [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License](#) lizenziert.

## Änderungshistorie

An dieser Stelle werden alle wichtigen Updates des Skripts gesammelt. Die Versionsnummer hat folgende Struktur: **major.minor.patch** Neue Kapitel erhöhen die **minor** Stelle, kleinere, aber signifikante Korrekturen werden als Patches gekennzeichnet.

Datum	Version	Wichtigste Änderungen
19.10.19	0.1.0	Erste Version veröffentlicht
03.11.19	0.2.0	Markdown-Anhang hinzugefügt
04.11.19	0.3.0	Anhänge zur Wiederholung grundlegender Statistik hinzugefügt
06.11.19	0.4.0	Kapitel zu linearen Modellen hinzugefügt
18.11.19	0.5.0	Kapitel zur Datenaufbereitung und Visualisierung hinzugefügt; kleinere Korrekturen im Kapitel zu lin. Modellen
20.11.19	0.5.1	Korrektur von kleineren Rechtschreib/Grammatikfehlern; Fix für Problem mit html Version auf HP
03.12.19	0.6.0	Kapitel zu formalen Konzepten hinzugefügt; kleinere Korrekturen
10.12.19	0.6.1	Herleitung OLS hinzugefügt; bessere Beispiele bei Formalie; Konsolidierung Notation Kap. 4 und 7
06.01.20	0.7.0	Kapitel zur fortgeschrittenen Themen der Regression und nichtlinearen Schätzern hinzugefügt
12.01.20	0.7.1	Ergänzung Beweise im Kapitel zu fortgeschrittenen Themen der Regression
29.01.20	0.7.2	Korrektur der Tabelle in "Wahl der funktionalen Form" in Kapitel 8 im Bezug auf log-log Modelle

# Chapter 1

## Vorbemerkungen

### 1.1 Warum R?

Im Folgenden gebe ich einen kurzen Überblick über die Gründe, in diesem Buch spezifisch die Programmiersprache R für sozio-ökonomische Forschung vorzustellen. Die Liste ist sicherlich nicht abschließend (siehe auch [Wickham \(2019\)](#)).

- Die R Community gilt als besonders freundlich und hilfsbereit. Gerade weil viele Menschen, die R benutzen, praktizierende Datenwissenschaftler\*innen sind werden praktische Probleme breit und konstruktiv in den einschlägigen Foren diskutiert und es ist in der Regel leicht Lösungen für Probleme zu finde, sobald man selbst ein bestimmtes Level an Programmierkenntnissen erlangt hat.
  - Auch gibt es großartige Online Foren und Newsletter, die es einem einfacher und unterhaltsamer machen, seine R Kenntnisse stetig zu verbessern und zusätzlich viele neue Dinge zu lernen. Besonders empfehlen kann ich [R-Bloggers](#), eine Sammlung von Blog Artikeln, die R verwenden und neben Inspirationen für die Verwendung von R häufig inhaltlich sehr interessant sind; [rweekly](#), ein Newsletter, der ebenfalls interessante Infos zu R enthält sowie die [R-Ladies Community](#), die sich besonders das Empowerment von Minderheiten in der Programmierwelt zur Aufgabe gemacht hat.
  - Selbstverständlich werden zahlreiche R Probleme auch auf [StackOverflow](#) diskutiert. Häufig ist das der Ort, an dem man Antworten auf seine Fragen findet. Allerdings ist es gerade am Anfang unter Umständen schwierig die häufig sehr fortgeschrittenen Lösungen zu verstehen.
- R ist eine offene und freie Programmiersprache, die auf allen bekannten Betriebssystemen läuft. Im Gegensatz dazu stehen Programme wie SPSS und STATA, für die Universitäten jedes Semester viele Tausend Euro bezahlen müssen und die dann umständlich über Serverlizenzen abgerufen werden müssen. Auch für Studierende sind die Preise alles andere als gering. R dagegen ist frei und inklusiv, und auch Menschen mit weniger Geld können sie benutzen. Gerade vor dem Hintergrund der Rolle von Wissenschaft in einer demokratischen und freien Gesellschaft und in der Kooperation mit Wissenschaftler\*innen aus ärmeren Ländern ist dies extrem wichtig.
- R verfügt über ein hervorragendes Package System. Das bedeutet, dass es recht einfach ist, neue Pakete zu schreiben und damit die Funktionalitäten von R zu erweitern. In der Kombination mit der Open Source Charakter von R bedeutet das, dass R nie wirklich *out of date* ist, und dass neuere Entwicklungen der Statistik und Datenwissenschaften, und immer mehr auch in der VWL, recht zügig in R implementiert werden. Ins-

besondere wenn es um statistische Analysen, *machine learning*, Visualisierungen oder Datenmanagement und -manipulation geht: für alles gibt es Pakete in R. Irgendjemand hat Ihr Problem mit hoher Wahrscheinlichkeit schon einmal gelöst und Sie können davon profitieren.

- R ist - zusammen mit Python - mittlerweile die *lingua franca* im Bereich Statistik und Machine Learning.
- Integration mit Git, Markdown, Latex und anderen Tools erlaubt einen integrierten Workflow. Beispielsweise ist es mit R Markdown möglich, das Coding und Schreiben der Antworten im gleichen Dokument vorzunehmen. Auch dieses Skript wurde vollständig in R Markdown geschrieben.
- R ist eine sehr flexible Programmiersprache, die es Ihnen erlaubt in verschiedenen ‘Stilen’ zu programmieren. Für Anfänger\*innen mag es egal sein, dass R sowohl für objektorientierte und funktionale Programmierstile geeignet ist, aber gerade fortgeschrittene Nutzer\*innen werden dieses Feature sehr zu schätzen wissen - auch wenn die Stärke von R sicherlich eher in der funktionalen Programmierung zu finden sind.
- Für besondere Aufgaben ist es recht einfach R mit high-performance Sprachen wie C, Fortran oder C++ zu integrieren.

## 1.2 Besonderheiten von R

R ist keine typische Programmiersprache, denn sie wird vor allem von Statistiker\*innen benutzt und weiterentwickelt, und nicht von Programmierer\*innen. Das hat den Vorteil, dass die Funktionen oft sehr genau auf praktische Herausforderungen ausgerichtet sind und es für alle typischen statistischen Probleme Lösungen in R gibt. Gleichzeitig hat dies auch dazu geführt, dass R einige unerwünschte Eigenschaften aufweist, da die Menschen, die Module für R programmieren keine ‘genuine’ Programmierer\*innen sind.

Im Folgenden möchte ich einige Besonderheiten von R aufführen, damit Sie im Laufe Ihrer R-Karriere nicht negativ von diesen Besonderheiten überrascht werden. Während es sich für Programmier-Neulinge empfiehlt die Liste zu einem späteren Zeitpunkt zu inspizieren sollten Menschen mit Erfahrungen in anderen Sprachen gleich einen Blick darauf werfen.

- R wird dezentral über viele benutzergeschriebene Pakete (‘libraries’ oder ‘packages’) konstant weiterentwickelt. Das führt wie oben erwähnt dazu, dass R quasi immer auf dem neuesten Stand der statistischen Forschung ist. Gleichzeitig kann die schiere Masse von Paketen auch verwirrend sein, insbesondere weil es für die gleiche Aufgabe häufig deutlich mehr als ein Paket gibt. Das führt zwar auch zu einer positiven Konkurrenz und jede\*r kann sich seinen oder ihren Geschmäckern gemäß für das eine oder andere Paket entscheiden, es bringt aber auch mögliche Inkonsistenzen und schwerer verständlichen Code mit sich.
- Im Gegensatz zu Sprachen wie Python, die trotz einer enormen Anzahl von Paketen eine interne Konsistenz nicht verloren haben gibt es in R verschiedene ‘Dialekte’, die teilweise inkonsistent sind und gerade für Anfänger durchaus verwirrend sein können. Besonders die Unterscheidungen des **tidyverse**, einer Gruppe von Paketen, die von der R Studio Company sehr stark gepusht werden und vor allem zur Verarbeitung von Datensätzen gedacht sind, implementieren viele Routinen des ‘klassischen R’ (‘base R’) in einer neuen Art und Weise. Das Ziel ist, die Arbeit mit Datensätzen einfacher und leichter verständlich zu machen, allerdings wird die recht aggressive ‘Vermarktung’ und die teilweise inferiore Performance des Ansatzes auch kritisiert.<sup>1</sup>

---

<sup>1</sup>Zum einen bin ich ein großer Fan von vielen **tidyverse** Paketen, gleichzeitig ist der Fokus von R Studio auf diese Pakete sehr gefährlich. Meiner Meinung nach hilft **tidyverse** allerdings bei der Einsteigerfreundlichkeit, da diese Pakete die Arbeit mit Datensätzen sehr einfach machen. Für kleine Datensätze (<500MB) benutze ich das **tidyverse** auch in meiner eigenen Forschung. Aufgrund der Einsteigerfreundlichkeit werden wir hier für die Arbeit mit Datensätzen trotz allem mit dem **tidyverse** arbeiten. Ich weise jedoch auf die kritische Diskussion im entsprechenden Kapitel des Skripts hin.

- Da viele der Menschen, die R Pakete herstellen keine Programmierer\*innen sind, sind viele Pakete von einem Programmierstandpunkt aus nicht sonderlich effizient oder elegant geschrieben. Gleichzeitig gibt es aber auch viele Ausnahmen zu dieser Regel und viele Pakete werden über die Zeit hinweg signifikant verbessert.
- R an sich ist nicht die schnellste Programmiersprache, insbesondere wenn man seinen Code nicht entsprechend geschrieben hat. Auch bedarf eine R Session in der Regel recht viel Speicher. Hier sind selbst andere High-Level Sprachen wie Julia oder Python deutlich performanter, auch wenn Pakete wie `data.table` diesen Nachteil häufig abschwächen. Zudem ist er für die meisten Probleme, die Sozioökonom\*innen in ihrer Forschungspraxis bearbeiten, irrelevant.

Alles in allem ist R also eine hervorragende Wahl wenn es um quantitative sozialwissenschaftliche Forschung geht. Auch in der Industrie ist R extrem beliebt und wird im Bereich der *Data Science* nur noch von Python ernsthaft in den Schatten gestellt. Allerdings verwenden die meisten Menschen, die in diesem Bereich arbeiten, ohnehin beide Sprachen, da sie unterschiedliche Vor- und Nachteile haben. Entsprechend ist jede Minute, die Sie in das Lernen von R investieren eine exzellente Investition, egal wo Sie in Ihrem späteren Berufsleben einmal landen werden.

Das Wichtigste am Programmieren ist in jedem Fall Spaß und die Bereitschaft zu sowie die Freude an der Zusammenarbeit mit anderen. Denn das hat R mit anderen offenen Sprachen wie Python gemeinsam: Programmieren und das Lösen von statistischen Fragestellungen sollte immer ein kollaboratives Gemeinschaftsprojekt sein!



# Teil I

## Grundlagen in R



# Chapter 2

## Einrichtung

### 2.1 Installation von R und R-Studio

Die Installation von R ist in der Regel unproblematisch. Auf der [R homepage](#) wählt man unter dem Reiter ‘Download’ den Link ‘CRAN’ aus, wählt einen Server in der Nähe und lädt sich dann die R Software herunter. Danach folgt man den Installationshinweisen.

Im zweiten Schritt muss noch das Programm ‘R-Studio’ installiert werden. Hierbei handelt es sich um eine grafische Oberfläche für R, welche uns die Arbeit enorm erleichtern wird. Das Programm kann [hier](#) heruntergeladen werden. Bitte darauf achten ‘RStudio Desktop’ zu installieren.

### 2.2 Die R Studio Oberfläche

Nach dem Installationsprozess öffnen wir R Studio zum ersten Mal. Abbildung 2.1 zeigt die verschiedenen Elemente der Oberfläche, deren Funktion im Folgenden kurz erläutert wird. Vieles ergibt sich hier aber auch durch *learning by doing*. Im Folgenden werden nur die Bereiche der Oberfläche beschrieben, die am Anfang unmittelbar relevant für uns sind.

- Der **Skriptbereich** (1) ist ein Texteditor wie Notepad - nur mit zusätzlichen Features wie Syntax Highlighting für R, sodass es uns leichter fällt R Code zu schreiben. Hier werden wir unsere Skripte verfassen.
- Die **Konsole** (2) erlaubt es uns über R direkt mit unserem Computer zu interagieren. R ist eine Programmiersprache. Das bedeutet, wenn wir den Regeln der Sprache folgen und uns in einer für den Computer verständlichen Art und Weise ausdrücken, versteht der Computer was wir von ihm wollen und führt unsere Befehle aus. Wenn wir in die Konsole z.B. `2+2` eingeben, dann ist das valider R Code. Wenn wir dann Enter drücken versteht der Computer unseren Befehl und führt die Berechnung aus. Die Konsole ist sehr praktisch um den Effekt von R Code direkt zu beobachten. Wenn wir etwas in der Console ausführen wollen, das wir vorher im **Skriptbereich** geschrieben haben, können wir den Text markieren und dann auf den Button **Run** (3) drücken: dann kopiert R Studio den Code in die Konsole und führt ihn aus.
- Für den Bereich oben rechts haben wir in der Standardkonfiguration von R Studio drei Optionen, die wir durch Klicken auf die Reiter auswählen können. Der Reiter **Environment** (4) zeigt uns alle bisher definierten Objekte an (mehr dazu später). Der Reiter **History** (5) zeigt an, welchen Code wir in der Vergangenheit ausgeführt haben. Der Reiter **Connections** (6) braucht uns aktuell nicht zu interessieren.

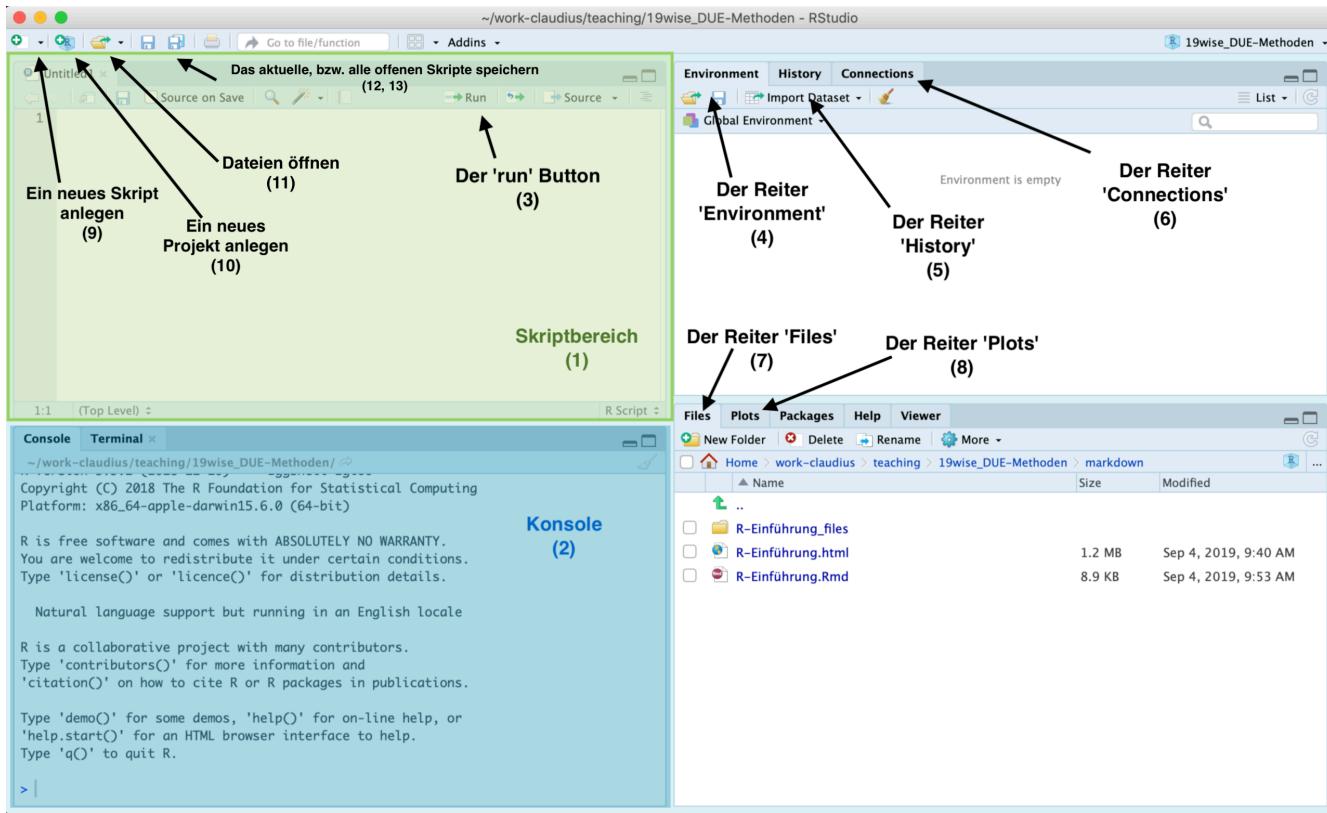


Figure 2.1: Die Benutzeroberfläche von R-Studio.

- Auch für den Bereich unten rechts haben wir mehrere Optionen: Der Bereich **Files** (7) zeigt uns unser Arbeitsverzeichnis mit allen Ordnern und Dateien an. Das ist das Gleiche, was wir auch über den File Explorer unseres Betriebssystems sehen würden. Der Bereich **Plots** (8) zeigt uns eine Vorschau der Abbildungen, die wir durch unseren Code produzieren. Die anderen Bereiche brauchen uns aktuell noch nicht zu interessieren.
- Wenn wir ein neues R Skript erstellen wollen, können wir das über den Button **Neu** (9) erledigen. Wir klicken darauf und wählen die Option ‘R Skript’. Mit den alternativen Dateiformaten brauchen wir uns aktuell nicht beschäftigen.
- Der Botton **Neues Projekt anlegen** (10) erstellt eine neue R Studio Projekt - mehr dazu in Kürze.
- Der Button **Öffnen** (11) öffnet Dateien im Skriptbereich.
- Die beiden Buttons **Speichern** (12) und **Alles speichern** (13) speichern das aktuelle, bzw. alle im Skriptbereich geöffneten Dateien.

Die restlichen Buttons und Fenster in R Studio werden wir im Laufe der Zeit kennenlernen.

Es macht Sinn, sich einmal die möglichen Einstellungsmöglichkeiten für R Studio anzuschauen und gegebenenfalls eine andere Darstellungsversion zu wählen.

## 2.3 Einrichtung eines R Projekts

Im Folgenden werden wir lernen wie man ein neues R Projekt anlegt, R Code schreiben und ausführen kann.

Wann immer wir ein neues Programmierprojekt starten, sollten wir dafür einen eigenen Ordner anlegen und ein

sogenanntes ‘R Studio Projekt’ erstellen. Das hilft uns den Überblick über unsere Arbeit zu behalten, und macht es einfach Code untereinander auszutauschen.

Ein Programmierprojekt kann ein Projekt für eine Hausarbeit sein, die Mitschriften für eine Vorlesungseinheit, oder einfach der Versuch ein bestimmtes Problem zu lösen, z.B. einen Datensatz zu visualisieren.

Die Schritte zur Erstellung eines solchen Projekts sind immer die gleichen:

1. Einen Ordner für das Projekt anlegen.
2. Ein R-Studio Projekt in diesem Ordner erstellen.
3. Relevante Unterordner anlegen.

Wir beschäftigen uns mit den Schritten gleich im Detail, müssen vorher aber noch die folgenden Konzepte diskutieren: (1) das Konzept eines *Arbeitsverzeichnisses* (*working directory*) und (2) die Unterscheidung zwischen *absoluten* und *relativen* Pfaden.

### 2.3.1 Arbeitsverzeichnisse und Pfade

Das **Arbeitsverzeichnis** ist ein Ordner auf dem Computer, in dem R standardmäßig sämtlichen Output speichert und von dem aus es auf Datensätze und anderen Input zugreift. Wenn wir mit Projekten arbeiten ist das Arbeitsverzeichnis der Ordner, in dem das R-Projektfile abgelegt ist, ansonsten ist es das Benutzerverzeichnis. Wir können uns das Arbeitsverzeichnis mit der Funktion `getwd()` anzeigen lassen. In meinem Fall ist das Arbeitsverzeichnis das Folgende:

```
#> [1] "/Volumes/develop/packages/RforSocioEcon"
```

Wenn ich R nun sagen würde, es solle ein File unter dem Namen `test.pdf` speichern, dann würde es am folgenden Ort gespeichert werden:

```
#> [1] "/Volumes/develop/packages/RforSocioEcon/test.pdf"
```

R geht in einem solchen Fall immer vom Arbeitsverzeichnis aus. Da wir im vorliegenden Fall den Speicherort relativ zum Arbeitsverzeichnis angegeben haben, sprechen wir hier von einem **relativen Pfad**.

Alternativ können wir den Speicherort auch als **absoluten Pfad** angeben. In diesem Fall geben wir den kompletten Pfad, ausgehend vom **Root Verzeichnis** des Computers, an. Wir würden R also *explizit* auffordern, das File an folgendem Ort zu speichern:

```
#> [1] "/Volumes/develop/packages/RforSocioEcon/test.pdf"
```

Wir werden hier **immer** relative Pfade verwenden. Relative Pfade sind fast immer die bessere Variante, da es uns erlaubt den gleichen Code auf verschiedenen Computern zu verwenden. Denn wie man an den absoluten Pfaden erkennen kann, sehen diese auf jedem Computer anders aus und es ist dementsprechend schwierig, Code miteinander zu teilen.

Wir lernen mehr über dieses Thema wenn wir uns später mit Dateninput und -output beschäftigen.

### 2.3.2 Schritt 1: Projektordner anlegen

Zuerst müssen Sie sich für einen Ordner auf Ihrem Computer entscheiden, in dem alle Daten, die mit ihrem Projekt zu tun haben, also Daten, Skripte, Abbildungen, etc. gespeichert werden sollen und diesen Ordner gegebenenfalls neu erstellen. Es macht Sinn, einen solchen Ordner mit einem informativen Namen ohne Leer- und Sonderzeichen zu versehen, z.B. `Learning-R`.

Dieser Schritt kann theoretisch auch gemeinsam mit Schritt 2 erfolgen.

### 2.3.3 Schritt 2: Ein R-Studio Projekt im Projektordner erstellen

Wir möchten nun R Studio mitteilen den in Schritt 1 erstellten Ordner als R Projekt zu behandeln. Damit wird nicht nur dieser Ordner als Root-Verzeichnis festgelegt, man kann auch die Arbeitshistorie eines Projekts leicht wiederherstellen und es ist einfacher, das Projekt auf verschiedenen Computern zu bearbeiten.

Um ein neues Projekt zu erstellen klicken Sie in R Studio auf den Button **Neues Projekt** (Nr. 10 in Abbildung 2.1) und Sie sollten das in Abbildung 2.2 dargestellte Fenster sehen.

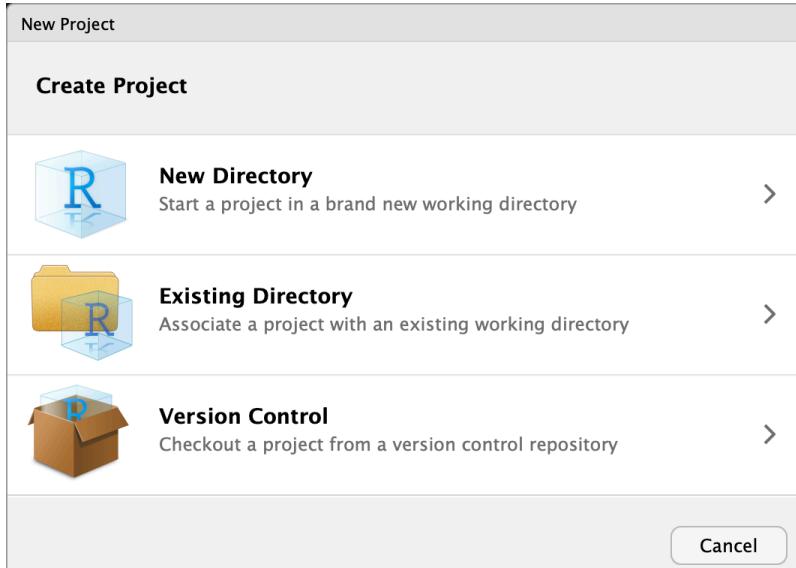


Figure 2.2: Ein neues Projekt erstellen.

Falls Sie in Schritt 1 den Projektordner bereits erstellt haben wählen Sie hier **Existing Directory**, ansonsten erstellen Sie einen neuen Projektordner gleich gemeinsam mit dem Projektfile indem Sie **New Directory** auswählen.

Falls Sie **Existing Directory** gewählt haben, landen Sie in einem Fenster, welches dem linken Feld der Abbildung 2.3 entspricht. Hier wählen Sie einfach den vorher erstellten Ordner aus und klicken auf **Create Project**.

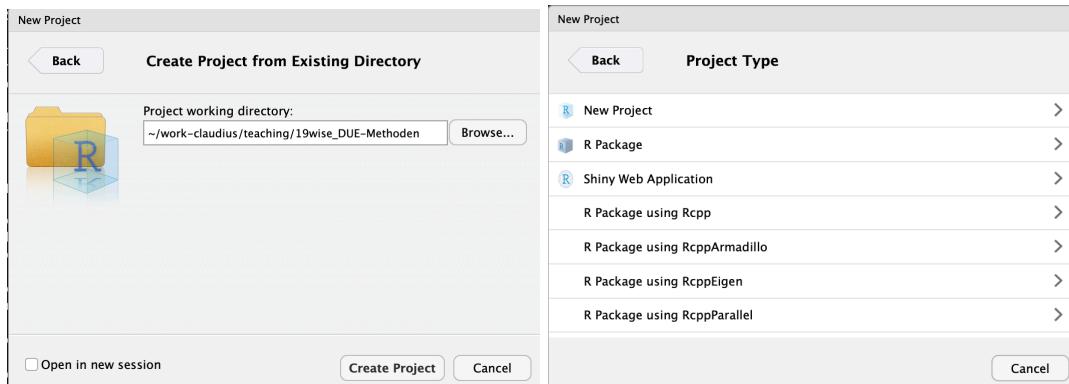


Figure 2.3: Ein neues R-Projekt aus einem existierenden (links) oder in einem neuen Projektordner (rechts) erstellen.

Falls Sie **New Directory** gewählt haben, landen Sie dann auf dem rechten in Abbildung 2.3 dargestellten Fenster. Hier wählen Sie **New Project** aus, geben dem Projekt im folgenden Fenster einen Namen (das wird der Name des

Projektordners sein), wählen den Speicherort für den Ordner aus und klicken auf **Create Project**.

In beiden Fällen wurde nun ein Ordner erstellt, in dem sich ein File **\*.Rproj** befindet. Damit ist die formale Erstellung eines Projekts abgeschlossen. Es empfiehlt sich jedoch dringend gleich eine sinnvolle Unterordnerstruktur mit anzulegen.

### 2.3.4 Schritt 3: Relevante Unterordner erstellen

Eine sinnvolle Unterordnerstruktur hilft (1) den Überblick über das eigene Projekt nicht zu verlieren, (2) mit anderen über verschiedene Computer hinweg zu kollaborieren und (3) Kollaborationsplattformen wie Github zu verwenden und replizierbare und für andere nachvollziehbare Forschungsarbeit zu betreiben.

Die folgende Ordnerstruktur ist eine Empfehlung. In manchen Projekten werden Sie nicht alle hier vorgeschlagenen Unterordner brauchen, in anderen bietet sich die Verwendung von mehr Unterordnern an. Nichtsdestotrotz ist es ein guter Ausgangspunkt, den ich in den meisten meiner Forschungsprojekte auch so verwende.

Insgesamt sollten die folgenden Ordner im Projektordner erstellt werden:

- Ein Ordner **data**, der alle Daten enthält, die im Rahmen des Projekts verwendet werden. Hier empfiehlt es sich zwei Unterordner anzulegen: Einen Ordner **raw**, der die Rohdaten enthält, so wie sie aus dem Internet runtergeladen wurden. Diese Rohdaten sollten **niemals** verändert werden, ansonsten wird Ihre Arbeit nicht vollständig replizierbar werden und es kommt gegebenenfalls zu irreparablen Schäden. Alle Veränderungen der Daten sollten durch Skripte dokumentiert werden, die die Rohdaten als Input, und einen modifizierten Datensatz als Output generieren. Dieser modifizierte Datensatz sollte dann im Unterordner **tidy** gespeichert werden.

Beispiel: Sie laden sich Daten zum BIP in Deutschland von Eurostat und Daten zu Arbeitslosigkeit von AMECO herunter. Beide Datensätze sollten im Unterordner **data/raw** gespeichert werden. Mit einem Skript lesen Sie beide Datensätze ein und erstellen den kombinierten Datensatz **macro\_data.csv**, den Sie im Ordner **data/tidy** speichern und für die weitere Analyse verwenden. Dadurch kann jede\*r nachvollziehen wie die von Ihnen verwendeten Daten sich aus den Rohdaten ergeben haben und Ihre Arbeit bleibt komplett transparent.

- Ein Ordner **R**, der alle R Skripte enthält, also alle Textdokumente, die R Code enthalten.
- Ein Ordner **output**, in dem der Output ihrer Berechnungen, z.B. Tabellen oder Plots, gespeichert werden können. Der Inhalt dieses Ordners sollte sich komplett mit den Inhalten der Ordner **data** und **R** replizieren lassen.
- Ein Ordner **text**, in dem Sie Ihre Verschriftlichungen speichern, z.B. das eigentliche Forschungspapier, ihre Hausarbeit oder Ihre Vorlesungsmitschriften.
- Einen Ordner **misc** in den Sie alles packen, was in keinen der anderen Ordner passt. Ein solcher Ordner ist wichtig und Sie sollten nicht zuordbare Dateien nie in den übergeordneten Projektordner als solchen speichern.

Wenn wir annehmen unser Projektordner heißt **2019-Methoden** ergibt sich damit insgesamt die in Abbildung 2.4 dargestellte Ordner und Datenstruktur.

In einem vierten Schritt sollten Sie nun noch eine **.here**-Datei erstellen. Da dies jedoch die Verwendung von Paketen voraussetzt sollten Sie den nächsten Abschnitt zunächst überspringen, wenn Sie noch nicht wissen wie man R-Pakete verwendet und den Abschnitt zu einem späteren Zeitpunkt lesen (R-Pakete werden in Abschnitt ?? eingeführt).

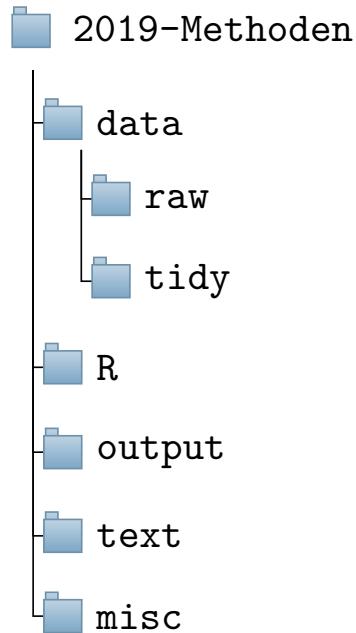


Figure 2.4: Die vorgeschlagene Ordnerstruktur für R-Projekte.

## 2.4 Optional: Schritt 4 und das here-Paket

Es gibt einen weiteren Schritt, den Sie bei der Einrichtung eines neuen Projekts immer durchführen sollten. Dieser Schritt beinhaltet die Verwendung des R-Pakets [here](#) (Müller, 2017). Falls Sie noch nicht mit der Verwendung von R-Paketen vertraut sind, sollten Sie diesen Abschnitt zunächst überspringen. Pakete werden in Abschnitt ?? eingeführt und es macht Sinn, wenn Sie später nach Lektüre dieses Abschnitts noch einmal hierher zurückkehren.

Wie weiter oben beschrieben sollten Sie in der alltäglichen Arbeit immer *relative* Pfade verwenden. In zwei Situationen kann die Verwendung von relativen Pfaden aber problematisch sein: (1) in der Zusammenarbeit mit anderen und (2) bei der Verwendung von R-Markdown.<sup>1</sup>

Gerade wenn Sie ein Skript einmal an eine\*n Kollege\*in schicken wollen und diese\*r nicht das ganze R Projekt öffnet kommt es schnell zu Fehlermeldungen. Glücklicherweise gibt es eine recht einfache Lösung für alle derartigen Probleme: das Paket [here](#) (Müller, 2017). Durch Einlesen des Pakets erhalten Sie Zugriff auf die Funktion (Überraschung!) `here()`. Diese Funktion nimmt nur ein einziges Argument und zwar einen relativen Pfad. Die Funktion wandelt diesen relativen Pfad automatisch in einen absoluten Pfad um. Dabei berücksichtigt sie die individuelle Ordnerstruktur von dem Computer, auf dem sie aktuell ausgeführt wird. Das bedeutet, dass Sie für unterschiedliche Computer unterschiedliche Ergebnisse liefert. Anhand eines Beispiels lässt sich das am einfachsten nachvollziehen. Gehen wir davon aus, dass Birte und Claudius gemeinsam an einem R-Skript arbeiten. Im Code kommt dabei folgende Zeile vor:

```
write(c(1,2,3), file = "output/vektor.txt")
```

Hier wird ein Vektor in einer Textdatei im Unterordner `output/` gespeichert. Damit R die Datei am richtigen Ort speichert müssen Birte und Claudius das gleiche Arbeitsverzeichnis verwenden, sonst wirkt der relative Pfad auf

<sup>1</sup>Mit R Markdown können Sie in R direkt Texte schreiben und somit die statistische Analyse und die Beschreibung der Ergebnisse in einem Dokument integrieren. Dieses Buch ist z.B. auch vollständig in R-Markdown geschrieben. Eine kurze Einführung finden Sie in Kapitel ??.

ihren jeweiligen Computern anders. Alternativ könnte Claudius den Pfad auch absolut angeben:

```
write(c(1,2,3), file = "/Users/claudius/projekte/R-Projekt-Birte/output/vektor.txt")
```

Dann würde die Datei auf seinem Computer immer am richtigen Ort gespeichert, egal in welchem Arbeitsverzeichnis er sich gerade befindet. Wenn Birte nun allerdings diesen Code bei sich ausführt wird sie mit ziemlicher Sicherheit eine Fehlermeldung erhalten. Denn auf ihrem Computer liegt das Skript bestimmt an einem anderen Ort. Sie müsste also z.B. schreiben:

```
write(c(1,2,3), file = "/Users/birte/projekte/R-Projekt-Claudius/output/vektor.txt")
```

Dieser Code wiederum würde bei Claudius nicht funktionieren. Natürlich könnten die beiden den Code immer individuell vor dem Ausführen auf ihrem Computer anpassen. Das wäre allerdings nervig und Sie sollten Ihren Code immer so schreiben, dass er ohne Modifikation auf unterschiedlichen Computern funktioniert. Sonst macht Zusammenarbeit wenig Spaß.

Die Funktion `here()` löst dieses Problem. Hier könnten die beiden einfach schreiben:

```
write(c(1,2,3), file = here("output/vektor.txt"))
```

Die Funktion `here()` baut dann automatisch einen absoluten Pfad. Der Output von `here("output/vektor.txt")` sähe auf Claudius' Computer so aus:

`~/claudius-projekte/R-Projekt-Birte/output/vektor.txt`

Und auf Birte's Computer aber so:

`~/birte-projekte/R-Projekt-Claudius/output/vektor.txt`

So können die beiden den Code einfach untereinander tauschen ohne den Code jeweils auf ihren Computern verändern zu müssen. Daher, und aufgrund einiger anderer potenzieller Schwierigkeiten, denen wir erst später begegnen werden, lohnt es sich immer das Paket `here` zu verwenden und relative Pfade immer gemeinsam mit der Funktion `here()` anzugeben. Das ist ein kleiner Mehraufwand, der auf Dauer aber viel Ärger spart.

Damit das problemlos funktioniert lohnt es sich, dem `here` Paket den Ausgangspunkt für die relativen Pfade explizit mitzuteilen. Dafür muss beim Erstellen eines Arbeitsverzeichnis *einmalig* eine Datei `.here` erstellt werden, und zwar in dem Ordner, der als Ausgangspunkt für die relativen Pfade fungieren soll. Das ist in aller Regel der Ordner, in dem auch die `.Rproj` Datei liegt.<sup>2</sup>

Dazu führen wir nach Schritt 3 noch folgende Befehle aus:

```
here::here()
```

Wenn der angezeigte Pfad mit dem Pfad zu Ihrem gewünschten Arbeitsverzeichnis übereinstimmt, führen Sie nun folgenden Befehl aus:

```
here::set_here()
```

Falls nicht geben Sie der Funktion `here::set_here()` den absoluten Pfad zu Ihrem Arbeitsverzeichnis als Argument, z.B.:

---

<sup>2</sup>Strikt genommen ist das Erstellen der `.here` Datei nicht nötig, da `here()` im Zweifel von dem Ordner als Ausgangspunkt ausgeht, in dem es die nächste `.Rproj`-Datei findet. Es ist aber besser hier explizit zu seine `.here`-Datei beim Einrichten eines neuen Projekts immer mit zu erstellen. Dann funktioniert der Code in jedem Fall immer.

```
here::set_here("/Users/claudius/projekte/R-Projekt-Birte")
```

Sie sollten nun eine Nachricht bekommen haben, dass in Ihrem Arbeitsverzeichnis eine neue Datei `.here` erstellt wurde. Damit ist die Einrichtung Ihres Projekts vollständig abgeschlossen.

## 2.5 Abschließende Bemerkungen

Eine gute Ordnerstruktur ist nicht nur absolut essenziell um selbst einen Überblick über seine Forschungsprojekte zu behalten, sondern auch wenn man mit anderen Menschen kollaborieren möchte, da jegliche Kollaboration durch eine gut durchdachte Ordnerstruktur massiv erleichtert wird. In einem solchen Fall sollte man auf jeden Fall eine Versionskontrolle wie Git und GitHub verwenden. Eine (optionale) kurze Einführung in Git und Github finden Sie in Kapitel ??.

# Chapter 3

## Erste Schritte in R

Nach den (wichtigen) Vorbereitungsschritten im vorangehenden Kapitel wollen wir an dieser Stelle mit dem eigentlichen Programmieren anfangen. Zu diesem Zweck müssen wir uns mit der Syntax von R vertraut machen, also mit den Regeln, denen wir folgen müssen, wenn wir Code schreiben, damit der Computer versteht, was wir ihm eigentlich in R sagen wollen.

Das Kapitel ist dabei folgendermaßen aufgebaut: zunächst lernen wir in Abschnitt 3.1 wie wir mit Hilfe von R und R-Studio mit unserem Computer ‘kommunizieren’ können. Danach lernen wir in Abschnitt 3.2 nicht nur die zentralen Elemente der Programmiersprache R kennen, nämlich Objekte und Funktionen, sondern lernen auch wie wir solche Objekte erstellen und ihnen Namen zuweisen können. Ein großer Teil des Kapitels (Abschnitt 3.3) ist dann den unterschiedlichen Arten von Objekten in R gewidmet. Wir lernen zum Beispiel wie sich Vektoren von Listen unterscheiden, und welche Pendants in R es zu ‘Zahlen’ und ‘Wörtern’ in unserer Alltagssprache gibt. Abgeschlossen wird das Kapitel mit einer kurzen Einführung von ‘Paketen’ in Abschnitt 3.4. Pakete sind ein wichtiger Bestandteil der Open-Source Sprache R: hier handelt es sich um Code, den andere Menschen geschrieben und der Allgemeinheit frei zugänglich gemacht haben. Dadurch ist sichergestellt, dass R immer auf dem neuesten Stand der Forschung und Praxis ist.

### 3.1 Befehle in R an den Computer übermitteln

Grundsätzlich können wir über R Studio auf zwei Arten mit dem Computer “kommunizieren”: über die Konsole direkt, oder indem wir im Skriptbereich ein Skript schreiben und dies dann ausführen.

Als Beispiel für die erste Möglichkeit wollen wir mit Hilfe von R die Zahlen 2 und 5 miteinander addieren. Zu diesem Zweck können wir einfach `2 + 2` in die Konsole eingeben, und den Befehl mit ‘Enter’ an den Computer senden. Da es sich beim Ausdruck `2 + 3` um korrekten R Code handelt, ‘versteht’ der Computer was wir von ihm wollen und gibt uns das entsprechende Ergebnis aus:

```
2 + 3
```

```
#> [1] 5
```

Die Zeichenkombination `#>` am Beginn der Zeile zeigt an, dass es sich bei dieser Zeile um den Output eines R-Befehls handelt. Das kann bei Ihrem Computer durchaus anders aussehen. Das Ergebnis von `2+3` ist eine Zahl (genauer: ein ‘Skalar’). In R werden Skalare immer als Vektor der Länge 1 dargestellt. Die `[1]` gibt also an, dass hier ein

Vektor der Länge 1 angezeigt wird. Wäre das Ergebnis unserer Berechnung ein Vektor der Länge 2 würde die Outputzeile dementsprechend mit `#> [2]` eingeleitet werden.

Auf diese Art und Weise können wir R als einfachen Taschenrechner verwenden, denn für alle einfachen mathematischen Operationen können wir bestimmte Symbole als Operatoren verwenden. An dieser Stelle sei noch darauf hingewiesen, dass das Symbol `#` in R einen Kommentar einleitet, das heißt alles was in einer Zeile nach `#` steht wird vom Computer ignoriert und man kann sich an dieser Stelle Notizen im Code machen.

### `2 + 2 # Addition`

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

### `2/2 # Division`

```
#> [1] 1
```

### `4*2 # Multiplikation`

```
#> [1] 8
```

### `3**2 # Potenzierung`

```
#> [1] 9
```

Alternativ können wir die Befehle in einem Skript aufschreiben, und dieses Skript dann ausführen. Während die Interaktion über die Konsole sinnvoll ist um die Effekte bestimmter Befehle auszuprobieren, bietet sich die Verwendung von Skripten an, wenn wir mit den Befehlen später weiter arbeiten wollen, oder sie anderen Menschen zugänglich zu machen. Denn das Skript können wir als Datei auf unserem Computer speichern, vorzugsweise im Unterordner `R` unseres R-Projekts (siehe Abschnitt [Relevante Unterordner erstellen](#)), und dann später weiterverwenden.

Die Berechnungen, die wir bislang durchgeführt haben sind zugegebenermaßen nicht sonderlich spannend. Um fortgeschrittene Operationen in R durchzuführen und verstehen zu können müssen wir uns zunächst mit den Konzepten von **Objekten**, **Funktionen** und **Zuweisungen** beschäftigen.

## 3.2 Objekte, Funktionen und Zuweisungen

To understand computations in R, two slogans are helpful: Everything that exists is an object. Every-  
thing that happens is a function call.

— John Chambers

Mit der Aussage ‘Alles in R ist ein Objekt’ ist gemeint, dass jede Zahl, jede Funktion, oder jeder Buchstabe in R ein Objekt ist, das irgendwo auf dem Speicher Ihres Rechners abgespeichert ist.

In der Berechnung `2 + 3` ist die Zahl `2` genauso ein Objekt wie die Zahl `3` und die Additionsfunktion, die durch den Operator `+` aufgerufen wird.

Mit der Aussage ‘Alles was in R passiert ist ein Funktionsaufruf’ ist gemeint, dass wir R eine Berechnung durchführen lassen indem wir eine Funktion aufrufen.

**Funktionen** sind Algorithmen, die bestimmte Routinen auf einen *Input* anwenden und dabei einen *Output* produzieren. Die Additionsfunktion, die wir in der Berechnung `2 + 3` aufgerufen haben hat als Input die beiden Zahlen `2` und `3` aufgenommen, hat auf sie die Routine der Addition angewandt und als Output die Zahl `5` ausgegeben. Der Output `5` ist dabei in R genauso ein Objekt wie die Inputs `2` und `3`, sowie die Funktion `+`.

Ein ‘Problem’ ist, dass R im vorliegenden Fall den Output der Berechnung zwar ausgibt, wir danach aber keinen Zugriff darauf mehr haben:

```
2 + 3
```

```
#> [1] 5
```

Falls wir den Output weiterverwenden wollen, macht es Sinn, dem Output Objekt einen Namen zu geben, damit wir später wieder darauf zugreifen können. Der Prozess einem Objekt einen Namen zu geben wird **Zuweisung** oder **Assignment** genannt und durch die Funktion **assign** vorgenommen:

```
assign("zwischenergebnis", 2 + 3)
```

Wir können nun das Ergebnis der Berechnung  $2 + 3$  aufrufen, indem wir in R den Namen des Output Objekts eingeben:

```
zwischenergebnis
```

```
#> [1] 5
```

Da Zuweisungen so eine große Rolle spielen und sehr häufig vorkommen gibt es auch für die Funktion **assign** eine Kurzschreibweise, nämlich  $\leftarrow$ . Entsprechend sind die folgenden beiden Befehle äquivalent:

```
assign("zwischenergebnis", 2 + 3)
zwischenergebnis <- 2 + 3
```

Daher werden wir Zuweisungen immer mit dem  $\leftarrow$  Operator durchführen.<sup>1</sup>

Wir können in R nicht beliebig Namen vergeben. Gültige (also: syntaktisch korrekte) Namen

- enthalten nur Buchstaben, Zahlen und die Symbole . und \_
- fangen nicht mit . oder einer Zahl an!

Zudem gibt es einige Wörter, die schlicht nicht als Name verwendet werden dürfen, z.B. **function**, **TRUE**, oder **if**. Die gesamte Liste verbotener Worte kann mit dem Befehl **?Reserved** ausgegeben werden.

Wenn man einen Namen vergeben möchte, der nicht mit den gerade formulierten Regeln kompatibel ist, gibt R eine Fehlermeldung aus:

```
TRUE <- 5
```

```
#> Error in TRUE <- 5: invalid (do_set) left-hand side to assignment
```

Zudem sollte man Folgendes beachten:

- Namen sollten kurz und informativ sein; entsprechend ist **sample\_mean** ein guter Name, **vector\_2** dagegen eher weniger
- Man sollte **nie Umlaute in Namen verwenden**
- R ist *case sensitive*, d.h. **mean\_value** ist ein anderer Name als **Mean\_Value**
- Auch wenn möglich, sollte man nie von R bereit gestellte Funktionen überschreiben. Eine Zuweisung wie **assign <- 2** ist zwar möglich, führt in der Regel aber zu großem Unglück, weil man nicht mehr ganz einfach auf die zugrundeliegende Funktion zurückgreifen kann.

<sup>1</sup>Theoretisch kann  $\leftarrow$  auch andersherum verwendet werden:  $2 + 3 \rightarrow zwischenergebnis$ . Das mag zwar auf den ersten Blick intuitiver erscheinen, da das aus  $2 + 3$  resultierende Objekt den Namen **zwischenergebnis** bekommt, also immer erst das Objekt erstellt wird und dann der Name zugewiesen wird. Es führt jedoch zu deutlich weniger lesbarem Code und sollte daher nie verwendet werden. Ebensowenig sollten Zuweisungen durch den = Operator vorgenommen werden, auch wenn es im Fall **zwischenergebnis = 2 + 3** funktionieren würde.

**Hinweis:** Alle aktuellen Namenszuweisungen sind im Bereich `Environment` in R Studio (Nr. 4 in Abbildung 2.4 oben) aufgelistet und können durch die Funktion `ls()` angezeigt werden.

**Hinweis:** Ein Objekt kann mehrere Namen haben, aber kein Name kann zu mehreren Objekten zeigen, da im Zweifel eine neue Zuweisung die alte Zuweisung überschreibt:

```
x <- 2
y <- 2 # Das Objekt 2 hat nun zwei Namen
print(x)

#> [1] 2

print(y)

#> [1] 2

x <- 4 # Der Name 'x' zeigt nun zum Objekt '4', nicht mehr zu '2'
print(x)

#> [1] 4
```

**Hinweis:** Wie Sie vielleicht bereits bemerkt haben wird nach einer Zuweisung kein Wert sichtbar ausgegeben:

```
2 + 2 # Keine Zuweisung, R gibt das Ergebnis in der Konsole aus

#> [1] 4

x <- 2 + 2 # Zuweisung, R gibt das Ergebnis in der Konsole nicht aus
```

An dieser Stelle wollen wir das bisher Gelernte kurz zusammenfassen:

- Wir können Befehle in R Studio an den Computer übermitteln indem wir (a) den R Code in die Konsole schreiben und Enter drücken oder (b) den Code in ein Skript schreiben und dann ausführen
- Alles was in R *existiert* ist ein Objekt, alles was in R *passiert* ist ein Funktionsaufruf
- Wir können einem Objekt mit Hilfe von `<-` einen Namen geben und dann später wieder aufrufen. Den Prozess der Namensgebung nennen wir **Assignment** und wir können uns alle aktuell von uns vergebenen Namen mit der Funktion `ls()` anzeigen lassen
- Eine Funktion ist ein Objekt, das auf einen Input eine bestimmte Routine anwendet und einen Output produziert

An dieser Stelle sei noch auf die Hilfefunktion `help()` hingewiesen. Über diese können Sie weitere Informationen über ein Objekt bekommen. Wenn Sie z.B. genauere Informationen über die Verwendung der Funktion `assign` erhalten wollen, können Sie Folgendes eingeben:

```
help(assign)
```

### 3.3 Grundlegende Objekte in R

Wir haben bereits gelernt, dass alles was in R existiert ein Objekt ist. Wir haben aber auch schon gelernt, dass es unterschiedliche Typen von Objekten gibt: Zahlen, wie `2` oder `3` und Funktionen wie `assign`.<sup>2</sup> Tatsächlich gibt

---

<sup>2</sup>Wie wir unten lernen werden sind `2` und `3` in erster Linie keine Zahlen, sondern Vektoren der Länge 1, und gelten erst in nächster Instanz als ‘Zahl’ (genauer: ‘double’).

es noch viel mehr Arten von Objekten. Ein gutes Verständnis der Objektarten ist Grundvoraussetzung um später anspruchsvolle Programmieraufgaben zu lösen. Daher wollen wir uns im Folgenden mit den wichtigsten Objektarten in R auseinandersetzen.

### 3.3.1 Funktionen

Wie oben bereits kurz erwähnt handelt es sich bei Funktionen um Algorithmen, die bestimmte Routinen auf einen *Input* anwenden und dabei einen *Output* produzieren.

Die Funktion `log()` zum Beispiel nimmt als Input eine Zahl und gibt als Output den Logarithmus dieser Zahl aus:

```
log(2)
```

```
#> [1] 0.6931472
```

#### Eine Funktion aufrufen

In R gibt es prinzipiell vier verschiedene Arten Funktionen aufzurufen. Nur zwei davon sind allerdings aktuell für uns relevant.

Die bei weitem wichtigste Variante ist die so genannte *Prefix-Form*. Dies ist die Form, die wir bei der überwältigenden Anzahl von Funktionen verwenden werden. Wir schreiben hier zunächst den Namen der Funktion (im Folgenden Beispiel `assign`), dann in Klammern und mit Kommata getrennt die Argumente der Funktion (hier der Name `test` und die Zahl 2):

```
assign("test", 2)
```

Eine hin und wieder auftretende Form ist die sogenannte *Infix-Form*. Hier wird der Funktionsname zwischen die Argumente geschrieben. Dies ist, wie wir oben bereits bemerkt haben, bei vielen mathematischen Funktionen wie `+`, `-` oder `/` der Fall. Streng genommen ist die Infix-Form aber nur eine *Abkürzung*, denn jeder Funktionsaufruf in Infix-Form kann auch in Prefix-Form geschrieben werden, wie folgendes Beispiel zeigt:

```
2 + 3
```

```
#> [1] 5
```

```
+ (2,3)
```

```
#> [1] 5
```

#### Die Argumente einer Funktion

Die Argumente einer Funktion stellen zum einen den *Input* für die in der Funktion implementierte Routine dar.

Die Funktion `sum` zum Beispiel nimmt als Argumente eine beliebige Anzahl an Zahlen (ihr ‘Input’) und berechnet die Summe dieser Zahlen:

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

```
#> [1] 10
```

Darüber hinaus akzeptiert `sum()` noch ein *optionales Argument*, `na.rm`, welches entweder den Wert `TRUE` oder `FALSE` annehmen kann. Die Buchstaben `na` stehen hier für “not available”, und bezeichnen fehlende Werte. Wenn wir das Argument nicht explizit spezifizieren nimmt es automatisch `FALSE` als den Standardwert an.

Dieses optionale Argument ist kein klassischer Input, sondern kontrolliert das genaue Verhalten der Funktion. Wenn `na.rm` den Wert `TRUE` hat, dann werden im Falle von `sum()` die fehlende Werte, also die `NA`, ignoriert bevor die Summe der Inputs gebildet wird:

```
sum(1,2,3,4,NA)
#> [1] NA
sum(1,2,3,4,NA, na.rm = TRUE)
#> [1] 10
```

Wenn wir wissen wollen, welche Argumente eine Funktion akzeptiert, ist es immer eine gute Idee über die Funktion `help()` einen Blick in die Dokumentation zu werfen!

Im Falle von `sum()` sehen wir hier sofort, dass die Funktion neben den zu addierenden Zahlen ein optionales Argument `na.rm` akzeptiert, welches den Standardwert `FALSE` annimmt.

### Eigene Funktionen definieren

Sehr häufig möchten wir selbst Funktionen definieren. Das können wir mit dem reservierten Keyword `function` machen. Als Beispiel wollen wir eine Funktion `pythagoras` definieren, die als Argumente die Seitenlängen der Katheten eines rechtwinkligen Dreiecks annimmt und über den [Satz des Pythagoras](#) die Länge der Hypotenuse bestimmt:

```
pythagoras <- function(kathete_1, kathete_2){
  hypo_quadrat <- kathete_1**2 + kathete_2**2
  hypotenuse <- sqrt(hypo_quadrat) # sqrt() zieht die Quadratwurzel
  return(hypotenuse)
}
```

Wir definieren eine Funktion durch die Funktion `function()`. In der Regel beginnen wir die Definition indem wir die zu erstellende Funktion mit einem Namen assoziieren (hier: ‘`pythagoras`’) damit wir sie später auch verwenden können.

Die Argumente für `function` sind dann die Argumente, welche die zu definierende Funktion annehmen soll, in diesem Fall `kathete_1` und `kathete_2`. Danach beginnen wir den ‘function body’, also den Code für die Routine, welche die Funktion ausführen soll, mit einer geschweiften Klammer.

Innerhalb des *function bodies* wird dann die entsprechende Routine implementiert. Im vorliegenden Beispiel definieren wir zunächst die Summe der Werte von `kathete_1` und `kathete_2` als ein Zwischenergebnis, welches hier `hypo_quadrat` genannt wird. Dies ist der häufig unter  $c^2 = a^2 + b^2$  bekannte Teil des Satz von Pythagoras. Da wir an der ‘normalen’ Länge der Hypotenuse interessiert sind, ziehen wir mit der Funktion `sqrt()` noch die Wurzel von `hypo_quadrat`, und geben dem resultierenden Objekt den Namen `hypotenuse`, welches in der letzten Zeile mit Hilfe des Keywords `return` als der Wert definiert wird, den die Funktion als Output ausgibt.<sup>3</sup>

Am Ende der Routine kann man mit dem Keyword `return` explizit machen welchen Wert die Funktion als Output ausgeben soll. Wenn wir die Funktion nun aufrufen wird die oben definierte Routine ausgeführt:

```
pythagoras(2, 4)
```

```
#> [1] 4.472136
```

---

<sup>3</sup>Das ist strikt genommen nicht notwendig, aber der Übersichtlichkeit werden wir immer `return` verwenden. Eine interessante Debatte darüber ob man `return` verwenden sollte oder nicht findet sich [hier](#).

Beachten Sie, dass alle Objektnamen, die innerhalb des *function bodies* verwendet werden, nach dem Funktionsaufruf verloren gehen, weil Funktionen ihr eigenes **environment** haben. Deswegen kommt es im vorliegenden Falle zu einem Fehler, da `hypo_quadrat` nur innerhalb des *function bodies* existiert:

```
pythagoras <- function(kathete_1, kathete_2){
  hypo_quadrat <- kathete_1**2 + kathete_2**2
  hypotenuse <- sqrt(hypo_quadrat) # sqrt() zieht die Quadratwurzel
  return(hypotenuse)
}
x <- pythagoras(2, 4)
hypo_quadrat
```

```
#> Error in eval(expr, envir, enclos): object 'hypo_quadrat' not found
```

Es ist immer eine gute Idee, die selbst definierten Funktionen zu dokumentieren - nicht nur wenn wir sie auch anderen zur Verfügung stellen wollen, sondern auch damit wir selbst nach einer möglichen Pause unseren Code noch gut verstehen können. Nichts ist frustrierender als nach einer mehrwöchigen Pause viele Stunden investieren zu müssen, den eigens programmierten Code zu entschlüsseln!

Die Dokumentation von Funktionen kann mit Hilfe von einfachen Kommentaren erfolgen, ich empfehle jedoch sofort sich die [hier beschriebenen Konventionen](#) anzugewöhnen. In diesem Falle würde eine Dokumentation unserer Funktion `pythagoras` folgendermaßen aussehen:

```
'# Berechne die Länge der Hypotenuse in einem rechtwinkligen Dreieck
#'
#' Diese Funktion nimmt als Argumente die Längen der beiden Katheten eines
#' rechtwinkligen Dreiecks und berechnet daraus die Länge der Hypotenuse.
#' @param kathete_1 Die Länge der ersten Kathete
#' @param kathete_2 Die Länge der zweiten Kathete
#' @return Die Länge der Hypotenuse des durch a und b definierten
#' rechtwinkligen Dreieckst
pythagoras <- function(kathete_1, kathete_2){
  hypo_quadrat <- kathete_1**2 + kathete_2**2
  hypotenuse <- sqrt(hypo_quadrat) # sqrt() zieht die Quadratwurzel
  return(hypotenuse)
}
```

Die Dokumentation wird also direkt vor die Definition der Funktion gesetzt. In der ersten Zeile gibt man der Funktion einen maximal einzeiligen Titel, der nicht länger als 80 Zeichen sein sollte und die Funktion prägnant beschreibt.

Dann, nach einer Leerzeile wird genauer beschrieben was die Funktion macht. Danach werden die Argumente der Funktion beschrieben. Für jedes Argument beginnen wir die Reihe mit `@param`, gefolgt von dem Namen des Arguments und dann einer kurzen Beschreibung.

Nach den Argumenten beschreiben wir noch kurz was der Output der Funktion ist. Diese Zeile wird mit `@return` begonnen.

Die Dokumentation einer Funktion sollte also zumindest die Parameter und die Art des Outputs erklären.

## Gründe für die Verwendung eigener Funktionen

Eigene Funktionen zu definieren ist in der Praxis extrem hilfreich und es ist empfehlenswert Routinen, die mehrere Male verwendet werden grundsätzlich als Funktionen zu schreiben. Dafür gibt es mehrere Gründe:

1. **Der Code wird kürzer und transparenter.** Zwar ist kurzer Code nicht notwendigerweise leichter zu verstehen als langer, aber Funktionen können besonders gut dokumentiert werden (am besten indem man den hier beschriebenen Konventionen folgt).
2. **Funktionen bieten Struktur.** Funktionen fassen in der Regel Ihre Vorstellung davon zusammen, wie ein bestimmtes Problem zu lösen ist. Da man sich diese Gedanken nicht ständig neu machen möchte ist es sinnvoll sie einmalig in einer Funktion zusammenzufassen.
3. **Funktionen erleichtern Korrekturen.** Wenn Sie merken, dass Sie in der Implementierung einer Routine einen Fehler gemacht haben müssen Sie im besten Falle nur einmal die Definition der Funktion korrigieren - im schlimmsten Falle müssen Sie in Ihrem Code nach der Routine suchen und sie in jedem einzelnen Anwendungsfall erneut korrigieren.

Es gibt noch viele weitere Gründe dafür, Funktionen häufig zu verwenden. Viele hängen mit dem Entwicklerprinzip **DRY** (“Don’t Repeat Yourself”) zusammen.

### 3.3.2 Vektoren

Vektoren sind einer der wichtigsten Objekttypen in R. Quasi alle Daten mit denen wir in R arbeiten werden als Vektoren behandelt.

Was Vektoren angeht gibt es wiederum die wichtige **Unterscheidung von atomaren Vektoren und Listen**. Beide bestehen ihrerseits aus Objekten und sie unterscheiden sich dadurch, dass atomare Vektoren nur aus Objekten des gleichen Typs bestehen können, Listen dagegen auch Objekte unterschiedlichen Typs beinhalten können.

Entsprechend kann jeder atomare Vektor einem Typ zugeordnet werden, je nachdem welchen Typ seine Bestandteile haben. Hier sind insbesondere vier Typen relevant:

- **logical** (logische Werte): es gibt zwei logische Werte, **TRUE** und **FALSE**, welche auch mit **T** oder **F** abgekürzt werden können
- **integer** (ganze Zahlen): das sollte im Prinzip selbsterklärend sein, allerdings muss den ganzen Zahlen in R immer der Buchstabe **L** folgen, damit die Zahl tatsächlich als ganze Zahl interpretiert wird.<sup>4</sup> Beispiele sind **1L**, **400L** oder **10L**.
- **double** (Dezimalzahlen): auch das sollte selbsterklärend sein; Beispiele wären **1.5**, **0.0**, oder **-500.32**.
- Ganze Zahlen und Dezimalzahlen werden häufig unter der Kategorie **numeric** zusammengefasst. Dies ist in der Praxis aber quasi nie hilfreich und man sollte diese Kategorie möglichst nie verwenden.
- Wörter (**character**): sie sind dadurch gekennzeichnet, dass sie auch Buchstaben enthalten können und am Anfang und Ende ein " " haben. Beispiele hier wären **"Hallo"**, **"500"** oder **"1\_2\_Drei"**.
- Es gibt noch zwei weitere besondere ‘Typen’, die strikt gesehen keine atomaren Vektoren darstellen, allerdings in diesem Kontext schon häufig auftauchen: **NULL**, was strikt genommen ein eigener Datentyp ist und immer die Länge 0 hat, sowie **NA**, das einen fehlenden Wert darstellt.

Hieraus ergibt sich die in Abbildung 3.1 aufgezeigte Aufteilung von Vektoren.

Wir werden nun die einzelnen Typen genauer betrachten. Vorher wollen wir jedoch noch die Funktion **typeof**

---

<sup>4</sup>Diese auf den ersten Blick merkwürdige Syntax hat historische Gründe: als der **integer** Typ in die R Programmiersprache eingeführt wurde war er sehr stark an den Typ **long integer** in der Programmiersprache ‘C’ angelehnt. In C wurde ein solcher ‘long integer’ mit dem Suffix ‘l’ oder ‘L’ definiert, diese Regel wurde aus Kompatibilitätsgründen auch für R übernommen, jedoch nur mit ‘L’, da man Angst hatte, dass ‘l’ mit ‘i’ verwechselt wird, was in R für die imaginäre Komponente komplexer Zahlen verwendet wird.

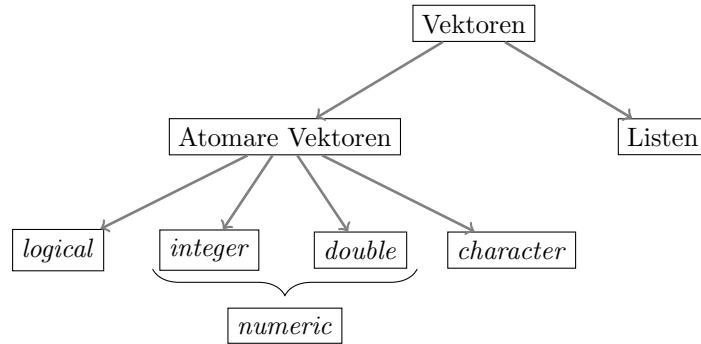


Figure 3.1: Arten von Vektoren in R

einführen. Sie hilft uns in der Praxis den Typ eines Objekts herauszufinden. Dafür rufen wir einfach die Funktion `typeof` mit dem zu untersuchenden Objekt oder dessen Namen auf:

```
typeof(2L)
```

```
#> [1] "integer"
x <- 22.0
typeof(x)
```

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

Wir können auch explizit testen ob ein Objekt tatsächlich ein Objekt eines bestimmten Typs ist. Die generelle Syntax hierfür ist: `is.*()`, also z.B.:

```
x <- 1.0
is.integer(x)

#> [1] FALSE
is.double(x)

#> [1] TRUE
```

Diese Funktion gibt als Output also immer einen logischen Wert aus, je nachdem ob die Inputs des entsprechenden Typs sind oder nicht.

Bestimmte Objekte können auch in einen anderen Typ transformiert werden. Hier spricht man von `coercion` und die generelle Syntax hierfür ist: `as.*()`, also z.B.:

```
x <- "2"
print(
  typeof(x)
)

#> [1] "character"
x <- as.double(x)
print(
  typeof(x)
)
```

```
)
```

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

Allerdings ist eine Transformation nicht immer möglich:

```
as.double("Hallo")
```

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

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

Da R nicht weiß wie man aus dem Wort ‘Hallo’ eine Dezimalzahl machen soll, transformiert R das Wort in einen ‘Fehlenden Wert’, der in R als **NA** bekannt ist und unten noch genauer diskutiert wird.

Für die Grundtypen ergibt sich folgende logische Hierarchie an trivialen Transformationen: **logical** → **integer** → **double** → **character**, d.h. man kann eine Dezimalzahl ohne Probleme in ein Wort transformieren, aber nicht umgekehrt:

### Warum überhaupt transformieren?

Für eine Programmiersprache sind Datentypen extrem wichtig, weil sonst unklar bliebe wie mathematische Operationen auf unterschiedliche Objekte wie Zahlen oder Wörter anzuwenden wären. Selbst transformieren werden Sie Objekte vor allem wenn Sie eine bestimmte, nur für eine bestimmte Objektart definierte Operation verwenden wollen und das Objekt bislang als ein anderer Typ gespeichert ist. Das kann zum Beispiel passieren wenn Sie Daten einlesen oder Wörter selbst in Zahlenwerte übersetzen. Wenn in Ihrem Code unerwartete Fehler mit kryptischen Fehlermeldungen auftauchen ist es immer eine gute Idee, erst einmal die Typen der verwendeten Objekte zu checken und die Objekte ggf. zu transformieren.

```
x <- 2
y <- as.character(x)
print(y)
```

```
#> [1] "2"
```

```
z <- as.double(y) # Das funktioniert
print(z)
```

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

```
k <- as.double("Hallo") # Das nicht
```

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

```
print(k)
```

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

Bei der Transformation logischer Werte wird **TRUE** übrigens zu 1 und **FALSE** zu 0, eine Tatsache, die wir uns später noch zunutze machen werden:

```
x <- TRUE
as.integer(x)

#> [1] 1

y <- FALSE
as.integer(y)

#> [1] 0
```

Da nicht immer ganz klar ist wann R bei Transformationen entgegen der gerade eingeführten Hierarchie eine Warnung ausgibt und wann nicht sollte man hier immer besondere Vorsicht walten lassen!

Zudem ist bei jeder Transformation Vorsicht geboten, da sie häufig Eigenschaften der Objekte implizit verändert. So führt eine Transformation von einer Dezimalzahl hin zu einer ganzen Zahl teils zu unerwartetem Rundungsverhalten:

```
x <- 1.99
as.integer(x)

#> [1] 1
```

Auch führen Transformationen, die der eben genannten Hierarchie zuwiderlaufen, nicht zwangsläufig zu Fehlern, sondern ‘lediglich’ zu unerwarteten Änderungen, die in jedem Fall vermieden werden sollten:

```
z <- as.logical(99)
print(z)

#> [1] TRUE
```

Häufig transformieren Funktionen ihre Argumente automatisch, was meistens hilfreich ist, manchmal aber auch gefährlich sein kann:

```
x <- 1L # Integer
y <- 2.0 # Double
z <- x + y
typeof(z)

#> [1] "double"
```

Bei einer Addition werden logische Werte ebenfalls automatisch transformiert:

```
x <- TRUE
y <- FALSE
z <- x + y # TRUE wird zu 1, FALSE zu 0
print(z)
```

```
#> [1] 1
```

Daher sollte man immer den Überblick behalten, mit welchen Objekttypen man gerade arbeitet.

Einen Überblick zu den Test- und Transformationsbefehlen finden Sie in Tabelle 3.1.

Table 3.1: Ein Überblick zu Test- und Transformationsbefehlen in R.

Typ	Test	Transformation
logical	<code>is.logical</code>	<code>as.logical</code>
double	<code>is.double</code>	<code>as.double</code>
integer	<code>is.integer</code>	<code>as.integer</code>
character	<code>is.character</code>	<code>as.character</code>
function	<code>is.function</code>	<code>as.function</code>
NA	<code>is.na</code>	NA
NULL	<code>is.null</code>	<code>as.null</code>

Ein letzter Hinweis zu **Skalaren**. Unter Skalaren verstehen wir in der Regel ‘einzelne Zahlen’, z.B. 2. Dieses Konzept gibt es in R nicht. 2 ist ein Vektor der Länge 1. Wir unterscheiden also vom Typ her nicht zwischen einem Vektor, der nur ein oder mehrere Elemente hat.

**Hinweis:** Um längere Vektoren zu erstellen, verwenden wir die Funktion `c()`:

```
x <- c(1, 2, 3)
x
```

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

Dabei können Vektoren auch miteinander verbunden werden:

```
x <- 1:3 # Shortcut für: x <- c(1, 2, 3)
y <- 4:6
z <- c(x, y)
z
```

```
#> [1] 1 2 3 4 5 6
```

Da atomare Vektoren immer nur Objekte des gleichen Typs enthalten, könnte man erwarten, dass es zu einem Fehler kommt, wenn wir Objekte unterschiedlichen Type kombinieren wollen:

```
x <- c(1, "Hallo")
```

Tatsächlich transformiert R die Objekte allerdings nach der oben beschriebenen Hierarchie `logical → integer → double → character`. Da hier keine Warnung oder kein Fehler ausgegeben wird, sind derlei Transformationen eine gefährliche Fehlerquelle!

**Hinweis:** Die Länge eines Vektors kann mit der Funktion `length` bestimmt werden:

```
x = c(1, 2, 3)
len_x <- length(x)
len_x
```

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

### 3.3.3 Logische Werte (logical)

Die logischen Werte `TRUE` und `FALSE` sind häufig das Ergebnis von logischen Abfragen, z.B. ‘Ist 2 größer als 1?’ Solche Abfragen kommen in der Forschungspraxis häufig vor und es macht Sinn, sich mit den häufigsten logischen Operatoren vertraut zu machen. Einen Überblick finden Sie in Tabelle 3.2.

Table 3.2: Zentrale logische Abfragen in R.

Operator	Funktion in R	Beispiel
größer	<code>&gt;</code>	<code>2&gt;1</code>
kleiner	<code>&lt;</code>	<code>2&lt;4</code>
gleich	<code>==</code>	<code>4==3</code>
größer gleich	<code>&gt;=</code>	<code>8&gt;=8</code>
kleiner gleich	<code>&lt;=</code>	<code>5&lt;=9</code>
nicht gleich	<code>!=</code>	<code>4!=5</code>
und	<code>&amp;</code>	<code>x&lt;90 &amp; x&gt;55</code>
oder	<code> </code>	<code>x&lt;90   x&gt;55</code>
entweder oder	<code>xor()</code>	<code>xor(2&lt;1, 2&gt;1)</code>
nicht	<code>!</code>	<code>!(x==2)</code>
ist wahr	<code>isTRUE()</code>	<code>isTRUE(1&gt;2)</code>

Das Ergebnis eines solches Tests ist immer ein logischer Wert:

```
x <- 4
y <- x == 8
typeof(y)
```

```
#> [1] "logical"
```

Es können auch längere Vektoren getestet werden:

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

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

Tests können beliebig miteinander verknüpft werden:

```
x <- 1L
x>2 | x<2 & (is.double(x) & x!=0)
```

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

Da für viele mathematischen Operationen `TRUE` als die Zahl 1 interpretiert wird, ist es einfach zu testen wie häufig eine bestimmte Bedingung erfüllt ist:

```
x <- 1:50
smaller_20 <- x<20
print(
  sum(smaller_20) # Wie viele Elemente sind kleiner als 20?
)
```

```
#> [1] 19
print(
  sum(smaller_20/length(x)) # Wie hoch ist der Anteil von diesen Elementen?
)
#> [1] 0.38
```

### 3.3.4 Wörter (character)

Wörter werden in R dadurch gebildet, dass an ihrem Anfang und Ende das Symbol ' oder " steht:

```
x <- "Hallo"
typeof(x)

#> [1] "character"

y <- 'Auf Wiedersehen'
typeof(y)

#> [1] "character"
```

Wie andere Vektoren können sie mit der Funktion `c()` verbunden werden:

```
z <- c(x, "und", y)
z

#> [1] "Hallo"           "und"            "Auf Wiedersehen"
```

Nützlich ist in diesem Zusammenhang die Funktion `paste()`, die Elemente von mehreren Vektoren in Wörter transformiert und verbindet:

```
x <- 1:10
y <- paste("Versuch Nr.", x)
y

#> [1] "Versuch Nr. 1"  "Versuch Nr. 2"  "Versuch Nr. 3"  "Versuch Nr. 4"
#> [5] "Versuch Nr. 5"  "Versuch Nr. 6"  "Versuch Nr. 7"  "Versuch Nr. 8"
#> [9] "Versuch Nr. 9"  "Versuch Nr. 10"
```

Die Funktion `paste()` akzeptiert ein optionales Argument `sep`, mit dem wir den Wert angeben können, der zwischen die zu verbindenden Elemente gesetzt wird (der Default ist `sep = " "`):

```
tag_nr <- 1:10
x_axis <- paste("Tag", tag_nr, sep = ": ")
x_axis

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

Hinweis: Hier haben wir ein Beispiel für das sogenannte ‘Recycling’ gesehen: da der Vektor `c("Tag")` kürzer war als der Vektor `tag_nr` wird `c("Tag")` einfach kopiert damit die Operation mit `paste()` Sinn ergibt. Recycling ist oft praktisch, aber manchmal auch schädlich, nämlich dann, wenn man eigentlich davon ausgeht eine Operation mit zwei gleich langen Vektoren durchzuführen, dies aber tatsächlich nicht tut. In einem solchen Fall führt Recycling dazu, dass keine Fehlermeldung ausgegeben wird. Ein Beispiel

dafür gibt folgender Code, in dem die Intention klar die Verbindung aller Wochentage zu Zahlen ist und einfach ein Wochentag vergessen wurde:

```
tage <- paste("Tag ", 1:7, ":", sep="")
tag_namen <- c("Montag", "Dienstag", "Mittwoch", "Donnerstag", "Freitag", "Samstag")
paste(tage, tag_namen) # default ist sep=" "

#> [1] "Tag 1: Montag"      "Tag 2: Dienstag"    "Tag 3: Mittwoch"
#> [4] "Tag 4: Donnerstag" "Tag 5: Freitag"     "Tag 6: Samstag"
#> [7] "Tag 7: Montag"
```

### 3.3.5 Fehlende Werte und NULL

Fehlende Werte werden in R als `NA` kodiert. `NA` erfüllt gerade in statistischen Anwendungen eine wichtige Rolle, da ein bestimmter Platz in einem Vektor aktuell fehlend sein müsste, aber als Platz dennoch existieren muss.

**Beispiel:** Der Vektor `x` enthält einen logischen Wert, der zeigt ob eine Person die Fragen auf einem Fragebogen richtig beantwortet hat. Wenn die Person die dritte Frage auf dem Fragebogen nicht beantwortet hat, sollte dies durch `NA` kenntlich gemacht werden. Einfach den Wert komplett wegzulassen macht es im Nachhinein unmöglich festzustellen welche Frage die Person nicht beantwortet hat.

Die meisten Operationen die `NA` als einen Input bekommen geben auch als Output `NA` aus, weil unklar ist wie die Operation mit unterschiedlichen Werten für den fehlenden Wert ausgehen würde:

```
5 + NA
```

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

Einige Ausnahmen sind Operationen, die unabhängig vom fehlenden Wert einen bestimmten Wert annehmen:

```
NA | TRUE # Gibt immer TRUE, unabhängig vom Wert für NA
```

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

Um zu testen ob ein Vektor `x` fehlende Werte enthält sollte die Funktion `is.na` verwendet werden, und nicht etwa der Ausdruck `x==NA`:

```
x <- c(NA, 5, NA, 10)
print(x == NA) # Unklar, da man nicht weiß, ob alle NA für den gleichen Wert stehen
```

```
#> [1] NA NA NA NA
print(
  is.na(x)
)
```

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

Wenn eine Operation einen nicht zu definierenden Wert ausgibt, ist das Ergebnis nicht `NA` sondern `NaN` (*not a number*):

```
0 / 0
```

```
#> [1] NaN
```

Eine weitere Besonderheit ist `NULL`, welches in der Regel als Vektor der Länge 0 gilt, aber häufig zu besonderen Zwecken verwendet wird:

```
x <- NULL
length(x)
```

```
#> [1] 0
```

`NULL` wird häufig verwendet um zu signalisieren, dass etwas nicht existiert. So ist ein leerer Vektor `NULL`:

```
x <- c()
x
```

```
#> NULL
```

```
length(x)
```

```
#> [1] 0
```

Damit unterscheidet er sich von einem Vektor mit einem (oder mehreren) fehlenden Werten:

```
y <- NA
length(y)
```

```
#> [1] 1
```

Auch im Programmieren von Funktionen wird `NULL` häufig für optionale Argumente verwendet. Solche fortgeschrittenen Konzepte werden aber erst an späterer Stelle behandelt. Für jetzt reicht die Idee, `NULL` als einen Vektor der Länge 0 zu verstehen.

### 3.3.6 Indizierung und Ersetzung

Einzelne Elemente von atomaren Vektoren können mit eckigen Klammern extrahiert werden:

```
x <- c(2,4,6)
x[1]
```

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

Auf diese Weise können auch bestimmte Elemente modifiziert werden:

```
x <- c(2,4,6)
x[2] <- 99
x
```

```
#> [1] 2 99 6
```

Es kann auch mehr als ein Element extrahiert werden:

```
x[1:2]
```

```
#> [1] 2 99
```

Negative Indizes sind auch möglich, diese eliminieren die entsprechenden Elemente:

```
x[-1]
```

```
#> [1] 99 6
```

Um das letzte Element eines Vektors zu bekommen verwendet man einen Umweg über die Funktion `length()`:

```
x[length(x)]
```

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

### 3.3.7 Nützliche Funktionen für atomare Vektoren

Hier sollen nur einige Funktionen erwähnt werden, die im Kontext von atomaren Vektoren besonders praktisch sind,<sup>5</sup> insbesondere wenn es darum geht solche Vektoren herzustellen, bzw. Rechenoperationen mit ihnen durchzuführen.

**Herstellung von atomaren Vektoren:**

Eine Sequenz ganzer Zahlen wird in der Regel sehr häufig gebraucht. Entsprechend gibt es den hilfreichen Shortcut `:`, den wir bei der Besprechung von Vektoren bereits kennengelernt haben:

```
x <- 1:10
```

```
x
```

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

```
y <- 10:1
```

```
y
```

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

Häufig möchten wir jedoch eine kompliziertere Sequenz bauen. In dem Fall hilft uns die allgemeinere Funktion `seq()`:

```
x <- seq(1, 10)
```

```
print(x)
```

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

In diesem Fall ist `seq()` äquivalent zu `:`. Die Funktion `seq` erlaubt aber mehrere optionale Argumente: so können wir mit `by` die Schrittänge zwischen den einzelnen Zahlen definieren.

```
y <- seq(1, 10, by = 0.5)
```

```
print(y)
```

```
#> [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0 6.5 7.0 7.5 8.0
```

```
#> [16] 8.5 9.0 9.5 10.0
```

Wenn wir die Länge des resultierenden Vektors festlegen wollen und die Schrittänge von R automatisch festgelegt werden soll, können wir dies mit dem Argument `length.out` machen:

```
z <- seq(2, 8, length.out = 4)
```

```
print(z)
```

```
#> [1] 2 4 6 8
```

Und wenn wir einen Vektor in der Länge eines anderen Vektors erstellen wollen, bietet sich das Argument `along.with` an. Dies wird häufig für das Erstellen von Indexvektoren verwendet.<sup>6</sup> In einem solchen Fall müssen

<sup>5</sup>Für viele typische Aufgaben gibt es in R bereits eine vordefinierte Funktion. Am einfachsten findet man diese durch googlen.

<sup>6</sup>Ein Indexvektor `x` zu einem beliebigen Vektor `y` mit `N` Elementen enthält die ganzen Zahlen von 1 bis `N`. Der `n`-te Wert von `x` korrespondiert also zum Index des `n`-ten Wert von `y`.

wir die Indexzahlen nicht direkt angeben:

```
z_index <- seq(along.with = z)
print(z_index)
```

```
#> [1] 1 2 3 4
```

Auch häufig möchten wir einen bestimmten Wert wiederholen. Das geht mit der Funktion `rep`:

```
x <- rep(NA, 5)
print(x)
```

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

## Rechenoperationen

Es gibt eine Reihe von Operationen, die wir sehr häufig gemeinsam mit Vektoren anwenden. Häufig interessiert und die **Länge** eines Vektors. Dafür können wir die Funktion `length()` verwenden:

```
x <- c(1,2,3,4)
length(x)
```

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

Wenn wir den **größten** oder **kleinsten Wert** eines Vektors erfahren möchten geht das mit den Funktionen `min()` und `max()`:

```
min(x)
```

```
#> [1] 1
```

```
max(x)
```

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

Beide Funktionen besitzen ein optionales Argument `na.rm`, das entweder TRUE oder FALSE sein kann. Im Falle von TRUE werden alle NA Werte für die Rechenoperation entfernt:

```
y <- c(1,2,3,4,NA)
min(y)
```

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

```
min(y, na.rm = TRUE)
```

```
#> [1] 1
```

Den **Mittelwert** bzw die **Varianz/Standardabweichung** der Elemente bekommen wir mit `mean()`, `var()`, bzw. `sd()`, wobei alle Funktionen auch das optionale Argument `na.rm` akzeptieren:

```
mean(x)
```

```
#> [1] 2.5
```

```
var(y)
```

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

```
var(y, na.rm = T)
```

```
#> [1] 1.666667
```

Ebenfalls häufig sind wir an der **Summe**, bzw, dem **Produkt** aller Elemente des Vektors interessiert. Die Funktionen **sum()** und **prod()** helfen weiter und auch sie kennen das optionale Argument **na.rm**:

```
sum(x)
```

```
#> [1] 10
```

```
prod(y, na.rm = T)
```

```
#> [1] 24
```

### 3.3.8 Listen

Im Gegensatz zu atomaren Vektoren können Listen Objekte verschiedenen Typs enthalten. Sie werden mit der Funktion **list()** erstellt:

```
l_1 <- list(
  "a",
  c(1,2,3),
  FALSE
)
typeof(l_1)
```

```
#> [1] "list"
```

```
l_1
```

```
#> [[1]]
#> [1] "a"
#>
#> [[2]]
#> [1] 1 2 3
#>
#> [[3]]
#> [1] FALSE
```

Wir können Listen mit der Funktion **str()** (kurz für “structure”) inspizieren. In diesem Fall erhalten wir unmittelbar Informationen über die Art der Elemente:

```
str(l_1)
```

```
#> List of 3
#> $ : chr "a"
#> $ : num [1:3] 1 2 3
#> $ : logi FALSE
```

Die einzelnen Elemente einer Liste können auch benannt werden:

```
l_2 <- list(
  "erstes_element" = "a",
  "zweites_element" = c(1,2,3),
  "drittes_element" = FALSE
)
```

Die Namen aller Elemente in der Liste erhalten wir mit der Funktion `names()`:

```
names(l_2)
```

```
#> [1] "erstes_element" "zweites_element" "drittes_element"
```

Um einzelne Elemente einer Liste auszulesen müssen wir `[[` anstatt `[` verwenden. Wir können dann Elemente entweder nach ihrer Position oder nach ihren Namen auswählen:

```
l_2[[1]]
```

```
#> [1] "a"
```

```
l_2[["erstes_element"]]
```

```
#> [1] "a"
```

Im Folgenden wollen wir uns noch mit drei speziellen Typen beschäftigen, die weniger fundamental als die bislang diskutierten Typen sind, jedoch häufig in der alltäglichen Arbeit vorkommen: Faktoren, Matrizen und Data Frames.

### 3.3.9 Faktoren

Faktoren werden verwendet um ordinale oder kategoriale Daten darzustellen. Ein Faktor kann nur einen von mehreren vorher definierten Werten annehmen, so genannten *Levels*. Faktoren werden über die Funktion `factor()` erstellt. Sie nimmt als erstes Argument die Werte für den Faktor:

```
x <- c("Frau", "Mann", "Frau")
x <- factor(c("Frau", "Mann", "Frau"))
x
```

```
#> [1] Frau Mann Frau
```

```
#> Levels: Frau Mann
```

Wenn wir Levels definieren wollen, die aber aktuell noch keine Ausprägung haben können wir dies mit dem Argument `levels` bewerkstelligen:

```
x <- c("Frau", "Mann", "Frau")
x <- factor(c("Frau", "Mann", "Frau"),
            levels=c("Divers","Frau", "Mann"))
x
```

```
#> [1] Frau Mann Frau
```

```
#> Levels: Divers Frau Mann
```

Wenn wir das Argument `levels` verwenden werden dort nicht genannte Ausprägungen den Wert `NA` erhalten:

```
x <- c("Frau", "Mann", "Frau")
x <- factor(c("Frau", "Mann", "Frau", "Divers"),
```

```

  levels=c("Frau", "Mann"))

x

#> [1] Frau Mann Frau <NA>
#> Levels: Frau Mann

```

Die Reihenfolge der einzelnen Levels spielt meist keine Rolle. Bei ordinalen Daten möchten wir aber eine sinnvolle Wertigkeit der Ausprägungen sicherstellen. Das geht mit der Funktion `factor()` und dem Argument `ordered`:

```

x <- c("Hoch", "Hoch", "Gering", "Hoch")
x <- factor(x,
            levels = c("Gering", "Mittel", "Hoch"),
            ordered = TRUE)
x

#> [1] Hoch   Hoch   Gering Hoch
#> Levels: Gering < Mittel < Hoch

```

Häufig handelt es sich bei den Ausprägungen von Faktoren um Wörter, also Objekte vom Type `character`. Technisch gesehen werden Faktoren aber als `integer` gespeichert: um Speicherplatz zu sparen wird jedem Level auf dem Computer eine ganze Zahl zugewiesen, die dann auf den eigentlichen Wert gemapt wird. Gerade wenn die Ausprägungen als solche große Zahlen oder lange Wörter sind spart das Speicher, weil diese Ausprägungen nur einmal gespeichert werden müssen, und jedes Element des Faktors nur noch eine einfache Zahl ist. Daher gibt `typeof()` für Faktoren auch `integer` aus:

```

x <- factor(c("Frau", "Mann", "Frau"),
            levels=c("Mann", "Frau", "Divers"))
typeof(x)

#> [1] "integer"

```

Um zu überprüfen ob es sich bei einem Objekt um einen Faktor handelt verwenden wir die Funktion `is.factor()`:

```
is.factor(x)
```

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

Manche Operationen, die für `integer` definiert sind, funktionieren bei Faktoren aber nicht, z.B. Addition:

```
x[1] + x[2]
```

```
#> Warning in Ops.factor(x[1], x[2]): '+' not meaningful for factors
```

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

Dafür können wir andere nützliche Dinge mit Faktoren anstellen, z.B. die absoluten Häufigkeiten über die Funktion `table` anzeigen:

```
table(x)
```

```

#> x
#>   Mann   Frau Divers
#>     1      2      0

```

Faktoren werden vor allem in der Arbeit mit ordinalen und kategorialen Daten verwendet (siehe Kapitel 4).

### 3.3.10 Matrizen

Bei Matrizen handelt es sich um zweidimensionale Objekte mit Zeilen und Spalten, bei denen es sich jeweils um atomare Vektoren handelt.

#### Erstellen von Matrizen

Matrizen werden mit der Funktion `matrix()` erstellt. Diese Funktion nimmt als erstes Argument die Elemente der Matrix und dann die Spezifikation der Anzahl von Zeilen (`nrow`) und/oder der Anzahl von Spalten (`ncol`):

```
m_1 <- matrix(11:20, nrow = 5)
m_1
```

```
#>      [,1] [,2]
#> [1,]    11   16
#> [2,]    12   17
#> [3,]    13   18
#> [4,]    14   19
#> [5,]    15   20
```

Wir können die Zeilen und Spalten sowie einzelne Werte folgendermaßen extrahieren und gegebenenfalls Ersetzungen vornehmen:

```
m_1[,1] # Erste Spalte
```

```
#> [1] 11 12 13 14 15
```

```
m_1[1,] # Erste Zeile
```

```
#> [1] 11 16
```

```
m_1[2,2] # Element [2,2]
```

```
#> [1] 17
```

**Optionaler Hinweis:** Matrizen sind weniger ‘fundamental’ als atomare Vektoren. Entsprechend gibt uns `typeof()` für eine Matrix auch den Typ der enthaltenen atomaren Vektoren an:

```
typeof(m_1)
```

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

Um zu testen ob es sich bei einem Objekt um eine Matrix handelt verwenden wir entsprechend `is.matrix()`:

```
is.matrix(m_1)
```

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

```
is.matrix(2.0)
```

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

Die Grundlagen der Matrizenalgebra und ihre Implementierung in R wird später in Kapitel 6 erläutert. Zudem gibt es im Internet zahlreiche gute Überblicksartikel zum Thema Matrizenalgebra in R, z.B. [hier](#) oder in größerem Umfang [hier](#).

### 3.3.11 Data Frames

Der `data.frame` ist eine besondere Art von Liste und ist ein in der Datenanalyse regelmäßig auftretender Datentyp. Im Gegensatz zu einer normalen Liste müssen bei einem `data.frame` alle Elemente die gleiche Länge aufweisen. Das heißt man kann sich einen `data.frame` als eine rechteckig angeordnete Liste vorstellen.

Wegen der engen Verwandschaft können wir einen `data.frame` direkt aus einer Liste erstellen indem wir die Funktion `as.data.frame()` verwenden:

```
l_3 <- list(
  "a" = 1:3,
  "b" = 4:6,
  "c" = 7:9
)
df_3 <- as.data.frame(l_3)
```

Wenn wir R nach dem Typ von `df_3` fragen, sehen wir, dass es sich weiterhin um eine Liste handelt:

```
typeof(df_3)
```

```
#> [1] "list"
```

Allerdings können wir testen ob `df_3` ein `data.frame` ist indem wir `is.data.frame` benutzen:

```
is.data.frame(df_3)
```

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

```
is.data.frame(l_3)
```

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

Wenn wir `df_3` ausgeben sehen wir unmittelbar den Unterschied zur klassischen Liste:

```
l_3
```

```
#> $a
```

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

```
#>
```

```
#> $b
```

```
#> [1] 4 5 6
```

```
#>
```

```
#> $c
```

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

```
df_3
```

```
#>   a b c
```

```
#> 1 1 4 7
```

```
#> 2 2 5 8
```

```
#> 3 3 6 9
```

Die andere Möglichkeit einen `data.frame` zu erstellen ist direkt über die Funktion `data.frame()`, wobei es hier in der Regel ratsam ist das optionale Argument `stringsAsFactors` auf `FALSE` zu setzen, da sonst Wörter in so

genannte Faktoren umgewandelt werden:<sup>7</sup>

```
df_4 <- data.frame(
  "gender" = c(rep("male", 3), rep("female", 2)),
  "height" = c(189, 175, 180, 166, 150),
  stringsAsFactors = FALSE
)
df_4
```

```
#>   gender height
#> 1 male    189
#> 2 male    175
#> 3 male    180
#> 4 female  166
#> 5 female  150
```

Data Frames sind das klassische Objekt um eingelesene Daten zu repräsentieren. Wenn Sie sich z.B. Daten zum BIP in Deutschland aus dem Internet runterladen und diese Daten dann in R einlesen, werden diese Daten zunächst einmal als `data.frame` repräsentiert.<sup>8</sup> Diese Repräsentation erlaubt dann eine einfache Analyse und Manipulation der Daten.

Zwar gibt es ein eigenes Kapitel zur Bearbeitung von Daten (siehe Kapitel 4), wir wollen aber schon hier einige zentrale Befehle im Zusammenhang von Data Frames einführen.

An dieser Stelle sei schon angemerkt, dass um Zeilen, Spalten oder einzelne Elemente auszuwählen die gleichen Befehle wie bei Matrizen verwendet werden können:

```
df_4[, 1] # erste Spalte
#> [1] "male"    "male"    "male"    "female"  "female"
df_4[, 2] # Werte der zweiten Spalte
#> [1] 189 175 180 166 150
```

Die Abfrage funktioniert nicht nur mit Indices, sondern auch mit Spaltennamen:<sup>9</sup>

```
df_4[["gender"]]
#> [1] "male"    "male"    "male"    "female"  "female"
```

Wenn wir `[` anstatt von `[[` verwenden erhalten wir als Output einen (reduzierten) Data Frame:

```
df_4[["gender"]]
```

```
#>   gender
#> 1 male
#> 2 male
#> 3 male
```

---

<sup>7</sup>Zur Geschichte dieses wirklich ärgerlichen Verhaltens siehe [diesen Blog](#).

<sup>8</sup>Das ist nicht ganz korrekt, weil es mittlerweile Erweiterungen gibt, welche den `data.frame` mit effizienteren Objekten ersetzen, z.B. dem `tibble` oder dem `data.table`. Der Umgang mit diesen Objekten ist jedoch sehr ähnlich zum `data.frame`.

<sup>9</sup>Anstelle von `[[` kann auch der Shortcut `$` verwendet werden. Das werden wir aufgrund der größeren Transparenz von `[[` hier jedoch nicht verwenden.

```
#> 4 female
#> 5 female
```

Es können auch mehrere Zeilen ausgewählt werden:

```
df_4[1:2, ] # Die ersten beiden Zeilen
```

```
#>   gender height
#> 1 male     189
#> 2 male     175
```

Oder einzelne Werte:

```
df_4[2, 2] # Zweiter Wert der zweiten Spalte
```

```
#> [1] 175
```

Dies können wir uns zu Nutze machen um den Typ der einzelnen Spalten herauszufinden:

```
typeof(df_4[["gender"]])
```

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

Gerade bei sehr großen Data Frames möchte man oft nur die ersten paar Zeilen inspizieren. Das ist mit der Funktion `head()` möglich. Das erste Argument ist immer der Name des Data Frames. Das zweite (optionale) Argument ist ein `integer`, der die Anzahl der anzuzeigenden Zeilen angibt (Standardwert: 5):

```
head(df_4, 2) # gibt die ersten zwei Zeilen aus
```

```
#>   gender height
#> 1 male     189
#> 2 male     175
```

## 3.4 Pakete

Bei Paketen handelt es sich um eine Kombination aus R Code, Daten, Dokumentationen und Tests. Sie sind der beste Weg, reproduzierbaren Code zu erstellen und frei zugänglich zu machen. Zwar werden Pakete häufig der Öffentlichkeit zugänglich gemacht, z.B. über GitHub oder CRAN, es ist aber genauso hilfreich, Pakete für den privaten Gebrauch zu schreiben, z.B. um für bestimmte Routinen Funktionen zu programmieren, zu dokumentieren und in verschiedenen Projekten verfügbar zu machen.<sup>10</sup>

Die Tatsache, dass viele Menschen statistische Probleme lösen indem sie bestimmte Routinen entwickeln, diese dann generalisieren und über Pakete der ganzen R Community frei verfügbar machen, ist einer der Hauptgründe für den Erfolg und die breite Anwendbarkeit von R.

Wenn man R startet haben wir Zugriff auf eine gewisse Anzahl von Funktionen, vordefinierten Variablen und Datensätzen. Die Gesamtheit dieser Objekte wird in der Regel `base R` genannt, weil wir alle Funktionalitäten ohne Weiteres nutzen können.

Die Funktion `assign`, zum Beispiel, ist Teil von `base R`: wir starten R und können sie ohne Weiteres verwenden.

Im Prinzip kann so gut wie jedwede statistische Prozedur in `base R` implementiert werden. Dies ist aber häufig zeitaufwendig und fehleranfällig: wie wir am Beispiel von Funktionen gelernt haben, sollten häufig verwendete

---

<sup>10</sup>Wickham and Bryan (2019) bietet eine exzellente Einführung in das Programmieren von R Paketen.

Routinen im Rahmen von einer Funktion implementiert werden, die dann immer wieder angewendet werden kann. Das reduziert nicht nur Fehler, sondern macht den Code besser verständlich.

Pakete folgen dem gleichen Prinzip, nur tragen sie die Idee noch weiter: hier wollen wir die Funktionen auch über ein einzelnes R Projekt hinaus nutzbar machen, sodass sie nicht in jedem Projekt neu definiert werden müssen, sondern zentral nutzbar gemacht und dokumentiert werden.

Um ein Paket in R zu nutzen, muss es zunächst installiert werden. Für Pakete, die auf der zentralen R Pakete Plattform CRAN verfügbar sind, geht dies mit der Funktion `install.packages`. Wenn wir z.B. das Paket `data.table` installieren wollen geht das mit dem folgenden Befehl:

```
install.packages("data.table")
```

Das Paket `data.table` enthält viele Objekte, welche die Arbeit mit großen Datensätzen enorm erleichtern. Beispielsweise ist darunter eine verbesserte Version des `data.frame`, der `data.table`. Wir können einen `data.frame` mit Hilfe der Funktion `as.data.table()` in einen `data.table` umwandeln.

Allerdings haben wir selbst nach erfolgreicher Installation von `data.table` nicht direkt Zugriff auf diese Funktion:

```
x <- data.frame(
  a=1:5,
  b=21:25
)
as.data.table(x)

#> Error in as.data.table(x): could not find function "as.data.table"
```

Wir haben zwei Möglichkeiten auf die Objekte im Paket `data.table` zuzugreifen: zum einen können wir mit dem Operator `::` arbeiten:

```
y <- data.table::as.data.table(x)
y

#>     a   b
#> 1: 1 21
#> 2: 2 22
#> 3: 3 23
#> 4: 4 24
#> 5: 5 25
```

Wir schreiben also den Namen des Pakets, direkt gefolgt von `::` und dann den Namen des Objekts aus dem Paket, das wir verwenden wollen.

Zwar ist das der transparenteste und sauberste Weg auf Objekte aus anderen Paketen zuzugreifen, allerdings kann es auch nervig sein wenn man häufig oder sehr viele Objekte aus dem gleichen Paket verwendet. Wir können alle Objekte eines Paketes direkt zugänglich machen indem wir die Funktion `library()` verwenden.

```
library(data.table)
y <- as.data.table(x)
```

Der Übersicht halber sollte das für alle in einem Skript verwendeten Pakete ganz am Anfang des Skripts gemacht werden. So sieht man auch unmittelbar welche Pakete für das Skript installiert sein müssen.

Grundsätzlich sollte man in jedem Skript nur die Pakete mit `library()` einlesen, die auch tatsächlich verwendet werden. Ansonsten lädt man unnötigerweise viele Objekte und verliert den Überblick woher eine bestimmte Funktion eigentlich kommt. Außerdem ist es schwieriger für andere das Skript zu verwenden, weil unter Umständen viele Pakete unnötigerweise installiert werden müssen.

Da Pakete dezentral von verschiedenen Menschen hergestellt werden, besteht die Gefahr, dass Objekte in unterschiedlichen Paketen den gleichen Namen bekommen. Da in R ein Name nur zu einem Objekt gehören kann, werden beim Einladen mehrerer Pakete eventuell Namen überschrieben, oder ‘maskiert’. Dies wird am Anfang beim Einlesen der Pakete mitgeteilt, gerät aber leicht in Vergessenheit und kann zu sehr kryptischen Fehlermeldungen führen.

Wir wollen das kurz anhand der beiden Pakete `dplyr` und `plm` illustrieren:

```
library(dplyr)

library(plm)

#>
#> Attaching package: 'plm'

#> The following objects are masked from 'package:dplyr':
#>
#>     between, lag, lead

#> The following object is masked from 'package:data.table':
#>
#>     between
```

In beiden Paketen gibt es Objekte mit den Namen `between`, `lag` und `lead`. Bei der Verwendung von `library` maskiert das später eingelesene Paket die Objekte des früheren. Wir können das illustrieren indem wir den Namen des Objekts eingeben:

```
lead

#> function (x, k = 1, ...)
#> {
#>     UseMethod("lead")
#> }
#> <bytecode: 0x7fa0ab07ceb8>
#> <environment: namespace:plm>
```

Aus der letzten Zeile wird ersichtlich, dass `lead` hier aus dem Paket `plm` kommt.

Wenn wir die Funktion aus `dplyr` verwenden wollen, müssen wir `::` verwenden:

```
dplyr::lead

#> function (x, n = 1L, default = NA, order_by = NULL, ...)
#> {
#>     if (!is.null(order_by)) {
#>         return(with_order(order_by, lead, x, n = n, default = default))
#>     }
#>     if (length(n) != 1 || !is.numeric(n) || n < 0) {
```

```
#>     bad_args("n", "must be a nonnegative integer scalar, ",
#>               "not {friendly_type_of(n)} of length {length(n)}.")
#>   }
#>   if (n == 0)
#>     return(x)
#>   xlen <- vec_size(x)
#>   n <- pmin(n, xlen)
#>   inputs <- vec_cast_common(default = default, x = x)
#>   vec_c(vec_slice(inputs$x, -seq_len(n)), vec_rep(inputs$default,
#>             n))
#> }
#> <bytecode: 0x7fa0cce49460>
#> <environment: namespace:dplyr>
```

Wenn es zu Maskierungen kommt ist es also der Transparenz wegen besser in beiden Fällen `::` zu verwenden, also `plm::lead` und `dplyr::lead`.

**Hinweis:** Alle von Konflikten betroffenen Objekte können mit der Funktion `conflicts()` angezeigt werden.

**Optionale Info:** Um zu überprüfen in welcher Reihenfolge R nach Objekten sucht, kann die Funktion `search()` verwendet werden. Wenn ein Objekt aufgerufen wird schaut R zuerst im ersten Element des Vektors nach, der globalen Umgebung. Wenn das Objekt dort nicht gefunden wird, schaut es im zweiten, etc. Wie man hier auch erkennen kann, werden einige Pakete standardmäßig eingelesen. Wenn ein Objekt nirgends gefunden wird gibt R einen Fehler aus. Im vorliegenden Fall zeigt uns die Funktion, dass R erst im Paket `plm` nach der Funktion `lead()` sucht, und nicht im Paket `dplyr`:

`search()`

```
#> [1] ".GlobalEnv"           "package:plm"        "package:dplyr"
#> [4] "package:data.table"  "package:tufte"      "package:stats"
#> [7] "package:graphics"     "package:grDevices" "package:utils"
#> [10] "package:datasets"    "package:methods"   "Autoloads"
#> [13] "package:base"
```

**Weiterführender Hinweis:** Um das Maskieren besser zu verstehen sollte man sich mit dem Konzept von *namespaces* und *environments* auseinandersetzen. Eine gute Erklärung bietet [Wickham and Bryan \(2019\)](#).

**Weiterführender Hinweis:** Das Paket `conflicted` führt dazu, dass R immer einen Fehler ausgibt wenn nicht eindeutige Objektnamen verwendet werden.

Der besseren Transparenz wegen wird in diesem Buch ab jetzt immer die Notation mit `::` verwendet, auch wenn dies nicht unbedingt nötig wäre. So sehen Sie bei jedem Code-Beispiel unmittelbar aus welchem Paket die verwendeten Funktionen stammen. Lediglich bei den Basispaketen werden wir auf `::` verzichten.

## Chapter 4

# Datenkunde und Datenaufbereitung

In diesem Kapitel geht es um den auf den ersten Blick unspannendsten Teil der Forschung: Datenaufbereitung und -management. Gleichzeitig ist es einer der wichtigsten Schritte: ohne Daten können viele Forschungsfragen nicht angemessen beantwortet werden. Abbildung 4.1 zeigt den typischen Arbeitsablauf eines Forschungsprojekts.

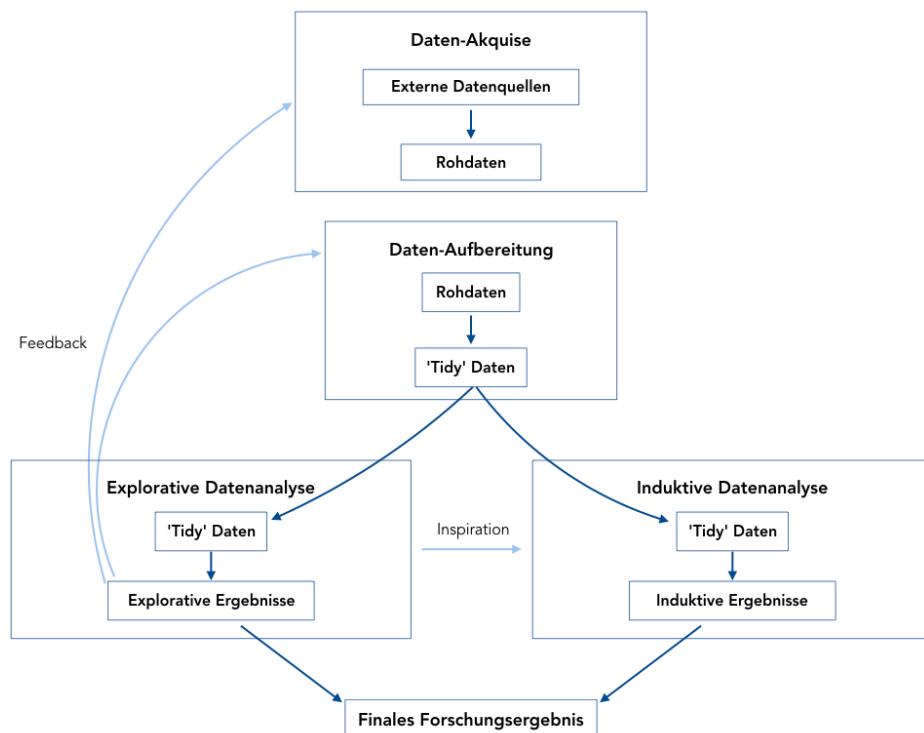


Figure 4.1: Typischer Arbeitsablauf eines Forschungsprojekts

In diesem Kapitel liegt der Fokus auf den ersten beiden Abschnitten, der Akquise und der Aufbereitung Ihrer Daten. Laut [dieser Umfrage](#) verwenden Datenspezialist\*innen regelmäßig 80% ihrer Arbeitszeit auf diese beiden Schritte. Um hier also Zeit und Nerven zu sparen ist es wichtig, sich mit den grundlegenden Arbeitsschritten und Algorithmen vertraut zu machen. Zum Glück ist R sehr gut zur reproduzierbaren Datenaufbereitung geeignet und stellt dank vieler hilfreicher Pakete eine große Hilfe in diesem wichtigen Prozess dar.

Ein zentrales Anliegen dieses Abschnitts liegt darin, Ihnen Methoden zur *reproduzierbaren* und *transparenten* Date-

naufbereitung an die Hand zu geben. Für eine glaubwürdige Forschungsarbeit ist es unerlässlich, dass der Weg von der Datenerhebung hin zum Forschungsergebnis, also der gesamte Prozess in der obigen Abbildung, transparent und nachvollziehbar ist. Daher muss der Datenaufbereitungsprozess gut dokumentiert werden. Dank skriptbasierter Sprachen wie R ist das im Prinzip ein Kinderspiel.

Wenn Sie nämlich alle Arbeitsschritte nach der Datenerhebung in R durchführen, müssen Sie einfach nur Ihre Skripte aufheben - und schon haben Sie die beste Dokumentation, die man sich wünschen kann. Das Wichtigste bei diesem Prozess: Sie dürfen **nie die Rohdaten selbst verändern**.

Alle Änderungen an den Rohdaten müssen durch ein R Skript vorgenommen werden, und die veränderten Daten müssen unter neuem Namen gespeichert werden. Wenn Sie sich das einmal angewöhnt haben, können Sie nicht nur vollkommen transparent in Ihrer Forschung sein, Sie können auch nicht aus Versehen und unwiderruflich Ihre wertvollen Rohdaten zerstören.

Und wenn Sie sich mit den grundlegenden Algorithmen einmal vertraut gemacht haben kann Datenaufbereitung wider Erwarten auch wirklich Spaß machen!

Dieses Kapitel folgt dem typischen Arbeitsablauf eines Forschungsprojektes und beschäftigt sich mit den ersten beiden Abschnitten aus der obigen Grafik, der Daten-Akquise und der Daten-Aufbereitung, wobei Letztere im Mittelpunkt stehen soll. Entsprechend ist das Kapitel folgendermaßen strukturiert:

Als erstes werden wir uns einen Überblick über die verschiedenen [Arten von Daten](#) verschaffen. Danach geht es los mit der [Datenakquise](#). Hier lernen wir, wie man Daten aus häufig verwendeten Datenbanken direkt über R herunterlädt. Als nächstes werden Funktionen zum [Lesen und Schreiben von Datensätzen](#) und typische Herausforderungen in diesem Prozess besprochen. Danach kommt ein sehr umfangreicher Block zum Thema [Datenaufbereitung](#), in dem Sie lernen, wie Sie Ihre Rohdaten in ein Format überführen, das für die statistische Analyse geeignet ist. Zum Abschluss des Kapitels wird noch die [Rolle des Datenmanagements für transparente Forschung](#) verdeutlicht und auf die Debatten über die Ko-Existenz [verschiedener Pakete für die Datenaufbereitung in R](#) hingewiesen.

## Verwendete Pakete

```
library(countrycode)
library(here)
library(WDI)
library(tidyverse)
library(data.table)
library(R.utils)
library(haven)
```

**Hinweis:** In diesem Kapitel verwenden wir für die Arbeit mit Daten vor allem Pakete aus dem sogenannten [tidyverse](#). Ich habe mich für diese Pakete entschieden, weil sie meiner Meinung nach die für R-Beginner am einfachsten zu lernenden Pakete sind und sie zu sehr einfach zu lesendem Code führen. Zudem sind sie sehr weit verbreitet. Es gibt aber auch sehr gute Alternativen und gerade für sehr große Datensätze kommen Sie nicht an dem Paket [data.table](#) vorbei. Die Rolle des [tidyverse](#) und der Debatte um die Pakete in R wird am Ende des Kapitels beschrieben. Bis dahin verweise ich häufig auf weitere Quellen, in denen die Implementierung der Arbeitsschritte in anderen Paketen als dem [tidyverse](#) beschrieben wird.

## 4.1 Arten von Daten

Es gibt verschiedene mehr oder weniger konsistente Klassifizierungen von Daten, die jeweils auf unterschiedliche Aspekte von Daten oder auch Variablen abzielen.

Eine sehr prominente Unterscheidung wird zwischen **quantitativen** und **qualitativen Daten** getroffen. Bei *quantitativen* Daten handelt es sich grob gesagt um *numerische* Daten, also Daten, die Sie in Zahlen ausdrücken können. ‘Größe’, ‘Preis’, ‘BIP’ oder ‘Gehalt’ sind typische Beispiele. *Qualitative* Daten werden intuitiv *nicht-numerisch* ausgedrückt. Häufig handelt es sich um text-basierte oder beschreibende Daten. In der Praxis werden Sie aber merken, dass die Grenze zwischen quantitativen und qualitativen Daten häufig deutlich schwammiger ist, als man das auf den ersten Blick glauben möchte, denn häufig werden qualitative Beschreibungen quantifiziert und dann mit typischen quantitativen Methoden analysiert. Auch werden sogenannte *mixed methods*-Ansätze immer beliebter, welche qualitative und quantitative Methoden kombinieren.

Vor allem in der Psychologie unterscheidet man zwischen **manifesten** und **latenten Variablen**. *Manifeste* Variablen sind direkt beobachtbar und ihre Bedeutung ist häufig klar. Die *Körpergröße* ist z.B. eindeutig messbar und jede\*r weiß was damit gemeint ist.

*Latente* Variablen sind **nicht** direkt beobachtbar und sind häufig erklärungsbedürftig. *Nutzen* ist zum Beispiel nicht beobachtbar.<sup>1</sup> Zudem muss in der Regel erst einmal deutlich gemacht werden, was mit dem Begriff genau gemeint ist.

Ein großer Teil von Forschungsarbeit ist die **Operationalisierung** einer latenten Variable durch eine oder mehrere manifeste Variablen. Wir sprechen dann davon, dass eine oder mehrere manifeste Variablen als Indikator für eine latente Variable verwendet werden. *Wirtschaftliche Entwicklung* z.B. ist als solche nicht direkt beobachtbar und wird häufig durch das BIP operationalisiert.<sup>2</sup>

Der *Human Development Index* ist der Versuch, wirtschaftliche Entwicklung durch mehr als eine manifeste Variable zu operationalisieren, also durch beobachtbare Variablen messbar zu machen. Eine solche Operationalisierung ist natürlich immer kritisch zu hinterfragen und ist nicht selten ein Einfallstor für subjektive und manchmal auch manipulative Wertentscheidungen.

In der Praxis sehr relevant ist zudem die Unterscheidung der **vier Skalenniveaus von Daten**, da die Art der Skala bestimmt, welche Methode angemessen ist um die Daten zu analysieren. Hier wird zwischen **nominal**, **ordinal**, **intervall** und **verhältnis** skalierten Daten unterschieden, wobei intervall- und verhältnisskalierte Daten häufig unter dem Label **kardinalskalierte** Daten zusammen gefasst werden, wie aus der Abbildung 4.2 hervorgeht:

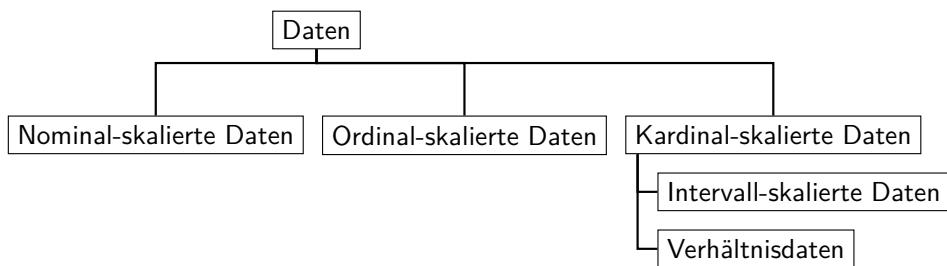


Figure 4.2: Skalenniveaus von Daten

<sup>1</sup>Das klassische Beispiel in der Psychologie ist ‘Intelligenz’.

<sup>2</sup>Interessanterweise hat der maßgeblich an der Entwicklung des Indikators BIP beteiligte Simon Kuznets in *Kuznets (1934)* davon abgeraten, diese Operationalisierung als Indikator für wirtschaftliche Entwicklung zu verwenden. Für mehr Infos dazu siehe z.B. *Lepenies (2016)*.

Wir sprechen von **nominalskalierten** Daten wenn wir den einzelnen Ausprägungen der Daten zwar bestimmte Werte oder eindeutige Beschreibungen zuordnen können, diese aber keine natürliche Rangfolge aufweisen. So können wir einer Person eine Haarfarbe zuordnen, allerdings die verschiedenen Haarfarben in keine natürliche Rangfolge einordnen. Ebenso können wir z.B. Tiere in die Kategorien “Hund”, “Katze” und “Sonstiges” einordnen, aber eine natürliche Rangfolge dieser Kategorien gibt es nicht. Als Konsequenz können wir die einzelnen Ausprägungen zwar zählen, aber sonst keine komplexeren mathematischen Operationen - wie z.B. die Berechnung eines Mittelwerts - ausführen.

In R werden solche Daten in der Regel als **character** oder als **factor** beschrieben. In der Regel ist es einfacher die Daten erst als **character** zu erstellen und dann mit der Funktion **factor()** in Faktoren umzuwandeln:

```
beobachtete_haarfarben <- c("Blond", "Braun", "Schwarz",
                           "Blond", "Braun", "Braun")
```

```
typeof(beobachtete_haarfarben)
```

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

```
beobachtete_haarfarben <- factor(beobachtete_haarfarben)
```

```
beobachtete_haarfarben
```

```
#> [1] Blond  Braun  Schwarz Blond  Braun  Braun
```

```
#> Levels: Blond Braun Schwarz
```

Beachten Sie, dass Faktoren besondere Arten von **integer** sind. Um zu testen, ob eine Variable als Faktor kodiert ist, können Sie die Funktion **is.factor()** verwenden:

```
typeof(beobachtete_haarfarben)
```

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

```
is.factor(beobachtete_haarfarben)
```

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

Die einzelnen Ausprägungen eines Faktors können mit der Funktion **table** gezählt werden. Der häufigste wird dabei ‘Modus’ genannt:

```
table(beobachtete_haarfarben)
```

```
#> beobachtete_haarfarben
```

```
#>   Blond  Braun Schwarz
```

```
#>     2      3      1
```

Mehr Informationen zur Arbeit mit Objekten des Typs **factor** finden Sie in Abschnitt [3.3.9](#).

Bei **ordinalskalierten** Daten können die einzelnen Ausprägungen in eine klare Rangfolge gebracht werden, aber die Abstände sind nicht sinnvoll interpretierbar. Das klassische Beispiel sind Schulnoten: eine ‘1’ ist besser als eine ‘2’, aber weder ist eine 1 ‘doppelt so gut’ wie eine 2, noch sind zwei einser genauso gut wie eine 2.

Ordinalskalierte Daten werden in R am besten auch als **factor** behandelt, allerdings müssen Sie die Reihenfolge explizit spezifizieren indem Sie die Level über das Argument **levels** explizit angeben und dem Argument **ordered** den Wert **TRUE** übergeben:

```
noten <- c(rep(1, 3), rep(2, 4), rep(3, 6), rep(4, 2), rep(5, 3))
noten

#> [1] 1 1 1 2 2 2 3 3 3 3 3 4 4 5 5 5

noten <- factor(noten, levels = 1:6, ordered = T)
noten

#> [1] 1 1 1 2 2 2 3 3 3 3 3 4 4 5 5 5
#> Levels: 1 < 2 < 3 < 4 < 5 < 6
```

Dass der Faktor geordnet ist erkennen wir daran, dass bei der Auflistung der Levels das Symbol < verwendet wird um die Reihenfolge zu illustrieren. Um bei bestehenden Faktoren die Reihenfolge zu spezifizieren, verwenden Sie die Funktion `ordered()` (das und weitere technische Besonderheiten von Objekten des Typs `factor` werden in Abschnitt 3.3.9 genauer beschrieben):

```
noten <- factor(noten, levels = 1:6, ordered = F)
noten

#> [1] 1 1 1 2 2 2 3 3 3 3 3 4 4 5 5 5
#> Levels: 1 2 3 4 5 6

noten <- ordered(noten, levels = 1:6)
noten

#> [1] 1 1 1 2 2 2 3 3 3 3 3 4 4 5 5 5
#> Levels: 1 < 2 < 3 < 4 < 5 < 6
```

Da wir ordinal-skalierte Daten ordnen können, ist es hier z.B. auch möglich empirische Quantile zu berechnen. Allerdings müssen wir bei der Funktion noch das Argument `type=1` oder `type=3`<sup>3</sup> ergänzen, um einen Quantilsalgorithmus zu wählen, der auch mit Faktoren funktioniert:

```
quantile(noten, type = 1)

#> 0% 25% 50% 75% 100%
#> 1    2    3    4    5
#> Levels: 1 < 2 < 3 < 4 < 5 < 6
```

Bei **intervallskalierten** Daten können wir die Ausprägungen nicht nur in eine Rangfolge bringen, sondern auch die Abstände zwischen den Ausprägungen sinnvoll interpretieren. Während es bei Noten also keinen Sinn ergibt, mathematische Operationen wie ‘Addition’ oder ‘Subtraktion’ zu verwenden (und die Abstände entsprechend nicht konsistent zu interpretieren sind), ist dies bei intervallskalierten Daten wie z.B. Jahreszahlen möglich: zwischen 1999 und 2005 liegt der gleiche Abstand wie zwischen 2009 und 2015. Entsprechend werden intervallskalierte Daten in der Regel als `integer` oder `double` gespeichert und wir können Kennzahlen wie den Mittelwert oder die Varianz berechnen.

Allerdings verfügen intervallskalierte Daten über keinen absoluten Nullpunkt, sodass Divisionen und Multiplikationen keinen Sinn machen. Das ist bei **verhältnisskalierten** Daten wie Gewicht, Preis oder Alter anders. Das kann man am besten an folgendem Beispiel illustrieren:

**Beispiel: Inverall- vs. verhältnisskalierte Temperaturen** Wenn wir die Temperatur in Grad Celsius messen haben wir eine Skala ohne absoluten Nullpunkt. Entsprechend können wir nicht sagen,

---

<sup>3</sup>Eine Beschreibung der unterschiedlichen Algorithmen finden Sie über `help(quantile)`.

dass 40 Grad Celsius doppelt so warm sind wie 20 Grad Celsius, nur das der Abstand der Gleiche ist wie zwischen 10 und 30 Grad Celsius. Das wird deutlich, wenn wir uns fragen ob 10 Grad Celsius doppelt so warm wären wie -10 Grad Celsius. Eine Lösung ist die Temperatur in Kelvin anzugeben, denn für Kelvin ist ein absoluter Nullpunkt definiert. Entsprechend können wir auch sagen, dass 20 Kelvin halb so warm ist wie 40 Kelvin - wobei beides ziemlich kalt wäre.

Da sowohl intervall- als auch verhältnisskalierte Daten als `double` oder `integer` repräsentiert werden, ist Vorsicht geboten: wir müssen immer selbst entscheiden welche Maße wir für die Daten berechnen und R gibt uns keinen Fehler aus, wenn wir für zwei intervallskalierte Variablen ein Verhältnis berechnen wollen.

Tabelle 4.1 gibt nochmal einen Überblick über die Skalenniveaus und die dazugehörigen Objekte. Wie oben erwähnt bestimmt das Skalenniveau die anwendbaren statistischen Operationen und Maße. Tabelle 4.2 fasst das für die uns bislang bekannten statistischen Maße kurz zusammen. Alle diese Konzepte werden im Kapitel ?? vertiefter behandelt.

Table 4.1: Skalenniveaus und die dazugehörigen R-Objekte.

Skalenniveau	Beispiel	Messbare Eigenschaften	Typisches R Objekt
Nominal	Haarfarbe,	Häufigkeit	<code>character</code> , <code>factor</code>
	Telefonnummer		
Ordinal	Schulnote,	Häufigkeit, Rangfolge	<code>factor</code>
	Zufriedenheit		
Intervall	Temperatur in C°, Jahreszahl	Häufigkeit, Rangfolge, Abstand	<code>integer</code> , <code>double</code>
	Preise, Alter	Häufigkeit, Rangfolge, Abstand, abs. Nullpunkt	

Table 4.2: Skalenniveaus und die anwendbaren statistischen Operationen.

	Nominal	Ordinal	Intervall	Verhältnis
<b>Modus</b>	✓	✓	✓	✓
<b>Quantile</b>	✗	✓	✓	✓
<b>Interquartilsabstand</b>	✗	✓	✓	✓
<b>Rankkorrelation</b>	✗	✓	✓	✓
<b>Mittelwert</b>	✗	✗	✓	✓
<b>Varianz</b>	✗	✗	✓	✓
<b>Pearson-Korrelation</b>	✗	✗	✓	✓

Wahrscheinlich kennen Sie auch noch die Unterscheidung zwischen **diskreten** und **stetigen** Werten. Diese Kategorisierungen ist nicht vollkommen konsistent mit den Skalenniveaus: zwar sind kardinale Daten in der Tendenz eher stetig und nominale, bzw. ordinale Daten eher diskret, allerdings gibt es auch diskrete kardinale Daten (aber keine stetigen nominalen Daten).

**Hinweis zum Angeben:** Aus der Skalierung oben wird ersichtlich, dass man mit ordinalskalierten Daten keine Durchschnitte bilden darf - man kann sie ja noch nicht einmal addieren. Ein Bereich wo

dieser fundamentalen Regel ständig Gewalt angetan wird ist die Schule: wer hat noch nie von einer Durchschnittsnote gehört? Zum Glück gehört das an der Universität der Vergangenheit an...

## 4.2 Datenakquise

Der erste Schritt in der Arbeit mit Daten ist immer die Akquise der Daten. Je nach verwendeter Methode und Fragestellung ist das mehr oder weniger Arbeit. Im einfachsten Fall sind die von Ihnen benötigten Daten bereits erhoben und über das Internet frei zugänglich. Das trifft z.B. auf viele makroökonomische Indikatoren, wie das BIP, den Gini-Index oder die Arbeitslosigkeit zu. In diesem Falle müssen Sie einfach nur noch die passende Quelle finden,<sup>4</sup> laden die Daten herunter und machen beim nächsten Schritt zum [Einlesen von Datensätzen](#) weiter, oder überlegen ob sie die Daten sogar [direkt mit R herunterladen](#) wollen.

### 4.2.1 Exkurs 1: Ländercodes übersetzen

Gerade wenn Sie mit makroökonomischen Daten arbeiten werden Sie häufig in Kontakt mit Ländercodes kommen. In vielen Datensätzen werden Länder unterschiedlich abgekürzt. So mögen manche Datensätze zwar ausgeschriebene Ländernamen wie "Deutschland" verwenden, andere verwenden aber eher den [iso3c-Code](#) "DEU", während wieder andere den [iso2c-Code](#) "DE" verwenden. Wenn Sie sich dann Daten vom IWF herunterladen wundern Sie sich vielleicht, dass Deutschland dort mit der Zahl 134 kodiert wird.

Zum Glück gibt es ein R-Paket, das die Übersetzung der Codes kinderleicht macht: [countrycode](#) ([Arel-Bundock et al., 2018](#)). Es stellt Ihnen unter anderem die Funktion `countrycode()` zur Verfügung, mit der Sie die Codes einfach übersetzen können. Die Funktion benötigt die folgenden Argumente: `sourcevar` akzeptiert einen `character` oder einen Vektor mit den zu übersetzenden Ländercodes. `origin` gibt die Form dieser Codes an und `destination` spezifiziert den Code in den Sie die `sourcevar` übersetzen wollen. Die Abkürzungen finden Sie in der Hilfefunktion von `countrycode()`.

Nehmen wir einmal an, wir möchten die `iso2c`-Codes für Frankreich und die Schweiz herausfinden. Das geht folgendermaßen:

```
countrycode::countrycode(
  sourcevar = c("Frankreich", "Schweiz"),
  origin = "country.name.de",
  destination = "iso3c")

#> [1] "FRA" "CHE"
```

In diesem Fall verdeutlicht `origin="country.name.de"`, dass wir die Originalnamen auf Deutsch angegeben haben und `destination="iso3c"` dass wir in `iso3c` übersetzen wollen.

Wenn wir wissen wollen welches Land sich hinter der IWF Nummer 112 verbirgt schreiben wir:

```
countrycode::countrycode(
  sourcevar = c("112"),
  origin = "imf",
  destination = "country.name.de")
```

---

<sup>4</sup>Das bedeutet natürlich weder, dass Sie (a) diesen Daten blind vertrauen sollten, noch dass (b) Ihre Daten tatsächlich die [latente Variable](#) messen, an der Sie interessiert sind. Häufig besteht großer Dissens mit welchem Maß welche latente Variable gemessen werden kann. Entsprechend geht der Auswahl der Daten häufig viel Zeit des theoretischen Überlegens voraus. Hier gehen wir davon aus, dass Sie sich über die richtigen Daten schon im Klaren sind.

```
#> [1] "Großbritannien"
```

Die Funktion `countrycode()` kennt bereits alle wichtigen Ländercodes. Schauen Sie in der Hilfefunktion nach wie die Codes abgekürzt werden. Grundsätzlich empfehle ich Ihnen in Ihrer Arbeit möglichst auf das Ausschreiben von Ländernamen zu verzichten und stattdessen mit eindeutigen Kürzeln zu arbeiten. Ich arbeite z.B. immer mit den `iso3c`-Codes, da sie trotz Abkürzung sehr intuitiv lesbar sind.

Das Problem mit ausgeschriebenen Ländernamen lässt sich anhand der Tschechischen Republik gut verdeutlichen. Der `iso3c`-Code ist hier eindeutig `CZE`, allerding verwenden manche Datenbanken den Namen ‘Czechia’ und andere ‘Czech Republik’. Das `countrycode`-Paket übersetzt beide Namen in `CZE`:

```
countrycode::countrycode("Czech Republic", "country.name", "iso3c")
```

```
#> [1] "CZE"
```

```
countrycode::countrycode("Czechia", "country.name", "iso3c")
```

```
#> [1] "CZE"
```

Das kann manchmal zu Problemen beim Zusammenführen von Datensätzen führen, da R nicht von sich aus weiß, dass ‘Czechia’ und ‘Czech Republik’ das gleiche Land bezeichnen. Da die Ländercodes immer eindeutig sind empfehle ich daher immer mit den Kürzeln zu arbeiten und beim ersten Übersetzen immer besonders vorsichtig zu sein.

Das Anwendungsgebiet von `countrycode()` geht übrigens weit über das Übersetzen von Länderkürzeln hinaus: manchmal möchten Sie vielleicht eine ganz andere Übersetzung durchführen. In einem solchen Falle können Sie `countrycode()` über das Argument `custom_dict` auch einen `data.frame` mit dem neuen Code übergeben und die Funktion ansonsten äquivalent nutzen. Das kann z.B. passieren wenn Sie Daten von einer Quelle verwenden, die Länder oder andere Beobachtungen nach einer eigenen Klassifizierung kodiert und diese Kodierung durch eine eigene Tabelle beschreibt. Im folgenden Beispiel ist genau das der Fall: wie haben Daten über die Exporte verschiedener Güterarten. Die Güter sind jedoch nach dem *Harmonized System* der UN kodiert (siehe [hier](#)):

```
head(export_data)
```

```
#>      Good Exports
#> 1:     1   43592
#> 2:     3  234293
#> 3:     5  23842
#> 4:     6 123103
```

Wir können uns aber eine *Korrespondenztabelle* herunterladen, welche die Codes in konkrete Beschreibungen übersetzt. Eine solche Tabelle sieht folgendermaßen aus:

```
head(correspondence_table, 3)
```

```
#>      Code          Description
#> 1:     1 Food and beverages
#> 2:     2 Industrial supplies nes
#> 3:     3 Fuels and lubricants
```

Wir können nun die Funktion `countrycode()` zur Übersetzung der Codes in ihre Beschreibung verwenden (dabei müssen Tabellen in anderen Formaten immer mit `as.data.frame()` in einen Data Frame umgewandelt werden):

```

export_data <- dplyr::mutate(
  export_data,
  Good_Description=countrycode(
    Good, "Code", "Description",
    custom_dict = as.data.frame(correspondence_table)
  )
)
export_data

#>      Good Exports                      Good_Description
#> 1:     1   43592                      Food and beverages
#> 2:     3   234293                     Fuels and lubricants
#> 3:     5  23842 Transport equipment, and parts and accessories thereof
#> 4:     6  123103                    Consumption goods

```

### 4.2.2 Exkurs 2: Daten direkt mit R herunterladen

Manchmal können Sie sich viel Arbeit sparen indem Sie die Daten direkt in R über eine so genannte API herunterladen. Das bedeutet, dass Sie über R einen direkten Zugang zum Server mit den Daten herstellen und die Daten direkt in R einladen. Das hat den Vorteil, dass die Daten in der Regel bereits in einem gut weiterzuverarbeitenden Zustand sind und dass aus Ihrem Code unmittelbar ersichtlich wird wo Ihre Rohdaten herkommen.<sup>5</sup>

Es lohnt sich daher, gerade wenn Sie aus einer Quelle mehrere Daten beziehen wollen, nachzuschauen ob ein R Paket oder eine besondere API verfügbar ist. Im Folgenden möchte ich das Vorgehen mit dem Paket WDI ([Arel-Bundock, 2019](#)) illustrieren, welches Ihnen Zugriff auf die [Weltbankdaten](#) ermöglicht.

Das Paket WDI stellt Funktionen sowohl zum Suchen als auch zum direkten Download von Daten aus der Datenbank der Weltbank zur Verfügung. Diese Datenbank ist extrem nützlich, weil sie makroökonomische Indikatoren für die ganze Welt aus verschiedenen Quellen bündelt.

Als erstes müssen Sie den Code des von Ihnen gewünschten Indikators herausfinden. Dazu gehen Sie am besten auf die [Startseite](#) der Weltbankdatenbank und suchen dort nach den Indikatoren ihrer Wahl. Nehmen wir einmal an, Sie wollen Daten zum Export und zur Arbeitslosigkeit für Deutschland und Österreich für die Jahre 2012-2014 haben.

Sie suchen also nach den Indikatoren und lesen den Code aus der URL des Indikators ab, wie in Abbildung 4.3 dargestellt:<sup>6</sup>

Über die Weltbankseite finden Sie heraus, dass die beiden von Ihnen gesuchten Indikatoren mit NE.EXP.GNFS.ZS und SL.UEM.TOTL.ZS kodiert sind. Nun verwenden Sie die Funktion `WDI::WDI()` um direkt auf die Daten zuzugreifen. Die Funktion benötigt dabei die folgenden Argumente: `country` verlangt nach einem Vektor mit Länderkürzeln. Der `countrycode`-Code für die von der Weltbank geforderten Kürzel ist `wb` und es ist entsprechend einfach diesen Vektor zu erstellen. Das zweite relevante Argument ist `indicator` und benötigt einen Vektor der gewünschten Indikatoren. Über die Argumente `start` und `stop` geben Sie das erste und letzte gewünschte Beobachtungsjahr an. Die weiteren Argumente sind nicht von unmittelbarem Interesse.

<sup>5</sup>Da ein solcher Code nur funktioniert wenn Sie mit dem Internet verbunden sind und Sie die Daten ja nicht jedes Mal von neuem herunterladen wollen macht es Sinn, die Daten nach dem Runterladen abzuspeichern, auch um den konkreten Datensatz, mit dem Sie Ihre Ergebnisse bekommen haben, zu konservieren.

<sup>6</sup>Zwar gibt es im WDI-Paket auch die Funktion `WDI::WDIsearch()`, mit der Sie Datensätze direkt suchen können, allerdings funktioniert das meiner Erfahrung nach nicht optimal.

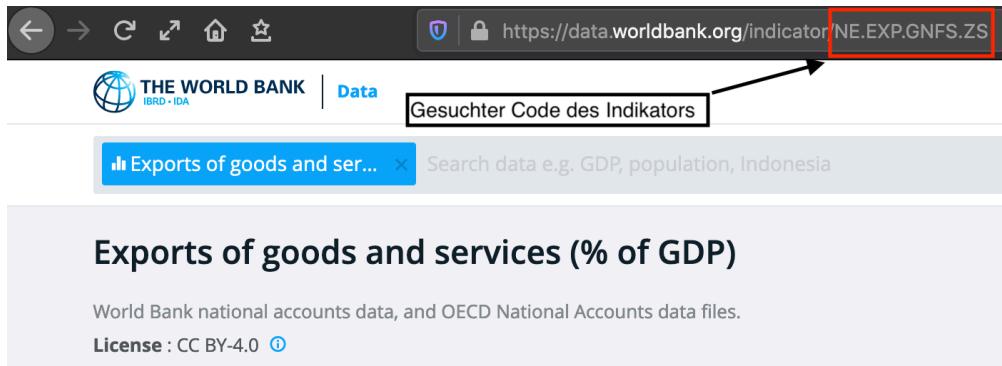


Figure 4.3: Ablesen des Codes aus der URL

Nun können Sie die Funktion `WDI::WDI()` folgendermaßen verwenden um Export- und Arbeitslosendaten für Deutschland und Österreich zwischen 2012 und 2014 zu bekommen:

```
t_beginn <- 2012
t_ende <- 2014
laender <- countrycode::countrycode(c("Germany", "Austria"),
                                      "country.name", "wb")
indikatoren <- c("NE.EXP.GNFS.ZS", "SL.UEM.TOTL.ZS")

daten <- WDI::WDI(
  country = laender,
  indicator = indikatoren
  start = t_beginn,
  end = t_ende
)

daten
```

	iso2c	country	year	NE.EXP.GNFS.ZS	SL.UEM.TOTL.ZS
#> 1:	AT	Austria	2012	53.97368	4.865
#> 2:	AT	Austria	2013	53.44129	5.335
#> 3:	AT	Austria	2014	53.38658	5.620
#> 4:	DE	Germany	2012	45.98254	5.379
#> 5:	DE	Germany	2013	45.39788	5.231
#> 6:	DE	Germany	2014	45.64482	4.981

Mit derlei Paketen können Sie sich häufig viel Zeit sparen, insbesondere wenn Sie mehrere Datensätze von der gleichen Quelle benötigen.

## 4.3 Daten einlesen und schreiben

### 4.3.1 Einlesen von Datensätzen

Wenige Arbeitsschritte können so frustrierend sein wie das Einlesen von Daten. Sie können sich gar nicht vorstellen was hier alles schiefgehen kann! Aber kein Grund zur übertriebenen Sorge: wir können viel Frustration vermeiden wenn wir am Anfang unserer Karriere ausreichend Zeit in die absoluten Grundlagen von Einlesefunktionen

investieren. Also, auch wenn die nächsten Zeilen etwas trocken wirken: sie werden Ihnen später viel Zeit ersparen!

Das am weitesten verbreitete Dateiformat ist csv. ‘csv’ steht für ‘comma separated values’ und diese Dateien sind einfache Textdateien, in denen Spalten mit bestimmten Symbolen, in der Regel einem Komma, getrennt sind. Aufgrund dieser Einfachheit sind diese Dateien auf allen Plattformen und quasi von allen Programmen ohne Probleme lesbar.

In R gibt es verschiedene Möglichkeiten csv-Dateien einzulesen. Die mit Abstand beste Option ist dabei die Funktion `data.table::fread()` aus dem Paket `data.table`, da sie nicht nur sehr flexibel spezifiziert werden kann, sondern auch deutlich schneller als andere Funktionen arbeitet.

Wir gehen im Folgenden davon aus, dass wir die Datei `data/tidy/export_daten.csv` einlesen wollen. Die Datei sieht folgendermaßen aus:

```
iso2c,year,Exporte
AT,2012,53.97
AT,2013,53.44
AT,2014,53.38
```

Es handelt sich also um eine sehr standardmäßige csv-Datei, die wir einfach mit der Funktion `data.table::fread()` einlesen können. Dazu übergeben wir `data.table::fread()` nur das einzige wirklich notwendige Argument: den Dateipfad. Der besseren Übersicht halber sollte dieser immer separat definiert werden:

```
daten_pfad <- here::here("data/tidy/export_daten.csv")
daten <- data.table::fread(daten_pfad)
daten

#>   iso2c year Exporte
#> 1:    AT 2012  53.97
#> 2:    AT 2013  53.44
#> 3:    AT 2014  53.38
```

Vielleicht fragen Sie sich wie `data.table::fread()` die Spalten bezüglich ihres **Datentyps** interpretiert hat? Das können wir folgendermaßen überprüfen:

```
typeof(daten$year)

#> [1] "integer"
```

In der Regel funktioniert die automatische Typerkennung von `data.table::fread()` sehr gut. Ich empfehle dennoch die Typen im Zweifel manuell zu spezifizieren, aus folgenden Gründen: (1) Sie merken leichter wenn es mit einer Spalte ein Problem gibt, z.B. wenn in einer Spalte, die ausschließlich aus Zahlen besteht ein Wort vorkommt. Wenn Sie diese Spalte nicht manuell als `double` spezifizieren würden, würde `data.table::fread()` sie einfach still und heimlich als `character` interpretieren und Sie wundern sich später, warum Sie für die Spalte keinen Durchschnitt berechnen können; (2) Ihr Code wird leichter lesbar; und (3) der Einlesevorgang wird deutlich beschleunigt da `data.table::fread()` die Typen nicht selbst ‘erraten’ muss.

Sie können die Spaltentypen manuell über das Argument `colClasses` einstellen, indem Sie einfach einen Vektor mit den Datentypen angeben:

```
daten_pfad <- here::here("data/tidy/export_daten.csv")
daten <- data.table::fread(daten_pfad,
```

```

    colClasses = c("character", "double", "double"))
typeof(daten$year)

#> [1] "double"

```

Da es bei sehr großen Dateien einen extremen Unterschied macht ob Sie die Spaltentypen angeben oder nicht macht es in einem solchen Fall häufig Sinn, zunächst mal nur die erste Zeile des Datensatzes einzulesen, sich anzuschauen welche Typen die Spalten haben sollten und dann den gesamten Datensatz mit den richtig spezifizierten Spaltentypen einzuladen. Sie können nur die erste Zeile einladen indem Sie das Argument `nrows` verwenden:

```

daten_pfad <- here("data/tidy/export_daten.csv")
daten <- data.table::fread(daten_pfad,
                           colClasses = c("character", "double", "double"),
                           nrows = 1)
daten

#>     iso2c year Exporte
#> 1:     AT 2012   53.97

```

Manchmal möchten Sie auch nur eine bestimmte Auswahl an Spalten einlesen. Auch das kann bei großen Datensätzen viel Zeit sparen. Wenn wir nur das Land und die Anzahl der Exporte haben wollen, spezifizieren wir das über das Argument `select`:

```

daten_pfad <- here::here("data/tidy/export_daten.csv")
daten <- data.table::fread(daten_pfad,
                           colClasses = c("character", "double", "double"),
                           nrows = 1,
                           select = c("iso2c", "Exporte"))
daten

#>     iso2c Exporte
#> 1:     AT   53.97

```

Die Beispiel-Datei oben war sehr angenehm formatiert. Häufig werden aber andere Spalten- und Dezimalkennzeichen verwendet. Gerade in Deutschland ist es verbreitet, Spalten mit ; zu trennen und das Komma als Dezimaltrenner zu verwenden. Unsere Beispiel-Datei oben sähe dann so aus:

```

iso2c;year;Exporte
AT;2012;53,97
AT;2013;53,44
AT;2014;53,38

```

Zum Glück können wir das Spaltentrennzeichen über das Argument `sep` und das Kommatrennzeichen über das Argument `dec` manuell spezifizieren:<sup>7</sup>

```

daten_pfad <- here::here("data/tidy/export_daten_dt.csv")
daten <- data.table::fread(daten_pfad,
                           colClasses = c("character", "double", "double"),
                           sep = ";",
                           dec = ",",
                           header = TRUE)

```

---

<sup>7</sup>Auch hier gilt, dass die automatische Erkennung von `data.table::fread()` schon sehr gut funktioniert, aber die manuelle Eingabe immer sicherer und transparenter ist.

```

        dec = ","
)
daten

#>   iso2c year Exporte
#> 1:   AT 2012  53.97
#> 2:   AT 2013  53.44
#> 3:   AT 2014  53.38

```

`data.table::fread()` verfügt noch über viele weitere Spezifizierungsmöglichkeiten, mit denen Sie sich am besten im konkreten Anwendungsfall vertraut machen. Auch ein Blick in die Hilfeseite ist recht illustrativ. Für die meisten Anwendungsfälle sind Sie jetzt aber gut aufgestellt.

**Anmerkungen zu komprimierten Dateien:** Häufig werden Sie auch komprimierte Dateien einlesen wollen. Gerade komprimierte csv-Dateien kommen häufig vor. In den meisten Fällen können Sie diese Dateien direkt mit `data.table::fread()` einlesen. Falls nicht, können Sie `data.table::fread()` aber auch dem entsprechenden UNIX-Befehl zum Entpacken als Argument `cmd` übergeben, also z.B. `data.table::fread("unzip -p data/gezippte_daten.csv.bz2")`. Weitere Informationen finden Sie sehr einfach im Internet.

Auch wenn csv-Dateien die am weitesten verbreiteten Daten sind: es gibt natürlich noch viele weitere Formate mit denen Sie in Kontakt kommen werden. Hier möchte ich exemplarisch auf drei weitere Formate (`.rds`, `.rdata` und `.dta`) eingehen:

R verfügt über zwei ‘hauseigene’ Formate, die sich extrem gut zum Speichern von größeren Daten eignen, aber eben nur von R geöffnet werden können. Diese Dateien enden mit `.rds`, bzw. mit `.RData` oder `.Rda`, wobei `.Rda` nur eine Abkürzung für `.RData` ist.

Dabei gilt, dass `.rds`-Dateien einzelne R-Objekte enthalten, z.B. einen einzelnen Datensatz, aber auch jedes andere Objekt (Vektor, Liste, etc.) kann als `.rds`-Datei gespeichert werden. Solche Dateien können mit der Funktion `readRDS()` gelesen werden, die als einziges Argument den Dateinamen annimmt:

```

daten_pfad <- here::here("data/tidy/export_daten.rds")
daten <- readRDS(daten_pfad)
daten

```

```

#>   Land Jahr BIP
#> 1  DEU 2011  1
#> 2  DEU 2012  2

```

`.RData`-Dateien können auch mehrere Objekte enthalten. Zudem gibt die entsprechende Funktion `load()` kein Objekt aus, dem Sie einen Namen zuweisen können. Vielmehr behalten die Objekte den Namen, mit dem sie ursprünglich gespeichert wurden. In diesem Fall wurden in der Datei `data/tidy/test_daten.RData` der Datensatz `test_dat` und der Vektor `test_vec` gespeichert. Entsprechend sind sie nach dem Einlesen verfügbar:

```

load(here::here("data/tidy/test_daten.RData"))
test_dat

```

```

#>   a b
#> 1 1 3
#> 2 2 4

```

```
test_vec
```

```
#> [1] "Test Vektor"
```

Die Verwendung von `.RData` ist besonders dann hilfreich, wenn Sie mehrere Objekte speichern wollen und wenn einige dieser Objekte keine Datensätze sind, für die auch andere Formate zur Verfügung stehen.

Ein in der Ökonomik häufig verwendetes Format ist das von der Software **STATA** verwendete Format `.dta`. Um Dateien in diesem Format lesen zu können verwenden Sie die Funktion `read_dta()` aus dem Paket **haven** (Wickham and Miller, 2019), die als einziges Argumente den Dateinamen akzeptiert:

```
dta_datei <- here::here("data/tidy/export_daten.dta")
```

```
dta_daten <- haven::read_dta(dta_datei)
```

```
head(dta_daten, 2)
```

```
#> # A tibble: 2 x 3
#>   iso2c   year Exporte
#>   <chr> <dbl>   <dbl>
#> 1 AT      2012    54.0
#> 2 AT      2013    53.4
```

Das Paket **haven** stellt auch Funktionen zum Lesen von SAS oder SPSS-Dateien bereit.

### 4.3.2 Speichern von Daten

Im Vergleich zum Einlesen von Daten ist das Speichern deutlich einfacher, weil sich die Daten ja bereits in einem vernünftigen Format befinden. Die größte Frage hier ist also: in welchem Dateiformat sollten Sie Ihre Daten speichern?

In der großen Mehrheit der Fälle ist diese Frage klar mit `.csv` zu beantworten. Dieses Format ist einfach zu lesen und absolut plattformkompatibel. Es hat auch nicht die schlechtesten Eigenschaften was Lese- und Schreibgeschwindigkeit angeht, insbesondere wenn man die Daten komprimiert.

Die schnellste und meines Erachtens mit Abstand beste Funktion zum Schreiben von csv-Dateien ist die Funktion `fwrite()` aus dem Paket **data.table**. Angenommen wir haben einen Datensatz `test_data`, den wir im Unterordner `data/tidy` als `test_data.csv` speichern wollen. Das geht mit `data.table::fwrite()` ganz einfach:

```
datei_name <- here::here("data/tidy/test_data.csv")
```

```
data.table::fwrite(test_data, file = datei_name)
```

Neben dem zu schreibenden Objekt als erstem Argument benötigen Sie noch das Argument `file`, welches den Namen und Pfad der zu schreibenden Datei spezifiziert. Der Übersicht halber ist es oft empfehlenswert diesen Pfad zuerst als `character`-Objekt zu speichern und dann an die Funktion `data.table::fwrite()` zu übergeben.

`data.table::fwrite()` akzeptiert noch einige weitere optionale Argumente, die Sie im Großteil der Fälle aber nicht benötigen. Schauen Sie bei Interesse einfach einmal in die Hilfefunktion!

Falls Ihr Datensatz im csv-Format doch zu groß ist, Sie aber aufgrund von Kompatibilitätsanforderungen kein spezialisiertes Format benutzen wollen, bietet es sich an die csv-Datei zu komprimieren. Natürlich könnten Sie das händisch in Ihrem Datei-Explorer machen, aber das ist vollkommen überholt. Sie können das gleich in R miterledigen indem Sie z.B. die Funktion `gzip` aus dem Paket **R.utils** (Bengtsson, 2019) verwenden:

```
csv_datei_name <- here::here("data/tidy/test_data.csv")
data.table::fwrite(test_data, file = csv_datei_name)
R.utils::gzip(csv_datei_name,
  destname=paste0(csv_datei_name, ".gz"),
  overwrite = TRUE, remove=TRUE)
```

Diese Funktion akzeptiert als erstes Argument den Pfad zu der zu komprimierenden Datei, also zweites Argument (`destname`) den Namen, den die komprimierte Datei tragen soll und einige weitere optionale Argumente. Häufig bietet sich `overwrite = TRUE` an, um alte Versionen der komprimierten Datei im Zweifel zu überschreiben, und `remove=TRUE` um die un-komprimierte Datei nach erfolgter Komprimierung zu löschen.

**Hinweise zu verschiedenen zip-Formaten:** Die Funktion `R.utils::gzip()` komprimiert eine Datei mit dem [GNU zip Algorithmus](#). Die resultierende komprimierten Dateien sollten mit der zusätzlichen Endung `.gz` gekennzeichnet werden. `R.utils::gzip()` ist eine relativ schnell arbeitende Funktion, allerdings mit mäßigen Kompressionseigenschaften. Wenn Sie bereit sind längere Arbeitszeit für ein besseres Kompressionsergebnis in Kauf zu nehmen, sollten Sie sich die Funktion `R.utils::bzip2()` ansehen, welche den [bzip2-Algorithmus](#) implementiert. Dieser hat eine deutlich bessere Kompressionsrate (die komprimierten Dateien sind also deutlich kleiner), allerdings ist `R.utils::bzip2()` auch deutlich langsamer als `R.utils::gzip()`. Dateien, die mit `R.utils::bzip2()` komprimiert wurden, sollten mit der Endung `.bz2` gekennzeichnet werden. Entsprechend sieht der Code von oben mit `R.utils::bzip2()` anstatt `R.utils::gzip()` folgendermaßen aus:

```
csv_datei_name <- here::here("data/tidy/test_data.csv")
data.table::fwrite(test_data, csv_datei_name)
R.utils::bzip2(csv_datei_name,
  destname=paste0(csv_datei_name, ".bz2"),
  overwrite = TRUE)
```

Einen Vergleich der Kompressionseigenschaften und Lese- und Schreibgeschwindigkeiten ist immer auch kontextabhängig, im Internet finden sich viele Diskussionen zu dem Thema. Am Anfang sind Sie mit `R.utils::gzip()` und `R.utils::bzip2()` aber eigentlich für alle relevanten Fälle gut aufgestellt.

Die oben bereits vorgestellten R-spezifischen Formate `.Rdata` und `.rds` verfügen über deutliche Geschwindigkeits- und Komprimierungsvorteile gegenüber dem `csv`-Format und sind dabei trotzdem vollkommen plattformkompatibel. Einziger Nachteil: alle Irren, die nicht R benutzen, können Ihre Daten nicht öffnen. Manchmal mag das eine verdiente Strafe, manchmal aber auch ein Ausschlusskriterium sein.

```
saveRDS(object = test_data, file = here("data/tidy/export_daten.rds"))
```

Wie Sie sehen sind zwei Argumente zentral: das erste Argument, `object` spezifiziert das zu speichernde Objekt und `file` den Dateipfad. Darüber hinaus können Sie mit dem optionalen Argument `compress` hier die Kompressionsart auswählen. Ähnlich wie oben gilt, dass `gz` am schnellsten und `bz` am stärksten ist. `xz` liegt in der Mitte.

Wenn Sie mehrere Objekte auf einmal speichern möchten können Sie das über das Format `.RData` machen. Die entsprechende Funktion ist `save()`. Zwar können Sie einfach alle zu speichernden Objekte als die ersten Argumente an die Funktion übergeben, es ist aber übersichtlicher das über das Argument `list` zu erledigen. Der folgende Code speichert die beiden Objekte `test_data` und `daten` in der Datei "data/tidy/datensammlung.Rdata":

```
save(list=c("test_data", "daten"),
  file=here("data/tidy/datensammlung.RData"))
```

Wie `saveRDS()` können Sie bei `save()` über das Argument `compress` den Kompressionsalgorithmus auswählen, allerdings können Sie mit `compression_level` zusätzlich noch die Stärke von 1 (schnell, aber wenig Kompression) bis 9 (langsamer, aber starke Kompression) auswählen.

Da, wie oben erwähnt, gerade in der Ökonomik auch häufig mit der kostenpflichtigen Software **STATA** gearbeitet wird, möchte ich noch kurz erläutern, wie man einen Datensatz im STATA-Format `.dta` speichern kann. Dazu verwenden wir die Funktion `write_dta()` aus dem Paket [haven](#).

```
haven::write_dta(test_data,
  here::here("data/tidy/test_daten.dta"))
```

Für SAS- und SPSS-Daten gibt es ähnliche Funktionen, die ebenfalls durch das [haven](#)-Paket bereitgestellt werden.

**Hinweis:** Gerade bei großen Datensätzen kommt es wirklich sehr auf die Lese- und Schreibgeschwindigkeit von Funktionen an. Auch stellt sich hier die Frage nach dem besten Dateiformat noch einmal viel deutlicher als das bei kleinen Datensätzen der Fall ist und sich die Formatfrage vor allem um das Thema ‘Kompatibilität’ dreht. Einige nette Beiträge, die verschiedene Funktionen und Formate bezüglich ihrer Geschwindigkeit vergleichen finden Sie z.B. [hier](#) oder [hier](#).

## 4.4 Verarbeitung von Daten (‘data wrangling’)

Nachdem Sie Ihre Daten erhoben haben, müssen Sie die Rohdaten in eine Form bringen, mit der Sie sinnvoll weiterarbeiten können. Dieser Prozess wird oft als ‘Datenaufbereitung’ bezeichnet und stellt häufig einen der zeitaufwendigsten Arbeitsschritte in der Forschungsarbeit dar: Laut [dieser Umfrage](#) macht es sogar 60 % der Arbeitszeit von Datenspezialist\*innen aus. Entsprechend wichtig ist es, sich mit den typischen Arbeitsschritten und Algorithmen vertraut zu machen um in diesem aufwendigen Arbeitsschritt Zeit zu sparen.

Ein großes Problem in der Forschungspraxis ist häufig, dass Forscher\*innen den Datenaufbereitungsprozess nicht richtig dokumentieren. In diesem Fall ist unklar was für Änderungen an den Rohdaten vorgenommen wurden bevor die eigentliche Analyse begonnen wurde. Das führt zu unreproduzierbarer und intransparenter Forschung. Daher ist es wichtig, alle Änderungen, die Sie im Rahmen der Datenaufbereitung vornehmen zu dokumentieren.

Am einfachsten ist es, für die Datenaufbereitung einfach ein R-Skript zu schreiben, in dem Sie die Rohdaten einlesen und am Ende die aufbereiteten Daten unter neuem Namen speichern. Am besten legen Sie in Ihrem Ordner `data` zwei Unterordner an: Die Rohdaten speichern Sie dann in einem Unterordner `raw`, die bearbeiteten und aufbereiteten Daten in einem Unterordner `tidy`. So behalten Sie immer den Überblick. **Nie** sollten Sie Ihre Rohdaten überschreiben! Damit sind Sie in Ihrer Forschung vollkommen transparent und es entsteht Ihnen im Prinzip keine Mehrarbeit (siehe Abbildung 4.4).

In diesem Abschnitt lernen Sie Lösungen für Probleme, die typischerweise während der Datenaufbereitung auftreten. Dafür beschäftigen wir uns zunächst mit dem gewünschten Ergebnis: sogenannter `tidy data`. Diese Art von Datensätzen sollte das Ergebnis jeder Datenaufbereitung sein.

Auf dem Weg zu `tidy data` bedarf es häufig einer [Transformation von langen und breiten Datensätzen](#). Außerdem werden Sie häufig mehrere [Datensätze zusammenführen](#) und Ihre [Daten filtern, selektieren und aggregieren](#). Zudem möchten Sie manchmal Daten auch [reduzieren und zusammenfassen](#).

**Beispiel für berühmte Menschen mit miserabler Datenaufbereitung: Der Reinhart-Rogoff Skandal**

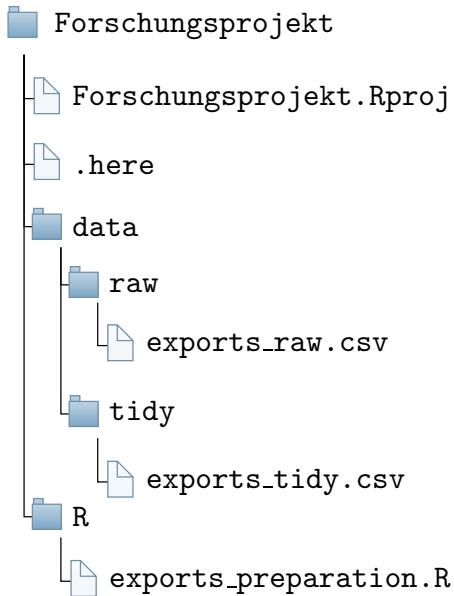


Figure 4.4: Eine übersichtliche Art und Weise Ihre Daten zu speichern. Die Dateien im Unterordner ‘raw‘ werden nie geändert. Die Datei ‘exports-tidy‘ im Ordner ‘tidy‘ wurde mit dem Skript ‘exports-preparation.R‘ aus dem Rohdatensatz ‘exports-raw.csv‘ erstellt.

Eines der dramatischsten Beispiele für Fehler in der Datenaufbereitung mit katastrophalen realweltlichen Implikationen ist der [Reinhart-Rogoff-Skandal](#). Carmen Reinhart und Kenneth Rogoff haben in ihrem einflussreichen Paper [Growth in a Time of Debt](#) einen negativen Effekt von übermäßiger Staatsverschuldung auf wirtschaftliches Wachstum festgestellt. Als der PhD-Student [Thomas Herndon](#) während eines Seminars das Paper replizieren sollte, bekam er Probleme. Dankenswerterweise sendete ihm Carmen Reinhart den Datensatz zu, allerdings stellte sich heraus, dass durch einen Excel-Fehler einige Länder aus der Stichprobe gefallen waren. Mit der kompletten Stichprobe löste sich der im ursprünglichen Paper identifizierte Zusammenhang auf ([Herndon et al., 2013](#)). Das ist besonders dramatisch, da das Paper nicht nur zahlreiche Preise gewonnen hat, sondern auch als wichtige Begründung für die in Europa implementierte Austeritätspolitik fungierte. Klar ist: wäre der Datenaufbereitungsprozess transparent und offen durchgeführt und dokumentiert worden, wäre der Fehler wahrscheinlich deutlich einfacher und früher gefunden worden.

#### 4.4.1 Das Konzept von ‘tidy data’

Die Rohdatensätze, die wir erheben oder aus dem Internet herunterladen haben oft eine abenteuerliche Form und wir können in der Regel nicht direkt mit der statistischen Analyse anfangen. Die meisten Statistik-Pakete und Funktionen setzen eine bestimmte ‘aufgeräumte’ Form der Daten voraus. [Wickham \(2014\)](#) beschreibt diese Form als **tidy data**<sup>8</sup> und es ist unser Ziel durch die Datenaufbereitung die verschiedenen Rohdatensätze in **tidy data** zu verwandeln. Die daraus resultierenden Datensätze können dann separat gespeichert werden, damit wir die Datenaufbereitung nicht jedes Mal erneut durchführen müssen (im Abschnitt [Abschließende Bemerkungen](#) wird ein entsprechender Vorschlag für eine hilfreiche Ordnerstruktur beschrieben).

<sup>8</sup>Wie [hier beschrieben](#) ist das Konzept von ‘tidy data’ nicht neu: Statistiker\*innen sprechen bei einem ‘tidy’ Datensatz häufig von einer ‘Datenmatrix’. Wer sich mehr mit der zugrundeliegenden Theorie beschäftigen möchte sollte zunächst die [12 Regeln von Edgar Codd](#) und ihre Begründung nachlesen.

Aber was zeichnet `tidy data` aus? Wie von [Wickham \(2014\)](#) beschrieben kann ein Datensatz auf vielerlei Art und Weise ‘unordentlich’ sein, aber nur auf eine Art und Weise ‘tidy’. Ein ‘tidy’ Datensatz ist durch folgende drei Eigenschaften gekennzeichnet:

1. Jede **Spalte** korrespondiert zu genau einer **Variable**
2. Jede **Zeile** korrespondiert zu genau einer **Beobachtung**
3. Jede **Zelle** korrespondiert zu einem einzelnen **Wert**

Punkt (1) verlangt, dass jede Spalte zu einer Variable korrespondiert und es keine Spalten gibt, die zu keiner Variable korrespondieren. Wenn wir also Daten zum BIP in verschiedenen Ländern über die Zeit erheben impliziert das, dass wir es mit drei Variablen zu tun haben: dem **Land**, dem **Jahr** und dem **BIP**. Entsprechend sollte unser Datensatz genau drei Spalten haben, die jeweils zu diesen Variablen korrespondieren.

Punkt (2) verlangt, dass jede Zeile zu genau einer Beobachtung korrespondiert. In unserem Beispiel sollte also jede Zeile zu der Beobachtung des BIP in genau einem Land zu genau einem Zeitraum korrespondieren - und z.B. nicht die Beobachtungen für ein einziges Land zu allen möglichen Zeiträumen zusammenln.

Punkt (3) ist meistens in unseren Anwendungsfällen ohnehin erfüllt. Er verlangt, dass jede Zelle in unserem Datensatz genau einen Wert enthält, und z.B. nicht nochmal eine Liste mit mehreren Werten, wie es ja bei einem `data.frame` auch [möglich wäre](#).

**Beispiel ‘tidy data’:** Der folgende Datensatz ist ‘tidy’ im gerade beschriebenen Sinn:

```
#>   Land Jahr  Exporte Arbeitslosigkeit
#> 1: AT 2013 53.44129      5.335
#> 2: AT 2014 53.38658      5.620
#> 3: DE 2013 45.39788      5.231
#> 4: DE 2014 45.64482      4.981
```

Wir haben vier Spalten, die jeweils zu einer der vier Variablen **Land**, **Jahr**, **Exporte** und **Arbeitslosigkeit** korrespondieren. Jede Zeile korrespondiert zur Beobachtung von **BIP** und **Exporte** in genau einem Jahr in genau einem Land. Und die einzelnen Zellen enthalten genau einen Wert, jeweils für das Land, das Jahr, die Exporte und die Arbeitslosigkeit.

**Beispiel: Verstoß gegen (1) :** Der folgende Datensatz, welcher nur Informationen zu den Exporten und für das Jahr 2014 enthält, ist nicht ‘tidy’, da er gegen Anforderung (1) verstößt:

```
#>   Land Variable 2014
#> 1: AT Exporte 53.38658
#> 2: DE Exporte 45.64482
```

Hier haben wir drei Variablen, **Land**, **Jahr** und **Exporte**, aber die Spalte 2013 korrespondiert zu einer Ausprägung der Variable **Jahr**, aber nicht zur Variablen als solchen. Die Bedeutung dieser Unterscheidung wird im nächsten Beispiel deutlich.

**Beispiel: Verstoß gegen (1) und (2):** Wenn wir in dem ersten Datensatz alle Informationen belassen, würde er in der gerade dargestellten Form sowohl gegen (1) als auch (2) verstößen:

```
#>   Land       Variable 2013     2014
#> 1: AT Arbeitslosigkeit 5.33500 5.62000
#> 2: AT           Exporte 53.44129 53.38658
#> 3: DE Arbeitslosigkeit 5.23100 4.98100
```

```
#> 4 DE Exporte 45.39788 45.64482
```

Jetzt ist nicht nur die Anforderung, dass jede Spalte zu einer Variable korrespondiert, verletzt, sondern auch die Anforderung, dass jede Zeile zu genau einer Beobachtung korrespondiert, da wir wegen der zwei Jahre in jeder Zeile zwei Beobachtungen haben. Ebenfalls sehr häufig kommt folgendes Format vor, das ebenfalls (1) und (2) widerspricht:

```
#>   Land Jahr      Variable     Wert
#> 1  AT 2013      Exporte 53.44129
#> 2  AT 2014      Exporte 53.38658
#> 3  DE 2013      Exporte 45.39788
#> 4  DE 2014      Exporte 45.64482
#> 5  AT 2013 Arbeitslosigkeit 5.33500
#> 6  AT 2014 Arbeitslosigkeit 5.62000
#> 7  DE 2013 Arbeitslosigkeit 5.23100
#> 8  DE 2014 Arbeitslosigkeit 4.98100
```

**Beispiel: Verstoß gegen (3)** Verstöße gegen die dritte Anforderung kommen in der Praxis in der Regel seltener vor, sind aber auch unschön:

```
d <- data.frame(Land=c("DE", "AT"))
d$`Wichtige Industrien` <- list(c("Autos", "Medikamente"), c("Stahlproduktion", "Holz"))
d
```

```
#>   Land Wichtige Industrien
#> 1  DE   Autos, Medikamente
#> 2  AT   Stahlproduktion, Holz
```

#### 4.4.2 Von langen und breiten Datensätzen

Die Datenaufbereitung umfasst häufig das Wechseln zwischen der so genannten 'langen' (oder 'gestapelten') und 'breiten' ('ungestapelten') Datenform. Die erste ist für die statistische Verarbeitung, die zweite für das menschliche Auge besser geeignet.

'Lange' Daten haben in der Regel viele Zeilen und wenige Spalten. Alle `tidy` Datensätze sind im langen Datenformat. 'Breite' Daten haben mehr Spalten und weniger Zeilen und sind häufig das, was wir aus dem Internet herunterladen. Im Folgenden ist der gleiche Datensatz einmal im langen und einmal im breiten Format dargestellt.

Zuerst das 'lange' Format, in dem wir verhältnismäßig viele Zeilen haben:

```
#>   Land Jahr  Exporte
#> 1:  AT 2013 53.44129
#> 2:  AT 2014 53.38658
#> 3:  DE 2013 45.39788
#> 4:  DE 2014 45.64482
```

Und hier das 'breite' Format mit verhältnismäßig mehr Spalten:

```
#>   Land Variable 2013 2014
#> 1  AT  Exporte 53.44129 53.38658
#> 2  DE  Exporte 45.39788 45.64482
```

Häufig werden Sie während Ihrer Datenaufbereitung mehrmals zwischen den beiden Formaten hin und her wechseln, da für manche statistischen Zwischenschritte das eine, für andere das andere Format besser ist.<sup>9</sup>

Um zwischen den Formaten hin und herzuwechseln verwenden wir vor allem die Funktionen `pivot_longer()` und `pivot_wider()` aus dem Paket `tidyverse` (Wickham and Henry, 2019), welches auch Teil des `tidyverse` ist.<sup>10</sup>

Wir verwenden `tidyverse::pivot_longer()` um einen Datensatz ‘länger’ zu machen. Wir verwenden dazu folgenden Datensatz als Ausgangsbeispiel, der Werte für die Arbeitslosigkeit in Deutschland und Österreich in zwei Jahren enthält:

```
data_wide
```

```
#>   Land 2013 2014
#> 1 AT 5.335 5.620
#> 2 DE 5.231 4.981
```

Das erste Argument für `tidyverse::pivot_longer()` heißt `data` und nimmt den Datensatz, den wir länger machen wollen. In unserem Beispiel also `data_wide`.

Das zweite Argument heißt `cols` und beschreibt die Spalten an denen Änderungen vorgenommen werden sollen. In unserem Falle sind das die Spalten `2013` und `2014`. Um hier eine Liste von Spaltennamen zu übergeben verwenden wir die Hilfsfunktion `one_of()`, die es uns erlaubt die Spaltennamen als `character` zu schreiben. Das Argument wird also als `cols=one_of("2013", "2014")` spezifiziert.

Das dritte Argument, `names_to` akzeptiert einen `character`, der den Namen der neu zu schaffenden Spalte beschreibt. In unserem Fall macht es Sinn, diese Spalte `Jahr` zu nennen.

Das vierte Argument, `values_to` spezifiziert den Namen der Spalte, welche die Werte des verlängerten Datensatzes beschreibt. In unserem Falle bietet sich der Name `Arbeitslosenquote` an, da es sich bei dem Datensatz um Arbeitslosenquotenstatistiken handelt.

Insgesamt erhalten wir damit den folgenden Funktionsaufruf:

```
data_long <- tidyverse::pivot_longer(data = data_wide,
                                      cols = one_of("2013", "2014"),
                                      names_to = "Jahr",
                                      values_to = "Arbeitslosenquote")
```

```
data_long
```

```
#> # A tibble: 4 x 3
#>   Land Jahr  Arbeitslosenquote
#>   <chr> <chr>      <dbl>
#> 1 AT    2013       5.34
#> 2 AT    2014       5.62
#> 3 DE    2013       5.23
#> 4 DE    2014       4.98
```

<sup>9</sup>Das steht nicht im Widerspruch zu dem oben formulierten Ziel am Ende `tidy` Daten zu haben. Es ist nur so, dass Sie für manche statistischen Transformationen als Zwischenschritt in ein anderes Format wechseln müssen, oder der Weg hin zu `tidy` Daten das Wechseln zwischen langen und breiten Datensätzen erforderlich macht. Das wird durch die Beispiele später in diesem Kapitel praktisch deutlich werden.

<sup>10</sup>Die Funktionen `pivot_longer()` und `pivot_wider()` wurden in der neuesten Version von `tidyverse` eingeführt. Achten Sie also darauf, dass Sie die neueste Version installiert haben. Sie ersetzen die Funktionen `tidyverse::spread()` und `tidyverse::gather()`, die natürlich noch weiterhin funktionieren und die Sie in älterem Code sicher noch häufig finden werden. In diesem Blog-Post beschreibt Chefentwickler Hadley Wickham die neuen Funktionen und grenzt sie von den älteren Implementierungen ab.

Wenn wir den umgekehrten Weg gehen wollen, also einen langen Datensatz 'breiter' machen wollen, verwenden wir die Funktion `tidyr::pivot_wider()`. Hier wird die Anzahl der Zeilen reduziert und die Anzahl der Spalten erhöht. Gehen wir einmal vom gerade produzierten Datensatz aus:

```
data_long
```

```
#> # A tibble: 4 x 3
#>   Land   Jahr  Arbeitslosenquote
#>   <chr> <chr>      <dbl>
#> 1 AT    2013     5.34
#> 2 AT    2014     5.62
#> 3 DE    2013     5.23
#> 4 DE    2014     4.98
```

Die Funktion `tidyr::pivot_wider()` verlangt als erstes Argument wieder `data`, also den zu manipulierenden Datensatz. Im Beispiel ist das `data_long`.

Das zweite Argument, `id_cols`, legt die Spalten fest, die nicht verändert werden sollen, weil sie die Beobachtung als solche spezifizieren. In unserem Fall ist das die Spalte `Land`, aber manchmal ist das auch mehr als eine Spalte. In dem Fall ist die Verwendung der Funktion `one_of()` wie im Beispiel oben notwendig, im Falle von einer Spalte wie hier ist das optional.

Das dritte Argument, `names_from` verlangt nach den Spalten, deren Inhalte im breiten Datensatz als einzelne Spalten aufgeteilt werden sollen. In unserem Falle wäre das die Spalte `Jahr`, weil wir in unserem breiten Datensatz separate Spalten für die einzelnen Jahre haben wollen.

Das vierte Argument ist `values_from` und spezifiziert die Spalte aus der die Werte für die neuen Spalten genommen werden sollen. In unserem Falle wäre das die Spalte `Arbeitslosenquote`, da wir ja in die Spalten für die einzelnen Jahre die Arbeitslosenquoten schreiben wollen.

Insgesamt sieht der Funktionsaufruf also so aus:

```
data_wide_neu <- tidyr::pivot_wider(data = data_long,
                                      id_cols = one_of("Land"),
                                      names_from = "Jahr",
                                      values_from = "Arbeitslosenquote")
```

```
data_wide_neu
```

```
#> # A tibble: 2 x 3
#>   Land   `2013` `2014`
#>   <chr>   <dbl>   <dbl>
#> 1 AT      5.34    5.62
#> 2 DE      5.23    4.98
```

Zum Schluss möchten wir uns noch ein Beispiel ansehen in dem wir beide Befehle nacheinander verwenden. Betrachten wir folgenden Datensatz, der Beobachtungen sowohl zur Arbeitslosenquote also auch zu den Exporten enthält:

```
#> # A tibble: 4 x 5
#>   Land   Variable      `2012` `2013` `2014`
#>   <chr> <chr>       <dbl>   <dbl>   <dbl>
#> 1 AT    Exporte      54.0    53.4    53.4
```

```
#> 2 AT    Arbeitslosigkeit  4.86  5.34  5.62
#> 3 DE    Exporte          46.0   45.4   45.6
#> 4 DE    Arbeitslosigkeit  5.38  5.23  4.98
```

Eine tidy Version dieses Datensatzes sähe so aus:

```
#> # A tibble: 6 x 4
#>   Land   Jahr  Exporte Arbeitslosigkeit
#>   <chr> <chr>   <dbl>           <dbl>
#> 1 AT     2012     54.0            4.86
#> 2 AT     2013     53.4            5.34
#> 3 AT     2014     53.4            5.62
#> 4 DE     2012     46.0            5.38
#> 5 DE     2013     45.4            5.23
#> 6 DE     2014     45.6            4.98
```

Leider ist diese Transformation nicht in einem Schritt zu machen. Als erstes müssen wir nämlich den Datensatz länger machen, indem die Jahre in ihre eigene Spalte gepackt werden, und dann muss der Datensatz breiter gemacht werden indem die Variablen **Exporte** und **Arbeitslosigkeit** ihre eigene Spalte bekommen. Wir machen uns dabei zu Nutze, dass wir dem Argument **cols** auch die Namen der Spalten geben können, die wir *nicht* transformieren wollen. Dazu stellen wir der Liste der Namen ein – voran und R wird entsprechend alle hier nicht genannten Spalten für die Transformation verwenden:

```
data_al_exp_longer <- tidyr::pivot_longer(data = data_al_exp,
                                             cols = -one_of("Land", "Variable"),
                                             names_to = "Jahr",
                                             values_to = "Wert")
head(data_al_exp_longer, 2)
```

```
#> # A tibble: 2 x 4
#>   Land   Variable Jahr   Wert
#>   <chr> <chr>    <chr> <dbl>
#> 1 AT     Exporte  2012    54.0
#> 2 AT     Exporte  2013    53.4
```

Beachten Sie wie wir diesmal das Argument **cols** spezifiziert haben: anstatt alle Jahre in die Funktion **one\_of()** zu schreiben, haben wir stattdessen die Spalten spezifiziert, die *nicht* bearbeitet werden sollen und das mit einem – vor **one\_of()** gekennzeichnet. Das ist vor allem dann hilfreich wenn wir sehr viele Spalten zusammenfassen wollen, was häufig vorkommt, wenn es sich bei den Spalten um Jahre handelt.

Um unser gewünschtes Endergebnis zu erhalten müssen wir diesen Datensatz nun nur noch breiter machen:

```
data_al_exp_tidy <- tidyr::pivot_wider(data = data_al_exp_longer,
                                         id_cols = one_of("Land", "Jahr"),
                                         values_from = "Wert",
                                         names_from = "Variable")
data_al_exp_tidy
```

```
#> # A tibble: 6 x 4
#>   Land   Jahr  Exporte Arbeitslosigkeit
```

```
#>   <chr> <chr>   <dbl>      <dbl>
#> 1 AT    2012    54.0     4.86
#> 2 AT    2013    53.4     5.34
#> 3 AT    2014    53.4     5.62
#> 4 DE    2012    46.0     5.38
#> 5 DE    2013    45.4     5.23
#> 6 DE    2014    45.6     4.98
```

Insgesamt sähe der Code damit folgendermaßen aus:

```
data_al_exp_longer <- tidyverse::pivot_longer(data = data_al_exp,
                                              cols = -one_of("Land", "Variable"),
                                              names_to = "Jahr",
                                              values_to = "Wert")

data_al_exp_tidy <- tidyverse::pivot_wider(data = data_al_exp_longer,
                                             id_cols = one_of("Land", "Jahr"),
                                             values_from = "Wert",
                                             names_from = "Variable")
```

Da die Kombination solcher Schritte in der Praxis sehr häufig vorkommt und man die vielen Zuweisungen der Übersicht halber vermeiden möchte, bieten die Pakete des `tidyverse` eine schöne Möglichkeit, den Code zu verkürzen: die so genannte **Pipe %>%**.

Mit `%>%` geben Sie ein Objekt direkt an die nächste Funktion weiter. Dort wird das Ergebnis des vorherigen Aufrufs automatisch als erstes Argument verwendet. Wir könnten also auch schreiben:

```
data_al_exp_tidy <- data_al_exp %>%
  tidyverse::pivot_longer(
  cols = -one_of("Land", "Variable"),
  names_to = "Jahr",
  values_to = "Wert") %>%
  tidyverse::pivot_wider(
  id_cols = one_of("Land", "Jahr"),
  values_from = "Wert",
  names_from = "Variable")
```

Das ist gleich viel besser lesbar! In der ersten Zeile schreiben wir nur das Ausgangsobjekt `data_al_exp`, welches über `%>%` dann unmittelbar als erstes Argument an `tidyverse::pivot_longer()` übergeben wird. Da es sich beim ersten Argument um `data` handelt ist das genau das was wir wollen.

Das Schreiben mit `%>%` führt in der Regel zu sehr transparentem und nachvollziehbarem Code, da Sie die einzelnen Manipulationsschritte schön von oben nach unten nachlesen können.

**Tipp:** Streng genommen gibt `%>%` den Output der aktuellen Zeile nicht automatisch als erstes Argument für den Funktionsaufruf der nächsten Zeile weiter. Das ist nur das Standardverfahren. Eigentlich gibt es den Output als `.` weiter. Das erlaubt Ihnen den Output auch in einem anderen als dem ersten Argument zu verwenden.

Wir könnten also auch expliziter schreiben:

```
data_al_exp_tidy <- data_al_exp %>%
  tidyverse::pivot_longer(
    data = .,
    cols = -one_of("Land", "Variable"),
    names_to = "Jahr",
    values_to = "Wert") %>%
  tidyverse::pivot_wider(
    data = .,
    id_cols = one_of("Land", "Jahr"),
    values_from = "Wert",
    names_from = "Variable")
```

Das ist hilfreich, wenn Sie den Output einer Zeile nicht als erstes, sondern z.B. als zweites Argument in der nächsten Funktion verwenden wollen. Dann verwenden Sie den `.` einfach explizit da wo Sie ihn brauchen. Da Sie die Argumente ja nicht in der richtigen Reihenfolge angeben müssen solange die Namen stimmen funktioniert also auch folgender Code:

```
data_al_exp_tidy <- data_al_exp %>%
  tidyverse::pivot_longer(
    cols = -one_of("Land", "Variable"),
    names_to = "Jahr",
    values_to = "Wert",
    data = .) %>%
  tidyverse::pivot_wider(
    id_cols = one_of("Land", "Jahr"),
    values_from = "Wert",
    names_from = "Variable",
    data = .)
```

Beide Funktionen, `tidyverse::pivot_wider()` und `tidyverse::pivot_longer()`, können noch viel komplexere Probleme lösen. Für weitere Anwendungen verweisen wir auf die offizielle [Dokumentation](#).

#### 4.4.3 Zusammenführen von Daten

Häufig möchten Sie mehrere Datensätze zusammenführen. Nehmen wir an, Sie hätten einen Datensatz, der Informationen über das BIP in verschiedenen Ländern über die Zeit enthält, und einen zweiten Datensatz, der Informationen über die Einkommensungleichheit in ähnlichen Ländern enthält.

```
#>   Jahr Land BIP
#> 1 2010 DEU 1
#> 2 2011 DEU 2
#> 3 2012 DEU 3
#> 4 2010 AUT 4
#> 5 2011 AUT 5
#> 6 2012 AUT 6

#>   year country Gini
#> 1 2010      DEU      1
```

```
#> 2 2011     DEU     2
#> 3 2012     AUT     3
#> 4 2013     AUT     4
```

Um den Zusammenhang zwischen Einkommensungleichheit und BIP zu untersuchen, möchten Sie die Datensätze zusammenführen, und dabei die Länder und Jahre richtig kombinieren.

Zum Glück hat das Paket `dplyr`, das ein Teil des `tidyverse` darstellt, für jede Situation die passende Funktion parat. Insgesamt gibt es im Paket die folgenden Funktionen, die alle dafür verwendet werden können, zwei Datensätze zusammenzuführen: `inner_join()`, `left_join()`, `right_join()`, `full_join()`, `semi_join()`, `nest_join()` und `anti_join()`.

Wir vergleichen nun das Verhalten der verschiedenen Funktionen mit Hilfe der beiden Beispiel-Datensätze zum BIP und zur Ungleichheit und fassen sie am Ende des Abschnitts nochmals in einer Tabelle (siehe Tabelle 4.3) zusammen.

Wie alle Funktionen der `*_join()`-Familie aus dem `dplyr` Paket verlangt `left_join()` zwei notwendige Argumente, `x` und `y`, welche die beiden zu verbindenden Datensätze spezifizieren. Wir nennen dabei `x` den 'linken' und `y` den 'rechten' Datensatz.

Die Funktion `dplyr::left_join()` sollten Sie verwenden, wenn Sie zu allen Zeilen in `x` (dem 'linken' Datensatz) die passenden Werte aus `y` hinzufügen wollen. Wenn eine Beobachtung nur in `y` vorkommt, wird diese im finalen Datensatz nicht berücksichtigt. Wenn eine Beobachtung nur in `x` vorkommt, wird in den Spalten aus `y` der Wert `NA` eingefügt. Man könnte sagen, der 'linke' Datensatz hat in `dplyr::left_join()` 'Priorität'.

Um zu spezifizieren gemäß welcher Spalten die Datensätze verbunden werden sollen können wir über das optionale Argument `id` die 'ID-Spalten' definieren. Diese Spalten identifizieren eine gemeinsame Beobachtung in `x` und `y`. In unserem Beispiel von oben wären das die Spalten `Jahr` (in `data_BIP`) und `year` (in `data_gini`) sowie `Land` (in `data_BIP`) und `country` (in `data_gini`). Um die Datensätze so zu kombinieren, dass wir die Daten den Ländern und Jahren entsprechend zusammenführen schreiben wir: `by=c("Jahr"="year", "Land"="country")`, also den Spaltennamen in `x` auf die linke Seite von `=` und das Pendant in `y` auf der rechten Seite vom `=`.

Im Falle von `dplyr::left_join()` ergibt sich also:

```
data_BIP_gini_left_join <- dplyr::left_join(data_BIP, data_gini,
                                              by=c("Jahr"="year", "Land"="country"))
data_BIP_gini_left_join
```

```
#>   Jahr Land BIP Gini
#> 1 2010  DEU   1    1
#> 2 2011  DEU   2    2
#> 3 2012  DEU   3   NA
#> 4 2010  AUT   4   NA
#> 5 2011  AUT   5   NA
#> 6 2012  AUT   6    3
```

Verwenden wir dagegen `data_gini` als 'linken' und `data_BIP` als 'rechten' Datensatz gibt `dplyr::left_join()` einen kürzeren gemeinsamen Datensatz aus, da es nur die Beobachtungen aus dem rechten Datensatz übernimmt, für die es ein Pendant im linken Datensatz gibt.

```
data_gini_bip_left_join <- dplyr::left_join(data_gini, data_BIP,
                                             by=c("year"="Jahr", "country"="Land"))
data_gini_bip_left_join
```

```
#>   year country Gini BIP
#> 1 2010    DEU    1    1
#> 2 2011    DEU    2    2
#> 3 2012    AUT    3    6
#> 4 2013    AUT    4    NA
```

Die Funktion `dplyr::inner_join()` unterscheidet sich von `dplyr::left_join()` darin, dass nur die Zeilen in den gemeinsamen Datensatz übernommen werden, die sowohl in `x` als auch in `y` enthalten sind:

```
data_bip_gini_inner_join <- dplyr::inner_join(data_BIP, data_gini,
                                              by=c("Jahr"="year", "Land"="country"))
data_bip_gini_inner_join
```

```
#>   Jahr Land BIP Gini
#> 1 2010  DEU   1    1
#> 2 2011  DEU   2    2
#> 3 2012  AUT   6    3
```

Das Verhalten von `dplyr::right_join()` ist analog zu `dplyr::left_join()`, nur hat hier der ‘rechte’ Datensatz, also der dem Argument `y` übergebene Datensatz Priorität:

```
data_gini_bip_right_join <- dplyr::right_join(data_gini, data_BIP,
                                               by=c("year"="Jahr", "country"="Land"))
data_gini_bip_right_join
```

```
#>   year country Gini BIP
#> 1 2010    DEU    1    1
#> 2 2011    DEU    2    2
#> 3 2012    AUT    3    6
#> 4 2012    DEU    NA   3
#> 5 2010    AUT    NA   4
#> 6 2011    AUT    NA   5
```

Wenn Sie keinem der beiden Datensätze eine Priorität einräumen möchten und alle Zeilen in jedem Fall behalten wollen, dann wählen Sie am besten die Funktion `dplyr::full_join()`:

```
data_bip_gini_full_join <- dplyr::full_join(data_BIP, data_gini,
                                             by=c("Jahr"="year", "Land"="country"))
data_bip_gini_full_join
```

```
#>   Jahr Land BIP Gini
#> 1 2010  DEU   1    1
#> 2 2011  DEU   2    2
#> 3 2012  DEU   3    NA
#> 4 2010  AUT   4    NA
#> 5 2011  AUT   5    NA
```

```
#> 6 2012 AUT 6 3
#> 7 2013 AUT NA 4
```

Die Funktionen `dplyr::semi_join()` und `dplyr::anti_join()` funktionieren ein wenig anders als die bisher vorgestellten Funktionen, da sie Datensätze strikt genommen nicht zusammenführen. Vielmehr filtern sie die Zeilen von `x` gemäß der in `y` vorkommenden Werte.

`dplyr::semi_join()` produziert einen Datensatz, der alle Spalten und Zeilen von `x` enthält, für die es auch in `y` einen entsprechenden Wert gibt. Der resultierende Datensatz enthält aber *nur die Spalten vom linken Datensatz* (`x`). Das kann hilfreich sein, wenn der Datensatz `x` deutlich kleiner ist und Sie in `x` keinerlei fehlende Werte haben wollen:

```
data_bip_gini_semi_join <- dplyr::semi_join(data_BIP, data_gini,
                                              by=c("Jahr"="year", "Land"="country"))
data_bip_gini_semi_join
```

```
#> Jahr Land BIP
#> 1 2010 DEU 1
#> 2 2011 DEU 2
#> 3 2012 AUT 6
```

`dplyr::anti_join()` ist quasi das 'Spiegelbild' zu `dplyr::semi_join()`: genau wie `semi_join()` produziert es einen Datensatz, der nur die Spalten von `x` enthält, und zwar diese, für die es in `y` **keinen** entsprechenden Wert gibt:

```
data_bip_gini_anti_join <- dplyr::anti_join(data_BIP, data_gini,
                                              by=c("Jahr"="year", "Land"="country"))
data_bip_gini_anti_join
```

```
#> Jahr Land BIP
#> 1 2012 DEU 3
#> 2 2010 AUT 4
#> 3 2011 AUT 5
```

Zum Schluss kommen wir mit `dplyr::nest_join()` zu der komplexesten Funktion in der `*_join()`-Familie. Hier wird für jede Zeile im linken Datensatz in einer neuen Spalte ein ganzer `data.frame`<sup>11</sup> hinzugefügt, der alle Zeilen vom rechten Datensatz enthält, die zu der entsprechenden linken Zeile passen:

```
data_bip_gini_nest_join <- dplyr::nest_join(data_BIP, data_gini,
                                              by=c("Jahr"="year", "Land"="country"))
data_bip_gini_nest_join
```

```
#> Jahr Land BIP data_gini
#> 1 2010 DEU 1 1
#> 2 2011 DEU 2 2
#> 3 2012 DEU 3
#> 4 2010 AUT 4
#> 5 2011 AUT 5
#> 6 2012 AUT 6 3
```

---

<sup>11</sup>Eigentlich ein `tibble`.

In der Spalte `y` finden sich nun also sechs Data Frames (einer pro Zeile):

```
data_bip_gini_nest_join[["y"]] # die neue Spalte
```

```
#> NULL
```

Jedes einzelne Element der Spalte `y` ist dabei ein eigener `data.frame`:

```
data_bip_gini_nest_join[["y"]][[1]] # erste Zeile der neuen Spalte
```

```
#> NULL
```

Das bedeutet, dass zur ersten Zeile des Datensatzes `data_BIP` aus dem Datensatz `data_gini` genau eine Spalte passt und diese Spalte den Wert 1 enthält.

In der Praxis werden Sie `nest_join()` wenig verwenden, es ist wegen seiner Flexibilität jedoch für das Programmieren extrem hilfreich.

Wie Sie vielleicht bemerkt haben, haben die Funktionen der `*_join()`-Familie sehr ähnliche Argumente: so verlangen alle `*_join()`-Funktionen als die ersten beiden Argumente `x` und `y` zwei Datensätze, die als `data.frame` oder vergleichbares Objekt vorliegen sollten, so wie `data_BIP` und `data_Gini` in unserem Beispiel.

Das dritte (optionale) Argument `by`, welches die ID-Spalten spezifiziert, ist ebenfalls bei allen Funktionen gleich. Achtung: wenn Sie `by` nicht explizit spezifizieren verwenden die Funktionen alle Spalten mit gleichen Namen als ID-Spalten. Zwar geben sie zur Info eine Warnung aus, aber Sie sollten das trotzdem immer vermeiden und möglichst explizit sein. Daher sollte `by` immer explizit gesetzt werden! Ansonsten erhalten Sie (ohne Warnung) solche ungewünschten Ergebnisse:

```
debt_data_IWF <- data.frame(Land=c("DEU", "GRC"),
                               Schulden=c(10, 50))
debt_data_WELTBANK <- data.frame(Land=c("GRC", "DEU"),
                                   Schulden=c(100, 25))
debt_data <- dplyr::full_join(debt_data_IWF, debt_data_WELTBANK)

#> Joining, by = c("Land", "Schulden")
debt_data

#>   Land Schulden
#> 1  DEU      10
#> 2  GRC      50
#> 3  GRC     100
#> 4  DEU      25
```

Darüber hinaus findet sich das optionale Argument `suffix` sowohl bei `dplyr::inner_join()`, `dplyr::left_join()`, `dplyr::right_join()` als auch `dplyr::full_join()`. Hier spezifizieren Sie eine Zeichenkette, die verwendet wird um Spalten, die in beiden Datensätzen vorkommen, aber keine ID-Spalten sind, im gemeinsamen Datensatz voneinander abzugrenzen. Standardmäßig ist dieses Argument auf `.x`, `.y` eingestellt. Das bedeutet, dass wenn beide Datensätze eine Spalte `Schulden` haben, diese aber nicht als ID-Spalte verwendet wird, beide Spalten als `Schulden.x` und `Schulden.y` in den gemeinsamen Datensatz aufgenommen werden:

```
debt_data_IWF <- data.frame(Land=c("DEU", "GRC"),
                             Schulden=c(10, 50))
```

```
debt_data_WELTBANK <- data.frame(Land=c("GRC", "DEU"),
                                    Schulden=c(100, 25))
debt_data <- dplyr::full_join(debt_data_IWF, debt_data_WELTBANK,
                               by=c("Land"))
debt_data

#>   Land Schulden.x Schulden.y
#> 1  DEU        10        25
#> 2  GRC        50       100
```

Oder wir geben für den finalen Datensatz ein explizites **suffix** an, damit die Variablennamen aussagekräftig werden:

```
debt_data <- dplyr::full_join(debt_data_IWF, debt_data_WELTBANK,
                               by=c("Land"),
                               suffix=c(".IWF", ".WELTBANK"))
debt_data

#>   Land Schulden.IWF Schulden.WELTBANK
#> 1  DEU        10        25
#> 2  GRC        50       100
```

Tabelle 4.3 fasst die gerade diskutierten Funktionen noch einmal zusammen.

Table 4.3: Überblick zu den Funktionen der `*_join()`-Familie des Pakets `dplyr`. 'DS' steht für 'Datensatz', mit `x` ist der linke und mit `y` der rechte Datensatz gemeint, wie in den Argumenten von `*_join()`.

Funktion	Effekt	Veränderung Anzahl Zeilen?
<code>left_join()</code>	<code>x</code> an <code>y</code> anhängen	Unmöglich
<code>right_join()</code>	<code>y</code> an <code>x</code> anhängen	Möglich
<code>inner_join()</code>	In <code>x</code> und <code>y</code> vorhandene Beobachtungen von <code>y</code> und <code>x</code> anhängen	Reduktion möglich
<code>full_join()</code>	<code>x</code> und <code>y</code> kombinieren	Vergrößerung möglich
<code>semi_join()</code>	Reduktion von <code>x</code> auf gemeinsame Beobachtungen	Reduktion möglich
<code>anti_join()</code>	Reduktion von <code>x</code> auf ungeteilte Beobachtungen	Reduktion möglich
<code>nest_join()</code>	Neue Spalte in <code>x</code> mit <code>data.frame</code> , der alle passenden Beobachtungen aus <code>y</code> enthält.	Unmöglich

**Tipp:** das Zusammenführen von Datensätzen ist extrem fehleranfällig. Häufig werden Probleme mit den Rohdaten hier offensichtlich. Daher ist es immer eine gute Idee, den zusammengeführten Datensatz genau zu inspizieren. Zumindest sollte man überprüfen ob die Anzahl an Zeilen so wie erwartet ist und ob durch das Zusammenführen Duplikate entstanden sind. Letzteres kann gerade in der Arbeit mit makroökonomischen Daten häufig vorkommen, wenn in einem Datensatz z.B. zwischen Ost-Deutschland und West-Deutschland Unterschiede bestehen und man vorher die Namen aber in Länderkürzel überführt

hat. In diesem Fall treten um 1990 herum häufig Duplikate auf. Damit kann man umgehen, man muss es aber erst einmal merken. Ich benutze z.B. immer die folgende selbst geschriebene Funktion um zu überprüfen ob es in einem neu generierten Datensatz Duplikate gibt:

```
'# Test uniqueness of data table
#
#' Tests whether a data.table has unique rows.
#
#' @param data_table A data frame or data table of which uniqueness should
#'   be tested.
#' @param index_vars Vector of strings, which specify the columns of
#'   data_table according to which uniqueness should be tested
#'   (e.g. country and year).
#' @return TRUE if data_table is unique, FALSE and a warning if it is not.
#' @import data.table
test_uniqueness <- function(data_table, index_vars, print_pos=TRUE){
  data_table <- data.table::as.data.table(data_table)
  if (nrow(data_table) != data.table::uniqueN(data_table, by = index_vars)){
    warning(paste0("Rows in the data.table: ", nrow(data_table),
                  ", rows in the unique data.table:",
                  data.table::uniqueN(data_table, by = index_vars)))
    return(FALSE)
  } else {
    if (print_pos){
      print(paste0("No duplicates in ", as.list(sys.call() [[2]])))
    }
    return(TRUE)
  }
}
```

Hier ein kleines Anwendungsbeispiel:

```
data_bip_gini_full_join <- dplyr::full_join(data_BIP, data_gini,
                                              by=c("Jahr"="year", "Land"="country"))
test_uniqueness(data_bip_gini_full_join,
                index_vars = c("Jahr", "Land"))

#> [1] "No duplicates in data_bip_gini_full_join"
#> [1] TRUE
```

Die folgende Situation tritt häufiger auf: in den Daten werden für die Wendezeit getrennte Daten für West-Deutschland und das vereinigte Deutschland angegeben, aber die `countrycode` Funktion differenziert nicht zwischen den Namen wenn Sie sie in Ländercodes übersetzen. In der Folge entstehen Duplikate, die beim Zusammenführen der Daten dann offensichtlich werden (können):

```
bip_data

#>   Land Jahr BIP
#> 1  DEU 1989  1
```

```
#> 2 DEU 1990 2
#> 3 DEU 1991 3
gini_data

#>           Land Jahr Gini
#> 1 West Germany 1989    1
#> 2      Germany 1989    2
#> 3 West Germany 1990    3
#> 4      Germany 1990    4
#> 5      Germany 1991    5

gini_data <- dplyr::mutate(gini_data, Land=countrycode(Land, "country.name", "iso3c"))
full_data <- dplyr::full_join(bip_data, gini_data,
                             by=c("Land", "Jahr"))
full_data

#>   Land Jahr BIP Gini
#> 1  DEU 1989  1    1
#> 2  DEU 1989  1    2
#> 3  DEU 1990  2    3
#> 4  DEU 1990  2    4
#> 5  DEU 1991  3    5

test_uniqueness(full_data,
                 index_vars = c("Land", "Jahr"))

#> Warning in test_uniqueness(full_data, index_vars = c("Land", "Jahr")): Rows in
#> the data.table: 5, rows in the unique data.table:3

#> [1] FALSE
```

**Alternative in data.table:** Eine Anleitung für das Zusammenführen von Datensätzen im `data.table`-Format findet sich [hier](#).

#### 4.4.4 Datensätze filtern und selektieren

Sehr häufig haben Sie einen Rohdatensatz erhoben und benötigen für die weitere Analyse nur einen Teil dieses Datensatzes. Zwei Szenarien sind denkbar. Zum einen möchten Sie bestimmte Spalten nicht verwenden. Wir sprechen dann davon den Datensatz zu *selektieren*. Zum anderen möchten Sie bestimmte Zeilen nicht verwenden. Sie wollen nur Beobachtungen verwenden, die eine bestimmte Bedingung erfüllen, z.B. im Zeitraum 2012-2014 erhoben zu sein. In diesem Fall sprechen wir von *filtern*.

Wir lernen hier wie wir diese beiden Aufgaben mit den Funktionen `filter()` und `select()` aus dem Paket `dplyr`, welches auch Teil des `tidyverse` ist, lösen können.

Betrachten wir folgenden Beispieldatensatz:

```
data_al_exp_tidy

#> # A tibble: 6 x 4
#>   Land   Jahr  Exporte Arbeitslosigkeit
```

```
#>   <chr> <chr>    <dbl>      <dbl>
#> 1 AT    2012     54.0      4.86
#> 2 AT    2013     53.4      5.34
#> 3 AT    2014     53.4      5.62
#> 4 DE    2012     46.0      5.38
#> 5 DE    2013     45.4      5.23
#> 6 DE    2014     45.6      4.98
```

Um einzelne Spalten zu selektieren verwenden wir die Funktion `dplyr::select()`. Diese verlangt als erstes Argument den zu manipulierenden Datensatz und danach die Namen oder Indices der Spalten, die behalten oder eliminiert werden sollen. Spalten die behalten werden sollen werden einfach benannt, bei Spalten, die eliminiert werden sollen schreiben Sie ein - vor den Namen:

```
head(
  dplyr::select(data_al_exp_tidy, Land, Exporte),
  2)
```

```
#> # A tibble: 2 x 2
#>   Land  Exporte
#>   <chr>    <dbl>
#> 1 AT       54.0
#> 2 AT       53.4
```

```
head(
  dplyr::select(data_al_exp_tidy, -Exporte),
  2)
```

```
#> # A tibble: 2 x 3
#>   Land  Jahr  Arbeitslosigkeit
#>   <chr> <chr>    <dbl>
#> 1 AT    2012     4.86
#> 2 AT    2013     5.34
```

Häufig ist es besser die Namen der Spalten als `character` zu übergeben. Das ist nicht nur besser lesbar, es wird später auch einfacher komplexe Vorgänge zu programmieren indem Sie Funktionen schreiben, die den Namen von Spalten als Argumente akzeptieren. In diesem Fall können Sie wieder die Hilfsfunktion `one_of()` verwenden:

```
head(
  dplyr::select(data_al_exp_tidy, one_of("Land", "Jahr")),
  2)
```

```
#> # A tibble: 2 x 2
#>   Land  Jahr
#>   <chr> <chr>
#> 1 AT    2012
#> 2 AT    2013
```

```
head(
  dplyr::select(data_al_exp_tidy, -one_of("Land", "Jahr")),
  2)
```

```
#> # A tibble: 2 x 2
#>   Exporte Arbeitslosigkeit
#>   <dbl>          <dbl>
#> 1 54.0           4.86
#> 2 53.4           5.34
```

**Tipp: Spalten auswählen:** Die Funktion `one_of()` erlaubt es Spalten mit sehr nützlichen Hilfsfunktionen auszuwählen. Manchmal möchten Sie z.B. alle Spalten auswählen, die mit `year_` anfangen, oder auf eine Zahl enden. Schauen Sie sich für solche Fälle einmal die `select_helpers` an.

Wie im Abschnitt zu [langen und weiten Daten](#) bereits beschrieben bietet sich in solchen Fällen die Pipe `%>%` an um Ihren Code zu vereinfachen und besser lesbar zu machen. Es hat sich eingebürgert in die erste Zeile immer den Ausgangsdatensatz zu schreiben und `select` dann in der nächsten Zeile mit implizitem ersten Argument zu verwenden:

```
data_al_exp_selected <- data_al_exp_tidy %>%
  dplyr::select(one_of("Land", "Jahr", "Exporte"))
head(data_al_exp_selected, 2)
```

```
#> # A tibble: 2 x 3
#>   Land Jahr Exporte
#>   <chr> <chr> <dbl>
#> 1 AT    2012   54.0
#> 2 AT    2013   53.4
```

Als nächstes wollen wir den Datensatz nach bestimmten Bedingungen filtern. Dabei ist es wichtig, die [logischen Operatoren](#) zu kennen, denn diese werden verwendet um Datensätze zu filtern.

Die Funktion `dplyr::filter()` akzeptiert als erstes Argument den Datensatz. Wie oben folgen wir der Konvention das Argument in der Regel implizit über `%>%` zu übergeben. Danach können wir beliebig viele logische Abfragen, jeweils durch Komma getrennt, an die Funktion übergeben. Wenn wir z.B. nur Beobachtungen für Österreich nach 2012 im Datensatz belassen wollen geht das mit:

```
data_al_exp_filtered <- data_al_exp_tidy %>%
  dplyr::filter(Land == "AT",
                Jahr > 2012)
data_al_exp_filtered
```

```
#> # A tibble: 2 x 4
#>   Land Jahr Exporte Arbeitslosigkeit
#>   <chr> <chr> <dbl>          <dbl>
#> 1 AT    2013   53.4           5.34
#> 2 AT    2014   53.4           5.62
```

Anstatt dem `,`, welches implizit für `&` steht, können wir auch beliebig komplizierte logische Abfragen einbauen. Wenn wir z.B. nur Beobachtungen wollen, die für Österreich im Jahr 2012 oder 2014 und für Deutschland 2013 erhoben wurden und in Deutschland zudem mit einer Arbeitslosigkeit über 5.3 % einhergehen, geht das mit:

```
data_al_exp_filtered <- data_al_exp_tidy %>%
  dplyr::filter(
    (Land == "AT" & Jahr %in% c(2012, 2014)) | (Land=="DE" & Arbeitslosigkeit>5.3)
```

```

)
data_al_exp_filtered

#> # A tibble: 3 x 4
#>   Land Jahr Exporte Arbeitslosigkeit
#>   <chr> <chr>    <dbl>        <dbl>
#> 1 AT   2012     54.0       4.86
#> 2 AT   2014     53.4       5.62
#> 3 DE   2012     46.0       5.38
```

Zuletzt wollen wir noch sehen wie wir einzelne **Spalten umbenennen** können. Das geht ganz einfach mit der Funktion `dplyr::rename()`, welche als erstes Argument den Datensatz, und dann die Umbenennungsvorgänge in der Form `Name_neu = Name_alt` verlangt.

Als Beispiel:

```

data_al_exp_tidy %>%
  dplyr::rename(country=Land,
               year_observation=Jahr,
               exports=Exporte,
               unemployment=Arbeitslosigkeit)
```

```

#> # A tibble: 6 x 4
#>   country year_observation exports unemployment
#>   <chr>    <chr>        <dbl>        <dbl>
#> 1 AT      2012         54.0       4.86
#> 2 AT      2013         53.4       5.34
#> 3 AT      2014         53.4       5.62
#> 4 DE      2012         46.0       5.38
#> 5 DE      2013         45.4       5.23
#> 6 DE      2014         45.6       4.98
```

Als abschließendes Beispiel kombinieren wir die neuen Funktionen und betrachten den Code, mit dem wir

1. aus dem Beispieldatensatz die Spalte zur Arbeitslosigkeit herausselektieren
2. nur die Beobachtungen für Deutschland nach 2012 betrachten und
3. die Spaltennamen dabei noch ins Englische übersetzen:

```

data_al_exp_tidy %>%
  dplyr::select(
    -one_of("Arbeitslosigkeit"))
  ) %>%
  dplyr::filter(
    Jahr>2012,
    Land=="DE"
  ) %>%
  dplyr::rename(
    country=Land,
```

```
year_observation=Jahr,
exports=Exporte)

#> # A tibble: 2 x 3
#>   country year_observation exports
#>   <chr>     <chr>        <dbl>
#> 1 DE        2013         45.4
#> 2 DE        2014         45.6
```

**Alternative Implementierung mit `data.table`:** wie diese Operationen mit dem high-performance Paket `data.table` durchgeführt werden können, wird [hier](#) sehr gut erläutert.

#### 4.4.5 Datensätze zusammenfassen

In diesem letzten Abschnitt werden Sie lernen wie Sie Datensätze erweitern oder zusammenfassen. So können Sie eine neue Variable als eine Kombination bestehender Variablen berechnen oder Ihren Datensatz zusammenfassen, z.B. indem Sie über alle Beobachtungen über die Zeit für einzelne Länder den Mittelwert bilden. Zu diesem Zweck werden wir hier die Funktionen `mutate()`, `summarise()` und `group_by()` aus dem Paket `dplyr` ([Wickham et al., 2019](#)) verwenden.

Wir verwenden `dplyr::mutate()` um bestehende Spalten zu verändern oder neue Spalten zu erstellen. Betrachten wir dafür folgenden Beispieldatensatz:

```
head(unemp_data_wb)

#>   country year laborforce_female workforce_total population_total
#> 1:      AT 2010       46.13933      4276558      8363404
#> 2:      AT 2011       46.33455      4305310      8391643
#> 3:      AT 2012       46.50653      4352701      8429991
#> 4:      AT 2013       46.57752      4394285      8479823
#> 5:      AT 2014       46.70688      4412800      8546356
#> 6:      AT 2015       46.67447      4460833      8642699
```

Angenommen wir möchten das Land mit den `iso3c`-Codes anstatt der `iso2c`-Codes angeben, dann könnten wir mit der Funktion `dplyr::mutate()` die Spalte `country` ganz einfach verändern:

```
unemp_data_wb2 <- unemp_data_wb %>%
  dplyr::mutate(
    country = countrycode::countrycode(country, "iso2c", "iso3c")
  )
head(unemp_data_wb, 2)
```

```
#>   country year laborforce_female workforce_total population_total
#> 1:      AT 2010       46.13933      4276558      8363404
#> 2:      AT 2011       46.33455      4305310      8391643
```

Links neben dem `=` steht der Name der neuen Spalte (wenn der Name bereits existiert wird die existierende Spalte verändert). Rechts neben dem `=` wird die Berechnung der neuen Werte spezifiziert. Wie das Beispiel zeigt, kann hier durchaus der ursprüngliche Wert der Spalte verwendet werden.

Wir können mit `dplyr::mutate()` aber auch einfach neue Spalten erstellen, wenn der Name links vom `=` noch nicht als Spalte im Datensatz existiert.

Wenn Sie nun z.B. wissen möchten, wie viele Frauen absolut in Deutschland und Österreich zur Erwerbsbevölkerung gehören, müssen wir den prozentualen Anteil mit der Anzahl an Erwerbstätigen multiplizieren. Das bedeutet, wir müssen die Spalten `laborforce_female` und `workforce_total` multiplizieren und durch 100 Teilen, da `laborforce_female` in Prozent angegeben ist. Das machen wir mit der Funktion `dplyr::mutate()`, wobei wir eine neue Spalte mit dem Namen `workers_female_total` erstellen wollen:

```
unemp_data_wb <- unemp_data_wb %>%
  dplyr::mutate(
    workers_female_total = laborforce_female*workforce_total/100
  )
```

```
head(unemp_data_wb, 2)
```

	country	year	laborforce_female	workforce_total	population_total
#> 1:	AT	2010	46.13933	4276558	8363404
#> 2:	AT	2011	46.33455	4305310	8391643

	workers_female_total
#> 1:	1973175
#> 2:	1994846

Vielleicht sind wir für unseren Anwendungsfall gar nicht so sehr an der Veränderung über die Zeit interessiert, sondern wollen die durchschnittliche Anzahl an Frauen in der Erwerbsbevölkerung berechnen. Das würde bedeuten, dass wir die Anzahl der Spalten in unserem Datensatz reduzieren - etwas das bei der Anwendung von `dplyr::mutate()` nie passieren würde. Dafür gibt es die Funktion `dplyr::summarise()`:<sup>12</sup>

```
unemp_data_wb_summarized <- unemp_data_wb %>%
  dplyr::summarise(
    fem_workers_avg = mean(workers_female_total)
  )
```

```
unemp_data_wb_summarized
```

```
#>   fem_workers_avg
#> 1      10761223
```

Wie Sie sehen, funktioniert die Syntax quasi äquivalent zu `dplyr::mutate()`, allerdings kondensiert `dplyr::summarise()` den gesamten Datensatz auf die definierte Zahl.

Im gerade berechneten Durchschnitt sind sowohl die Werte für Deutschland als auch Österreich eingegangen. Das erscheint erst einmal irreführend, es wäre wohl besser einen Durchschnittswert jeweils für Deutschland und Österreich getrennt zu berechnen. Das können wir erreichen, indem wir den Datensatz vor der Anwendung von `dplyr::summarise()` **gruppieren**. Das funktioniert mit der Funktion `dplyr::group_by()`, die als Argumente die Spalten, nach denen wir gruppieren wollen, akzeptiert. Sie sollten sich in jedem Fall angewöhnen, nach dem Gruppieren den Datensatz mit `dplyr::ungroup()` wieder in den ursprünglichen Zustand zurückzuführen:

```
unemp_data_wb %>%
  dplyr::group_by(country) %>%
  dplyr::summarise(
```

---

<sup>12</sup>Die Funktionen `dplyr::summarize()` und `dplyr::summarise()` sind Synonyme.

```
fem_workers_avg = mean(workers_female_total)
) %>%
dplyr::ungroup()

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

#> # A tibble: 2 x 2
#>   country fem_workers_avg
#>   <chr>          <dbl>
#> 1 AT            2042685.
#> 2 DE            19479761.
```

Natürlich können Sie `dplyr::group_by()` auch im Zusammenhang mit `dplyr::mutate()` oder anderen Funktionen verwenden. Wie Sie sehen ist der Effekt aber durchaus unterschiedlich:

```
unemp_data_wb %>%
  dplyr::group_by(country) %>%
  dplyr::mutate(
    fem_workers_avg = mean(workers_female_total)
  ) %>%
  dplyr::ungroup()

#> # A tibble: 14 x 7
#>   country year laborforce_fema~ workforce_total population_total
#>   <chr>     <dbl>          <dbl>          <dbl>          <dbl>
#> 1 AT        2010         46.1        4276558        8363404
#> 2 AT        2011         46.3        4305310        8391643
#> 3 AT        2012         46.5        4352701        8429991
#> 4 AT        2013         46.6        4394285        8479823
#> 5 AT        2014         46.7        4412800        8546356
#> 6 AT        2015         46.7        4460833        8642699
#> 7 AT        2016         46.7        4531193        8736668
#> 8 DE        2010         45.6        42014274       81776930
#> 9 DE        2011         45.9        41674901       80274983
#> 10 DE       2012         45.9        41767969       80425823
#> 11 DE       2013         46.1        42161170       80645605
#> 12 DE       2014         46.2        42415215       80982500
#> 13 DE       2015         46.3        42731868       81686611
#> 14 DE       2016         46.4        43182140       82348669
#> # ... with 2 more variables: workers_female_total <dbl>, fem_workers_avg <dbl>
```

Der Datensatz wird nicht verkleinert und keine Spalte geht verloren. Dafür wiederholen sich die Werte in der neu geschaffenen Spalte. Je nach Anwendungsfall ist also die Verwendung von `dplyr::mutate()` oder `dplyr::summarise()` im Zusammenspiel mit `dplyr::group_by()` angemessen.

Im Folgenden werden wir uns noch ein etwas komplexeres Beispiel anschauen: wir werden zunächst die jährliche Veränderung in der absoluten Anzahl der weiblichen Erwerbstätigen in Österreich und Deutschland ermitteln und dann vergleichen ob dieser Wert größer ist als das Bevölkerungswachstum in dieser Zeit. Dazu verwenden wir die

Funktion `dplyr::lag()` um den Wert vor dem aktuellen Wert zu bekommen.<sup>13</sup> Zuletzt wollen wir nur noch die berechneten Spalten im Datensatz behalten.

```
unemp_data_wb_growth <- unemp_data_wb %>%
  dplyr::group_by(country) %>%
  dplyr::mutate(
    pop_growth = population_total - lag(population_total)) / lag(population_total),
    fem_force_growth = (
      workers_female_total - lag(workers_female_total)) / lag(workers_female_total)
  ) %>%
  dplyr::ungroup() %>%
  dplyr::mutate(fem_force_growth_bigger = fem_force_growth > pop_growth) %>%
  dplyr::select(one_of("country", "year", "pop_growth",
    "fem_force_growth", "fem_force_growth_bigger"))
unemp_data_wb_growth
```

```
#> # A tibble: 14 x 5
#>   country year pop_growth fem_force_growth fem_force_growth_bigger
#>   <chr>     <dbl>       <dbl>           <dbl>      <lgl>
#> 1 AT        2010     NA             NA        NA
#> 2 AT        2011     0.00338        0.0110    TRUE
#> 3 AT        2012     0.00457        0.0148    TRUE
#> 4 AT        2013     0.00591        0.0111    TRUE
#> 5 AT        2014     0.00785        0.00700   FALSE
#> 6 AT        2015     0.0113         0.0102    FALSE
#> 7 AT        2016     0.0109         0.0166    TRUE
#> 8 DE        2010     NA             NA        NA
#> 9 DE        2011    -0.0184        -0.00206   TRUE
#> 10 DE       2012     0.00188        0.00311   TRUE
#> 11 DE       2013     0.00273        0.0145    TRUE
#> 12 DE       2014     0.00418        0.00813   TRUE
#> 13 DE       2015     0.00869        0.00993   TRUE
#> 14 DE       2016     0.00810        0.0126    TRUE
```

Besonders hilfreich sind die Versionen von `dplyr::mutate()` und `dplyr::summarize()`, welche mehrere Spalten auf einmal bearbeiten. Ich werde hier nicht im Detail darauf eingehen, sondern nur einen kurzen Einblick in diese Funktionalität geben. Angenommen Sie wollen das durchschnittliche Wachstum in Deutschland und Österreich sowohl für das Bevölkerungswachstum als auch das Wachstum der weiblichen Erwerbsbevölkerung berechnen. Ausgehend vom letzten Datensatz

```
unemp_data_wb_growth_avg <- unemp_data_wb_growth %>%
  dplyr::select(-fem_force_growth_bigger)
head(unemp_data_wb_growth_avg, 2)
```

---

<sup>13</sup>Es gibt neben den Funktionen `dplyr::lag()` und `dplyr::lead()` auch die Funktionen `dplyr::first()` und `dplyr::last()`, die Sie verwenden können um Änderungen über den gesamten Zeitraum zu berechnen. Achten Sie jedoch auf den möglichen Konflikt zwischen den Funktionen `data.table::first()` und `dplyr::first()` sowie `data.table::last()` und `dplyr::last()`!

## 4.5. ABSCHLIEßENDE BEMERKUNGEN ZUM UMGANG MIT DATEN INNERHALB EINES FORSCHUNGSPROJEKTS

```
#> # A tibble: 2 x 4
#>   country year pop_growth fem_force_growth
#>   <chr>     <dbl>      <dbl>            <dbl>
#> 1 AT        2010      NA                 NA
#> 2 AT        2011     0.00338           0.0110
```

geht das folgendermaßen mit der Funktion `dplyr::summarise_all()`:

```
unemp_data_wb_growth_avg %>%
  dplyr::select(-year) %>%
  dplyr::group_by(country) %>%
  dplyr::summarise_all(mean, na.rm=TRUE) %>%
  dplyr::ungroup()
```

```
#> # A tibble: 2 x 3
#>   country pop_growth fem_force_growth
#>   <chr>      <dbl>            <dbl>
#> 1 AT        0.00731          0.0118
#> 2 DE        0.00120          0.00771
```

Eine schöne Übersicht über diese praktischen Funktionen gibt es [hier](#).

Es gibt noch zahlreiche hilfreiche Erweiterungen zu den Funktionen `dplyr::mutate()`, `dplyr::summarize()`, `dplyr::group_by()` und Co. Schauen Sie doch mal auf die Homepage des Pakets `dplyr`. Ansonsten können Sie durch Googlen eigentlich immer eine passgenaue Lösung für Ihr Problem herausfinden - auch wenn es beim ersten Mal häufig ein wenig dauert.

## 4.5 Abschließende Bemerkungen zum Umgang mit Daten innerhalb eines Forschungsprojekts

Das zentrale Leitmotiv dieses Kapitels war die Idee, dass **die Datenaufbereitung vom ersten Schritt an reproduzierbar und transparent** sein sollte. Wenn Sie gefragt werden, wie Ihre Ergebnisse zustande gekommen sind, sollten Sie in der Lage sein, jeden einzelnen Arbeitsschritt seit der ersten Akquise der Daten offenzulegen, bzw. nachvollziehbar zu machen.

Es ist ein zentraler Nachteil von *point-and-click*-Software (wie z.B. SPSS), dass Sie für eine Reproduktion jeden einzelnen Mausklick vor dem Rechner wiederholen, bzw. erklären müssten. Zum Glück ist das mit Skript-basierten Sprachen wie R anders: Sie können einfach ein Skript `Datenaufbereitung.R` anlegen, in welchem Sie die aus dem Internet heruntergeladenen Daten in den für die Analyse aufbereiteten Datensatz umwandeln. Wenn jemand wissen möchte, wo die Daten herkommen, die Sie in Ihrer Analyse verwenden, brauchen Sie der Person nur die Quelle der Daten zu nennen und ihr Skript zu zeigen. So ist es für Sie auch leicht Ihre Analyse mit neuen Daten zu aktualisieren.

Daher hat sich in der Praxis häufig die in Abbildung 4.5 aufgezeigte oder eine ähnliche Ordnerstruktur bewährt:

Der Vorteil an dieser Ordnerstruktur ist, dass Sie die Rohdaten in einem separaten Ordner gespeichert haben und so explizit vom Rest Ihres Workflows abgrenzen. Denn: **Rohdaten sollten nie bearbeitet werden**. Zu leicht gerät in Vergessenheit welche Änderungen tatsächlich vorgenommen wurden und Ihre Forschung wird dadurch nicht mehr replizierbar - weder für Sie noch für andere. Alle weiteren Änderungen an den Rohdaten sollten über ein

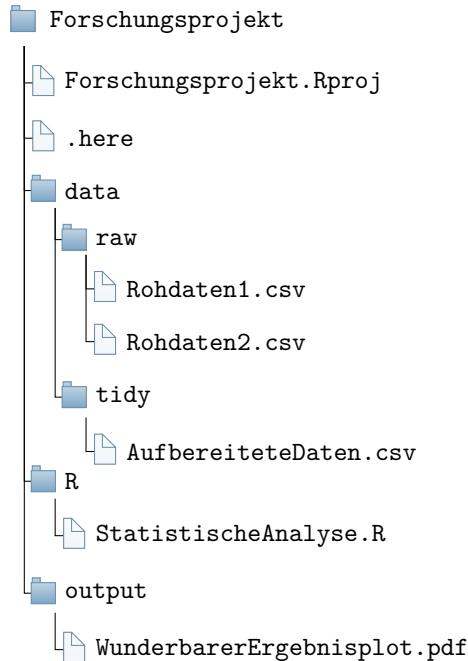


Figure 4.5: Bewährte Ordnerstruktur für Forschungsprojekte mit R.

Skript vorgenommen werden, sodass immer klar ist wie Sie von den Rohdaten zu den Analysedaten kommen.

Diese bearbeiteten Daten können in einem zweiten Unterordner (hier: `tidy`) gespeichert werden, damit Sie für Ihre Analyse nicht immer die Daten neu aufbereiten müssen. Gerade bei großen Datensätzen kann das nämlich sehr lange dauern. Wichtig ist aber, dass die Daten in `tidy` immer mit Hilfe eines Skripts aus den Daten in `raw` wiederhergestellt werden können.

In der Praxis würden Sie also aus den Daten in `raw`, die entweder direkt aus dem Internet geladen wurden oder direkt aus einem Experiment hervorgegangen sind, per Skript `Datenaufbereitung.R` den Datensatz `AufbereiteteDaten.csv` erstellen. Dabei können auch mehrere Rohdatensätze zusammengeführt werden. Dieser Datensatz kann dann in der weiteren Analyse verwendet werden, z.B. im Skript `StatistischeAnalyse.R`, das dann einen Output in Form einer Datei `WunderbarerErgebnisplot.pdf` produziert.

Der Vorteil: wenn jemand genau wissen möchte, wie `WunderbarerErgebnisplot.pdf` produziert wurde können Sie sämtliche Schritte ausgehend von den vollkommen unangetasteten Rohdaten transparent machen. Durch die Trennung unterschiedlicher Arbeitsschritte - wie Datenaufbereitung und statistische Analyse - bleibt Ihr Projekt zudem übersichtlich.

## 4.6 Anmerkungen zu Paketen

In diesem Kapitel wurden gleich mehrere Pakete aus dem `tidyverse`, einer Sammlung von Paketen, verwendet. Zwar schätze ich das `tidyverse` sehr, gleichzeitig ist der Fokus von R Studio auf diese Pakete zumindest potenziell problematisch. Dies wird in diesem [kritischen Blogpost](#) sehr schön beschrieben.

Was die Einsteigerfreundlichkeit vom `tidyverse` angeht, bin ich jedoch anderer Meinung als der Verfasser des Blogposts: meiner Meinung nach machen diese Pakete die Arbeit mit Datensätzen sehr einfach, und für kleine

Datensätze (<500MB) benutze ich das `tidyverse` auch in meiner eigenen Forschung. Es sollte jedoch klar sein, dass es nur eine Option unter mehreren ist, weswegen ich versuche in meinen Paketen vollständig auf das `tidyverse` zu verzichten - auch weil es in puncto Performance deutlich schlechter ist als z.B. `data.table` (Dowle and Srinivasan, 2019), das auch für mehrere hundert GB große Datensätze gut geeignet ist. Zur Aneignung des Pakets `data.table` (Dowle and Srinivasan, 2019) ist das [offizielle Tutorial](#) gut geeignet, macht m.E. aber auch deutlich, dass es für die ersten Schritte mit R etwas unintuitiver ist als das `tidyverse`.

Wenn Sie später einmal beide Ansätze beherrschen, können Sie das tun, was in einer diversen Sprache wie R das einzig Richtige ist: je nach Anwendungsfall das passende Paket wählen - ganz wie im Falle von Paradigmen in einer Pluralen Ökonomik.



# Chapter 5

## Visualisierung von Daten

### Verwendete Pakete

```
library(here)
library(tidyverse)
library(data.table)
library(ggpubr)
library(ggrepel)
library(scales)
library(tufte)
library(gapminder)
library(viridis)
library(latex2exp)
library(WDI)
library(countrycode)
```

### Einleitung

In diesem Kapitel lernen Sie mit Hilfe des Pakets `ggplot2` Ihre Daten ansprechend zu visualisieren.

Der erste Abschnitt ist dabei optional und beschäftigt sich mit den theoretischen Grundlagen von `ggplot2`. Hier diskutieren wir die Abgrenzung zwischen dem `ggplot2`- und `base`-Ansatz zur Datenvisualisierung in R und führen mit der in [Wickham \(2010\)](#) entwickelten [Grammatik für Grafiken](#) das theoretische Fundament für `ggplot2` ein. Diese beiden Abschnitte sind recht abstrakt, aber helfen Ihnen die interne Logik von `ggplot2` besser zu verstehen.

Im zweiten Abschnitt werden die grundlegenden [Elemente einer Grafik](#) in `ggplot2` beschrieben und eine erste Beispielgrafik Stück für Stück erstellt. Der dritte Abschnitt erläutert anhand von Beispielen wie die [gängigsten Visualisierungsarten](#) in `ggplot2` erstellt werden können.

Danach werden ausgewählte [fortgeschrittene Techniken](#), wie z.B. die Visualisierung von Regressionsergebnissen oder das Erstellen von Plots mit mehreren Abbildungen, eingeführt. Dabei greifen wir an einigen optionalen Stellen natürlich auf das Kapitel zur linearen Regression vor (siehe Kapitel ??), allerdings sollten die Inhalte hier auch ohne tieferen Kenntnisse der Regressionsanalyse verständlich sein.

Im fünften Abschnitt zeigen wir aufbauend auf [Schwabish \(2014\)](#) wie [typische Fehler](#) in der Datenvisualisierung vermieden werden können. Der sechste Abschnitt illustriert ausgewählte [Manipulationsstrategien](#) bei der Datenvisualisierung. Im letzten Abschnitt finden Sie Empfehlungen für [weiterführende Literatur](#).

## 5.1 Optional: Theoretische Grundlagen

### 5.1.1 ggplot2 vs. base plot

Wie so oft bietet R verschiedene Ansätze zur Datenvisualisierung. Die beiden prominentesten sind dabei die in der Basisversion von R integrierten Visualisierungsfunktionen, häufig als `base` bezeichnet, und das Paket `ggplot2` ([Wickham, 2016](#)).

Die Frage ‘Welcher Ansatz ist nun besser?’ ist nicht leicht zu beantworten, insbesondere da beiden Ansätzen eine sehr unterschiedliche Design-Philosophie zugrunde liegt: `base` funktioniert dabei wie ein Stift und ein Blatt Papier: Sie haben ein leeres Blatt, welches Sie mit dem Aufruf bestimmter `plot`-Funktionen beschreiben. Hierbei wird kein besonderes R-Objekt erstellt, in dem die Grafik gespeichert wird - vielmehr speichern Sie am Ende ihr ‘vollgemaltes Blatt’ entweder als Bild ab, oder Sie verwerfen es und beschreiben ein neues ‘Blatt’.

In `ggplot2` werden die Grafiken dagegen ‘scheibchenweise’ in einer Art Liste zusammengesetzt. Diese Liste enthält dann eine vollständige Beschreibung der Grafik im Sinne einer geschichteten [Grammatik für Grafiken](#). Dabei findet kein ‘Malprozess’ statt: die finale Grafik wird erst dann erstellt wenn auf die resultierende Liste eine `print`-Funktion angewandt wird.

Am Ende des Tages werden Sie wenige Dinge finden, die Sie nur mit `base` oder nur mit `ggplot2` erreichen können. Und wahrscheinlich gilt für die meisten, dass sie einfach bei dem Ansatz hängen bleiben, der Ihnen am Anfang intuitiv am besten gefallen hat. Ich habe in der [weiterführenden Literatur](#) einige Diskussionsbeiträge zum Thema `base` vs. `ggplot2` gesammelt und fasse mich hier daher kurz: in dieser Einführung verwenden wir `ggplot2`. Ich finde, dass die resultierenden Grafiken einen Tick schöner, die Syntax ein wenig einfacher und die Dokumentation im Internet ein wenig besser ist. Vor allem finde ich den Code leichter lesbar und den von [Wickham \(2010\)](#) vorgeschlagenen *Grammar of Graphics* Ansatz sehr intuitiv.

Wenn Sie dagegen lieber mit `base` arbeiten wollen - kein Problem. Es finden sich im Internet gerade auf Englisch viele exzellente Einführungen. Und im Endeffekt ist die einzige relevante Frage: haben Sie auf eine für Sie möglichst unterhaltsame Art und Weise einen guten Graphen produziert? Welches Paket Sie dafür verwendet haben, interessiert schlussendlich niemanden...

### 5.1.2 Einleitung zu Wickham’s *Grammar of Graphics*

Die Funktion von `ggplot2` ist leichter nachzuvollziehen wenn man weiß wodurch das Paket inspiriert wurde. In diesem Fall war es das Konzept der *Grammar of Graphics* ([Wilkinson, 1999](#)), beziehungsweise die Interpretation des Konzepts von [Wickham \(2010\)](#).

Dieses Konzept startet mit dem Wunsch eine ‘Grammatik’ für Grafiken zu entwickeln. Eine Grammatik wird hier als eine Sammlung von Konzepten verstanden aus denen sämtliche Grafiken hergestellt werden können- eine vollständige Beschreibung der Grafik sozusagen. So wie die Grammatik der deutschen Sprache eine Sammlung von Wörtern und Regeln darstellt, aus denen jede Menge (mehr oder weniger sinnvolle) Aussagen hergestellt werden können, verstehen wir unter einer Grammatik für Grafiken eine Sammlung von Konzepten und Regeln aus denen wir jede Menge (mehr oder weniger sinnvolle) Grafiken herstellen können.

Im Gegensatz zu der ursprünglich von [Wilkinson \(1999\)](#) vorgestellten Grammatik folgt die Grammatik von [Wickham \(2010\)](#) einer klar geordneten Struktur: jeder Teil der Grammatik ist unabhängig vom Rest, und eine Grafik wird vollends dadurch spezifiziert, dass die einzelnen Teile Stück für Stück zusammen geführt werden.

Nach Wickham's Grammatik besteht jede statistische Grafik aus den folgenden Komponenten:

1. Einem **Standard-Datensatz** gemeinsam mit den Funktionen (*engl.: mappings*), die bestimmten Variablen aus dem Datensatz eine so genannten **Ästhetik** (*engl.: aesthetic*) zuweisen. Die sogenannten *mappings* (es handelt sich dabei eigentlich um einfache Funktionen) verlinken eine Variable in den Daten mit einer Ästhetik in der Grafik. Beispielsweise könnten wir die Variable 'Jahr' in den Daten mit der Ästhetik 'x-Achse', die Variable 'BIP' mit der Ästhetik 'y-Achse' und die Variable 'Land' mit der Ästhetik 'Farbe' verlinken.
2. Eine oder mehrere **Ebenen**; jede Ebene besteht dabei aus einem geometrischen Objekt, einer statistischen Transformation, einer Positionszuweisung und, optionalerweise, einem von (1) abweichenden besonderen Datensatz und den entsprechenden *aesthetic mappings*.
- Von besonderer Relevanz sind dabei die geometrischen Objekte, **geoms**, denn sie bestimmen um was für einen Plot es sich handelt: verwenden wir als **geoms** Punkte bekommen wir ein Streudiagramm, bei Linien als **geoms** wird es ein Linienplot, usw. Die **geoms** visualisieren also die Ästhetiken, aber bestimmte **geoms** können natürlich nur bestimmte Ästhetiken repräsentieren: der **geom** 'Punkt' z.B. hat eine x und eine y-Komponente (also eine **Position**), eine **Größe**, eine **Form** und eine **Farbe**. Andere Ästhetiken ergeben für Punkte keinen Sinn.
- Da wir nicht notwendigerweise die exakten Werte der Variable an die Ästhetik weitergeben, wird die Möglichkeit einer *statistischen Transformation* offen gelassen: eventuell wird nicht der Variablenwert, sondern z.B. der Logarithmus dieses Wertes an die entsprechende Ästhetik weitergegeben. Natürlich kann die statistische Transformation auch weggelassen werden - in diesem Fall sprechen wir von der Transformation **identity** - die Daten werden nicht verändert, sondern direkt an die Ästhetik weitergegeben. Andere häufig verwendete Transformationen sind **boxplot** (wenn wir die Daten in einem Boxplot zusammenfassen wollen), **bin** (wenn wir die Daten in einem diskreten Histogramm darstellen wollen) oder **density** (wenn wir an der Wahrscheinlichkeitsdichte der Beobachtungen interessiert sind).
- Die Positionszuweisungen spielen nur eine Rolle wenn die Positionen der **geoms** angepasst werden müssen, z.B. um Überlappungen zu vermeiden. Ein typisches Beispiel ist auch das Schachteln von Balkendiagrammen.
3. Einer **Skala** für jedes *aesthetic mapping*. Sie beschreibt die genaue Art des Mappings zwischen Daten und Ästhetiken. Entsprechend handelt es sich bei einer Skala in diesem Sinne hier um eine **Funktion gemeinsam mit Parametern**. Am besten kann man sich das bei einer farblichen Skala vorstellen, die bestimmte Werte in einen Farbenraum abbildet.
4. Einem **Koordinatensystem**, welches zu den Daten und Ästhetiken und geometrischen Objekten passt. Am häufigsten wird hier sicher das kartesische Koordinatensystem verwendet, aber für Kuchendiagramme bietet sich z.B. das polare Koordinatensystem an.
5. Eine optionale **Facettenspezifikation** (*engl.: facet specification*), die verwendet werden kann um die Daten in verschiedene Teil-Datensätze aufzusplitten. So möchten wir vielleicht die Dynamik des BIP über die Zeit abbilden, aber einen separaten Unter-Plot für jedes einzelne Land erstellen. In diesem Fall verwenden wir eine Facettenspezifikation, die für jedes Land einen Teildatensatz erstellt.

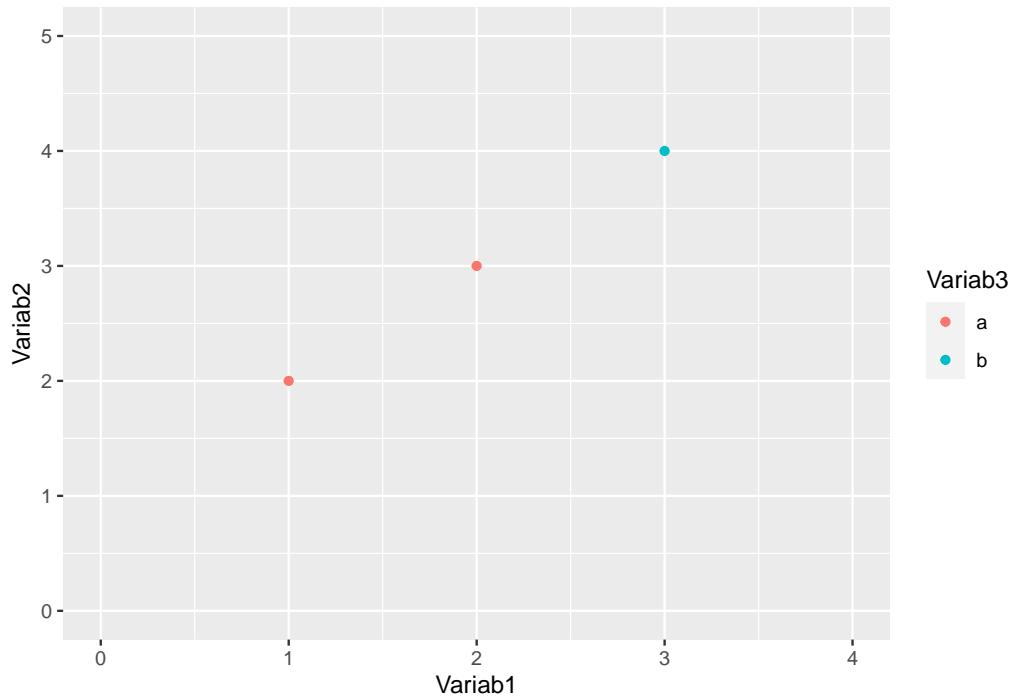
Alle Komponenten bleiben dabei unabhängig voneinander: die Daten z.B. sind unabhängig vom Rest, weil die gleiche Grafik für unterschiedliche Daten produziert werden kann: "Daten machen aus einer abstrakten Grafik eine konkrete Grafik" ([Wickham, 2010](#), p. 10).

Das Besondere an der so formulierten Grammatik ist, dass man mit den Komponenten 1 - 5 so ziemlich jede statistische Grafik beschreiben kann. Das Paket `ggplot2` macht sich das zunutze: es formalisiert diese Regeln in R, sodass Sie mit dem entsprechenden R Code quasi jede Grafik beschreiben können - und dann durch R erstellen lassen können. Dadurch ist auch die Vorgehensweise motiviert, zunächst ein Objekt mit der *Beschreibung* der Grafik zu erstellen und die Grafik dann am Ende durch Anwendung einer `print`-Funktion auf diese Beschreibung herzustellen. So können Sie das Objekt mit der Beschreibung vorher bereits speichern und weitergeben und dann zu einem späteren Zeitpunkt erst die eigentliche Grafik erstellen. Dieses Vorgehen machen wir uns später zunutze wenn wir mehrere Sub-Abbildungen in einer großen Grafik [gemeinsam abbilden wollen](#).

Wie Sie später sehen werden repräsentiert die Syntax von `ggplot2` genau diese theoretische Beschreibung von Grafiken. Hier greifen wir mit einem kleinen Beispiel vor:

```
example_data <- data.frame(
  Variab1=1:3,
  Variab2=2:4,
  Variab3=c("a", "a", "b")
)

ggplot2::ggplot(
  data = example_data,
  mapping = aes(x=Variab1,
                 y=Variab2,
                 color=Variab3)
) +
  ggplot2::layer(
    geom = "point",
    stat = "identity",
    position = "identity") +
  ggplot2::scale_color_discrete(
    aesthetics = c("color"))
) +
  ggplot2::coord_cartesian(
    xlim = c(0, 4),
    ylim = c(0, 5)
)
```



Die Funktion `ggplot2::ggplot()` erstellt eine Liste, in der die Grafik-Spezifikationen gespeichert werden und akzeptiert über die Argumente `data` und `mapping` die Standard-Daten und Standard-Mappings.<sup>1</sup> Es korrespondiert damit zu Punkt (1) oben.

Als nächstes wird mit `ggplot2::layer()` eine neue Ebene spezifiziert. Wie in der Theorie spezifizieren wir die Ebene über das Argument `geom` bezüglich der auf ihr abzubildenden geometrischen Objekte (hier: Punkte), über `stat` bezüglich der zu verwendeten statistischen Transformation (hier: keine Transformation, sondern die Daten identisch zu ihren Werten im Standard-Datensatz) und über `position` bezüglich der Positionszuweisungen (auch hier: keine besonderen Positionszuweisungen).

Als nächstes spezifizieren wir die Skala. Für die Ästhetik ‘Position’ der Variablen `Variab1` und `Variab2` ist keine Übersetzung notwendig, aber für den Link zwischen den Werten von Variable `Variab3` und der Ästhetik ‘Farbe’ müssen wir eine explizite Funktion verwenden. Mit der Funktion `ggplot2::scale_color_discrete()` weisen wir also jedem Wert der (diskreten) Variable `Variab3` eine Farbe zu.

Schließlich legen wir mit `ggplot2::coord_cartesian()` noch das zu verwendende Koordinatensystem fest indem wir mit den Argumenten `xlim` und `ylim` die Länge der x- und y-Achse spezifizieren. Eine besondere Facettenspezifikation verwenden wir hier dagegen nicht.

Wie Sie später sehen werden, verwenden wir in `ggplot2` häufig Abkürzungen für die in diesem Beispiel verwendeten ‘Originalfunktionen’. So gibt es für eine Ebene mit dem `geom` ‘Punkte’ die Abkürzung `ggplot2::geom_point()`. Auch muss nicht jedes Element explizit spezifiziert werden: da z.B. die meisten Grafiken ein kartesisches Koordinatensystem verwenden, ist dies als Standard-Koordinatensystem in `ggplot2` implementiert und Sie müssen nur dann explizit ein Koordinatensystem spezifizieren wenn Sie vom Standardwert abweichen wollen.

Wenn Sie sich genauer mit der hierarchischen Grammatik beschäftigen wollen, die `ggplot2` zugrundeliegt, kann ich Ihnen den Originalartikel von [Wickham \(2010\)](#) empfehlen.

<sup>1</sup>Normalerweise geben wir bei jeder Funktion die Paketzugehörigkeit über die `::`-Schreibweise explizit an. Wir werden darauf verzichten, wenn es sich um Hilfsfunktionen innerhalb anderer Funktionen handelt. So wird die Funktion `ggplot2::aes()` nur innerhalb der Funktion `ggplot2::ggplot()` aufgerufen. Daher werden wir bei `aes()` und vergleichbaren Funktionen auf den Zusatz `ggplot2::` im Sinne der besseren Lesbarkeit verzichten.

## 5.2 Grundlegende Elemente von ggplot2-Grafiken

### 5.2.1 Elemente eines ggplot

Analog zu der gerade vorgestellten [Theorie](#) besteht jeder `ggplot` aus den folgenden Komponenten:

- Dem **Basisobjekt**, welches einen leeren Plot erstellt und die **Standardwerte** für den zu verwendeten Datensatz und die entsprechenden Ästhetiken definiert.
- Verschiedenen **Ebenen** (`layer`), auf denen die - ggf. statistisch transformierten - Variablen der Daten auf bestimmten Ästhetiken (`aesthetics`) als geometrische Objekte (`geoms`) auf den entsprechenden Positionen (`position`) abgebildet werden.

Die folgenden Elemente sind ebenfalls Teil eines jeden Plots, werden aber nicht notwendigerweise explizit spezifiziert sondern einfach in der sich aus den Ebenen ergebenden Standard-Spezifikation übernommen:

- **Skalen**: Für jedes `mapping` zwischen einer Variable und einer Ästhetik gibt es eine Skala, die mit entsprechenden Funktionen geändert werden kann. So modifiziert die Funktion `ggplot2::scale_color_discrete()` das Mapping zwischen einer diskreten Variable und der Farbskala.
- **Labels**: Jeder Plot kann mit Labels, wie Titeln, Achsenbeschriftungen, Legenden oder sonstigem Text ergänzt werden.
- **Koordinaten**: Standardmäßig bilden wir Grafiken auf einem kartesischen Koordinatensystem ab. Sie können die Ausschnitte dieses Koordinatensystems beliebig anpassen, die Achsen transformieren, oder sogar ein anderes Koordinatensystem verwenden (siehe z.B. [hier](#)).
- **Facetten**: Wenn wir mehrere Facetten verwenden teilen wir die Daten gemäß einer Variable in mehrere Subdatensätze auf und bilden alle separat ab. Unten sehen Sie ein Beispiel wo wir separate Abbildungen für jedes Land im Datensatz erstellen.

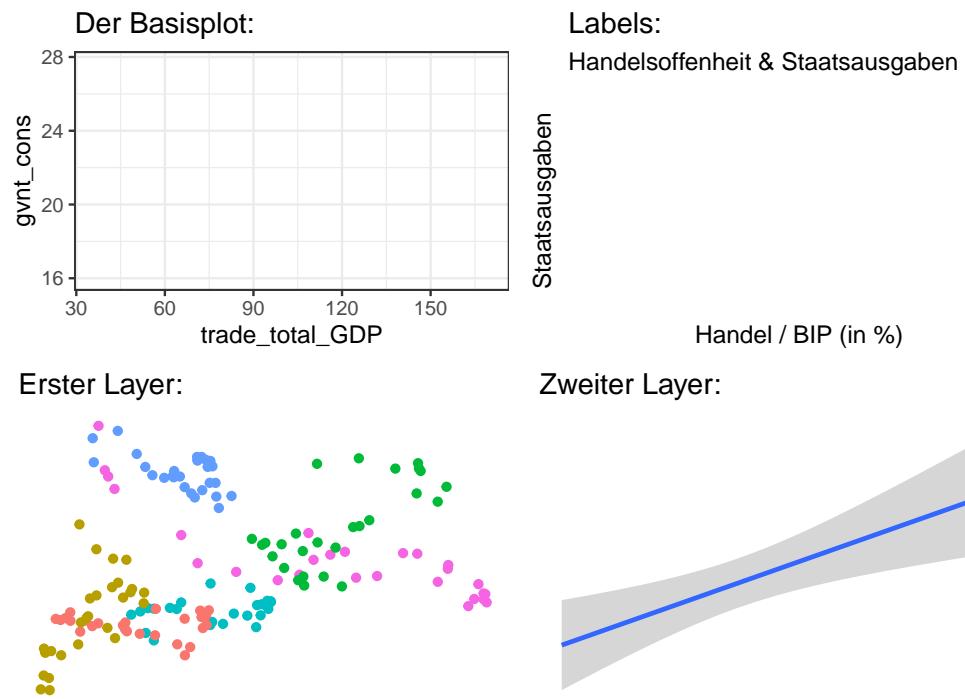
Hier ist eine Beispielimplementierung:

```
offenheit_plot <- ggplot2::ggplot( # <- Erstellt das Basisobjekt
  data = offenheit, # <- Spezifiziert Standard-Datensatz
  mapping = aes( # <- Spezifiziert die Mappings zu den Ästhetiken
    x=trade_total_GDP, # Verbinde Ästhetik 'x-Achse' & Variable 'trade_total_GDP'
    y=gvnt_cons) # Verbinde Ästhetik 'y-Achse' & Variable 'gvnt_cons'
) +
  ggplot2::layer( # <- Erstelle einen neuen Layer
    geom = "point", # Die Geoms auf diesem Layer sind Punkte
    stat = "identity", # Die Daten werden nicht statistisch transformiert
    position = "identity", # Positionen der Daten werden nicht geändert
    mapping = aes(color=Land) # Zusätzlich zur Standard-Ästhetik oben: verbinde
      # Variable 'Land' mit der Ästhetik 'color'
  ) +
  # Erstelle noch einen Layer mit der geom 'smooth' (Abkürzung für layer(...)):
  ggplot2::geom_smooth(
    method = "lm" # <- Verwende eine lineares Modell für die geom 'smooth'
  ) +
  # Gebe der Skala der x-Achse einen neuen Namen:
  ggplot2::scale_x_continuous(name = "Handel / BIP (in %)") +
  # Gebe der Skala der y-Achse einen neuen Namen:
```

```
ggplot2::scale_y_continuous(name = "Staatsausgaben") +
# Gebe der Farbskala einen neuen Namen:
ggplot2::scale_color_discrete(name="Land") +
ggplot2::labs(title = "Handelsoffenheit & Staatsausgaben 1990–2018") + # Ergänze Plot-Titel
ggplot2::coord_cartesian() + # Verwende eine kartesisches Koordinatensystem
ggplot2::facet_null() # Verwende nur eine Facette
```

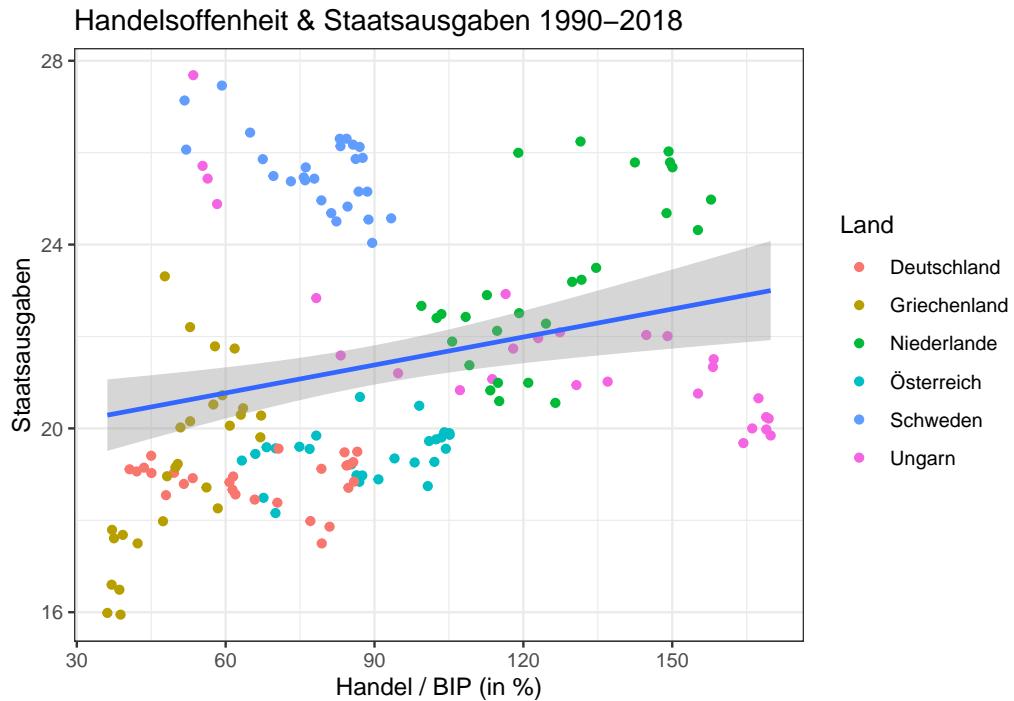
Dieser Code erstellt die einzelnen Elemente des Plots, die in `ggplot2` separat erstellt und am Ende übereinander gelegt werden:

```
#> `geom_smooth()` using formula 'y ~ x'
```



Daraus ergibt sich dann der Gesamtplot:

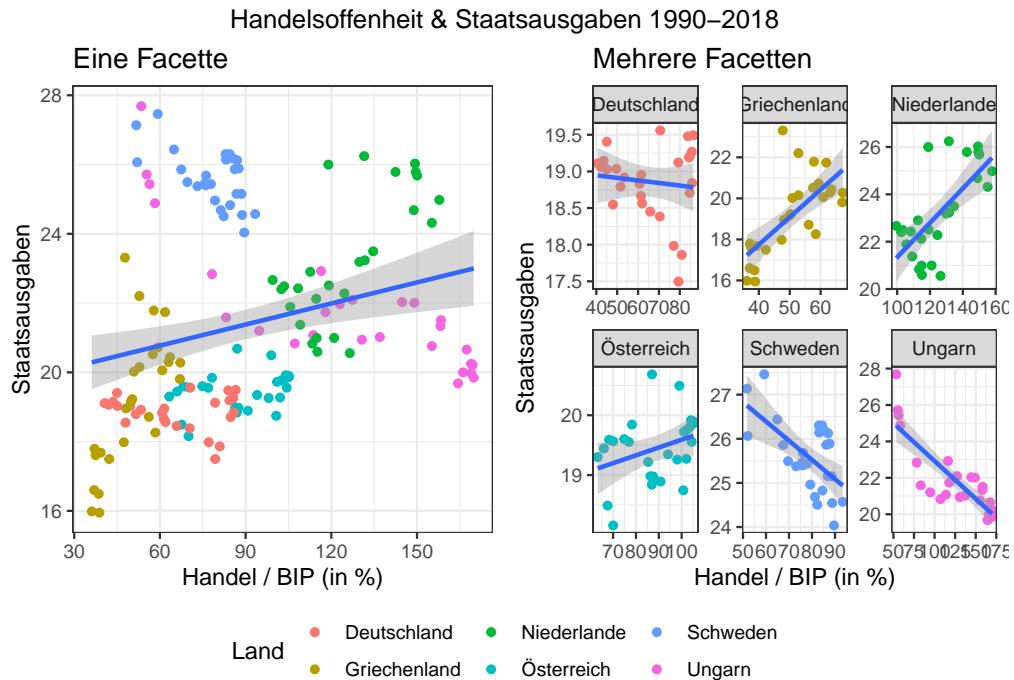
```
#> `geom_smooth()` using formula 'y ~ x'
```



```
#> `geom_smooth()` using formula 'y ~ x'
#> `geom_smooth()` using formula 'y ~ x'
```

Die Rolle der Facetten wird hier deutlich:

```
#> `geom_smooth()` using formula 'y ~ x'
#> `geom_smooth()` using formula 'y ~ x'
#> `geom_smooth()` using formula 'y ~ x'
```

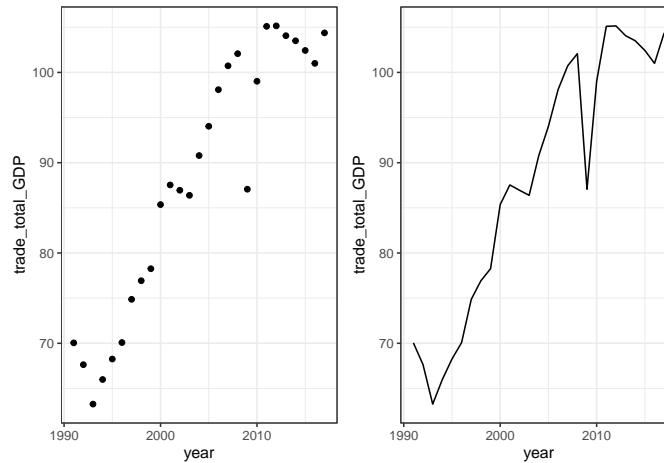


Der modulare Aufbau eines ggplot macht es einfach eine Grafik sukzessive zu ändern: wenn Sie z.B. von einem Streudiagramm zu einem Liniendiagramm wechseln wollen müssen Sie nur die `geoms` ändern - die restlichen Kom-

ponenten des Plots können identisch bleiben:

```
# Code für ein Streudiagramm
streudiagramm <- ggplot2::ggplot(offenheit_red,
                                    aes(x=year, y=trade_total_GDP)
                                    ) +
  ggplot2::geom_point() +
  ggplot2::theme_bw()

# Code für ein Liniendiagramm
liniendiagramm <- ggplot2::ggplot(offenheit_red,
                                    aes(x=year, y=trade_total_GDP)
                                    ) +
  ggplot2::geom_line() + # <- nur diese Zeile verändert
  ggplot2::theme_bw()
```



## 5.2.2 Beispiel Workflow

Hier betrachten wir den Workflow einer einfachen Grafik. Sie werden unten noch diverse Techniken lernen, wie Sie diese Grafik aufhübschen können. Übrigens ist die Reihenfolge der Schritte nicht weiter relevant, lediglich der erste Schritt muss vor den anderen kommen. Was den Rest angeht sind Sie aber in der Praxis recht flexibel, denn Sie erstellen ja am Anfang eine Liste, zu der Sie in den weiteren Schritten weitere Beschreibungsdetails hinzufügen. Die Grafik wird aus dieser durch `ggplot()` erstellten Liste erst bei Aufruf mit einer `print`-Funktion erstellt.

### 1. Schritt: Aufbereitung der Daten

Ihre Daten sollten ‘tidy’ sein, genauso wie im letzten Kapitel beschrieben. Im Folgenden gehen wir davon aus, dass wir einen entsprechend aufbereiteten Datensatz haben:

```
#>   Land Jahr HandelGDP
#> 1:  AUT 1965 48.23931
#> 2:  AUT 1966 48.92554
#> 3:  AUT 1967 48.30854
#> 4:  AUT 1968 49.01388
#> 5:  AUT 1969 52.72526
#> 6:  AUT 1970 54.86039
```

Dieser kleine Beispieldatensatz enthält Informationen über das Verhältnis von Handelsströmen und BIP in Österreich seit 1965.

**2. Schritt: Auswahl des Standarddatensatzes und der Variablen** Wir entscheiden uns, dass der gerade aufbereitete Datensatz die Basis für unsere Visualisierung darstellen soll. Natürlich können wir auch noch Daten aus anderen Datensätzen hinzufügen, aber dieser Datensatz soll unser *Standard-Datensatz* für die Grafik sein, die verwendet wird wenn wir nichts anderes spezifizieren. Genauso spezifizieren wir die *Standard-Ästhetik-Links* für die Abbildung. Eine Ästhetik ist z.B. die Größe, Farbe oder Achse der Abbildung. Es ist hilfreich am Anfang Standardwerte für die Verknüpfung von Variablen aus dem Datensatz mit Ästhetiken in der Grafik zu spezifizieren.

Im Beispiel wollen wir die Variable `Jahr` mit der x-Achse und die Variable `HandelGDP` mit der y-Achse verbinden. Da es sich um die Standardwerte handelt werden Sie in der Funktion `ggplot()` spezifiziert:

```
aut_trade_plot <- ggplot2::ggplot(
  data = aut_trade,
  mapping = aes(x = Jahr,
                 y = HandelGDP)
)
```

`ggplot2::ggplot()` erstellt das Grafik-Objekt, bei dem es sich um eine recht komplexe Liste handelt:

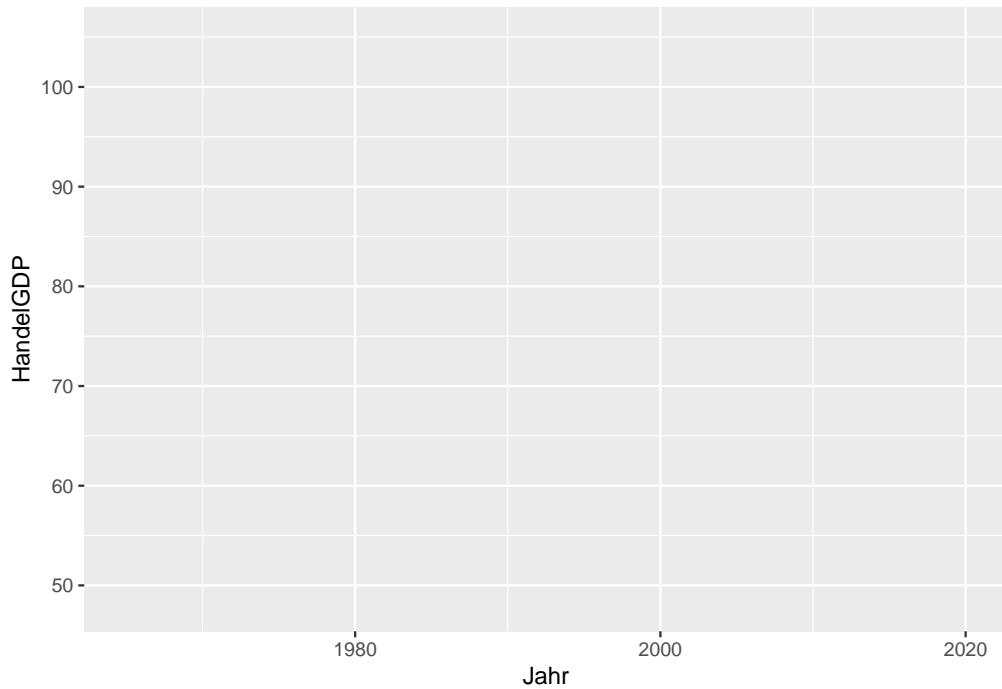
```
typeof(aut_trade_plot)
```

```
#> [1] "list"
```

Die Funktion `ggplot2::ggplot()` wird in der Regel mit zwei Argumenten verwendet: `data` spezifiziert den Standard-Datensatz für die Grafik und `mapping` die *aesthetic mappings*, welche die Variablen in `data` zu den ästhetischen Komponenten der Grafik verlinken. Wenn Sie den optionalen Abschnitt zur [Grammar of Graphics](#) gelesen haben, werden Sie die Konzepte sofort wiedererkennen!

Wie oben beschrieben wird die Grafik bei `ggplot2` erst erstellt, wenn Sie das Grafik-Objekt mit einer `print`-Funktion aufrufen. Das passiert automatisch, wenn Sie das Objekt als solches aufrufen:

```
aut_trade_plot
```



Da wir bislang nur die Standardwerte definiert haben ist die Grafik noch recht leer. Zumindest sehen wir, dass die Achsen die Variablen unseres Datensatzes repräsentieren.

### 3. Schritt: Hinzufügen von Ebenen mit geometrischen Objekten

Als nächstes wollen wir die geometrischen Objekte spezifizieren, mit denen die Ästhetiken auf dem Plot dargestellt werden sollen. Im vorliegenden Fall möchten wir z.B. unsere Beobachtungen mit einer Linie visualisieren. Das geht mit der Funktion `ggplot2::geom_line()`: sie fügt einen `geom` der Art 'Linie' hinzu. Im übrigen sind die Namen für alle verschiedenen `geoms` gleich aufgebaut, es ist immer `ggplot2::geom_*`, wobei \* für die Abkürzung des entsprechenden `geoms` steht.<sup>2</sup>

Die Funktionen `ggplot2::geom_*` verlangen in der Regel kein zusätzliches Argument, verwenden aber einige Standardwerte über die Sie Bescheid wissen sollten. Die Argumente `data` und `mapping` funktionieren wie oben beschrieben und haben als Standardwert die anfangs in `ggplot2::ggplot()` angegebenen Werte. Das Argument `stat` spezifiziert statistische Transformationen, die an den Daten vor dem Plotten vorgenommen werden sollen. Wenn die Daten bereits korrekt aufbereitet wurden ist das häufig nicht notwendig und der Standardwert `stat='identity'` ist ausreichend - in diesem Fall werden die Daten so abgebildet wie sie im Datensatz vorhanden sind.<sup>3</sup> Das Gleiche gilt für das Argument `position`: auch hier ist der Standardwert `position='identity'`, aber Sie können über verschiedene Funktionen die Position der `geoms` anpassen, z.B. um Überlappungen zu vermeiden.<sup>4</sup>

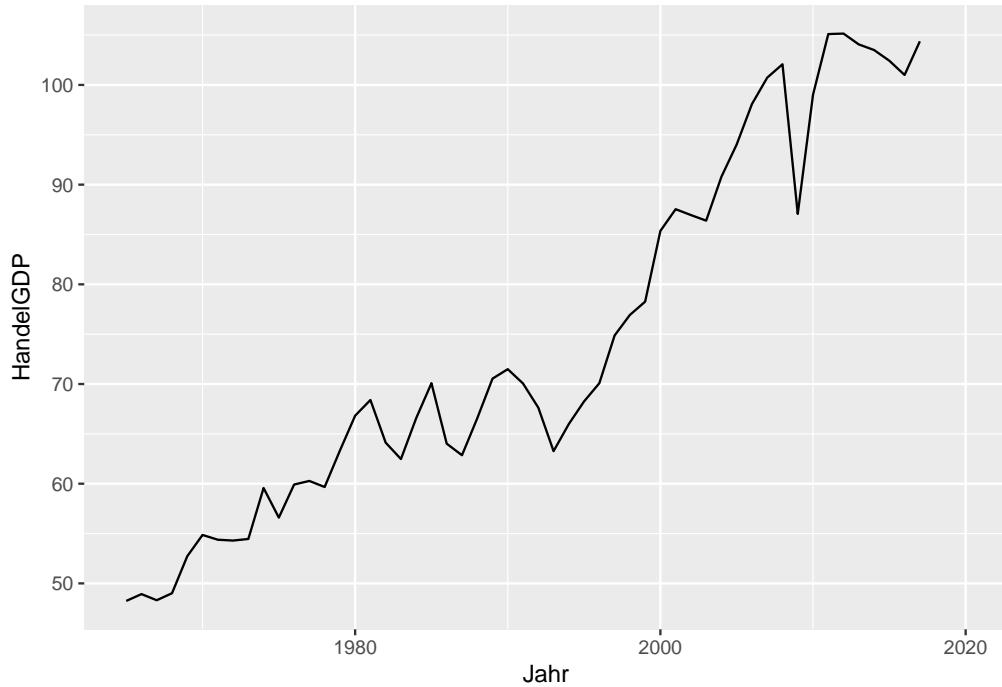
Da wir zu unserer Grafik `aut_trade_plot` eine Ebene hinzufügen wollen, verwenden wir einfach den Operator `+`:

```
aut_trade_plot <- aut_trade_plot +
  ggplot2::geom_line()
aut_trade_plot
```

<sup>2</sup>Eine Liste aller möglichen `geoms` finden Sie [hier](#).

<sup>3</sup>Alternativ zu `ggplot2::geom_*(stat="...")` können Sie auch immer schreiben `ggplot2::stat_*(geom="...")`. Entsprechend sind folgende Aufrufe äquivalent: `ggplot2::stat_identity(geom="line")` oder `ggplot2::geom_line(stat="identity")`. Was Sie verwenden ist komplett Ihnen überlassen, allerdings ist die Verwendung der `ggplot2::geom_*`-Funktionen üblicher.

<sup>4</sup>Die möglichen Werte für `position` sind: `identity` (der Standard, keine Anpassung der Positionen), `jitter` (Geoms werden über Zufallsfehler so verschoben, dass sie sich nicht überlappen), `dodge` (sich überlappende Geoms werden nebeneinander angeordnet), `fill` (die Geoms werden übereinander abgebildet und zu einer gleichmäßigen Summe normalisiert) und `stack` (die Geoms werden übereinander geplottet, aber nicht normalisiert). Die letzten drei Argumente werden vor allem bei Balkendiagrammen häufig verwendet.

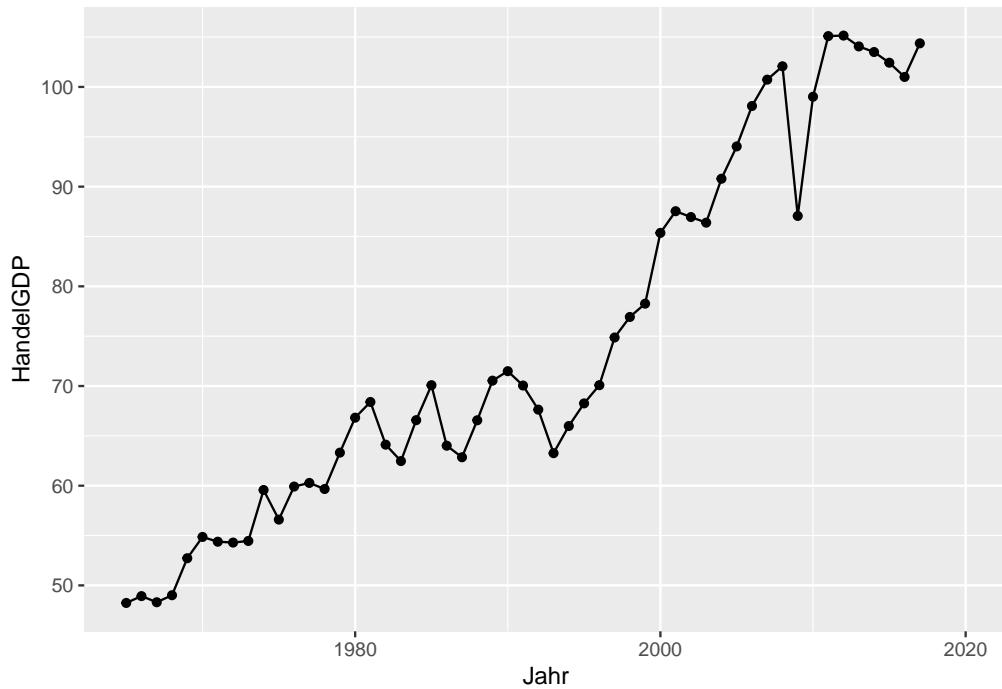


Am Anfang ein Grafikobjekt zu definieren und dann neue Elemente Stück für Stück mit `+` hinzuzufügen ist das Grundprinzip von `ggplot2`. Auch hier ist die Verbindung zu Wickham's [Grammar of Graphics](#) offensichtlich.

Im Beispiel haben wir `ggplot2::geom_line()` ohne ein einziges Argument aufgerufen. Wir könnten die Argumente `data` und `mapping` verwenden, aber da wir hier die in Schritt 1 definierten Standardwerte verwenden besteht dazu keine Veranlassung.

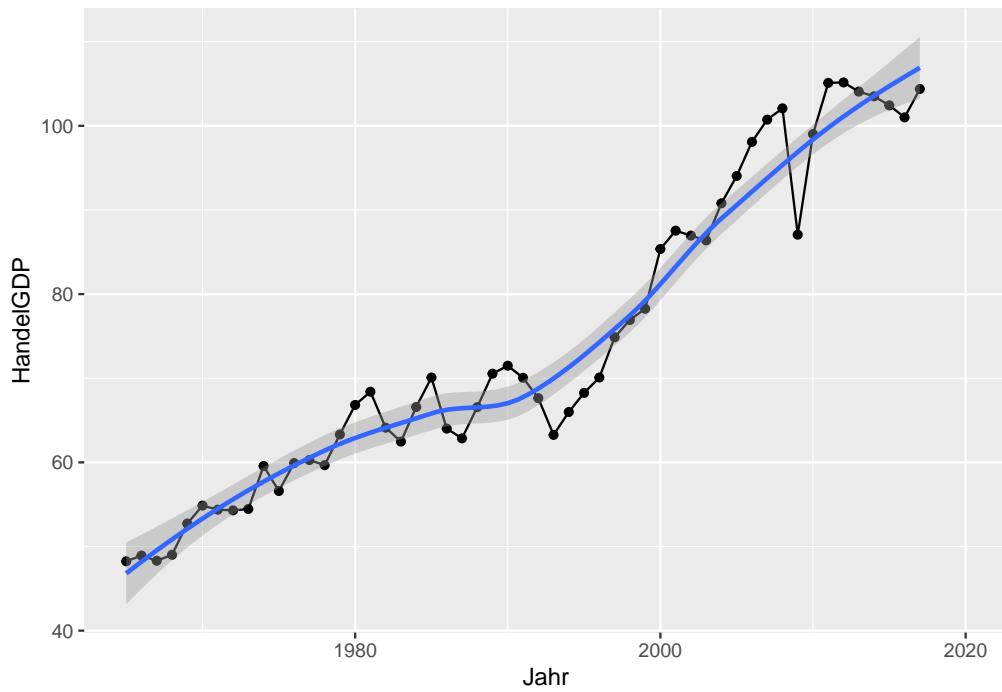
Wir können durchaus mehrere Ebenen nacheinander hinzufügen. Wenn wir die einzelnen Beobachtungen z.B. noch durch Punkte verdeutlichen wollen, dann können wir einfach eine weitere Ebene mit dem `geom` 'Punkt' hinzufügen. Das geht mit der Funktion `ggplot2::geom_point()` und da wir die gleichen Standardwerte wie vorher verwenden sind hier auch keine Argumente nötig:

```
aut_trade_plot <- aut_trade_plot +
  ggplot2::geom_point()
aut_trade_plot
```



Um den Trend der Entwicklung zu verdeutlichen möchten wir vielleicht noch einen Trend grafisch hinzufügen. Hierzu verwenden wir die Funktion `geom_smooth()`:

```
aut_trade_plot <- aut_trade_plot +  
  ggplot2::geom_smooth()  
aut_trade_plot
```



#### 4. Schritt: Anpassen der Skalen

Im nächsten Schritt wollen wir die *Skalen* der Abbildung anpassen. Für uns sind hier vor allem die Skalen der y-Achse und der x-Achse relevant.<sup>5</sup> Daher verwenden wir die Funktionen `ggplot2::scale_x_continuous()`

---

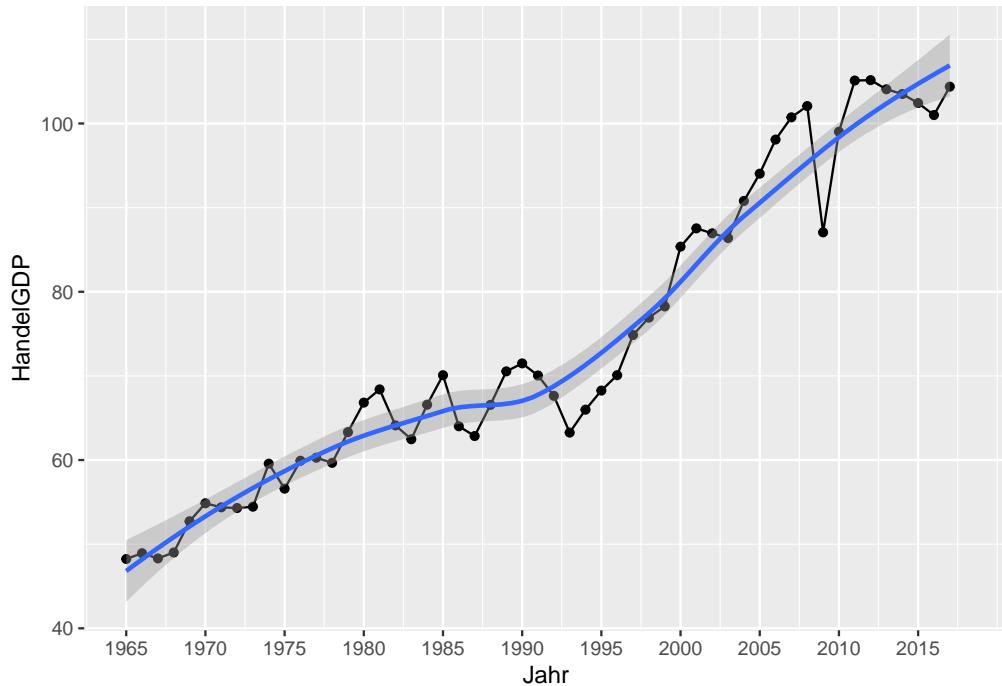
<sup>5</sup>Andere Skalen beziehen sich z.B. auf Farben, wenn wir Variablen zu einer farblichen Ästhetik gemapt hätten, oder die Formen der

und `ggplot2::scale_y_continuous()`, schließlich handelt es sich bei den auf diesen Skalen abgebildeten Variablen um kontinuierliche Variable. Wenn es diskrete Daten gewesen wären, würden wir die Funktionen `ggplot2::scale_x_discrete()` und `ggplot2::scale_y_discrete()` verwenden.

Beginnen wir mit der x-Achse. Hier möchten wir vor allem die auf der Skala angegeben Jahreszahlen anpassen und die Länge der Skala auf den Zeitraum 1965-2018 anpassen.

Die abzubildenden Jahre spezifizieren wir mit dem Argument `breaks`, dem wir einen Vektor mit den abzubildenden Jahreszahlen übergeben. Die Limits der Skala können wir mit dem Argument `limits` spezifizieren indem wir einen Vektor mit zwei Zahlen, dem unteren und dem oberen Limit, übergeben:

```
aut_trade_plot <- aut_trade_plot +
  ggplot2::scale_x_continuous(limits = c(1965, 2018),
                             breaks = seq(1965, 2017, 5))
aut_trade_plot
```

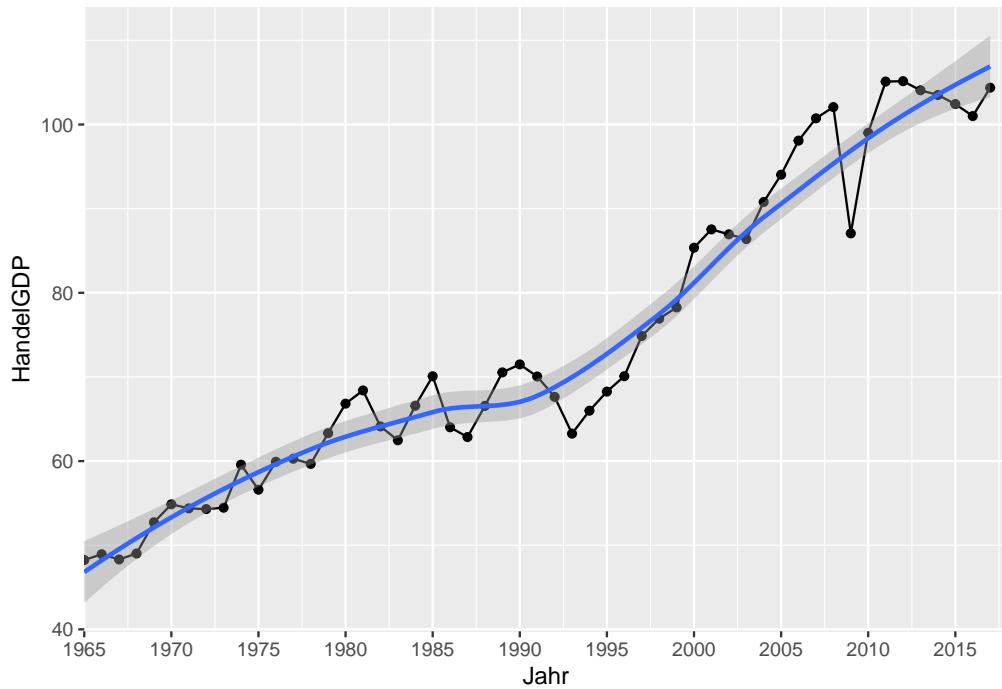


Unschön hier ist nur der ‘Rand’, den `ggplot2` automatisch an den jeweiligen Enden der Skalen hinzufügt. Dieser Rand kann durch das Argument `expand` geändert werden. Wie übergeben `expand` im einfachsten Falle einen Vektor mit zwei Werten: der erste Wert bestimmt eine Konstante, die auf beiden Seiten zur Skala hinzuaddiert wird, der zweite Wert einen Skalar der die Skala um den entsprechenden Wert multiplikativ streckt. In unserem Fall sollen beide Werte gleich 0 sein, denn wir wollen, dass die Skala 1960 anfängt und 2017 aufhört, so wie über das Argument `limits` vorher spezifiziert:

```
aut_trade_plot <- aut_trade_plot +
  ggplot2::scale_x_continuous(limits = c(1965, 2018),
                             breaks = seq(1960, 2017, 5),
                             expand = c(0, 0)
                           )
aut_trade_plot
```

---

`geoms`. Beispiele für so fortgeschrittene Anpassungen finden Sie [weiter unten](#) in diesem Kapitel.



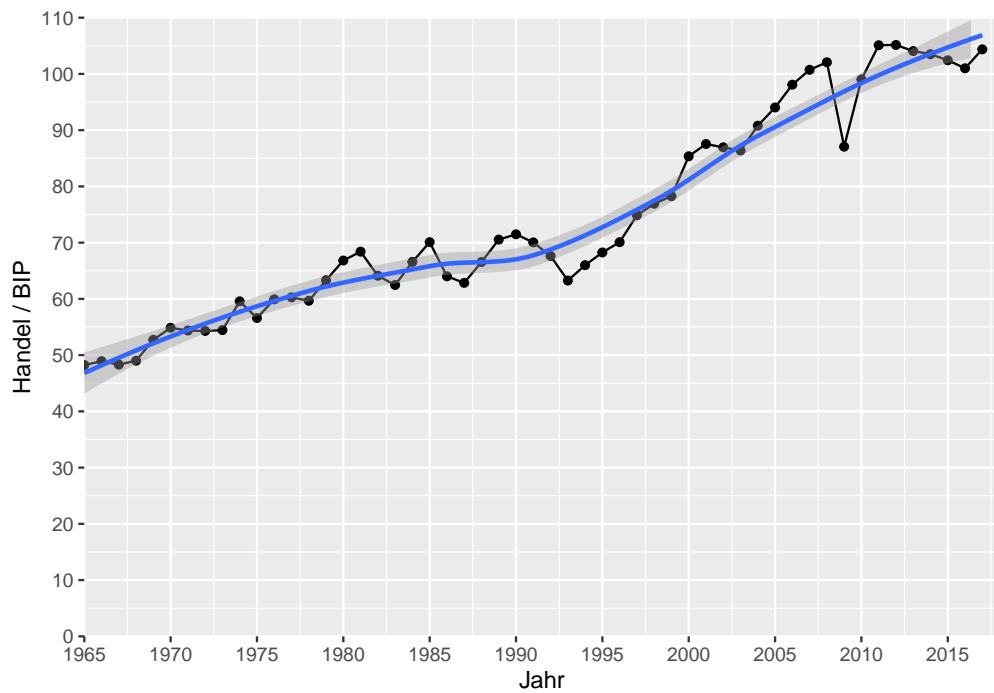
Das ist schon nicht so schlecht. Als nächstes beschäftigen wir uns mit der y-Achse. Hier möchten wir auch die Limits und die angegebenen Werte verändern, und zwar von 0 bis 110. Das geht wieder über die Argumente `limits` und `breaks`.

Darüber hinaus wäre es schön, den Namen der Achse anzupassen. Standardmäßig ist das der Name der Variable im Datensatz, aber hier wäre es schöner wenn dort einer 'Handel / BIP' stehen würde. Das erledigen wir mit dem Argument `name`.<sup>6</sup>

Auch möchten wir wieder den hässlichen Rand am oberen und unteren Ende der Skala eliminieren und verwenden dazu das Argument `expand` wie vorher:

```
aut_trade_plot <- aut_trade_plot +
  ggplot2::scale_y_continuous(name = "Handel / BIP",
    limits = c(0, 110),
    breaks = seq(0, 110, 10),
    expand = c(0, 0)
  )
aut_trade_plot
```

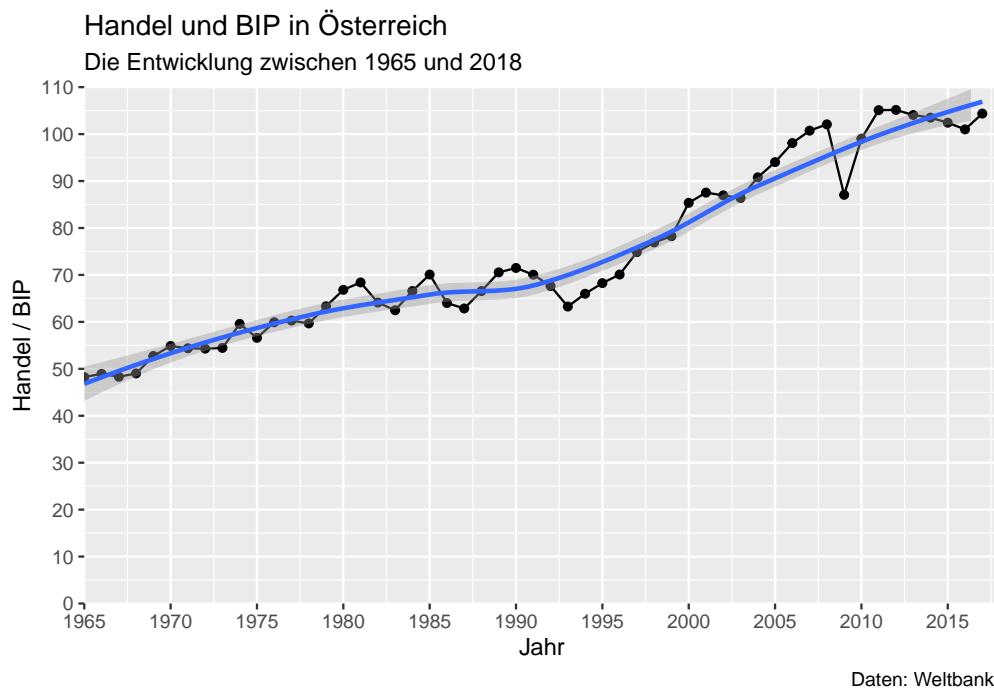
<sup>6</sup>Wenn wir nichts weiter an der Skala verändern wollen außer diesem so genannten Label, dann brauchen wir auch nicht die Funktion `ggplot2::scale_y_continuous()` aufrufen, sondern können einfach schreiben `ggplot2::ylab("Handel / BIP")`.



## 5. Schritt: Titel

Titel und andere so genannte ‘Labels’ können Sie mit der Funktion `ggplot2::labs()` sehr einfach hinzufügen. `ggplot2::labs()` akzeptiert drei optionale Argumente: `title` für den Titel, `subtitle` für den Untertitel und `caption` für eine Fußnote, die sich besonders gut eignet um die Quelle der Daten anzugeben.

```
aut_trade_plot <- aut_trade_plot +
  ggplot2::labs(title = "Handel und BIP in Österreich",
    subtitle = "Die Entwicklung zwischen 1965 und 2018",
    caption = "Daten: Weltbank.")
```



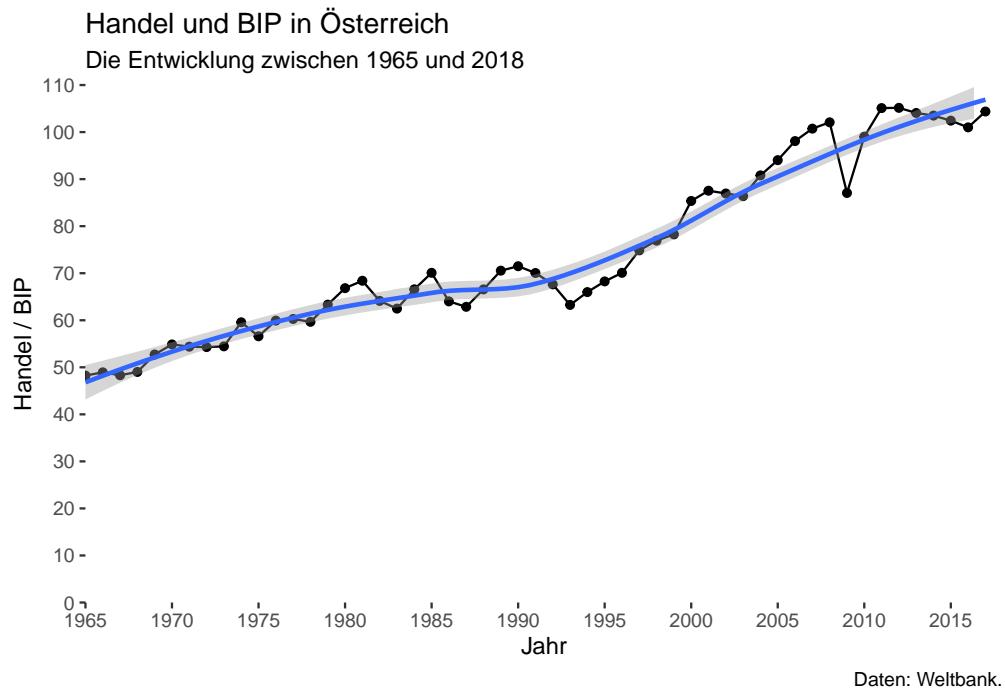
Weit verbreitet ist auch die Funktion `ggplot2::ggtitle()`, die genauso funktioniert, aber nur die Argumente `label` (für den Titel) und `subtitle` akzeptiert.

## 6. Schritt: Grundlegende Veränderungen mit `ggplot2::theme()`

Achtung, Kleinkram-Alarm! Zwar schaut die Grafik jetzt schon erträglich aus, aber es gibt natürlich noch diverse Dinge, die wir verschönern könnten. Warum der Hintergrund z.B. standardmäßig grau und die Linien in weiß sind, weiß niemand. Solcherlei Veränderungen können Sie über die Funktion `ggplot2::theme()` vornehmen. Wir betrachten hier nur ein paar Beispiele, eine Übersicht zu allen möglichen Argumenten finden Sie [hier](#).

Um den Hintergrund des Plot im Abbildungsbereich zu verändern verwenden wir das Argument `panel.background`. Solcherlei Veränderungen werden immer über bestimmte Funktionen durchgeführt, die sich nach der Art des zu veränderten Grafikbestandteils richten. Im Falle des Plot-Hintergrundes ist das ein Rechteck, sodass wir die Funktion `ggplot2::element_rect()` verwenden, die zahlreiche Gestaltungsmöglichkeiten erlaubt.<sup>7</sup> Hier wollen wir den Hintergrund weiß füllen, wir schreiben also `ggplot2::element_rect(fill = "white")`:

```
aut_trade_plot <- aut_trade_plot +
  ggplot2::theme(panel.background = element_rect(fill = "white"))
aut_trade_plot
```



Das ist besser, allerdings möchten wir schon einen Grid haben um die Achsen besser lesen zu können. Das entsprechende Argument ist `panel.grid`, bzw. `panel.grid.major` und `panel.grid.minor` für die Linien auf, bzw. zwischen den auf den Achsen aufgeschriebenen Werten.<sup>8</sup> Damit wir den Plot nicht überlasten malen wir aber nur auf die auf den Achsen auch tatsächlich abgebildeten Werte Linien, verwenden also das Argument `panel.grid.major`. Da es sich hier um Linien handelt verwenden wir die Funktion `ggplot2::element_line()`, die wir hier noch über die Farbe des Grids informieren: `ggplot2::element_line(colour = "grey")`. Auch die

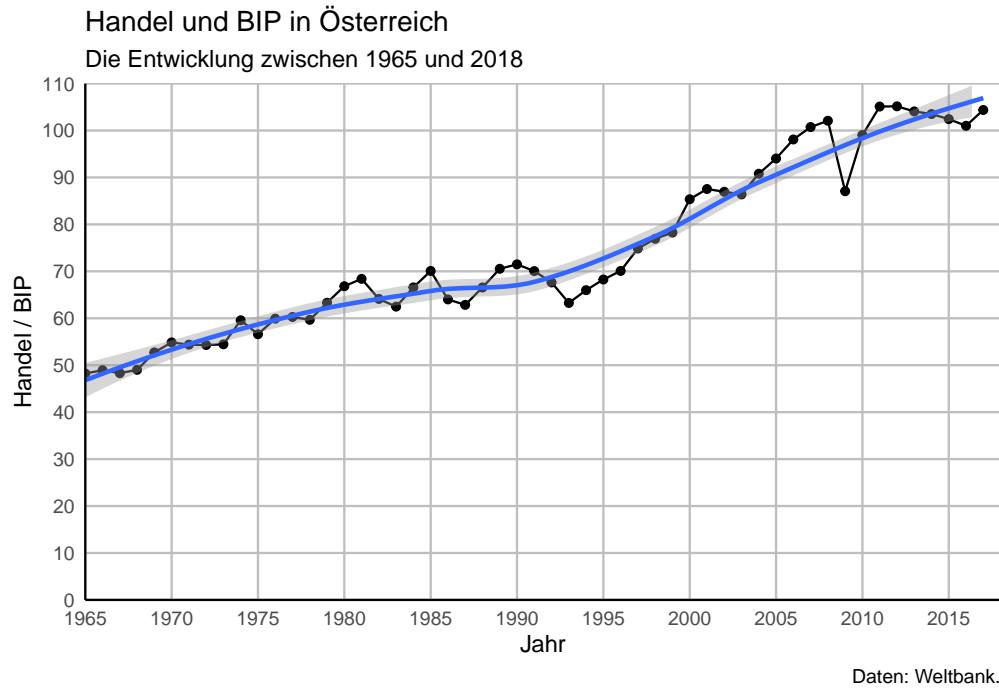
<sup>7</sup>Insgesamt gibt es die folgenden Hilfsfunktionen im ggplot2-Paket: `element_rect()` für Flächen und Kanten, `element_line()` für Linien und `element_text()` für Text. Wenn Sie einen Teil eliminieren wollen verwenden Sie `element_blank()`. Alle diese Funktionen bieten unzählbar viele Gestaltungsmöglichkeiten.

<sup>8</sup>Wenn Sie den horizontalen und vertikalen Grid separat ändern wollen verwenden Sie jeweils das Suffix `.x`, also `ggplot2::panel.grid.minor.x` bzw. `ggplot2::panel.grid.minor.y`.

fehlenden Achsenlinien machen den Plot nicht schöner. Wir fügen Sie über das Argument `axis.line` mit der Funktion `ggplot2::element_line()` explizit hinzu!

Sehr hässlich sind auch die kleinen schwarzen Zacken bei jedem Wert auf der x- und y-Achse. Diese werden mit `axis.ticks = ggplot2::element_black()` eliminiert. Sie verwenden die Funktion `ggplot2::element_blank()` ohne Argument immer wenn Sie einen bestimmten Teil der Grafik eliminieren wollen.<sup>9</sup> Somit bekommen wir insgesamt:

```
aut_trade_plot <- aut_trade_plot +
  ggplot2::theme(
    panel.background = element_rect(fill = "white"),
    panel.grid.major = element_line(colour = "grey"),
    panel.grid.minor = element_blank(),
    axis.line = element_line(colour = "black"),
    axis.ticks = element_blank()
  )
aut_trade_plot
```



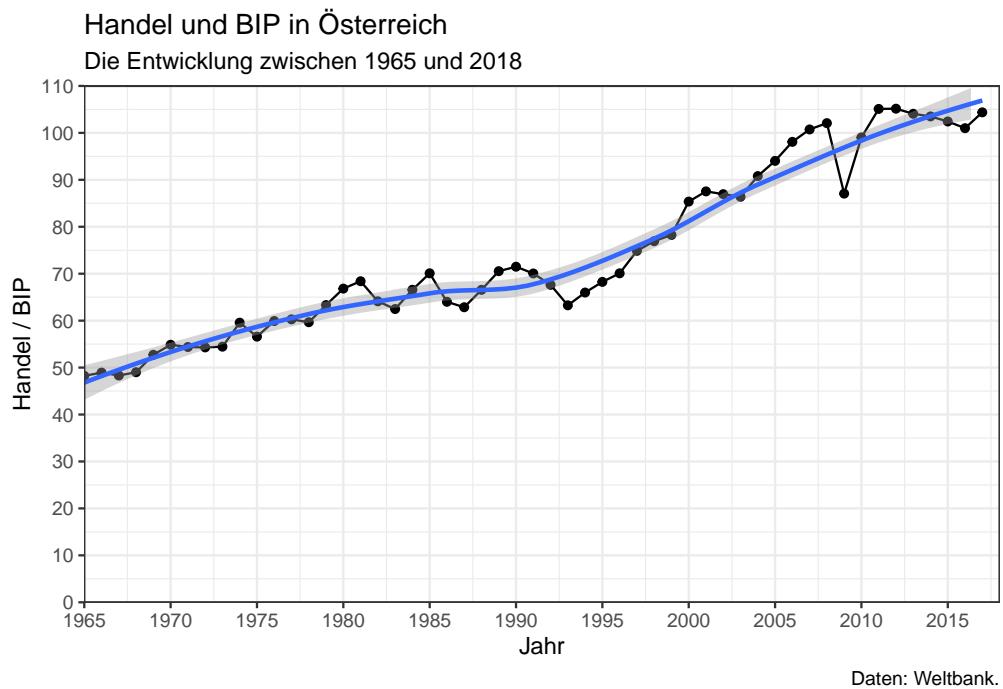
Sie merken bereits: mit `ggplot2::theme()` können Sie quasi alles an Ihrer Grafik ändern was Sie sich irgendwie vorstellen können. Einen Überblick über alle möglichen Parameter finden Sie [hier](#). Wie beschäftigten uns [unten](#) noch mit ausgewählten Argumenten etwas genauer.

Gleichzeitig mag es aber auch nervig sein, so viele Einstellungen immer manuell vorzunehmen. Daher gibt auch zahlreiche vorgefertigte Themen, die bestimmte Standard-Spezifikationen vornehmen. Eine Übersicht finden Sie [hier](#). Häufig wird z.B. das Theme `ggplot2::theme_bw()` verwendet:

```
aut_trade_plot + theme_bw()
```

---

<sup>9</sup>Auch im folgenden code wird die Zugehörigkeit der `element_*`-Funktionen zu `ggplot2` der einfachen Lesbarkeit halber nicht explizit deutlich gemacht.



Natürlich können Sie auch eigene Themen schreiben, in denen Sie Ihre Lieblingseinstellungen zusammenfassen.<sup>10</sup>

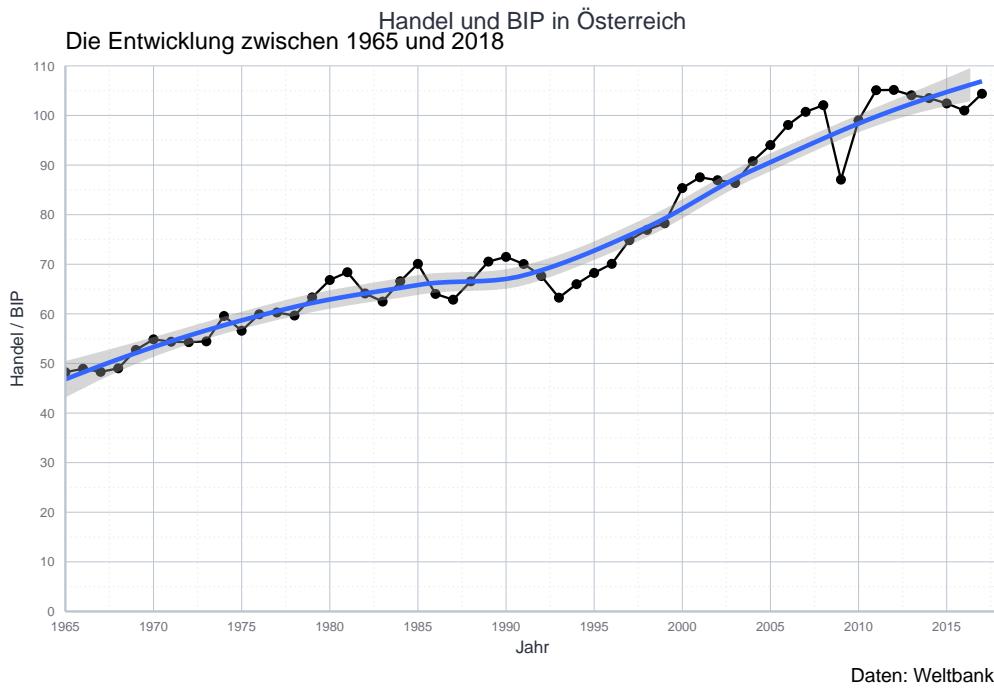
**Tipp:** Wenn Ihnen die Abbildungen im Skript bislang und auf den Slides gefallen haben können Sie gerne mein Standard-Thema verwenden. Sie können in `ggplot2` nämlich typische Anpassungen, die Sie mit `theme()` regelmäßig durchführen, auch automatisieren und eigene Themen verwenden. Das Thema, das ich verwende ist Teil des Pakets `icaeDesign` (Gräßner, 2019) und kann durch die Funktion `icaeDesign::theme_icae()` verwendet werden. Um das Paket `icaeDesign` zu installieren müssen Sie folgendermaßen vorgehen:

```
library(devtools)
devtools::install_github("graebnerc/icaeDesign")
```

Unser Beispielplot sähe damit folgendermaßen aus:

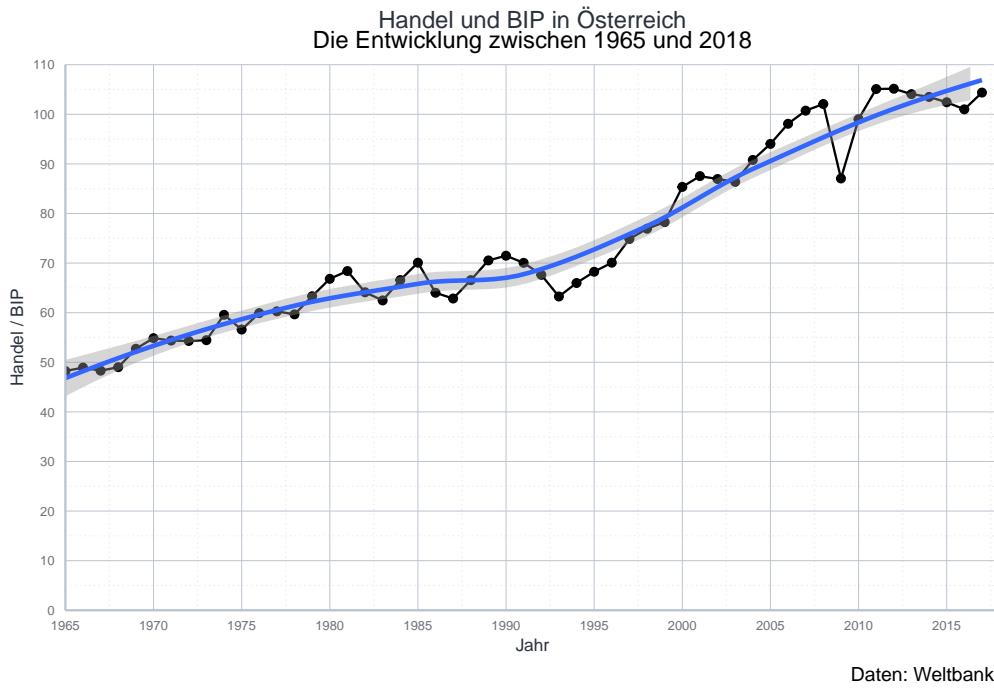
```
library(icaeDesign)
aut_trade_plot <- aut_trade_plot + theme_icae()
aut_trade_plot
```

<sup>10</sup>Eine gute Anleitung zu Erstellen eigener Themen finden Sie [hier](#).



Das ist nicht so schlecht, allerdings ist der Untertitel hässlich. Da ich selbst so gut wie nie Untertitel verwende ist das aktuell im Thema nicht berücksichtigt. Zum Glück können wir mit `ggplot2::theme()` auch nach einem benutzerdefinierten Theme noch weitere Modifikationen vornehmen. Da es sich beim Untertitel um Text handelt, verwenden wir die Funktion `ggplot2::element_text()`:

```
aut_trade_plot +
  ggplot2::theme(plot.subtitle = element_text(hjust = 0.5))
```



Dabei ist der Aufruf `icaeDesign::theme_icae()` eine Abkürzung für folgenden Aufruf von `ggplot2::theme()`. Sie müssen die Befehle nicht nachvollziehen, das ist nur zur Info:

```
ggplot2::theme_minimal() +
  ggplot2::theme(
    axis.line = element_line(
      color = rgb(188, 197, 207, maxColorValue = 255),
      linetype = "solid", size = 0.5
    ),
    legend.position = "bottom",
    legend.spacing.x = unit(0.2, "cm"),
    legend.title = element_blank(),
    plot.title = element_text(
      color = rgb(43, 49, 62, maxColorValue = 255),
      hjust = 0.5
    ),
    axis.title = element_text(
      color = rgb(43, 49, 62, maxColorValue = 255),
      size = rel(0.75)
    ),
    axis.text = element_text(
      color = rgb(110, 113, 123, maxColorValue = 255),
      size = rel(0.5)
    ),
    panel.grid.major = element_line(
      color = rgb(188, 197, 207, maxColorValue = 255),
      linetype = "solid"),
    panel.grid.minor = element_line(
      color = rgb(233, 234, 233, maxColorValue = 255),
      linetype = "dotted",
      size = rel(4)
    ),
    strip.text = element_text(
      size = rel(0.9),
      colour = rgb(43, 49, 62, maxColorValue = 255),
      margin = margin(t = 1, r = 1, b = 1, l = 1, unit = "pt")
    ),
    strip.text.x = element_text(
      margin = margin(t = 5, r = 1, b = 1, l = 1, unit = "pt"))
  )
```

## 7. Schritt: Ihre Grafik abspeichern

Zum Schluss können wir noch unsere Grafik speichern. Das machen wir ganz einfach mit der Funktion `ggplot2::ggsave()`. Die wichtigsten Argumente sind `filename` (für Dateinamen und Speicherort), `plot` (für den zu speichernden Plot), `width` (für die Breite der Abbildung) und `height` (für die Höhe der Figur).<sup>11</sup>

<sup>11</sup>Standardmäßig werden Breite und Höhe in Zoll angegeben. Mit der Funktion `unit()` aus dem Paket `units` (Pebesma et al., 2016) können Sie aber ganz einfach beliebige Einheiten verwenden, z.B. `width = unit(2, "cm")`. In der Praxis probieren Sie einfach herum bis Sie die richtige Kombination von Höhe und Breite gefunden haben. Für Abbildungen, die aus nur einem Plot bestehen ist `6:4` häufig ein guter Ausgangspunkt.

```
ggplot2::ggsave(filename = here::here("output/trade_ts.pdf"),
  plot = aut_trade_plot,
  width = 9,
  height = 6)
```

Achten Sie auf die Beibehaltung einer übersichtlichen Ordnerstruktur. Abbildungen sollten immer im Ordner `output` gespeichert werden!

**Tipp: Das richtige Format** Wenn nicht irgendwelche gewichtigen Gründe dagegen sprechen (z.B. dass Sie Ihre Grafik auf einer Website verwenden wollen) dann sollten Sie Ihre Grafik immer als PDF speichern. Da es sich dabei um eine `vektorbasierte Grafik` handelt bleiben Sie sehr flexibel was das spätere Vergrößern oder Verkleiner der Grafik angeht. Wenn Sie kein PDF verwenden können ist in der Regel PNG die erste Alternative.

## Zusammenfassung

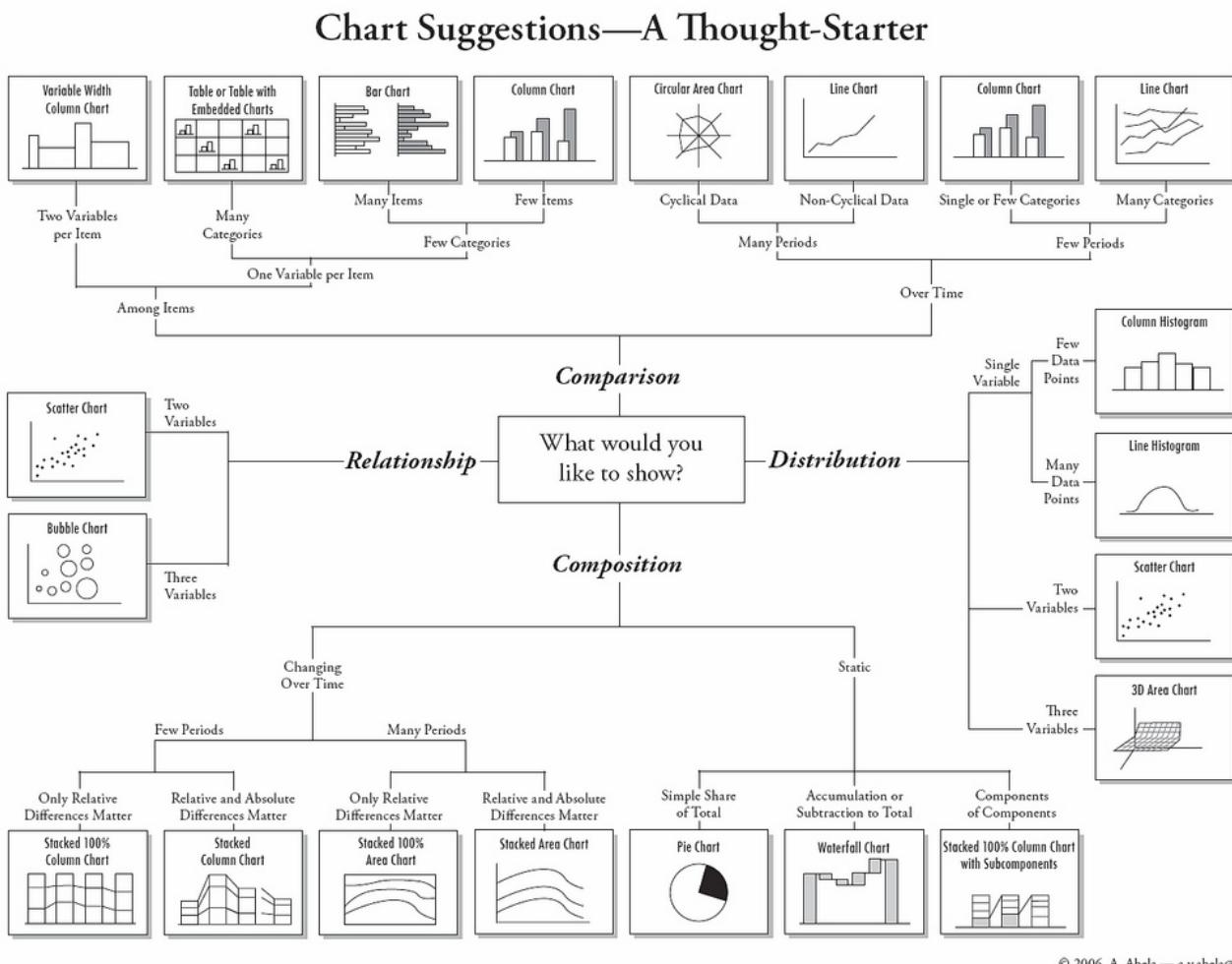
Abschließend noch einmal der komplette Code für unsere Abbildung:

```
aut_trade_plot <- ggplot2::ggplot(
  data = aut_trade,
  mapping = aes(x = Jahr,
                 y = HandelGDP)
) +
  ggplot2::geom_line() +
  ggplot2::geom_point() +
  ggplot2::geom_smooth() +
  ggplot2::scale_x_continuous(
    limits = c(1965, 2018),
    breaks = seq(1960, 2017, 5),
    expand = c(0, 0)
) +
  ggplot2::scale_y_continuous(
    name = "Handel / BIP",
    limits = c(0, 110),
    breaks = seq(0, 110, 10),
    expand = c(0, 0)
) +
  ggplot2::ggttitle(
    label = "Handel und BIP in Österreich",
    subtitle = "Die Entwicklung zwischen 1965 und 2018"
) +
  ggplot2::theme(
    panel.background = element_rect(fill = "white"),
    panel.grid.major = element_line(colour = "grey"),
    panel.grid.minor = element_blank(),
    axis.line = element_line(colour = "black"),
    axis.ticks = element_blank()
)
```

```
ggplot2::ggsave(filename = here::here("output/trade_ts.pdf"),
  plot = aut_trade_plot,
  width = 9,
  height = 6)
```

## 5.3 Arten von Datenvisualisierung

Es gibt viele verschiedene Arten wie Sie einen Datensatz visualisieren können. Bevor Sie sich für eine Art entscheiden müssen Sie sich immer fragen: "Welche Information möchte ich dem oder der Betrachter\*in mit dieser Abbildung vermitteln?" Die Antwort auf diese Frage in Kombination mit den Daten, die Sie zur Verfügung haben bestimmt dann die adequate Darstellungsform. Abbildung 5.1 kann dabei als erste Inspiration dienen:



© 2006 A. Abela — a.vabela@gmail.com

Figure 5.1: Mögliche Darstellungsformen. Quelle: <http://www.perceptualedge.com/blog/wp-content/uploads/2015/07/Abelas-Chart-Selection-Diagram.jpg>

Im Folgenden werde ich Ihnen einige Beispiel-Implementierungen mit `ggplot2` präsentieren. Am Ende werden die verschiedenen Visualisierungsmöglichkeiten noch einmal kurz in einer Tabelle **zusammengefasst**. Zuvor möchte ich Ihnen jedoch einige Hinweise dazu geben, wie Sie Grafiken grundsätzlich ein wenig ansprechender gestalten können.

### 5.3.1 Allgemeine Tipps zum Grafikdesign

Die folgenden Punkte sollten Sie beim Erstellen von Grafiken immer im Hinterkopf behalten:

- Entfernen Sie den Kasten um Ihre Abbildung, die normalen Achsen sind vollkommen ausreichend. Das geht über `ggplot2::theme()` mit `panel.border=element_blank()`. Dann sollten Sie allerdings die Achsen wieder mit `axis.line=element_line()` hinzufügen.
- Überlegen Sie sich gut ob Sie eine Legende brauchen und wo sie möglichst platzsparend platziert werden kann. Innerhalb von `ggplot2::theme()` geht das über das Argument `legend.position`, welches für Legenden außerhalb des Plots 'top', 'bottom', 'left' oder 'right', und für Legenden innerhalb des Plots die Koordinaten innerhalb des Plots mit `c(x, y)` akzeptiert.
- Vermeiden Sie ein zu enges Gitter für Ihren Plot, da dies für die Betrachter\*innen schnell anstrengend wird.
- Überhaupt gilt in der Regel ‘Weniger ist mehr’. Wenn Sie sich also nicht sicher sind ob Sie ein bestimmtes Element in Ihrer Abbildung brauchen, lassen Sie es weg.
- Das gilt auch für kleinere Elemente wie die Ticks auf den Achsen, denen man häufig keine Beachtung schenkt, die aber unbewusst sehr störend sind. Sie werden mit `axis.ticks=element_blank()` eliminiert.
- Verwenden Sie keine Spezialeffekte wie 3d-Balken oder ähnliches.
- Verwenden Sie ein angenehmes Farbschema, häufig sind weniger aggressive Farben besser geeignet (wie z.B. durch das Paket `icaeDesign` bereit gestellt)
- Auch ist es häufig besser leicht transparente Farben zu verwenden.
- Wenn Sie in Ihren Labels LaTeX-Code verwenden bietet sich das Paket `latex2exp` an.

Wie Sie ja oben gesehen haben können Sie mit `ggplot2::theme()` quasi jeden Teil Ihrer Grafik ändern und die Vorschläge entsprechend einfach implementieren. Um hier Zeit zu sparen können Sie, wie oben bereits erwähnt, auch [vorgefertigte Themen](#) verwenden oder Ihr eigenes Thema schreiben und dann immer wiederverwenden.

Im Folgenden werden einige Beispiel-Visualisierungsformen kurz eingeführt, aber nicht im Detail diskutiert. Weitere Ideen und tiefergehende Diskussionen finden Sie z.B. auf der exzellenten Homepage von [Holtz and Healy \(2020\)](#).

### 5.3.2 Streu- oder Blasendiagramm

**Besonders geeignet für:** Zusammenhang von 2 - 3 verhältnis-skalierten Variablen.

**Mögliche Probleme:** Negative Werte können in der Größendimension nicht dargestellt werden.

**Beispiel 1: Zwei Variablen in einem Streudiagramm**

Die dieser Abbildung zugrundeliegenden Daten beschreiben die Handelsoffenheit von Österreich über die Zeit:

```
head(offenheits_daten)
```

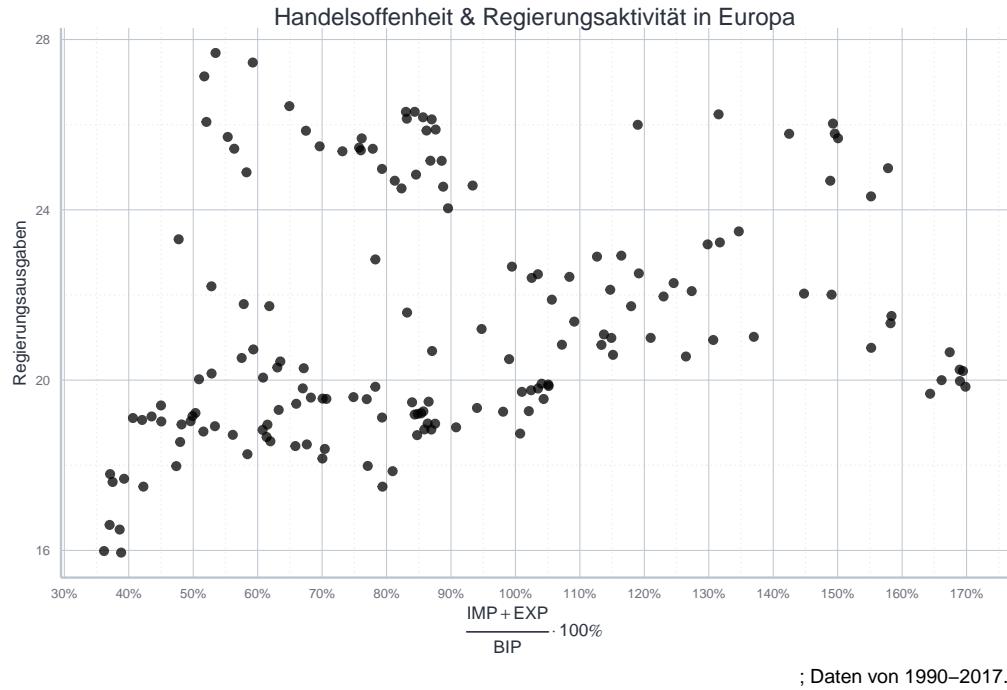
```
#>   year      Land trade_total_GDP gvnt_cons
#> 1: 1991 Österreich     70.04841 18.15780
#> 2: 1992 Österreich     67.63017 18.48991
#> 3: 1993 Österreich     63.26505 19.30042
#> 4: 1994 Österreich     65.98709 19.44437
#> 5: 1995 Österreich     68.25660 19.58966
#> 6: 1996 Österreich     70.08367 19.56574
```

```
streudiagramm <- ggplot2::ggplot(
  data = offenheits_daten,
```

```

mapping = aes(x=trade_total_GDP,
              y=gvnt_cons)
) +
ggplot2::geom_point(alpha=0.75) +
ggplot2::scale_y_continuous(name = "Regierungsausgaben") +
ggplot2::scale_x_continuous(name = TeX("$\\frac{IMP + EXP}{BIP} \\cdot 100\\%$"),
                             breaks = seq(30, 180, 10),
                             labels = scales::percent_format(accuracy = 1, scale = 1))
) +
ggplot2::labs(
  title = "Handelsoffenheit & Regierungsaktivität in Europa",
  caption = "; Daten von 1990–2017."
) +
icaeDesign::theme_icae()
streudiagramm

```



Mit dem Keyword `alpha` in `ggplot2::geom_point(alpha=0.75)` können Sie die Transparenz der Punkte kontrollieren. Gerade bei überlappenden Punkten, bzw. sehr dichten Punktewolken hilft das häufig, das Erscheinungsbild deutlich zu verbessern.

### Beispiel 2: Vier Dimensionen in einem Blasendiagramm

```
head(ausgangsdaten)
```

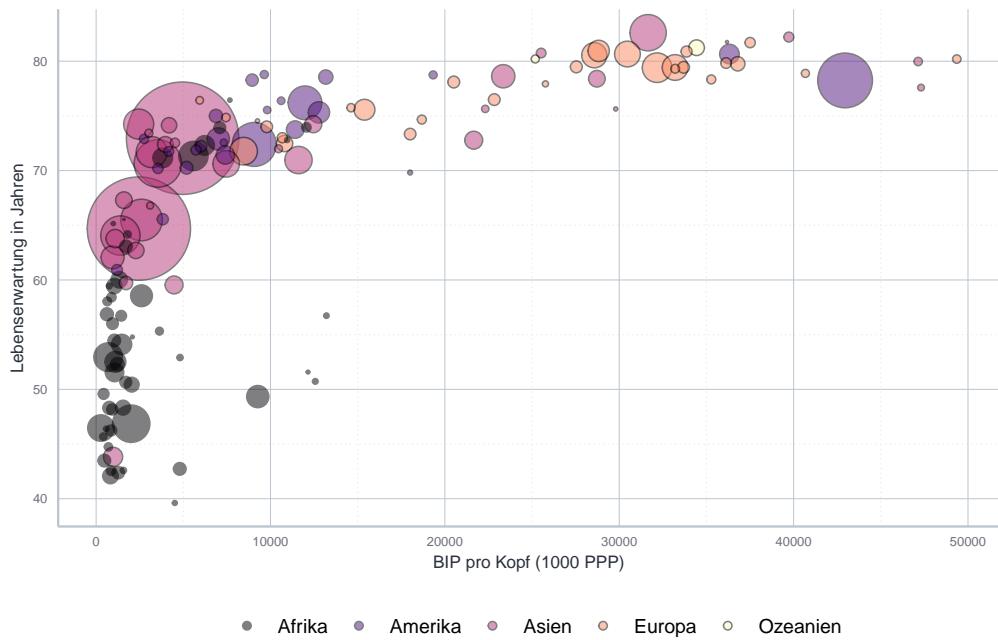
```

#> # A tibble: 6 x 5
#>   country      continent lifeExp      pop gdpPerCap
#>   <fct>        <chr>     <dbl>     <int>     <dbl>
#> 1 China        Asien      73.0    1318683096    4959.
#> 2 India         Asien      64.7    1110396331    2452.

```

```
#> 3 United States Amerika      78.2  301139947  42952.
#> 4 Indonesia     Asien       70.6  223547000  3541.
#> 5 Brazil        Amerika    72.4  190010647  9066.
#> 6 Pakistan      Asien       65.5  169270617  2606.

bubble_plot <- ggplot2::ggplot(
  data = ausgangsdaten,
  mapping = aes(x = gdpPercap,
                 y = lifeExp,
                 size = pop,
                 fill = continent)
) +
  ggplot2::geom_point(
    alpha=0.5, shape=21, color="black"
  ) +
  ggplot2::scale_size(
    range = c(0.1, 24), name="Bevölkerung", guide = FALSE
  ) +
  viridis::scale_fill_viridis(
    discrete=TRUE, option="A"
  ) +
  ggplot2::scale_y_continuous(
    name = "Lebenserwartung in Jahren"
  ) +
  ggplot2::scale_x_continuous(
    name = "BIP pro Kopf (1000 PPP)"
  ) +
  ggplot2::labs(
    caption = "Hinweis: Größe der Blasen repräsentiert Bevölkerungsanzahl. Quelle: Gapminder."
  ) +
  icaeDesign::theme_icae() +
  ggplot2::theme(
    legend.position="bottom",
    plot.caption = element_text(hjust = 0)
  )
bubble_plot
```



Hinweis: Größe der Blasen repräsentiert Bevölkerungsanzahl. Quelle: Gapminder.

### 5.3.3 Linienchart

**Besonders geeignet für:** Veränderungen weniger Variablen über die Zeit.

Die klassischen Liniengraphen haben Sie bereits häufiger kennen gelernt. Im folgenden wollen wir von mehreren Ländern über die Zeit den Durchschnitt berechnen und dann Mittelwert und Standardabweichung über die Zeit visualisieren. Zuerst aggregieren wir die Daten mit den im letzten Kapitel kennen gelernten Funktionen:

```
head(arbeitslosen_daten)
```

```
#>   year iso3c unemp_rate population_ameco      Gruppe
#> 1: 1995  AUT      4.2        7948.28 Kernländer
#> 2: 1996  AUT      4.7        7959.02 Kernländer
#> 3: 1997  AUT      4.7        7968.04 Kernländer
#> 4: 1998  AUT      4.7        7976.79 Kernländer
#> 5: 1999  AUT      4.2        7992.32 Kernländer
#> 6: 2000  AUT      3.9       8011.57 Kernländer
```

```
gewichtete_daten <- arbeitslosen_daten %>%
  dplyr::group_by(year, Gruppe) %>%
  dplyr::mutate(population_group=sum(population_ameco)) %>%
  dplyr::ungroup() %>%
  dplyr::mutate(pop_rel_group=population_ameco / population_group) %>%
  dplyr::group_by(year, Gruppe) %>%
  dplyr::summarise(
    unemp_rate_mean=weighted.mean(unemp_rate,
                                   pop_rel_group),
    unemp_rate_sd=sd(unemp_rate*pop_rel_group)
  ) %>%
```

```
dplyr::ungroup()

#> `summarise()` regrouping output by 'year' (override with `groups` argument)
head(gewichtete_daten)

#> # A tibble: 6 x 4
#>   year Gruppe      unemp_rate_mean unemp_rate_sd
#>   <dbl> <chr>          <dbl>           <dbl>
#> 1  1995 Kernländer       8.36            2.07
#> 2  1995 Peripherieländer 13.9             3.03
#> 3  1996 Kernländer       8.74            2.26
#> 4  1996 Peripherieländer 13.7             2.94
#> 5  1997 Kernländer       8.95            2.46
#> 6  1997 Peripherieländer 13.1             2.80
```

Nun erstellen wir den Plot. Die Markierung für die Standardabweichung fügen wir mit der Funktion `ggplot2::geom_ribbon()` ein, der wir mit `ymin` und `ymax` jeweils das obere und untere Ende der einzufärbenden Region als Argument übergeben. Da wir bereits eine Legende für den Mittelwert haben deaktivieren wir die Legende für die Markierung mit dem Argument `show.legend=FALSE`.

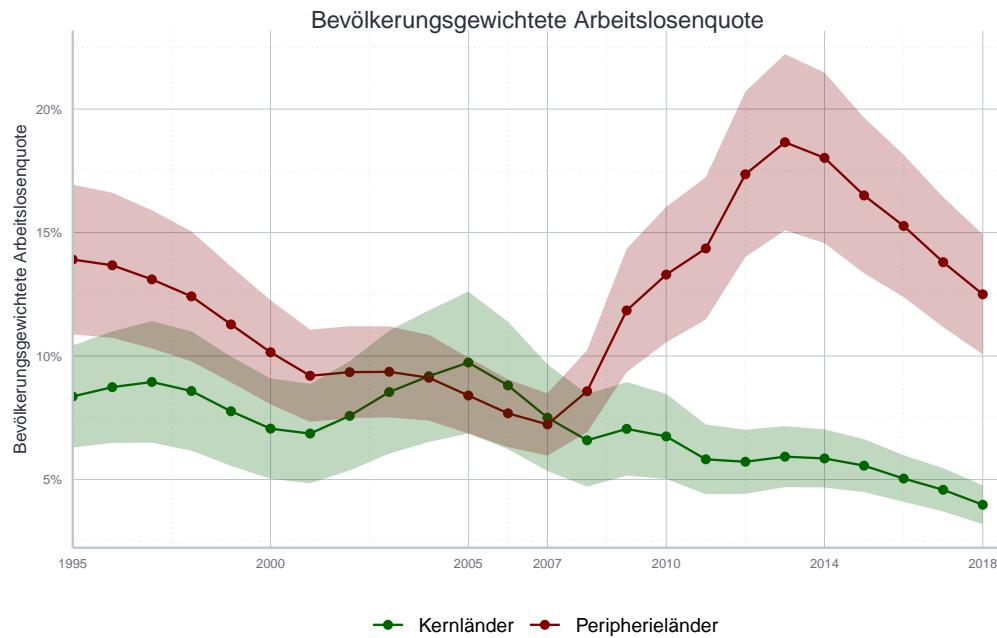
```
x_axis_breaks <- c(1995, 2000, 2005, 2007, 2010, 2014, 2018)

arbeitslosen_plot <- ggplot2::ggplot(
  data = gewichtete_daten,
  mapping = aes(x=year,
                 y=unemp_rate_mean,
                 color=Gruppe)
) +
  ggplot2::geom_point() +
  ggplot2::geom_line() +
  ggplot2::geom_ribbon(
    aes(ymin=unemp_rate_mean-unemp_rate_sd,
        ymax=unemp_rate_mean+unemp_rate_sd,
        linetype=NA, fill=Gruppe),
    alpha=0.25,
    show.legend = FALSE) +
  ggplot2::ylab("Bevölkerungsgewichtete Arbeitslosenquote") +
  icaeDesign::scale_color_icae(
    palette = "mixed",
    aesthetics=c("color", "fill"))
) +
  ggplot2::labs(
    title = "Bevölkerungsgewichtete Arbeitslosenquote",
    caption = "Quelle: Gräßner et al. (2019, CJE)")
) +
  ggplot2::scale_x_continuous(
```

```

breaks=x_axis_breaks,
expand = expansion(
  mult = c(0, 0), add = c(0, 0.5)
)
) +
ggplot2::scale_y_continuous(
  labels = scales::percent_format(accuracy = 1, scale = 1)
) +
icaeDesign::theme_icae() +
ggplot2::theme(axis.title.x = element_blank())
arbeitslosen_plot

```



Quelle: Gräßner et al. (2019, CJE)

Die Grafik stammt aus Gräßner et al. (2020). Bei den Kernländern handelt es sich um Österreich, Belgien, Finnland, Luxemburg, Deutschland und Holland. Die Periphereländer sind Griechenland, Irland, Italien, Portugal und Spanien.

### 5.3.4 Histogramme und Dichteplots

**Besonders geeignet für:** Verteilung einer Variable.

**Mögliche Probleme:** Die Breite der Balken hat in der Regel einen großen Einfluss auf das Erscheinungsbild und die Botschaft der Grafik. Die Entscheidung ist nicht einfach und es gibt [mehrere Heuristiken](#).

**Hinweis:** Wenn Sie extrem viele Datenpunkte haben können Sie die Daten als stetig interpretieren und gleich eine Wahrscheinlichkeitsdichte auf Basis Ihrer Daten berechnen. Dann sparen Sie sich das Problem der Balkenbreite.

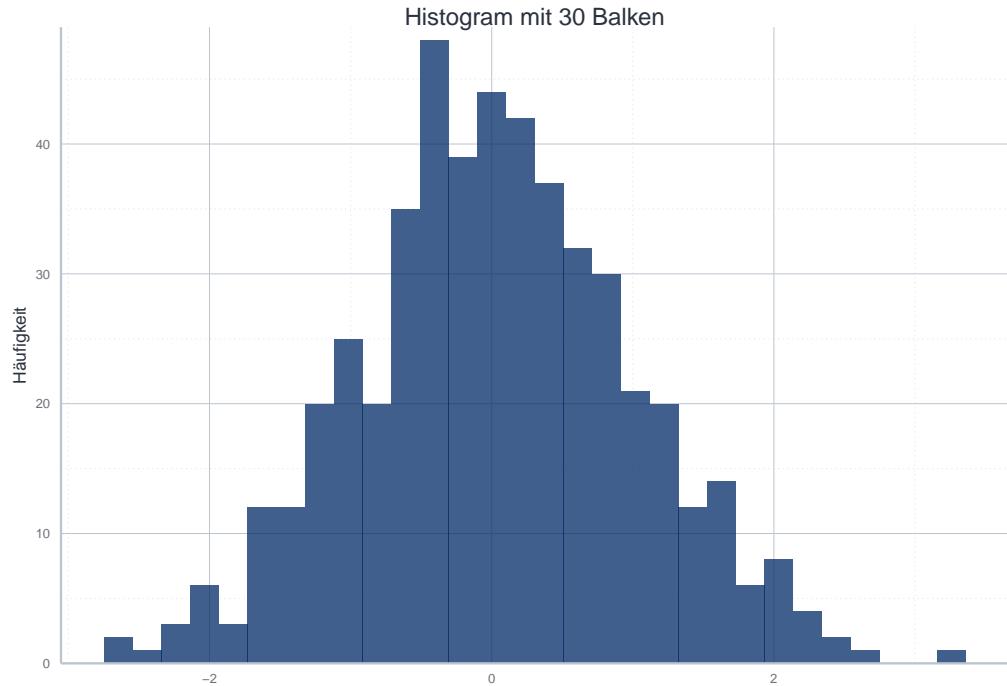
#### Beispiel 1: Einfaches Histogramm

```
head(histogram_daten)
```

```
#>      x
```

```
#> 1 -0.56047565
#> 2 -0.23017749
#> 3 1.55870831
#> 4 0.07050839
#> 5 0.12928774
#> 6 1.71506499

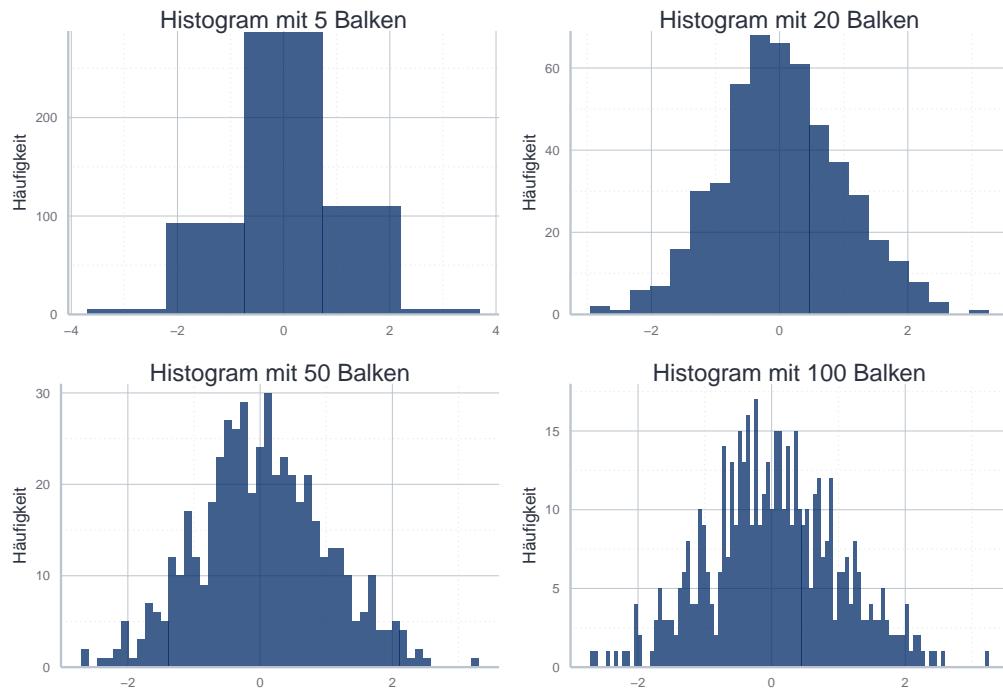
ggplot2::ggplot(data = histogram_daten,
  mapping = aes(x=x)) +
  ggplot2::geom_histogram(alpha=0.75, color=NA, fill="#002966") +
  ggplot2::scale_y_continuous(name = "Häufigkeit",
    expand = expansion(c(0, 0), c(0, 1))) +
  ggplot2::gtitle("Histogram mit 30 Balken") +
  icaeDesign::theme_icae() +
  ggplot2::theme(axis.title.x = element_blank())
```



Im Folgenden sehen Sie auch den großen Effekt unterschiedlicher Balkendicken:

```
bin_size <- c(5, 20, 50, 100)
hist_list <- list()
for (i in 1:length(bin_size)){
  hist_list[[i]] <- ggplot2::ggplot(data = histogram_daten,
    mapping = aes(x=x)) +
    ggplot2::geom_histogram(alpha=0.75, color=NA, fill="#002966", bins = bin_size[i]) +
    ggplot2::scale_y_continuous(name = "Häufigkeit",
      expand = expansion(c(0, 0), c(0, 1))) +
    ggplot2::gtitle(paste0("Histogram mit ", bin_size[i], " Balken")) +
    icaeDesign::theme_icae() +
    ggplot2::theme(axis.title.x = element_blank())
```

```
}
ggpubr::ggarrange(plotlist = hist_list, ncol = 2, nrow = 2)
```



### Beispiel 2: DichteVerteilung von Exportkörben

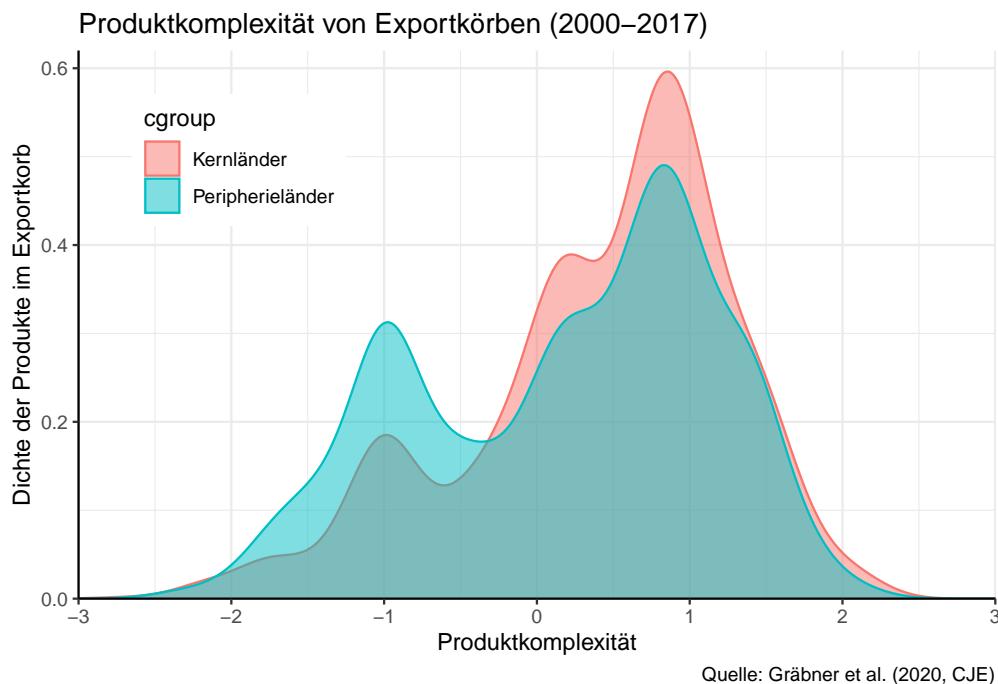
Diese Daten beschreiben die Zusammensetzung der Exportkörbe von Deutschland, Finnland und China bezüglich ihrer ökonomischen Komplexität:

```
#>          cgroup commoditycode      pci   exp_share
#> 1: Kernländer          0101  0.06424262 0.0001312370
#> 2: Peripherieländer    0101  0.06424262 0.0004639794
#> 3: Kernländer          0102 -0.49254290 0.0005162508
#> 4: Peripherieländer    0102 -0.49254290 0.0003700469
#> 5: Kernländer          0103  0.51082386 0.0005324995
#> 6: Peripherieländer    0103  0.51082386 0.0004082251
```

Aufgrund der großen Datenmenge kann die Verteilung der Exporte hier direkt über die Dichte dargestellt werden. Hierzu wird die Funktion `ggplot2::geom_density()` verwendet. Um die Güter nach ihrem tatsächlichen Exportwert zu gewichten verwenden wir die Ästhetik `weight`:

```
ggplot2::ggplot(data = exportzusammensetzung,
  mapping = aes(
    x=pci,
    color=cgroup,
    fill=cgroup)
  ) +
  ggplot2::geom_density(
  mapping = aes(weight=exp_share),
  alpha=0.5
  ) +
```

```
ggplot2::labs(
  title = "Produktkomplexität von Exportkörben (2000–2017)",
  caption = "Quelle: Gräßner et al. (2020, CJE)"
) +
  ggplot2::ylab("Dichte der Produkte im Exportkorb") +
  ggplot2::xlab("Produktkomplexität") +
  ggplot2::scale_y_continuous(limits = c(0, 0.62), expand = c(0, 0)) +
  ggplot2::scale_x_continuous(limits = c(-3, 3), expand = c(0, 0)) +
  ggplot2::theme_bw() +
  ggplot2::theme(legend.position = c(0.175, 0.8),
    panel.border = element_blank(),
    axis.line = element_line())
```



Auch diese Abbildung stammt ursprünglich aus Gräßner et al. (2020).

### 5.3.5 Balkendiagramme

**Besonders geeignet für:** Vergleich der Ausprägung der gleichen Variable in mehreren Gruppen.

Balkendiagramme sind auf den ersten Blick sehr ähnlich zu Histogrammen, sie geben jedoch nicht notwendigerweise Häufigkeiten an. Sie können häufig als Substitut für die zu vermeidenden **Kuchendiagramme** verwendet werden.

**Beispiel: Balkendiagramm für kumulierte Wachstumsraten in mehreren Ländern**

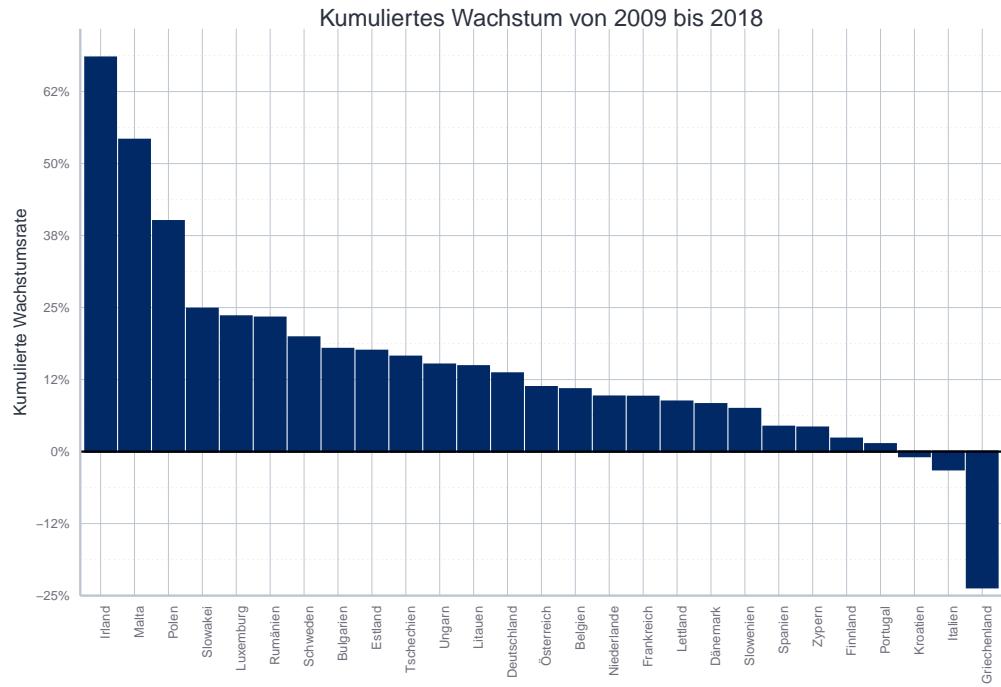
Eine häufige Herausforderung ist es, die Balken nach Größe zu sortieren. Das geht mit der Funktion `reorder()`, die sie innerhalb der Funktion `aes()` anwenden:

```
cum_growth_countries_full <- ggplot2::ggplot(
  data = daten_cum_growth) +
  ggplot2::geom_bar(
    aes(x=reorder(Land, -Wachstum.Land.kum)),
```

```

y=Wachstum.Land.kum),
color="#002966", fill="#002966",
stat = "identity"
) +
ggplot2::ylab("Kumulierte Wachstumsrate") +
ggplot2::ggtitle("Kumulierte Wachstumsrate von 2009 bis 2018") +
ggplot2::geom_hline(yintercept = 0) +
ggplot2::scale_y_continuous(
limits = c(-25, max(daten_cum_growth$Wachstum.Land.kum) + 5),
breaks = seq(-25, max(daten_cum_growth$Wachstum.Land.kum) + 5,
by=12.5),
expand = c(0, 0),
labels = scales::percent_format(accuracy = 1, scale = 1)
) +
icaeDesign::theme_icae() +
ggplot2::theme(axis.text.x = element_text(angle = 90, hjust = 1),
axis.title.x = element_blank(),
legend.position = "none")
cum_growth_countries_full

```



Die Abbildung stammt aus Kapeller et al. (2019), einer Studie, die sich mit Polarisierungstendenzen in Europa und möglichen Gegenmaßnahmen auseinandersetzt.

### 5.3.6 Kuchendiagramme

A table is nearly always better than a dumb pie chart; the only worse design than a pie chart is several of them, for then the viewer is asked to compare quantities located in spatial disarray both within and between charts [...] Given their low density and failure to order numbers along a visual dimension, pie

charts should never be used.

— Edward Tufte

Es gibt keine kontraproduktiveren Abbildungen als Kuchendiagramme. Entsprechend sollten Sie diese auch **nie** verwenden. Es gibt für jeden möglichen Anwendungsfall mit Sicherheit bessere Alternativen.

Warum Kuchendiagramme so grausig sind können Sie [hier](#), [hier](#), [hier](#) oder [hier](#) nachlesen.

### 5.3.7 Zusammenfassung

Tabelle 5.1 fasst die hier diskutierten Visualisierungsmöglichkeiten noch einmal kurz zusammen.

Table 5.1: Visualisierungsmöglichkeiten in R.

Art	Anwendungsgebiet	Relevante Funktion aus ggplot2
Balkendiagramm	Vergleich von Werten	geom_bar()
Linienchart	Dynamiken	geom_line(), geom_ribbon()
Histogram	Verteilungen weniger Variablen	geom_bar(), geom_hist(), geom_density()
Streu- und Blasendiagramm	Zusammenhänge zwischen 2-4 variablen	geom_point()
Kuchendiagramm	Nichts	Keine

## 5.4 Beispiele aus der Praxis und fortgeschrittene Themen

Die folgenden Arbeitsschritte tauchen in der Praxis sehr häufig auf und werden deshalb in etwas größerem Detail besprochen.

### 5.4.1 Regressionsgerade

In diesem Unterabschnitt werden einige Themen aus Kapitel ?? zum Thema Regressionsanalyse vorausgesetzt. Falls Sie noch nichts von linearen Regressionen gehört haben können Sie diesen Abschnitt einfach überspringen.

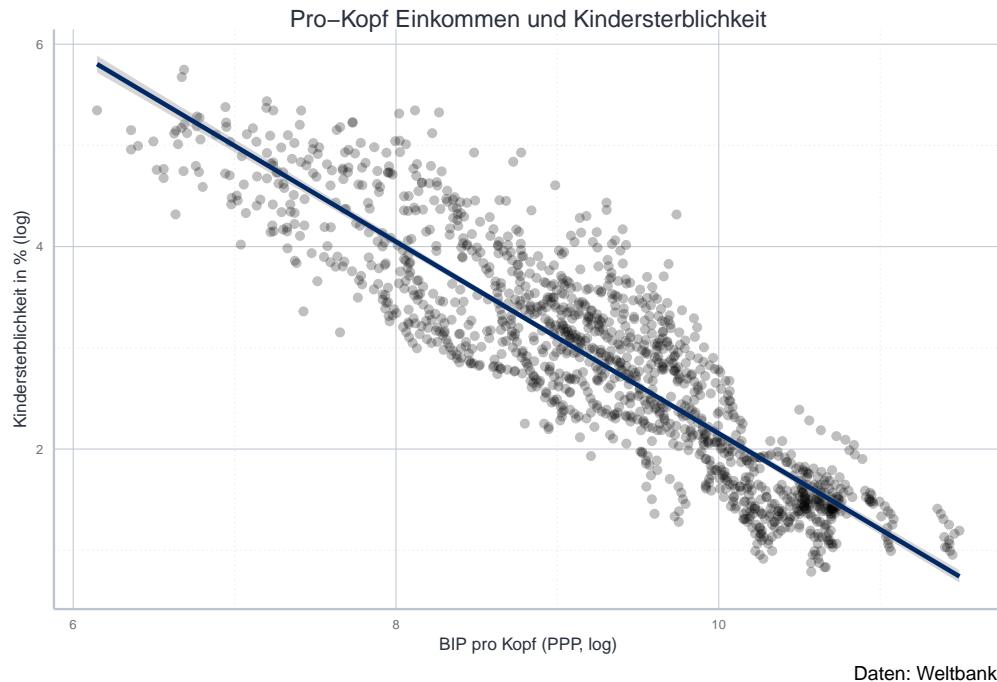
Oftmals möchten wir die Ergebnisse einer Regression in den Daten abbilden. Im einfachsten Falle soll es nur die aus einer linearen Regression resultierenden Gerade sein. Das können wir dann ganz einfach als eigenen Layer mit der Funktion `ggplot2::geom_smooth(method="lm")` hinzufügen. Mit den weiteren Argumenten können wir z.B. die Farbe der Linie (`color=black`) oder die Standardfehler um die Linie deaktivieren (`se=FALSE`):

```
mort_rate_plot <- ggplot2::ggplot(data = development_data,
  mapping = aes(x=log(GDP_PPPpc),
  y=log(MORTRATE))
) +
  ggplot2::geom_point(alpha=0.25) +
  ggplot2::labs(
    title = "Pro-Kopf Einkommen und Kindersterblichkeit",
    caption = "Daten: Weltbank."
) +
  ggplot2::xlab("BIP pro Kopf (PPP, log)") +
  ggplot2::ylab("Kindersterblichkeit in % (log)") +
```

```
icaeDesign::theme_icae()

mort_rate_plot + ggplot2::geom_smooth(method = "lm",
                                      color="#002966",
                                      se = TRUE)
```

#> `geom\_smooth()` using formula 'y ~ x'



Alternativ kann die Gerade auch mit Hilfe der Funktion `ggplot2::geom_abline()` eingezeichnet werden. Dazu müssen wir Regression vorher aber explizit mit `lm()` durchführen:

```
lm_obj <- lm(log(MORTRATE) ~ log(GDP_PPPpc),
              data = development_data)
summary(lm_obj)

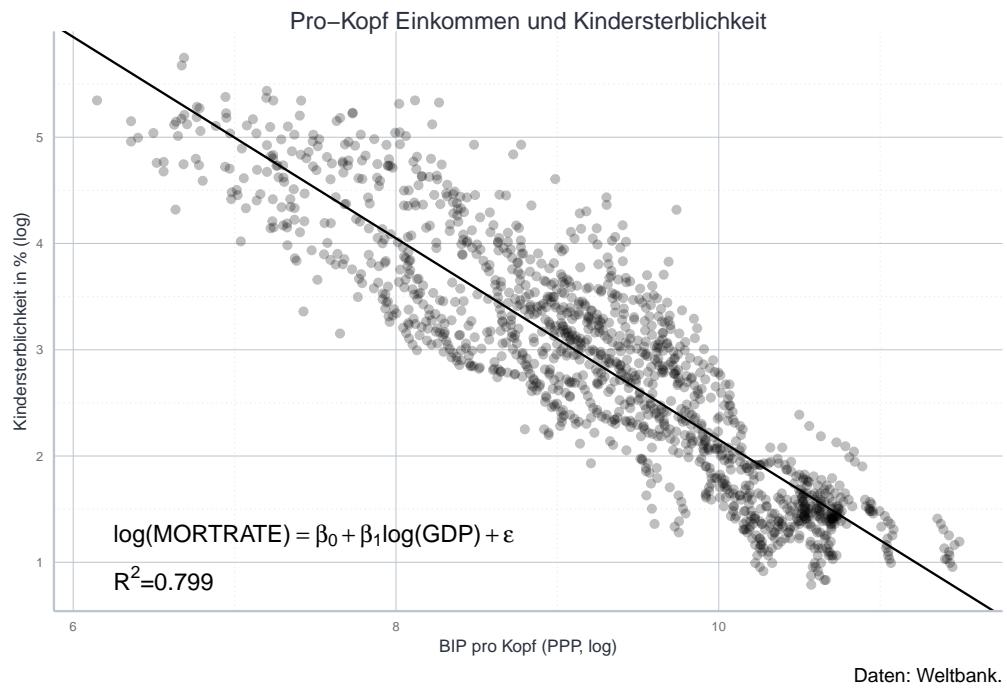
#>
#> Call:
#> lm(formula = log(MORTRATE) ~ log(GDP_PPPpc), data = development_data)
#>
#> Residuals:
#>    Min      1Q   Median      3Q     Max
#> -1.23149 -0.38749 -0.04103  0.35433  1.91519
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept) 11.62670   0.12008  96.83   <2e-16 ***
#> log(GDP_PPPpc) -0.94723   0.01287 -73.62   <2e-16 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
#>
#> Residual standard error: 0.5012 on 1363 degrees of freedom
#> Multiple R-squared:  0.799, Adjusted R-squared:  0.7989
#> F-statistic:  5420 on 1 and 1363 DF,  p-value: < 2.2e-16
```

Häufig möchten wir auch noch die Regressionsgleichung im Plot abbilden, und eventuell Kennzahlen der Regression, wie das  $R^2$  hinzufügen. Das können wir mit der Funktion `ggplot2::annotate()` machen. Als erstes Argument müssen wir mit `geom` die Art der Anmerkung spezifizieren (in diesem Falle: `geom='text'`). Danach werden über `x` und `y` die Koordinaten angegeben. Über `label` wird dann der eigentliche Text angegeben, der über `hjust` wie oben beschrieben noch formatiert werden kann.

Da eine Regressionsgleichung in der Regel leichter in LaTeX zu schreiben ist, empfiehlt sich hier die Verwendung der Funktion `\TeX()` aus dem Paket `latex2exp` (Meschiari, 2015). Hier können wir quasi normalen LaTeX-Code verwenden, müssen aber das häufig verwendete \ als \\ schreiben, damit es als \ interpretiert wird:

```
reg_eq <- "$\\log(MORTRATE) = \\beta_0 + \\beta_1 \\log(GDP) + \\epsilon$"
rsq <- paste0("$R^2=", round(summary(lm_obj)[["r.squared"]], 3), "$")
mort_rate_plot_marked <- mort_rate_plot +
  ggplot2::geom_abline(
    intercept = lm_obj[["coefficients"]][1],
    slope = lm_obj[["coefficients"]][2]) +
  ggplot2::annotate(geom = "text",
    x = 6.25,
    y = 1.25, hjust = 0,
    label = TeX(reg_eq)) +
  ggplot2::annotate(geom = "text",
    x = 6.25,
    y = 0.85, hjust = 0,
    label = TeX(rsq))
mort_rate_plot_marked
```



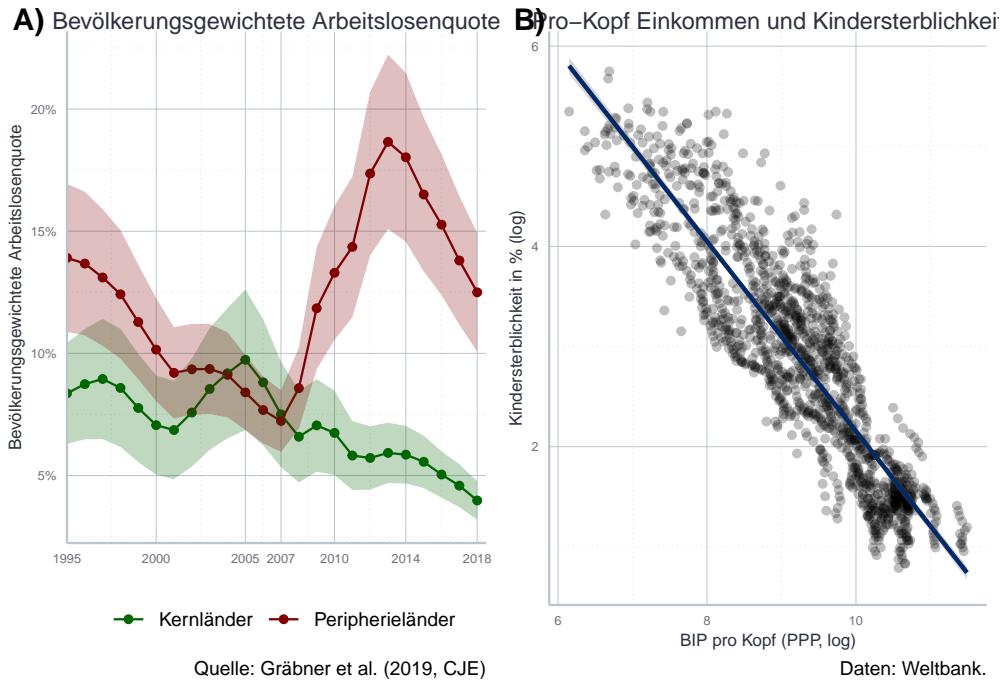
### 5.4.2 Mehrere Plots in einer Abbildung

Sehr häufig möchten wir in einer Grafik mehrere Plots unterbringen. Das ist mit dem Paket `ggpubr` (Kassambara, 2019) leicht zu machen. Dieses Paket bietet zahlreiche Gestaltungsmöglichkeiten. Für mehrere Plots ist die Funktion `ggpubr::ggarrange()` das Richtige. Sie akzeptiert zunächst einmal eine beliebige Anzahl an `ggplot2`-Objekten (oder eine Liste solcher Objekte über das Argument `plotlist`). Danach können noch einige optionale Argumente verwendet werden.

Die Argumente `ncol` bzw. `nrow` spezifizieren die Anzahl der Plots in einer Reihe, bzw. einer Spalte. Mit `labels` können Sie Anmerkungen wie 'a)', 'b)' hinzufügen und mit `font.label` die Schriftgröße und -art bestimmen. Mit `common.legend` können Sie angeben ob die Plots eine gemeinsame Legende haben sollen, oder in jedem Plot die plot-spezifische Legende abgebildet werden soll. Die Position der Legenden kann darüber hinaus über das Argument `legend` mit `top`, `bottom`, `left` oder `right` spezifiziert werden:

```
ggpubr::ggarrange(arbeitslosen_plot,
  mort_rate_plot+ggplot2::geom_smooth(color="#002966", method = "lm"),
  ncol = 2,
  labels = c("A)", "B"),
  font.label = list(face="bold"))
```

```
#> `geom_smooth()` using formula 'y ~ x'
```



### 5.4.3 Mehr zu den Skalen: `ggplot2::expansion()` und Skalentransformation

Häufig möchten Sie Ihre Skalen transformieren.

Bei eigentlich jedem Plot stehen Sie vor der Frage wie Sie mit den hässlichen Rändern umgehen, die `ggplot2::ggplot` standardmäßig an beide Enden der Achsen hinzufügt. Wir haben oben zwar bereits gelernt, dass wir diese Ränder mit `expand=c(0, 0)` innerhalb der Funktion `ggplot2::scale_*_continuous()` abschalten können, aber manchmal wollen wir das nur an einer Seite machen. In diesem Fall können wir die Hilfsfunktion `ggplot2::expansion()` verwenden. Sie akzeptiert zwei Argumente, `mult` und `add`, die wie oben beschrieben funktionieren. Entsprechend sind die folgenden beiden Aufrufe äquivalent:

```
ggplot2::scale_y_continuous(expand = c(0, 0))
ggplot2::scale_y_continuous(expand = expansion(mult = 0, add = 0))
```

Allerdings kann `ggplot2::expansion()` auch jeweils einen Vektor mit zwei Elementen verarbeiten, wobei dann die erste Zahl für den unteren und die zweite für den oberen Rand steht:

```
ggplot2::scale_y_continuous(expand = expansion(mult = c(0, 0), add = c(0, 2)))
```

Letzterer Code verändert die y-Achse nur in der Länge. Das ist nützlich, wenn wir um den Nullpunkt keinen, aber nach außen einen kleinen Rand haben wollen und wir häufig bei Histogrammen benutzt:

```
dichte_1 <- ggplot2::ggplot(
  data = exportzusammensetzung,
  mapping = aes(
    x=pci,
    color=cgroup,
    fill=cgroup)
) +
  ggplot2::geom_density(
    mapping = aes(weight=exp_share),
```

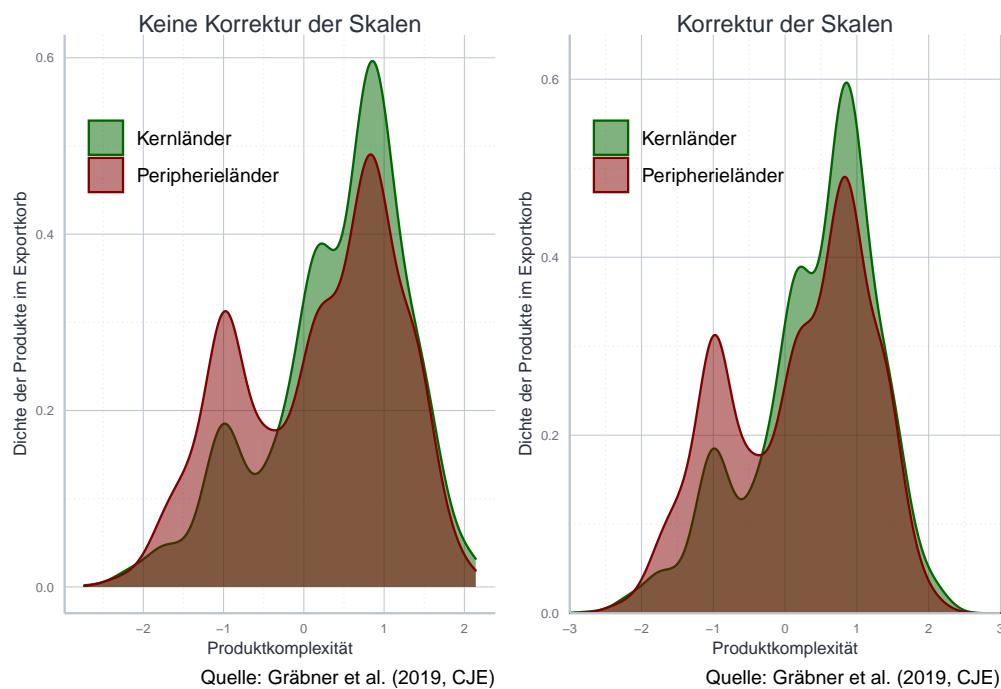
```

alpha=0.5
) +
ggplot2::labs(
  title = "Keine Korrektur der Skalen",
  caption = "Quelle: Gräßner et al. (2019, CJE)"
) +
ggplot2::ylab("Dichte der Produkte im Exportkorb") +
ggplot2::xlab("Produktkomplexität") +
icaeDesign::scale_color_icae(palette = "mixed",
                             aesthetics = c("color", "fill")) +
icaeDesign::theme_icae() +
ggplot2::theme(legend.position = c(0.275, 0.8))

dichte_2 <- dichte_1 +
ggplot2::ggtitle("Korrektur der Skalen") +
ggplot2::scale_y_continuous(limits = c(0, 0.6),
                           expand = expansion(mult = c(0, 0),
                                               add = c(0, 0.05))) +
ggplot2::scale_x_continuous(limits = c(-3, 3),
                           expand = expansion(mult = c(0, 0),
                                               add = c(0, 0)))

ggpubr::ggarrange(dichte_1, dichte_2, ncol = 2)

```



Auch werden Sie häufig die Labels auf Ihren Achsen ändern wollen. Gerade die Transformation hin zu Prozentwerten ist aber nicht immer ganz trivial. Am besten verwenden Sie die Funktion `percent_format()` aus dem Paket `scales` (Wickham, 2018) um das entsprechende Argument `labels` in `ggplot2::scale_*_continuous()` zu spezifizieren.

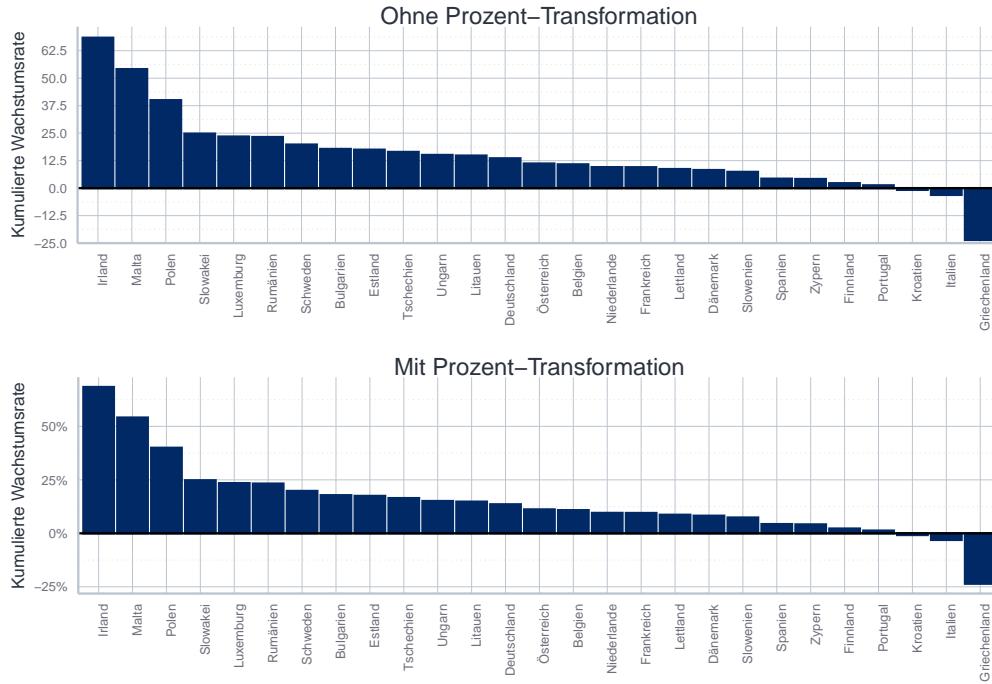
Die Funktion bedarf zweier Argumente `accuracy` und `scale`. Das Argument `accuracy` bezeichnet die Dezimalstelle auf die gerundet werden soll. Dies ist ein Einfallstor für viele Fehler, da die Funktion keine Fehler ausgibt wenn irreführende Werte angegeben werden. Vergleichen Sie immer die Skala vor und nach der Transformation um sicher zu gehen, dass sich keine Fehler eingeschlichen haben!

Das Argument `scale` bezeichnet die Skala in den Daten, also ob die Daten bereits in Prozent angegeben sind (in dem Falle wäre `scale=100`), oder ob der Wert 1 zu 100% korrespondiert (in diesem Falle wäre `scale=1`). Auch hier sollten Sie immer die Ache vor und nach der Transformation vergleichen.

Im Folgenden sehen sie ein Anwendungsbeispiel:

```
cum_growth_countries_full_percent <- cum_growth_countries_full +
  ggplot2::scale_y_continuous(
    labels = scales::percent_format(accuracy = 1, scale = 1)
  )

ggpubr::ggarrange(cum_growth_countries_full + ggplot2::ggtitle("Ohne Prozent-Transformation"),
  cum_growth_countries_full_percent + ggplot2::ggtitle("Mit Prozent-Transformation"),
  nrow = 2
)
```



Die weiteren Argumente sind relativ selbsterklärend und werden in der Regel nicht verwendet. Sie sind ähnlich zu den weiteren Formatierungsfunktionen in dem Paket. Überhaut bietet das Paket `scales` noch viele weitere Hilfsfunktionen an. Wenn Sie Probleme mit Skalierungen haben lohnt sich ein Blick auf die Paket-Homepage.

#### 5.4.4 Mehr zur Farbauswahl

Wie Sie bereits bemerkt haben können Sie in Ihren Abbildungen eine Vielzahl an Farben verwenden. Einen Überblick über alle in R definierten Farben finden Sie dabei leicht im Internet. Besonders attraktiv ist es jedoch, Farben durch ihren HEX Code anzugeben. Durch Angabe des HEX Codes können Sie quasi jede beliebige Farbe in R verwenden. Am einfachsten ist es, im Internet einen *Color Picker* zu verwenden und sich den HEX Code

ausgeben zu lassen und diesen dann in R zu verwenden. Einen empfehlenswerten *Color Picker* finden Sie z.B. unter [https://www.w3schools.com/colors/colors\\_picker.asp](https://www.w3schools.com/colors/colors_picker.asp).

Wenn Sie mehrere Farben verwenden wollen, die gut zueinander passen empfiehlt sich die Verwendung einer Farbpalette. Dabei handelt es sich um eine ‘Sammplung’ von zueinander passenden Farben. Es zahlreiche vordefinierte Paletten im Internet und alle funktionieren ähnlich. Eine sehr bekannte und nützliche Palette ist die so genannte *Viridis*-Palette von Stefan van der Walt and Nathaniel Smith. In der Grundversion schaut sie folgendermaßen aus:



Wenn Sie die Farben Ihrer Ästhetiken gemäß der Viridis-Palette verwenden wollen geht dies durch vordefinierte Funktionen. Für Füllfarben gibt es z.B. die Funktion `scale_fill_viridis()` und für Linienfarben `scale_color_viridis()`. Das gleiche Schema funktioniert aber auch für die meisten anderen Paletten, die Sie im Internet finden können.

Sie können sich auch HEX-Farbcodes direkt ausgeben lassen. Für den Fall von Viridis geht das über die Funktion `viridis::viridis()`. Das einzige notwendige Argument ist `n`. Es spezifiziert die Anzahl der HEX-Codes.

```
viridis::viridis(n = 3) # Drei Farben aus der Viridis-Palette
```

```
#> [1] "#440154FF" "#21908CFF" "#FDE725FF"
```

Darüber hinaus können Sie noch die Transparenz (`alpha`), den Ausschnitt der Palette (`begin` und `end`) und die Richtung der Palette (`direction`) spezifizieren. Mit der Richtung ist gemeint, dass normalerweise eine Palette die Farben von Dunkel nach Hell sortiert, diese Richtung aber natürlich auch umgedreht werden kann. Im folgenden Beispiel sind die Farben zu 50 Prozent transparent und es wird nur die Mitte der Palette verwendet. Zudem wird die Richtung der Palette umgedreht, also die Farben von Hell nach Dunkel sortiert:

```
viridis::viridis(n = 3, begin = 0.25, end = 0.5, direction = -1)
```

```
#> [1] "#21908CFF" "#2C728EFF" "#3B528BFF"
```

Gerade was die Wahl von Farben angeht empfiehlt sich das weitere Selbststudium und Ausprobieren anhand von Visualisierungsanleitungen und Blogs im Internet.

## 5.5 Typische Fehler in der Datenvisualisierung vermeiden

Hier implementieren wir einige der Beispiele aus [Schwabish \(2014\)](#). Eine wunderbare Seite mit typischen Visualisierungsfehlern und wie Sie sie vermeiden können finden Sie [hier](#).

### 5.5.1 Clutterplots und ihre Transformation zum beschrifteten Streudiagramm

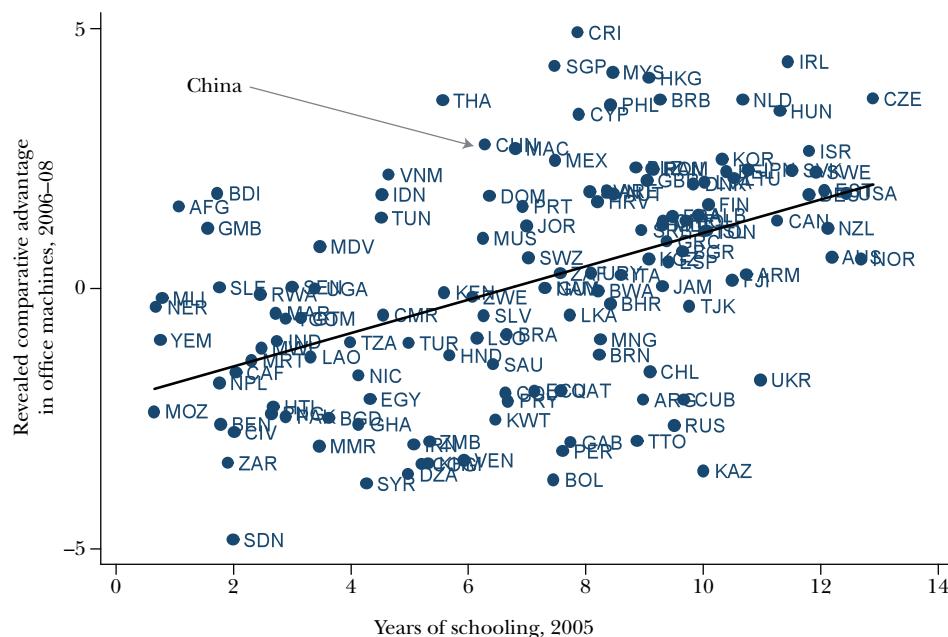
Schauen wir zunächst auf die folgende Abbildung 5.2 aus [Hanson \(2012, S. 55\)](#):

Da sich der Autor zusätzlich nicht erbarmt hat seinen Datensatz zu publizieren, müssen wir auch noch die der Abbildung zugrundeliegenden Daten selbst beschaffen - in diesen Momenten merken Sie wie wichtig es ist, zu jeder Publikation die Daten und den Code für die Abbildungen mit zu veröffentlichen. Zwar wurden die Datenquellen einigermaßen dokumentiert,<sup>12</sup> da es aber leider nicht vollständig nachzuvollziehen ist auf welchen Weltbankdatensatz er sich mit ‘Average years of schooling of the adult population’ bezieht und die genaue Quelle für die Exportdaten auch nicht genannt wurde<sup>13</sup> finden sich in der Replikation natürlich kleinere Abweichungen:

<sup>12</sup>Wie die Daten nacherhoben wurden können Sie bei Interesse über die [Github Repo](#) des Skripts selbst nachlesen.

<sup>13</sup>Hier verwende ich Daten von [The Growth Lab at Harvard University \(2019\)](#), die [hier](#) abzurufen sind.

*Figure 4*  
**Education and Exports of Office Machines**



Source: Author's calculations using (World Bank) World Development Indicators and UN Comtrade.

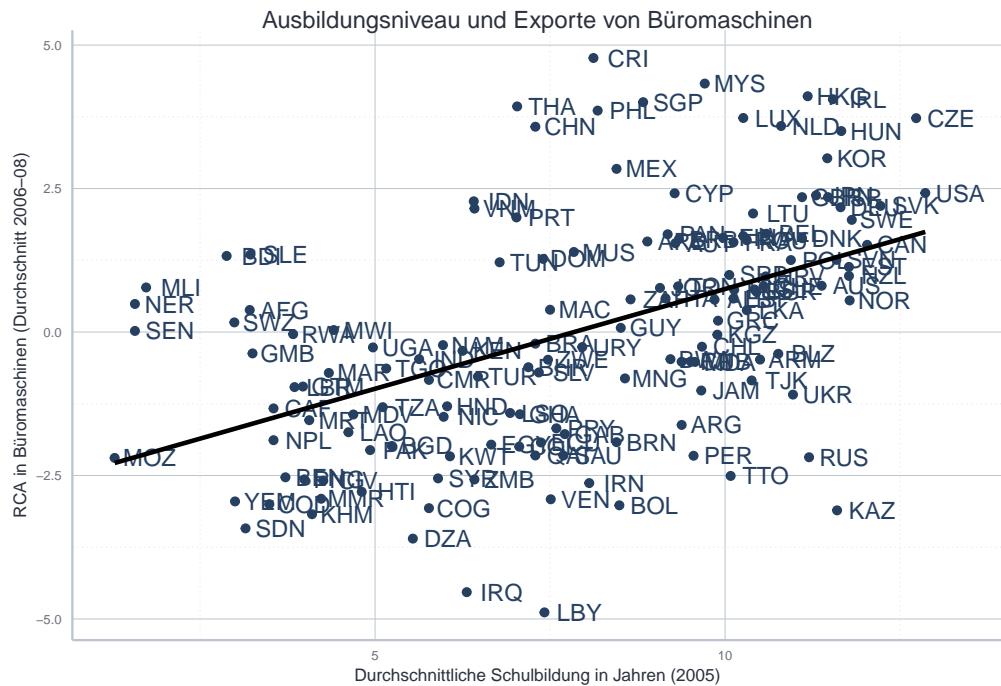
*Notes:* Figure 4 plots countries' revealed comparative advantage in office machines—Standard International Trade Classification (SITC) industry 75—averaged over 2006 to 2008, against the average years of schooling of the adult population in 2005. Revealed comparative advantage in computers is defined as the log ratio of a country's share of world exports of SITC 75 to its share of world exports of all merchandise. The countries are indicated by their World Bank abbreviations.

Figure 5.2: Abbildung nach Hanson, 2012

Zunächst replizieren wir das originale visuelle Verbrechen:

```
ggplot2::ggplot(data = hanson_data,
  mapping = aes(x=schooling, y=rca_purged)) +
  ggplot2::geom_point(color="#264062") +
  ggplot2::geom_text(aes(label=country), nudge_x = 0.5, color="#264062") +
  ggplot2::geom_smooth(method = "lm", se = FALSE, color="black") +
  ggplot2::gtitle("Ausbildungsniveau und Exporte von Büromaschinen") +
  ggplot2::scale_x_continuous(name = "Durchschnittliche Schulbildung in Jahren (2005)") +
  ggplot2::scale_y_continuous(name = "RCA in Büromaschinen (Durchschnitt 2006-08)") +
  icaeDesign::theme_icae()

#> `geom_smooth()` using formula 'y ~ x'
```



Abgesehen davon, dass es einfach hässlich ist so viele Überlappungen zu haben, setzt dieser Graph voraus, dass Sie fließend die `iso3c`-Codes beherrschen und schnell die fünf Länder finden, um die es im Text geht. Das ist nicht sonderlich leser\*innenfreundlich...

Wie Schwabish (2014) bilden wir zunächst einmal die Labels nur für die fünf uns interessierenden Länder ab. Das machen wir, indem wir die Funktion `ggplot2::geom_text()`, welche die Ländernamen abbildet, nicht den Standarddatensatz verwenden lassen, sondern einen reduzierten Datensatz übergeben. In diesem reduzierten Datensatz übersetzen wir die Ländernamen bereits ins Deutsche. Überhaupt ersetzen wir `ggplot2::geom_text()` besser mit `geom_label_repel()` aus dem Paket `ggrepel` (Slowikowski, 2019), welches quasi genauso funktioniert, aber den Text so verschiebt, dass es zu keinen Überschneidungen kommt.

Außerdem wählen wir eine stärkere Farbe für diese Namen aus. Damit es besser zu den Punkten passt, plotten wir die Punkte dieser Länder in der gleichen Farbe, und alle anderen Punkte in einem Grauton. Dazu verwenden wir einfach zwei unterschiedliche Layer, jeweils produziert durch `ggplot2::geom_point()`, aber mit unterschiedlichen Datensätzen.

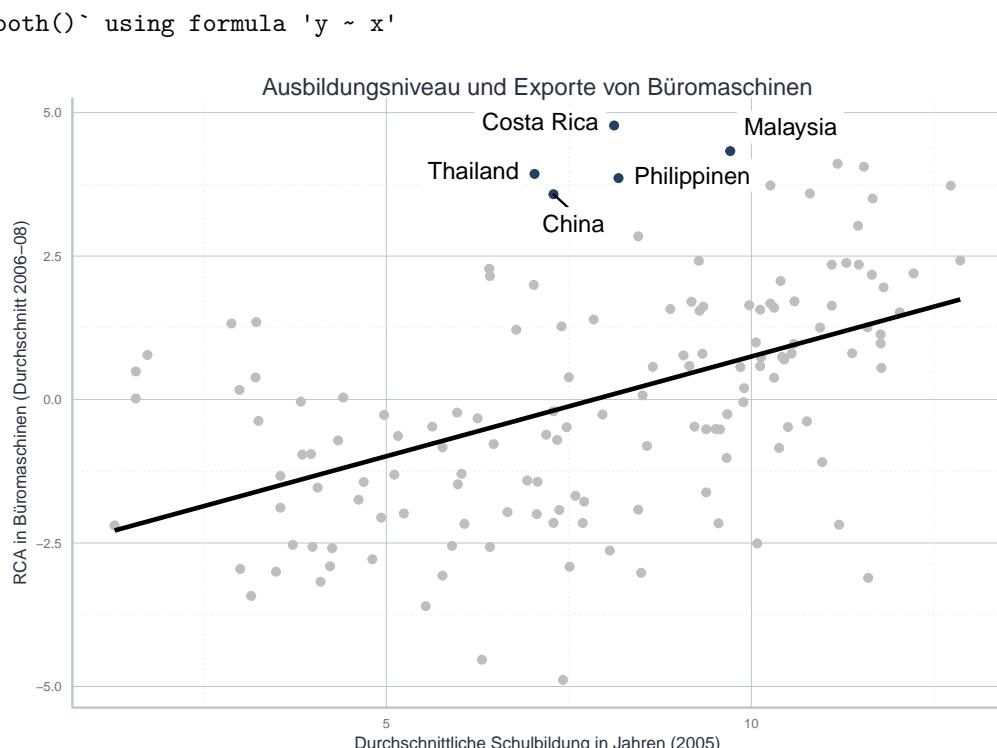
```

interessierende_laender <- countrycode(
  c("China", "Malaysia", "Costa Rica", "Philippines", "Thailand"),
  "country.name", "iso3c")

ggplot2::ggplot(data = hanson_data,
  mapping = aes(x=schooling, y=rca_purged)) +
  ggplot2::geom_point(
    data = filter(hanson_data,
      country %in% interessierende_laender),
    color="#264062") +
  ggplot2::geom_point(
    data = filter(hanson_data,
      !country %in% interessierende_laender),
    color="grey") +
  ggrepel::geom_label_repel(
    data = filter(hanson_data,
      country %in% interessierende_laender),
    aes(label=countrycode(country, "iso3c", "country.name.de")),
    color="black", label.size = NA
  ) +
  ggplot2::geom_smooth(method = "lm", se = FALSE, color="black") +
  ggplot2::ggtitle("Ausbildungsniveau und Exporte von Büromaschinen") +
  ggplot2::scale_x_continuous(name = "Durchschnittliche Schulbildung in Jahren (2005)") +
  ggplot2::scale_y_continuous(name = "RCA in Büromaschinen (Durchschnitt 2006-08)") +
  icaeDesign::theme_icae()

#> `geom_smooth()` using formula 'y ~ x'

```



Wie Sie merken werden diese Farben außerhalb von `mapping` definiert. Denn die Farben sollen ja für alle Variablen gleich sein, es handelt sich hier also nicht um ein *aesthetic mapping*, welches ja die Farbe abhängig vom Variablenwert vergeben würde.

Dies ist wieder ein schönes Beispiel für eine Grafik, die sehr davon profitiert, wenn man die abgebildeten Punkte auf das wirklich Wesentliche reduziert.

### 5.5.2 Ein ‘unbalancierter’ Plot

An anderes schönes Beispiel ist die folgende Abbildung 5.3, die angeblich von der NY Times und der OECD verwendet wurde. Zwar funktionieren alle angegebenen Links nicht mehr und der genaue Datensatz, welcher der Abbildung zurundeliegt bleibt ebenfalls unerwähnt (Sie sehen die Verbesserungsmöglichkeiten), allerdings ist er ein schönes Negativbeispiel:

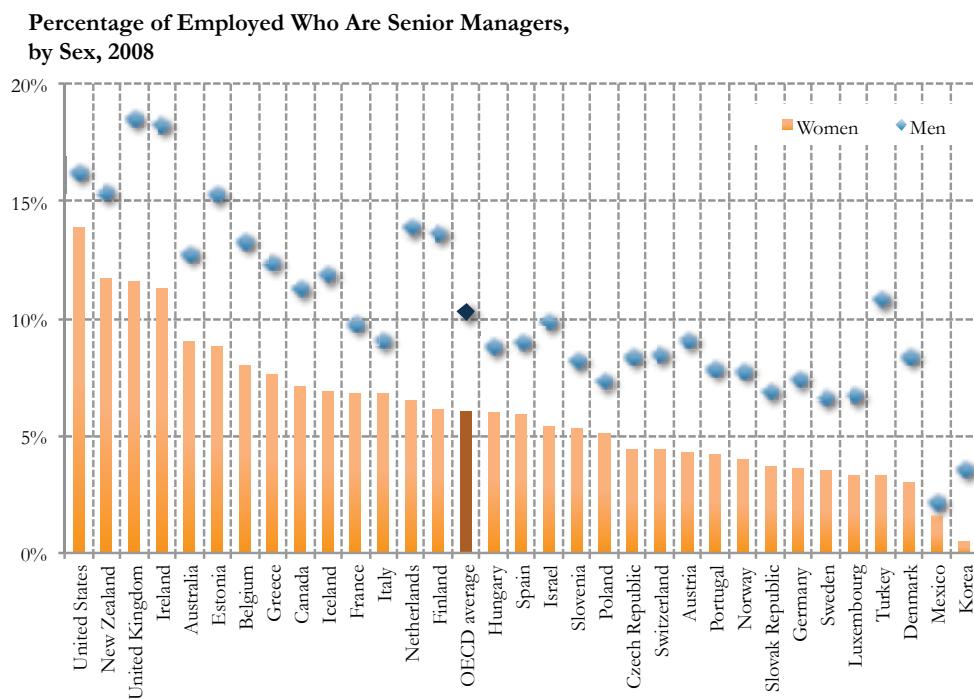


Figure 5.3: Beispiel für einen ‘unbalancierten’ Plot

Selbst mit der Beschreibung im Text ist schwer verständlich was uns diese Abbildung jetzt genau sagen soll. Wahrscheinlich versucht der Autor oder die Autorin zu zeigen, dass Frauen weniger in Führungspositionen vertreten sind als Männer. Warum dann allerdings die Werte für Frauen mit mehr Fläche dargestellt sind als die der Männer bleibt genauso schleierhaft wie die Begründung für die abartige Farbkombination und die übertriebenen Gitter. Zum Glück können wir die eigentlich wichtige Message viel besser darstellen!

Zuallererst geben wir mit [OECD \(2019\)](#) einmal die Quellen für unsere Daten korrekt an. Wie von [Schwabish \(2014\)](#) vorgeschlagen würde sich ein Balkendiagramm in dem die Balken von Männern und Frauen direkt nebeneinander liegen, gut anbieten. Hier nutzen wir aber die Change eine etwas exquisitere Darstellungsform kennen zu lernen, den [Lollipop-Graph](#).

Zuerst müssen jedoch die Daten in einen nutzbaren Zustand gebracht werden:

Diese Daten sehen im Rohzustand (nach Auswahl der relevanten Spalten) so aus:

```
head(oecd_data)
```

```
#>   COU   Sex Value
#> 1: AUT   Men   6.2
#> 2: AUT Women   2.9
#> 3: BEL   Men  10.4
#> 4: BEL Women   5.8
#> 5: CZE   Men   6.8
#> 6: CZE Women   3.6
```

Wir wissen ja aus letztem Kapitel wie wir hiermit umzugehen haben:

```
oecd_data <- oecd_data %>%
  tidyr::pivot_wider(names_from = "Sex",
                     values_from = "Value",
                     id_cols = "COU")
head(oecd_data)
```

```
#> # A tibble: 6 x 3
#>   COU     Men Women
#>   <chr> <dbl> <dbl>
#> 1 AUT      6.2   2.9
#> 2 BEL     10.4   5.8
#> 3 CZE      6.8   3.6
#> 4 DNK      3.4   1.4
#> 5 FIN      4.1   2.1
#> 6 FRA     9.3   4.6
```

Auch möchten wir die Ländernamen noch anpassen. Hier haben wir aber einen Fall in dem wir nicht einfach blind die Funktion `countrycode::countrycode()` verwenden können: zum einen enthält unser Datensatz das ‘Land’ OAVG, was der Durchschnitt aller OECD Länder ist. Diesen müssen wir separat übersetzen. Wir erledigen das mit der Funktion `ifelse()`. Diese Funktion erlaubt bedingte Befehle: wir formulieren als erstes Argument einen Test, als zweites Argument den Wert, den die Funktion ausgegen soll, wenn der Test erfüllt wird und als drittes Argument den Wert wenn der Test nicht erfüllt ist, so wie in folgendem Beispiel:

```
x <- 2
ifelse(x>2, "x ist größer als 2!", "x ist nicht größer als 2!")

#> [1] "x ist nicht größer als 2!"

x <- 4
ifelse(x>2, "x ist größer als 2!", "x ist nicht größer als 2!")

#> [1] "x ist größer als 2!"
```

Zudem ist die offizielle Bezeichnung für Südkorea “Korea, Republik von”. Das macht sich in einer Abbildung nicht sonderlich gut, daher passen wir auch das manuell an:

```
oecd_data_plot <- oecd_data %>%
  dplyr::mutate(COU = ifelse(COU=="OAVG", "OECD Durchschnitt",
                             countrycode::countrycode(COU, "iso3c", "country.name.de"))),
```

```
COU = ifelse(COU=="Korea, Republik von", "Südkorea", COU))
```

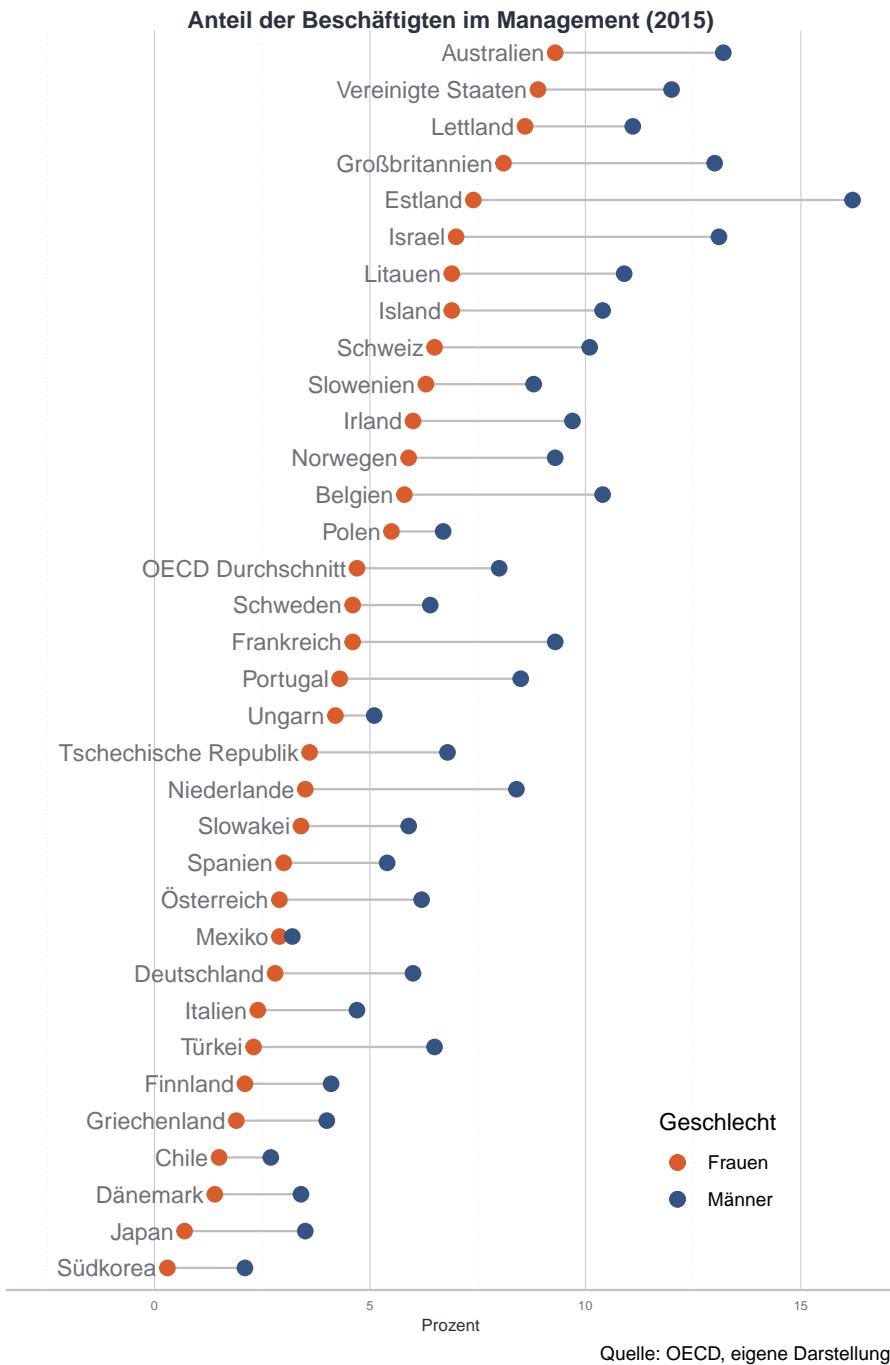
Mit diesen erstellen wir den Lollipop-Graphen folgendermaßen:<sup>14</sup>

```
farbe_m <- "#355383"
farbe_w <- "#d95d2c"

ggplot2::ggplot(oecd_data_plot) +
  ggplot2::geom_segment(aes(x=reorder(COU, Women),
                             xend=COU,
                             y=Women,
                             yend=Men),
                        color="grey") +
  ggplot2::geom_point(
    aes(x=COU,
        y=Women,
        color="Frauen"),
    size=3) +
  ggplot2::geom_point(
    aes(x=COU,
        y=Men,
        color="Männer"),
    size=3) +
  ggplot2::scale_color_manual(values = c("Männer"=farbe_m, "Frauen"=farbe_w), name="Geschlecht") +
  ggplot2::geom_text(
    aes(x=COU, y=Women, label=COU),
    nudge_y = -0.25, hjust=1, color=rgb(110, 113, 123, maxColorValue = 255)
  ) +
  ggplot2::scale_y_continuous(name = "Prozent",
                              expand = expansion(mult = c(0, 0),
                                                 add = c(3.5, 1)))
  ) +
  ggplot2::coord_flip() +
  ggplot2::labs(title = "Anteil der Beschäftigten im Management (2015)",
               caption = "Quelle: OECD, eigene Darstellung.") +
  icaeDesign::theme_icae() +
  ggplot2::theme(
    panel.grid.major.y = element_blank(),
    panel.grid.minor.y = element_blank(),
    legend.position = c(0.8, 0.1),
    legend.title = element_text(),
    panel.border = element_blank(),
    axis.title.y = element_blank(),
    axis.line.y = element_blank(),
    axis.text.y = element_blank(),
```

<sup>14</sup>Die Farben haben wir wieder vorher auf [https://www.w3schools.com/colors/colors\\_picker.asp](https://www.w3schools.com/colors/colors_picker.asp) herausgesucht.

```
plot.title = element_text(face = "bold")
)
```



Wie Sie sehen wird der Graph nicht durch eine eigene Funktion, sondern durch das sukzessive Hinzufügen von Strichen und Punkten erstellt. Besonders hervorzuheben am Code sind folgende Features:

- Wir verwenden die Funktion `reorder()` um die Werte auf der x-Achse nach Anteil der Frauen im Management zu ordnen
- Da wir mit der Funktion `ggplot2::coord_flip()` die Achsen umdrehen um eine horizontale Darstellung zu bekommen müssen wir bei allen Werten, die sich auf eine Achse beziehen umdenken
- Wir verwenden die Funktion `ggplot2::expansion()` wie oben eingeführt, da die x-Achse sonst nach links zu

wenig Platz für die Länderbezeichnungen lassen würde

- Das Argument `hjust=1` innerhalb von `ggplot2::geom_text()` sorgt dafür, dass der Text genau bei dem y-Wert aus `ggplot2::aes()` aufhört, also linksbündig formatiert wird (`hjust=0` korrespondiert entsprechend zu rechtsbündigem, `hjust=0.5` zu mittig formatierem Text).
- Mit `ggplot2::scale_color_manual()` erstellen wir eine manuelle Tabelle, da wir die Farben für Männer und Frauen in unterschiedlichen Layer plaziert haben. Wichtig ist, dass die Farbzuschreibung als *aesthetic mapping* definiert wird, da wir sonst keine Legende erstellen können. Die Syntax der Funktion ist dafür selbsterklärend.

## 5.6 Lügen mit grafischer Statistik

Grafiken können sehr leicht zur Manipulation der Betrachter\*innen eingesetzt werden. Im Folgenden wollen wir das an zwei klassischen Beispielen verdeutlichen. Eine schöne Übersicht finden Sie ansonsten in Krämer (2015)

### 5.6.1 Klassiker 1: Kontraintuitiver ‘Nullpunkt’

Sie möchten einen Unterschied konstruieren, der eigentlich gar nicht da ist? In diesem Fall könnten Sie sich ein Beispiel an Fox News nehmen (siehe Abbildung 5.4).



Figure 5.4: Quelle: <https://thenextweb.com/wp-content/blogs.dir/1/files/2015/05/viz3.jpg>

Die Autor\*innen haben ihre Manipulation hier entsprechend clever versteckt indem sie einfach gar keine Werte auf die y-Achse geschrieben haben. Das geht natürlich gar nicht, da wir intuitiv die beiden Flächen, bzw. Höhen der Balken ins Verhältnis setzen und uns weniger durch die abstrakten Zahlen beeinflussen lassen. Daher ist es gerade bei Histogrammen und Balkendiagrammen immer wichtig bei dem absoluten Nullpunkt zu starten.<sup>15</sup>

Im Folgenden sehen wir die manipulierende und korrekte Grafik nebeneinander:

```
data_used <- data.frame(Werte=c(6000000, 7066000), Art=c("Zustand", "Ziel"))

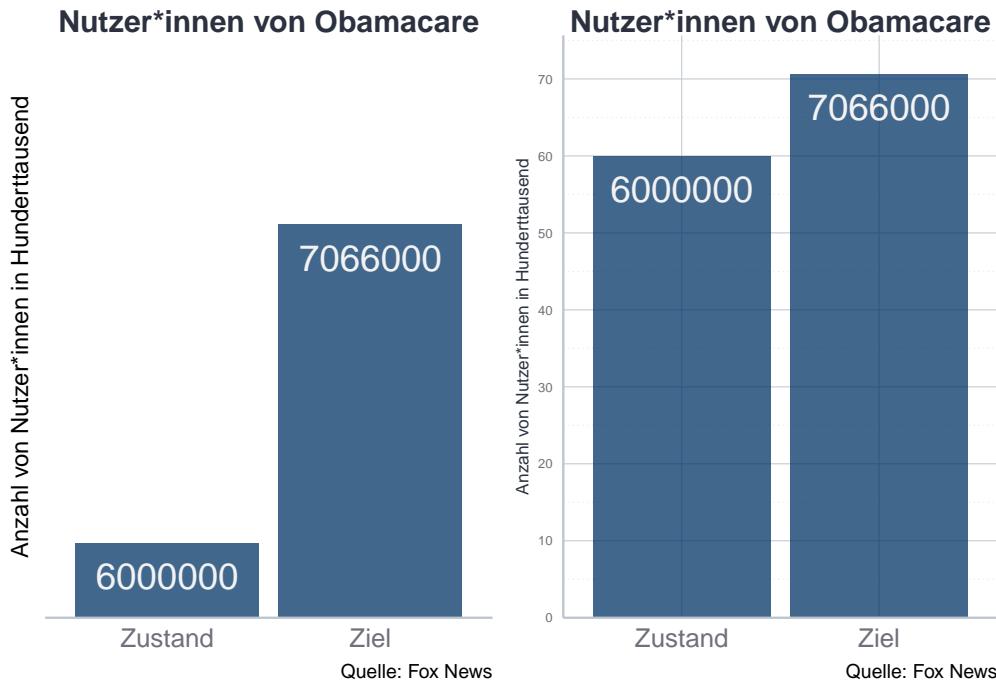
normal <- ggplot2::ggplot(data = data_used,
                           mapping = aes(x=reorder(Art, Werte), y=Werte)) +
```

<sup>15</sup>Wir wissen schließlich aus dem letzten Kapitel, dass solche Verhältnisvergleiche nur für verhältnis-skalierte Daten Sinn machen und diese durch die Existenz eines absoluten Nullpunkts definiert sind!

```
ggplot2::geom_bar(stat = "identity", fill="#003366", alpha=0.75) +
  ggplot2::geom_text(aes(label=as.character(format(Werte, scientific = FALSE))),
    size=6, vjust=1.75, color="#f2f2f2") +
  ggplot2::scale_y_continuous(
    name = "Anzahl von Nutzer*innen in Hunderttausend",
    breaks = seq(0, 8000000, 1000000),
    labels = seq(0, 80, 10),
    expand = expansion(c(0,0),
      c(0, 500000)))
  ) +
  ggplot2::labs(title = "Nutzer*innen von Obamacare",
    caption = "Quelle: Fox News") +
  icaeDesign::theme_icae() +
  ggplot2::theme(
    axis.title.y = element_text(),
    axis.text.x = element_text(size = 12),
    axis.title.x = element_blank(),
    plot.title = element_text(size=14, face = "bold")
  )

manipulativ <- normal +
  ggplot2::coord_cartesian(ylim=c(5750000, 7200000)) +
  ggplot2::theme(
    panel.grid = element_blank(),
    axis.title = element_blank(),
    axis.line.y = element_blank(),
    axis.text.y = element_blank()
  )

ggpubr::ggarrange(manipulativ, normal, ncol = 2)
```



Eine beliebte Variante ist es, die y-Achse zwar im Nullpunkt starten zu lassen, aber einfach die Achse zwischendrin abzuschneiden. Das Prinzip bleibt aber das gleiche und so etwas ist in keinem Fall eine gute Idee!

### 5.6.2 Klassiker 2: Geschickt gewählter Zeitraum und clever gewählte Achsenabschnitte

Sie möchten eine Tendenz zum Ausdruck bringen, die es gar nicht gibt? Grundsätzlich bieten sich hier drei Vorgehen an:

1. Sie wählen aus den ganzen Beobachtungen den Zeitraum aus in dem die Tendenz besteht.
2. Sie machen die Zeitachse möglichst kurz, dann wirken Veränderungen größer.
3. Sie zoomen in die y-Achse rein, auch das lässt Veränderungen größer werden.

Sehr gut funktioniert das bei schwankenden Größen wie der Arbeitslosigkeit. Gerade der erste Punkt funktioniert bei Arbeitslosenstatistiken immer sehr gut:

```
agenda_daten <- dplyr::filter(al_daten, year>2000)

manipulativ <- ggplot2::ggplot(data = agenda_daten,
                                mapping = aes(x=year, y=unemp_rate)
                                ) +
  ggplot2::geom_point() +
  ggplot2::geom_line() +
  ggplot2::geom_vline(xintercept = 2005) +
  ggplot2::scale_y_continuous(
    name = "Arbeitslosigkeit",
    labels = scales::percent_format(accuracy = 1, scale = 1)
  ) +
  ggplot2::labs(title = "Arbeitslosigkeit seit Einführung der Agenda 2010",
               caption = "Quelle: AMECO") +
```

```

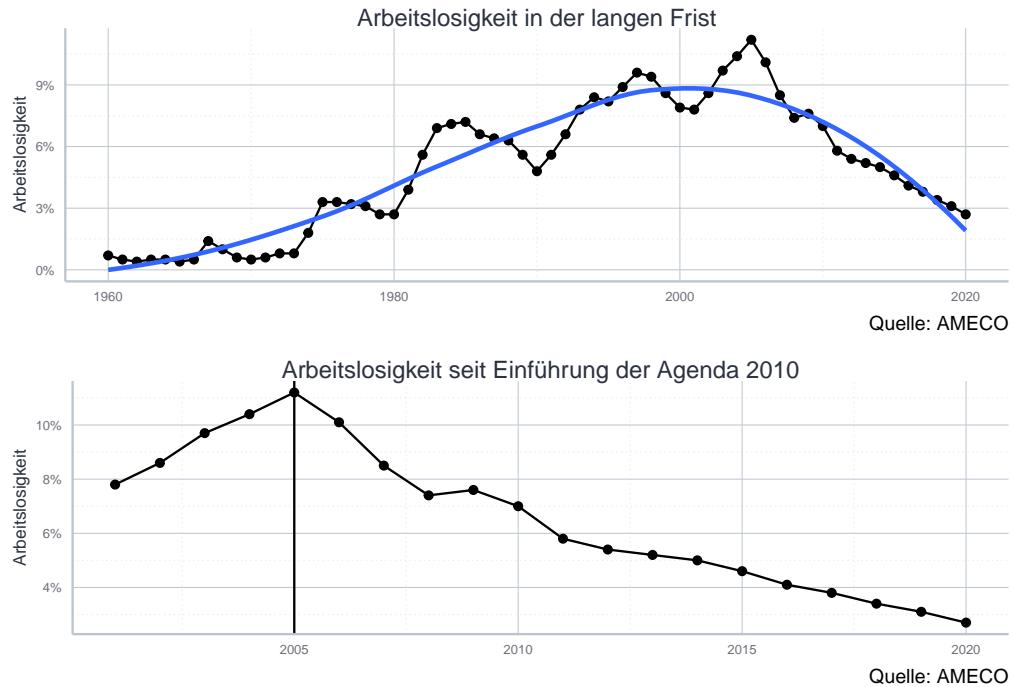
icaeDesign::theme_icae() +
ggplot2::theme(axis.title.x = element_blank())

normal <- ggplot2::ggplot(data = al_daten,
                           mapping = aes(x=year, y=unemp_rate)
                           ) +
ggplot2::geom_point() +
ggplot2::geom_line() +
ggplot2::geom_smooth(method = "loess", se = F) +
ggplot2::scale_y_continuous(
  name = "Arbeitslosigkeit",
  labels = scales::percent_format(accuracy = 1, scale = 1)
) +
ggplot2::labs(title = "Arbeitslosigkeit in der langen Frist",
             caption = "Quelle: AMECO") +
icaeDesign::theme_icae() +
ggplot2::theme(axis.title.x = element_blank())

ggpubr::ggarrange(normal, manipulativ, nrow=2)

```

#> `geom\_smooth()` using formula 'y ~ x'



Selbstverständlich ist der obere Graph auch nicht ganz manipulationsfrei. Aber es wird deutlich, wie viel Spielraum Sie nur über die Darstellung von bestimmten Grafiken haben.

Die weiteren beiden Punkte lassen sich anhand der Staatsausgaben in Deutschland auch sehr schön illustrieren. Die Rohdaten stammen von der [AMECO Homepage](#) und sind dem Kapitel “General Government/excessive deficit procedure” entnommen. Sie sind ein schönes Beispiel für die weit verbreiteten ‘breiten’ Daten, die wir erst einmal

in ein brauchbares Format bringen müssen:

```
ameco_data <- data.table::fread(here::here("data/raw/AMECO16.TXT"), fill = T, header = T) %>%
  dyplr::filter(
    TITLE=="Total current expenditure: general government :- Excessive deficit procedure",
    COUNTRY=="Germany",
    UNIT %in% c("(Percentage of GDP at current prices (excessive deficit procedure))",
               "Mrd ECU/EUR")) %>%
  dplyr::select(-one_of("CODE", "COUNTRY", "SUB-CHAPTER", "TITLE", "V68")) %>%
  dplyr::mutate(UNIT=ifelse(UNIT=="Mrd ECU/EUR", "Abs", "PercGDP")) %>%
  tidyverse::pivot_longer(names_to = "Jahr", values_to = "Wert", cols = -UNIT) %>%
  dplyr::filter(Jahr>1990) %>%
  tidyverse::pivot_wider(names_from = UNIT, values_from = Wert)
```

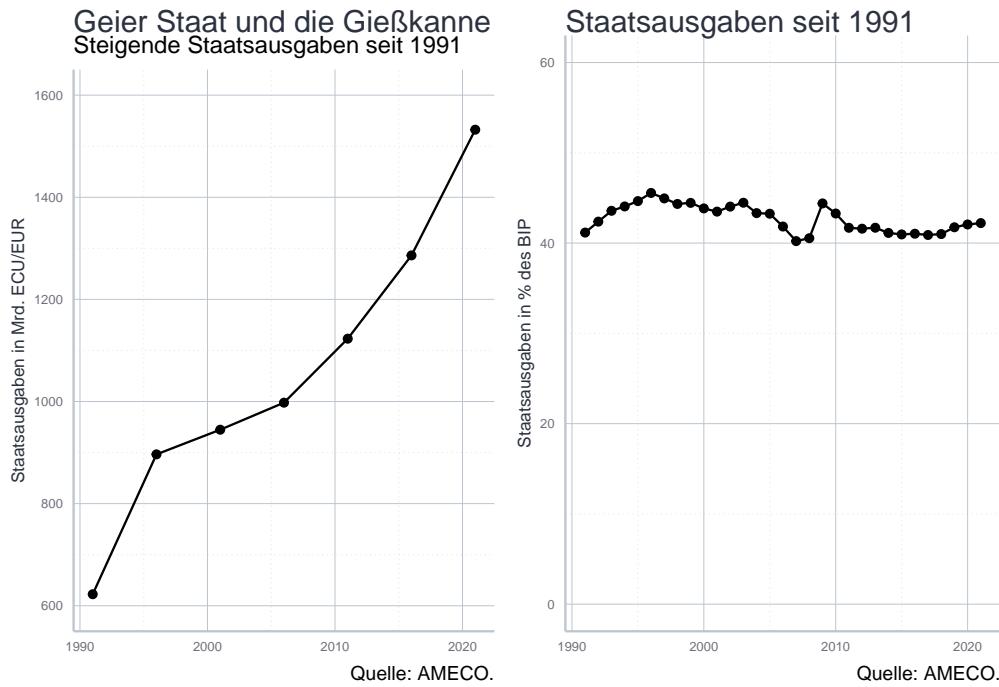
Jetzt können wir die Daten visualisieren:

```
ameco_geier_version <- ameco_data %>%
  dplyr::filter(Jahr %in% seq(1991, 2021, 5))

manipulativ <- ggplot2::ggplot(data = ameco_geier_version,
                                aes(x=Jahr, y=Abs)) +
  ggplot2::geom_point() +
  ggplot2::geom_line() +
  ggplot2::scale_y_continuous(name = "Staatsausgaben in Mrd. ECU/EUR",
                             limits = c(600, 1600)) +
  ggplot2::labs(title = "Geier Staat und die Gießkanne",
                subtitle = "Steigende Staatsausgaben seit 1991",
                caption = "Quelle: AMECO.") +
  icaeDesign::theme_icae() +
  ggplot2::theme(axis.title.x = element_blank(),
                plot.title = element_text(hjust = 0, size = 14))

normal <- ggplot2::ggplot(data = ameco_data,
                           aes(x=Jahr, y=PercGDP)) +
  ggplot2::geom_point() +
  ggplot2::geom_line() +
  ggplot2::scale_y_continuous(name = "Staatsausgaben in % des BIP",
                             limits = c(0, 60)) +
  ggplot2::labs(title = "Staatsausgaben seit 1991",
                caption = "Quelle: AMECO.") +
  icaeDesign::theme_icae() +
  ggplot2::theme(axis.title.x = element_blank(),
                plot.title = element_text(hjust = 0, size = 14))

ggpubr::ggarrange(manipulativ, normal, ncol = 2)
```



Falls Sie jetzt meinen: das ist ja eigentlich zu einfach um wirklich vorzukommen, dann schauen Sie mal in Abbildung 5.5 (bei der noch die Dummheit hinzukommt eine nominale Größe über die Zeit abzubilden),<sup>16</sup> in Ihre Tageszeitung oder in den sozialen Medien. Sie werden (leider) feststellen, dass solche visuellen Tricksereien sowohl in Schund- als auch Qualitätsmedien, aber auch in der Wissenschaft sehr weit verbreitet sind. Aber Sie wissen das ja jetzt und brauchen sich nicht mehr aufs Glatteis führen zu lassen.

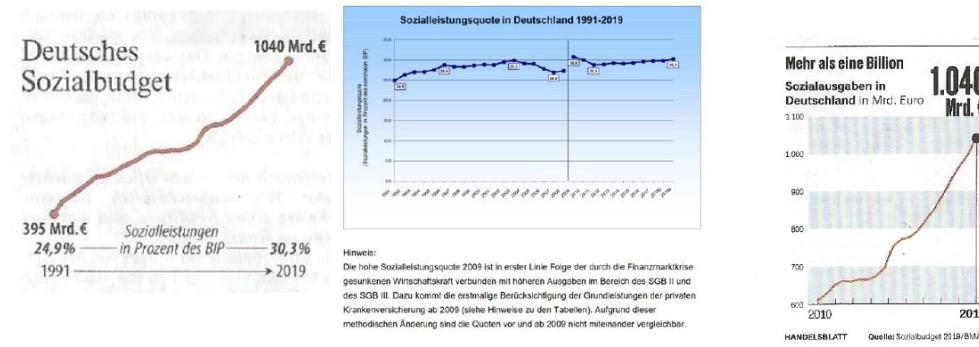


Figure 5.5: Quelle: FAZ vom 11. Juli 2020 (linkes Bild) und Bundesministerium für Arbeit und Soziales (mittleres Bild) und Handelsblatt vom 13. Juli 2020 (rechtes Bild).

## 5.7 Links und weiterführende Literatur

Einen guten Überblick über viele häufig verwendeten Befehle bietet [dieser Schummelzettel](#).

Die Debatte ob nun `base` oder `ggplot2` ‘besser’ ist kennt natürlich unzählbar viele Beiträge - die meisten davon geschrieben von Menschen mit starker Meinung und schwachen Argumenten. Ein recht häufig zitiert [pro-base Blog](#) von Jeff Leek findet hier eine [pro-ggplot2 Antwort](#). Nathan Yau bezieht sich auf beide Beiträge und vollzieht hier einen [sehr pragmatisch geschriebenen Vergleich](#). Auch wenn er das Potenzial von `ggplot2` nicht auch nur im

<sup>16</sup>Gefunden habe ich die Bilder über [diesen Tweet](#).

Ansatz ausnutzt ist es doch ein netter Vergleich mit in meinen Augen sinnvoller Conclusio: “There’s also no problem with using everything available to you. At the end of the day, it’s all R.”

Für alle die sich mit den theoretischen Grundlagen von `ggplot2` genauer befassen wollen: Die `ggplot2` zugrundeliegende Idee einer *grammar of graphics* geht, wie anfangs des Kapitels schon beschrieben, auf [Wilkinson \(1999\)](#) zurück und wird in [Wickham \(2010\)](#) theoretisch ausgeführt.

[Schwabish \(2014\)](#) wurde bereits erwähnt und ist eine konstruktive Auseinandersetzung mit typischen Visualisierungsfehlern, die auch tatsächlich in Top-Journalen gemacht wurden. Besonders wichtig: konstruktive Verbesserungsvorschläge sind gleich mit dabei.

[Krämer \(2015\)](#) ist eine klassische Sammlung manipulativer Grafiken und sicherlich empfehlenswert. Eine allgemeinere Diskussion von bestenfalls irreführenden Visualisierungen und ihre Implementierung in R findet sich [hier](#).

Falls Sie einen neuen Typ Grafik erstellen wollen ist es immer sinnvoll, sich Beispiele aus dem Internet anzuschauen, oder sogar bestehenden Code zu kopieren und für die eigenen Bedürfnisse anzupassen. Die [R Graph Gallery](#) ist dafür ein hervorragender Ausgangspunkt. Ansonsten bietet auch das [R Graphics Cookbook](#) zahlreiche sehr nützliche Ausgangsbeispiele.

Falls Sie geografische Daten visualisieren wollen finden Sie hier ein wunderbares [Eingangsbeispiel](#). Zur Visualisierung von Stromgrößen auf Karten finden Sie [hier](#) eine schöne Anleitung.



## Teil II

# Mathematische Grundlagen



# Chapter 6

## Formale Methoden der Sozioökonomie

Refusing to deal with numbers rarely serves the interest of the least well-off.

— Thomas Piketty

In diesem Kapitel werden ausgewählte formale Methoden, die in der sozioökonomischen Forschung besonders häufig verwendet werden, und ihre Implementierung in R eingeführt. Dabei gibt dieses Kapitel selbstverständlich nur einen ersten Einblick und die Auswahl ist notwendigerweise subjektiv.

Allerdings werden die in diesem Kapitel diskutierten Methoden Ihnen einen guten Einblick in die formale Forschung im Bereich der Sozioökonomik geben und Ihnen verdeutlichen wie vielseitig Sie R in Ihrer Forschungstätigkeit - auch abseits klassischer statistischer Anwendungen - verwenden können.

Zunächst werden wir uns mit der Berechnung von [Wachstumsraten](#) beschäftigen und dabei besonders die Verwendung von Logarithmen besprechen. Als nächstes werden Grundlagen der [Differentialrechnung](#) wiederholt und ihre Implementierung in R eingeführt. Besondere Beachtung findet dabei das Thema der Optimierung, das im Forschungsalltag eine besonders wichtige Rolle spielt.

Als nächstes illustrieren wir die Verwendung von Konzepten aus der [linearen Algebra](#), und werden anhand konkreter Beispiele noch einmal die Allgegenwärtigkeit der linearen Algebra verdeutlichen.

Den Schwerpunkt des Kapitels bildet dann der Abschnitt zu [Verteilungen](#). Die Analyse von Verteilungen spielt eine sehr wichtige Rolle in der Sozioökonomik, da Themen wie Einkommens- und Vermögensverteilung bzw. Ungleichheitsforschung traditionell ein wichtiges Kernthema der Sozioökonomik ausmachen.

### Verwendete Pakete

```
library(here)
library(tidyverse)
library(data.table)
library(ggrepel)
library(ggpubr)
library(latex2exp)
library(matlib)
library(fitdistrplus)
```

```
library(moments)
library(ineq)
library(rmutil)
library(viridis)
library(optimx)
```

**Hinweis:** Das Paket `matlib` (Friendly et al., 2019) verwenden wir für einige Matrizenoperationen und zum Lösen linearer Gleichungssysteme. Streng genommen ist das Paket nicht dringend nötig, da anstatt der Funktion `matlib::Solve()` auch die Funktion `base::solve()` verwendet werden kann. Der Output von `matlib::Solve()` ist aber schöner und etwas informativer.

## 6.1 Änderungsraten und die Rolle des Logarithmus

Die sozioökonomische Forschung beschäftigt sich häufig mit Veränderungen über die Zeit. Je nach Fragestellung sind dabei *absolute* oder *relative* Änderungen von Interesse.

Um die Änderungsrate einer Variable  $X$  zu berechnen wird folgende Formel verwendet:

$$\frac{X_t - X_{t-1}}{|X_{t-1}|} \cdot 100\% = \left( \frac{X_t}{|X_{t-1}|} - 1 \right) \cdot 100\%$$

Selbstverständlich können wir auch die Änderung über mehr als einen Zeitschritt berechnen. Für die **durchschnittliche Änderungsrate** verwenden wir:

$$\left( \left[ \frac{X_t}{X_{t-s}} \right]^{\frac{1}{s}} - 1 \right) \cdot 100\%$$

Umgekehrt können wir den tatsächlichen Wert der Variable  $X$  berechnen wenn wir Informationen über die jährliche Änderungsrate  $x$  haben. Hierbei gilt:

$$X_{t+s} = X_t (1 + x)^s$$

Diese Formel kann auch durch Verwendung der *Eulerschen Zahl*  $e$  approximiert werden:

$$X_{t+s} = X_t (1 + x)^s \approx X_t \cdot e^{xs}$$

Diese Approximation wird später hilfreich werden, wenn wir Wachstumsraten in logarithmierter Form darstellen wollen.

Wenn  $X_t = 4$ ,  $s = 5$  und  $x = 0.05$  ergibt sich für den Wert nach  $s$  Zeitschritten also  $X_{t+s} = 4 \cdot 1.05^5 = 5.11$ . Oder, unter Verwendung der vereinfachten Formel:  $4 \cdot e^{0.05 \cdot 5} = 5.13$ .

Natürlich können wir auch Änderungen von prozentualen Größen berechnen. Wenn die Inflation im Jahr 2010 bei 4% und 2011 bei 5% liegt können wir die Änderung folgendermaßen berechnen:

$$\frac{5\% - 4\%}{|4\%|} = 0.25 = 25\%$$

Hier von einer 25-prozentigen Änderung zu sprechen ist jedoch nicht eindeutig: damit könnte eine relative Änderung von 25% gemeint sein, oder aber eine absolute Änderung von 25%. Daher sprechen wir bei letzterem von einer Änderung in *Prozentpunkten*. Im Beispiel haben wir also eine absolute Änderung von einem Prozentpunkt, bzw. eine relative Änderung von 25%.

In R können wir die Funktionen `lag()` und `lead()` aus dem Paket `dplyr` (Wickham et al., 2019) verwenden um Änderungsraten zu berechnen.<sup>1</sup> Die Funktion `lag()` akzeptieren dabei zwei Argumente: den Vektor der Werte und die Anzahl der Schritte, die zurück bzw. vor gesprungen werden sollen.

Entsprechend können wir Änderungsraten folgendermaßen berechnen:

```
werte <- c(1, 2.2, 3.25, 0.5, 0.1, -0.1, 0.2)
rel_change <- (werte - dplyr::lag(werte)) / abs(dplyr::lag(werte)) * 100
rel_change
```

```
#> [1] NA 120.00000 47.72727 -84.61538 -80.00000 -200.00000 300.00000
```

Die gleiche Syntax können wir auch für die Arbeit mit einem `data.frame` verwenden. Hier müssen wir aber darauf achten, die Daten auch tatsächlich nach dem Beobachtungszeitpunkt zu sortieren, damit `data.frame::lag(x, 1)` auch den vorherigen Wert ausgibt. Dazu verwenden wir die Funktion `dplyr::arrange()`, welche die Zeilen eines `data.frame` gemäß einer oder mehrerer Variablen ordnet:

```
head(beispiel_daten_at, 4)
```

```
#>   country      BIP year
#> 1: Austria 37941.04 2018
#> 2: Austria 37140.79 2017
#> 3: Austria 36469.39 2016
#> 4: Austria 36129.03 2015
```

```
beispiel_daten_at <- beispiel_daten_at %>%
  dplyr::arrange(year)
head(beispiel_daten_at, 4)
```

```
#>   country      BIP year
#> 1: Austria 36123.43 2014
#> 2: Austria 36129.03 2015
#> 3: Austria 36469.39 2016
#> 4: Austria 37140.79 2017
```

```
beispiel_daten_at <- beispiel_daten_at %>%
  dplyr::mutate(BIP_Wachstum = (BIP-dplyr::lag(BIP))/abs(dplyr::lag(BIP))*100)
beispiel_daten_at
```

```
#>   country      BIP year BIP_Wachstum
#> 1: Austria 36123.43 2014          NA
#> 2: Austria 36129.03 2015  0.01550613
#> 3: Austria 36469.39 2016  0.94206769
#> 4: Austria 37140.79 2017  1.84100901
```

---

<sup>1</sup>Der Funktionsname ‘lag’ und ‘lead’ wird leider in sehr vielen Paketen verwendet, u.a. auch in `data.table`. Deswegen ist es gerade bei diesen Funktionen besser den expliziten Aufruf `dplyr::lag()` und `dplyr::lead()` zu verwenden.

```
#> 5: Austria 37941.04 2018 2.15464100
```

Falls wir innerhalb des Datensatzes unterschiedliche Beobachtungsobjekte haben, z.B. verschiedene Länder, müssen wir den Datensatz vor Berechnung der Wachstumsrate gruppieren:

```
head(beispiel_daten, 4)
```

```
#>   country      BIP year
#> 1: Austria 37941.04 2018
#> 2: Germany 35866.00 2018
#> 3: Austria 37140.79 2017
#> 4: Germany 35477.89 2017

beispiel_daten <- Beispiel_daten %>%
  dplyr::arrange(country, year) %>%
  dplyr::group_by(country) %>%
  dplyr::mutate(BIP_Wachstum = (BIP - dplyr::lag(BIP)) / abs(dplyr::lag(BIP)) * 100) %>%
  dplyr::ungroup()

beispiel_daten
```

```
#> # A tibble: 10 x 4
#>   country      BIP year BIP_Wachstum
#>   <chr>     <dbl> <int>      <dbl>
#> 1 Austria 36123.    2014       NA
#> 2 Austria 36129.    2015      0.0155
#> 3 Austria 36469.    2016      0.942
#> 4 Austria 37141.    2017      1.84
#> 5 Austria 37941.    2018      2.15
#> 6 Germany 34077.    2014       NA
#> 7 Germany 34371.    2015      0.862
#> 8 Germany 34859.    2016      1.42
#> 9 Germany 35478.    2017      1.78
#> 10 Germany 35866.   2018      1.09
```

Häufig werden Wachstumsraten in ihrer logarithmierten Form präsentiert. Wir können nämlich die Formel zur Berechnung von Änderungsprozessen folgendermaßen approximieren:

$$\left( \left[ \frac{X_t}{X_{t-s}} \right]^{\frac{1}{s}} - 1 \right) \approx \ln \left( \frac{X_t}{X_{t-s}} \right) / t = \frac{\ln(X_t) - \ln(X_{t-s})}{t}$$

Sie fragen sich vielleicht warum wir uns mit der Verwendung des Logarithmus überhaupt beschäftigen, wo durch die ‘Vereinfachung’ doch eine kleine Ungenauigkeit eingeführt wird? Tatsächlich ist die Verwendung des Logarithmus häufig hilfreich für die grafische Darstellung von Wachstumsraten.<sup>2</sup>

In Darstellung 6.1 gilt: die Steigung im logarithmierten Plot gibt die *relative* Änderung der Variable an. Das bedeutet wenn wir im logarithmierten Plot eine lineare Steigung haben wächst die Variable konstant mit der gleichen Wachstumsrate über die Zeit - so wie im obigen Beispiel.

---

<sup>2</sup>Zur Transformation der y-Achse verwenden wir in ggplot2 die Funktion `scale_y_continuous()` und setzen das Argument `trans = "log"`.

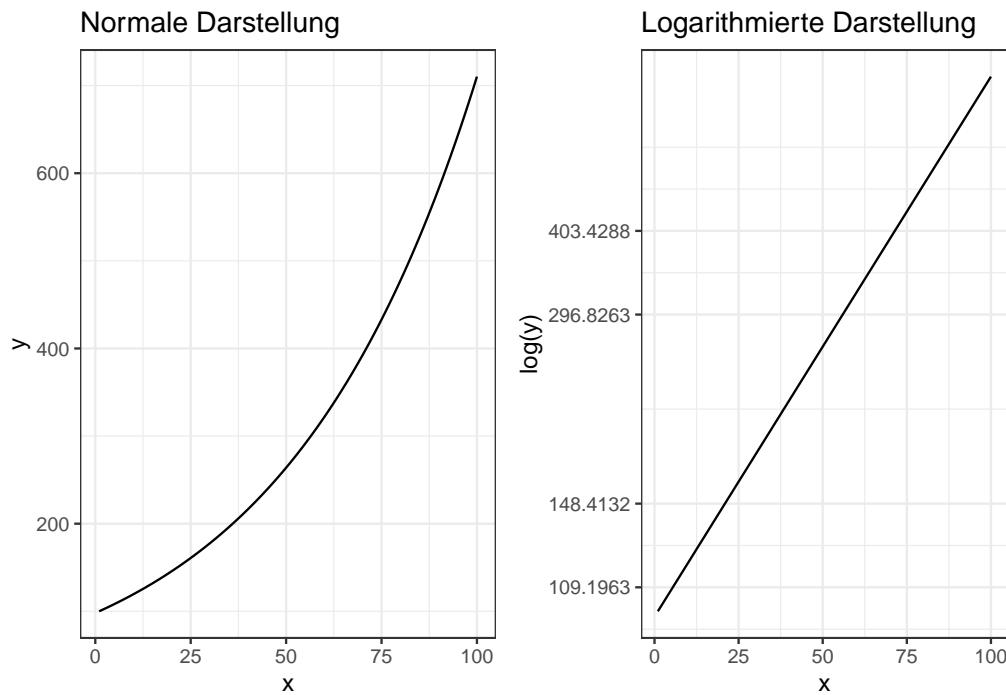


Figure 6.1: Vergleich normaler und logarithmierter Darstellung

Diese Art der Darstellung ist zum Beispiel bei der langfristigen Betrachtung von Wachstumsraten und dem Vergleich zwischen Ländern sehr hilfreich, da, wie in Abbildung 6.2 Unterschiede in der logarithmierten Darstellung besser erkennbar sind.

Abbildung 6.3 zeigt wie wichtig eine solche Darstellung sein kann um Events, die zu sehr unterschiedlichen Zeitpunkten stattgefunden haben, vergleichbar zu machen:

Während die absoluten Zahlen die Volatilität während der Großen Depression verschwindend gering erscheinen lassen wird im unteren Graph von Abbildung 6.3 deutlich, dass die Volatilität damals tatsächlich noch größer war.

Um die Achsen intuitiver verständlich zu machen habe ich von allen Werten den Wert für 1871 (die erste Beobachtung) abgezogen und den Wert für 1871 somit auf Null normiert. Zudem habe ich die Werte mit 100 multipliziert, sodass eine Änderung von 1 auf der y-Achse zu einer einprozentigen Änderung des S&P Kurses korrespondiert.

Im Rahmen der Regressionsanalyse werden wir zudem lernen, dass die logarithmierte Form die Analyse von Wachstumsraten in linearen Regressionsmodellen deutlich vereinfacht (siehe Kapitel ??).

## 6.2 Grundlagen der Differentialrechnung

### 6.2.1 Einleitung: Differential- und Integralrechnung

Die Differentialrechnung ist eng verwandt mit der Integralrechnung: in beiden Bereichen studiert man die Veränderungen von Funktionen. Während die Differentialrechnung sich mit der lokalen Änderung einer Funktion beschäftigt, also vor allem versucht die Steigung der durch die Funktion definierten Kurven zu berechnen, studiert die Integralrechnung die Flächen und Volumina, die durch eine Funktion definiert sind. Grafisch bedeutet dies, dass wir bei der Integralrechnung an den Flächen unter einer bzw. zwischen mehreren Kurven interessiert sind.

Die beiden Bereiche sind eng miteinander verbunden. Besonders deutlich wird das in dem so genannten *Fundamen-*

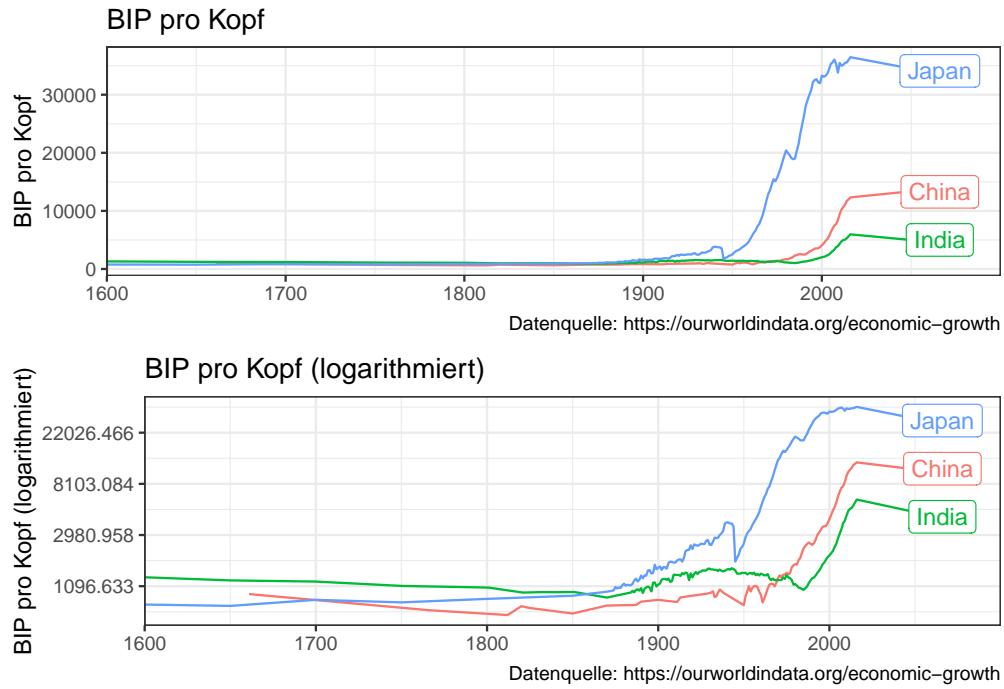


Figure 6.2: Vergleich normaler und logarithmierter Darstellung bei langfristiger vergleichender Betrachtung von Wachstumsraten

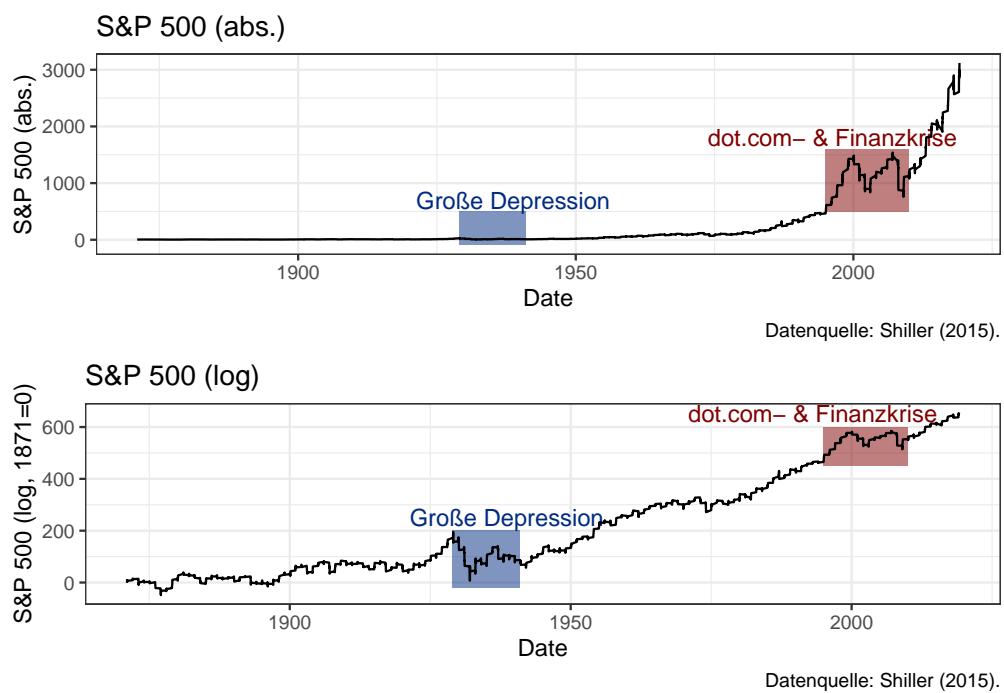


Figure 6.3: Vergleich normaler und logarithmierter Darstellung bei Events, die zu sehr unterschiedlichen Zeitpunkten stattgefunden haben

*talsatz der Analysis* (auch: *Hauptsatz der Differential- und Integralrechnung*) deutlich. In der Differentialrechnung leiten wir Funktionen *ab* und in der Integralrechnung leiten wir Funktionen *auf*. Der Fundamentalsatz der Analysis zeigt, dass die beiden Vorgehensweise jeweils die Umkehrung des anderen Darstellen: Die Ableitung einer Aufleitung führt zur gleichen Ausgangsfunktion, genauso wie die Aufleitung der Ableitung ebenfalls wieder zur Ausgangsfunktion führt.

In der Ökonomik spielen beide Bereiche eine wichtige Rolle, der Fokus wird in diesem Kapitel jedoch auf der Differentialrechnung liegen, deren Anwendungsgebiet noch einmal breiter ist: wann immer Sie eine Funktion maximieren oder minimieren bedienen Sie sich Methoden der Differentialrechnung. Und Maximierung spielt nicht nur in den herkömmlichen Modellen, die auf dem *homo oeconomicus* aufbauen, eine wichtige Rolle. Auch in zahlreichen anderen Modellierungsparadigmen und genauso in der Ökonometrie spielt die Maximierung eine wichtige Rolle.

### 6.2.2 Wiederholung: Ableitungsregeln

Für einfache Funktionen gibt es unmittelbare Ableitungsregeln, die uns für jeden Ausdruck die entsprechende Ableitung geben. Komplexere Ausdrücke versucht man über entsprechende Regeln auf diese einfacheren Ausdrücke zurückzuführen und Ableitungen von komplexeren Funktionen somit ‘Stück für Stück’ durchzuführen. Bei den komplexeren Ableitungsregeln handelt es sich insbesondere um die Summen-, Produkt-, Quotienten- und Kettenregel. Vorher wollen wir uns aber mit den einfachen Grundregeln vertraut machen.

Die Ableitung der Funktion  $f(x)$  wird als  $f'(x)$  oder mit  $\frac{\partial f(x)}{\partial x}$  bezeichnet. Letztere Formulierung ist besonders hilfreich wenn eine Funktion im Bezug auf verschiedene Variablen abgeleitet ist: durch diese Formulierung wird unter dem Bruchstrich noch einmal explizit angegeben nach welcher Variable die Funktion abgeleitet wird.

Grundsätzlich gilt, dass die Ableitung einer Konstanten gleich Null ist:

$$\frac{\partial a}{\partial x} = 0$$

Die Ableitung einer Potenz funktioniert folgendermaßen:

$$\frac{\partial x^n}{\partial x} = nx^{n-1}$$

Besteht unsere komplexere Funktion  $f(x)$  aus der Summe von Teilfunktionen verwenden wir die **Summenregel**. Diese besagt, dass die Ableitung von  $f(x) = u(x) + v(x)$  einfach die Summe der Ableitungen der Teilfunktionen  $u$  und  $v$  sind:

$$f'(x_0) = u'(x_0) + v'(x_0)$$

Wenn wir also die Funktion  $f(x) = 3x^2 + 4x$  ableiten wollen geht dies nach der Summenregel folgendermaßen:

$$f'(x) = u'(x) + v'(x) \tag{6.1}$$

$$u(x) = 3x^2, u'(x) = 6x \tag{6.2}$$

$$v(x) = 4x, v'(x) = 4 \tag{6.3}$$

$$f'(x) = 6x + 4 \tag{6.4}$$

Die Summenregel funktioniert natürlich äquivalent auch für den Fall in dem die Teilfunktionen substrikt werden. Werden die Teilfunktionen nicht summiert sondern multipliziert verwenden wir die **Produktregel**. Gehen wir wieder davon aus, dass wir eine komplexe Funktion  $f(x) = u(x)v(x)$  ableiten wollen. Ein Beispiel wäre  $f(x) = (4 + x^2)(1 - x^3)$ , wobei  $u(x) = (4 + x^2)$  und  $v(x) = (1 - x^3)$ .

Insbesondere gilt hier:

$$f'(x_0) = u'(x_0) \cdot v(x_0) + u(x_0) \cdot v'(x_0)$$

Wir können die komplexere Gesamtfunktion also ableiten indem wir die einzelnen Teile separat ableiten und jeweils mit den Ausgangsfunktionen multiplizieren. Für unser Beispiel mit  $f(x) = (4 + x^2)(1 - x^3)$  hätten wir also:

$$f'(x) = u'(x) \cdot v(x) + u(x) \cdot v'(x) \quad (6.5)$$

$$u(x) = (4 + x^2), u'(x) = 2x \quad (6.6)$$

$$v(x) = (1 - x^3), v'(x) = 3x \quad (6.7)$$

$$f'(x) = 2x(1 - x^3) + 3x(4 + x^2) = 2x - 2x^4 + 12x + 3x^3 = -2x^4 + 3x^3 + 14x \quad (6.8)$$

Wenn die beiden Teilfunktionen dagegen dividiert werden müssen wir die **Quotientenregel** anwenden. Hier gehen wir also von dem Fall  $f(x) = \frac{u(x)}{v(x)}$  aus, z.B. von  $f(x) = \frac{x^2}{2x}$ .

In diesem Fall gilt:

$$f'(x_0) = \frac{u'(x_0) \cdot v(x_0) - u(x_0) \cdot v'(x_0)}{(v(x_0))^2}$$

Für unser Beispiel hätten wir dann:

$$f'(x) = \frac{u'(x) \cdot v(x) - u(x) \cdot v'(x)}{(v(x))^2} \quad (6.9)$$

$$u(x) = x^2, u'(x) = 2x \quad (6.10)$$

$$v(x) = 2x, v'(x) = 2 \quad (6.11)$$

$$f'(x) = \frac{2x \cdot 2 - x^2 \cdot 2}{(2x)^2} = \frac{2x - 2x^2}{(2x)^2} \quad (6.12)$$

Zuletzt betrachten wir noch die **Kettenregel**, die es uns erlaubt geschachtelte Funktionen abzuleiten. Darunter verstehen wir Funktionen wie  $f(x) = u(x) \circ v(x) = u(v(x))$ .

Hier gilt:

$$(u \circ v)'(x_0) = u'(v(x_0)) \cdot v'(x_0)$$

Man leitet also die ‘innere’ Funktion  $v(x)$  normal ab und multipliziert diese Ableitung mit der Ableitung der ‘äußereren’ Funktion  $u(v)$  an der Stelle  $v(x_0)$ . Am einfachsten ist das mit einem Beispiel nachzuvollziehen in dem  $f(x) = (x^2 + 4)^2$ , also  $u(v) = v^2$  und  $v(x) = x^2 + 4$ .

Insgesamt bekommen wir also:

$$f'(x) = u' (v(x_0)) \cdot v'(x_0) \quad (6.13)$$

$$u(v) = v^2, u'(v) = 2v \quad (6.14)$$

$$v(x) = x^2 + 4, v'(x) = 2x \quad (6.15)$$

$$f'(x) = 2(x^2 + 4) \cdot 2x \quad (6.16)$$

### 6.2.3 Ableitungen in R

Sie müssen Ableitungen nicht händisch ausrechnen, sondern können die Funktionen auch in R direkt ableiten lassen. Dazu verwenden wir die Funktion `expression()` um unsere abzuleitende Funktion zu definieren und dann die Funktion `D()` um die Ableitung zu bilden.

Betrachten wir folgendes Beispiel:

$$f(x) = x^2 + 3x$$

Zunächst wird die Funktion in eine `expression` übersetzt:

```
f <- expression(x^2+3*x)
f
```

```
#> expression(x^2 + 3 * x)
```

Eine solche `expression` können Sie über die Funktion `eval()` für konkrete Werte ausrechnen lassen:

```
x <- 1:5
eval(f)
```

```
#> [1] 4 10 18 28 40
```

Zudem können wir mit der Funktion `D()` direkt die Ableitung einer `expression` berechnen:

```
D(f, "x")
```

```
#> 2 * x + 3
```

Wir haben also:

$$\frac{\partial f(x)}{\partial x} = 2x + 3$$

Wir können Aufrufe von `D()` auch verschachteln um höhere Ableitungen zu berechnen:

```
D(D(f, "x"), "x")
```

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

Für die zweite Ableitung erhalten wir also dementsprechend:

$$\frac{\partial f(x)}{\partial x^2} = 2$$

### 6.2.4 Maximierung: die analytische Perspektive

Eine der wichtigsten Anwendungen der Differentialrechnung ist die Berechnung von Minima und Maxima, so genannten Extrema, einer Funktion. Die interessierende Funktion wird in diesem Kontext in der Regel *Zielfunktion* genannt.

Die Differentialrechnung spielt hier eine wichtige Rolle, denn Extrema sind dadurch gekennzeichnet, dass die Ableitung einer Funktion an ihren Extrempunkten gleich Null ist. Weil die Nullstellen einer Funktion wiederum recht leicht zu finden sind, bietet es sich an, Extrema über die Ableitung einer Funktion zu suchen.

Die genauen Details des Verfahrens werden hier nicht besprochen, es gibt jedoch zahlreiche gute Lehrbücher. Hier soll es eher um die grundsätzliche Intuition gehen.

Wichtig ist die Unterscheidung zwischen *lokalen* und *globalen* Extremwerten. Das *globale Maximum* (Minimum) liegt an dem Punkt im Definitionsbereich einer Funktion, der zu dem größten (kleinsten) Wert im Wertebereich der Funktion führt. Das *lokale Maximum* (Minimum) ist für eine bestimmte Teilmenge des Definitionsbereichs der Funktion definiert und bezeichnet den Punkt mit dem größten (kleinsten) Wert *innerhalb dieser Teilmenge*.

Formal exakt können wir die Punkte folgendermaßen definieren, wenn wir von einer Funktion  $f$  mit Definitionsbereich  $D \subseteq \mathbb{R}$  und Wertebereich  $\mathbb{R}$ , also  $f : D \rightarrow \mathbb{R}$  ausgehen.

Dann hat  $f$  ein *lokales Minimum* im Intervall  $I = (a, b)$  am Punkt  $(x^*, f(x^*))$  wenn  $f(x^*) \leq f(x) \forall x \in I \cap D$ . Analog sprechen wir bei dem Punkt  $(x^*, f(x^*))$  von einem *lokalen Maximum* im Intervall  $I = (a, b)$  wenn  $f(x^*) \geq f(x) \forall x \in I \cap D$ .

Wir sprechen beim Punkt  $(x^*, f(x^*))$  von einem *globalen Minimum* wenn  $f(x^*) \leq f(x) \forall x \in D$  und von einem *globalen Maximum* wenn  $f(x^*) \geq f(x) \forall x \in D$ .

In Abbildung 6.4 sehen wir beispielhaft die Extremwerte der Funktion  $f(x) = 8x^2 + 2.5x^3 - 4.25x^4 + 2$ .

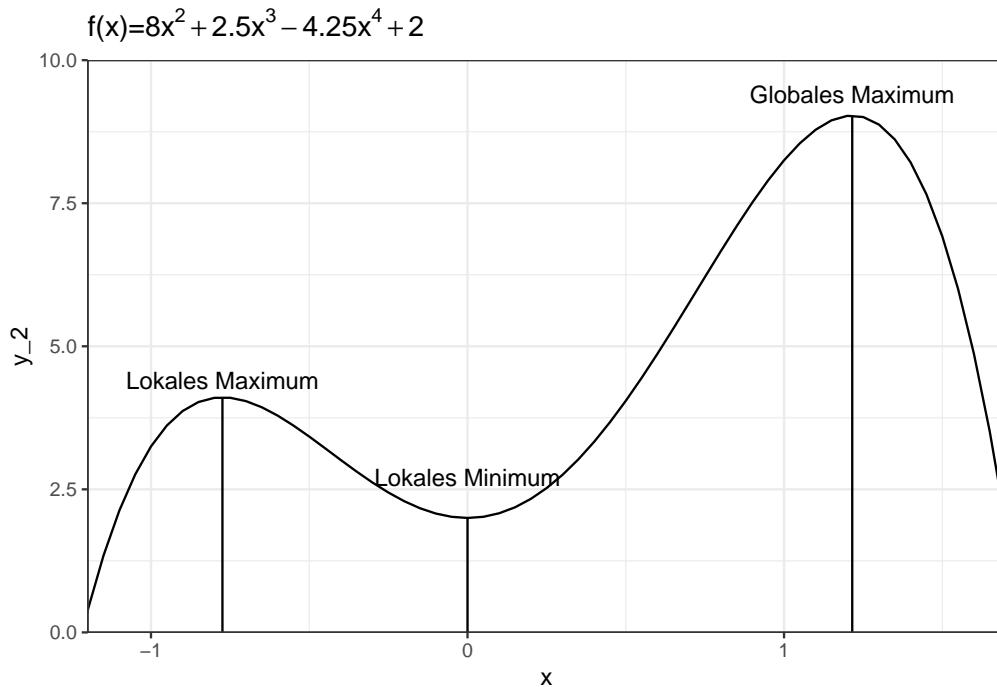


Figure 6.4: Beispiel für Extremwerte

Es kann gezeigt werden, dass eine **notwendige Bedingung** für die Existenz eines Extremwertes am Punkt  $x^*$  ist,

dass  $f'(x^*) = 0$ . Daher ist der erste Schritt bei der analytischen Suche nach Extremwerten immer die Ableitung der Funktion und die Identifikation der Nullstellen. Als nächstes untersucht man die **hinreichenden Bedingungen**, die einem genauere Informationen über den Punkt geben.

Hierbei hat sich die in Tabelle 6.1 zusammengefasste Heuristik in der Praxis bewährt:<sup>3</sup>

Table 6.1: Heuristik zur Untersuchung der hinreichenden Bedingung.

1. Ableitung	2. Ableitung	Ergebnis
$f'(x) = 0$	$f''(x) > 0$	Minimum
$f'(x) = 0$	$f''(x) < 0$	Maximum
$f'(x) = 0$	$f''(x) = 0$	Wendepunkt

Das Ganze funktioniert natürlich nur wenn eine Funktion auch tatsächlich eine Ableitung besitzt, es sich also um eine differenzierbare Funktion handelt. Daher wird das auch in vielen ökonomischen Modellen angenommen.

Um herauszufinden ob es sich um ein *globales* Extremum handelt müssen wir die Werte der Extrema vergleichen. Es gibt auch noch einige Heuristiken für besondere Sub-Klassen von Funktionen, die wir hier aber nicht genauer diskutieren wollen.

Wenn die Funktion unter bestimmten *Bedingungen* maximiert (minimiert) werden soll, sprechen wir von einer *Maximierung unter Nebenbedingung(en)*. Die Standard-Methode hier ist die sogenannte *Lagrange-Optimierung*. Details finden sich in zahlreichen Lehrbüchern, z.B. in [Wainwright and Chiang \(2005\)](#).

### 6.2.5 Maximierung: die algorithmische Perspektive

Bei vielen Funktionen wäre die analytische Berechnung von Extrema zu aufwendig oder gar nicht möglich. Daher verwendet man den Computer um die Extrema zu finden. Ironischerweise ist das gerade bei einfachen Funktionen kein großes Problem. Für die im linken Teil der Abbildung XX dargestellte Funktion kann der Computer einfach mit einem beliebigem Startwert  $x_0$  beginnen und sich auf dem Definitionsbereich in Richtung steigender Funktionswerte fortbewegt bis er den Punkt  $x_{globmax}^* = 0$  erreicht. Für Funktionen mit lokalen Extremwerten wie im rechten Teil von Abbildung funktioniert diese Strategie unter Umständen nicht mehr, da der Computer leicht auf lokalen Optima „steckenbleibt“. Im Beispiel in Abbildung 6.5 besteht bei einer unglücklichen Wahl des Startwertes die Gefahr, auf dem lokalen Extremum bei  $x = 0.77$  hängen zu bleiben.

Um das zu vermeiden verwenden die Optimierungsalgorithmen einige Tricks. Für die R-Funktion `optim()` können Sie z.B. zwischen sieben solcher ausfeilter Algorithmen wählen. Schauen Sie einmal in die Hilfefunktion wenn Sie mehr Informationen über diese Algorithmen bekommen möchten.<sup>4</sup>

Wichtig zu unterscheiden ist die Art der zu optimierenden Funktion und der Nebenbedingungen. Grob können wir zwischen den folgenden drei Fällen unterscheiden:

---

<sup>3</sup>Diese Klassifizierung ist nicht erschöpfend und in einigen Fällen uneindeutig. Tatsächlich gilt Folgendes: sei  $f^n(x)$  die  $n$ -te Ableitung von  $f(x)$ . Wenn  $f'(x) = 0$  und die erste von Null verschiedene höhere Ableitung eine Ableitung gerader Ordnung ist haben wir einen Extrempunkt, ansonsten einen Sattelpunkt. Ansonsten gilt auch, dass bei  $f''(x) > 0$  ein Minimum und bei  $f''(x) < 0$  ein Maximum vorliegt.

<sup>4</sup>Gerade bei komplexeren Methoden müssen Sie als Nutzer\*in jedoch in der Regel nachhelfen und der Opimierungsfunktion weitere Hinweise zur Funktion angeben. Für unsere Anwendungsbeispiele ist das nicht weiter relevant, Sie sollten die Problematik jedoch im Hinterkopf behalten.

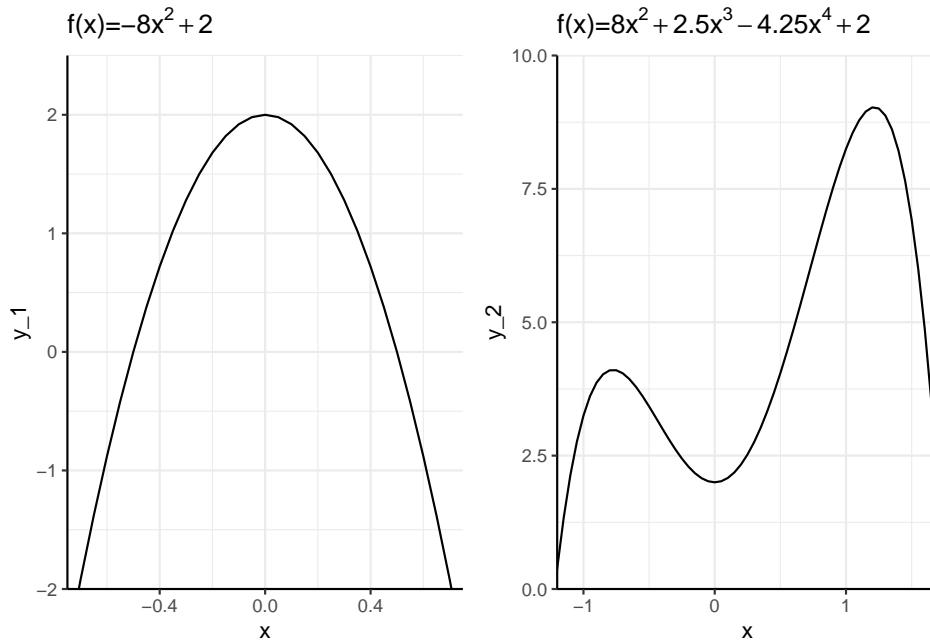


Figure 6.5: Beispiel für die Gefahr auf einem lokalen Extremum hängen zu bleiben (rechter Graph).

1. **Lineares Programmieren (LP)**: Sowohl Zielfunktion als auch Nebenbedingungen sind linear. Beispiel:  
 $\max s.t. Ax < b, x \geq 0$
2. **Quadratisches Programmieren (QP)** Zielfunktion ist quadratisch, Nebenbedingungen sind linear.  
 Beispiel:  $\max s.t. Ax < b, x \geq 0$
3. **Nicht-lineares Programmieren (NLP)**: Die Zielfunktion oder zumindest eine Nebenbedingung ist nicht-linear.

Die Unterscheidung spielt eine ähnliche Rolle wie die Unterscheidung verschiedener Skalenstufen bei der Datenanalyse: je nach Art des Problems müssen wir andere Methoden anwenden. In diesem Fall bedeutet das, dass wir für unterschiedliche Arten von Funktionen andere Pakete verwenden müssen um Extremwerte zu finden. Zusätzlich gibt es aber auch noch ein paar *general-purpose*-Funktionen, die wir auf alle Klassen anwenden können - auf Kosten der Performance. Diese sind in Tabelle 6.2 zusammengefasst.<sup>5</sup>

Das Schöne ist, dass trotz der Vielzahl an Paketen alle Optimierungsfunktionen nach einem sehr ähnlichen Schema aufgebaut sind. Die ersten beiden Argumente sind immer die Zielfunktion und die Nebenbedingungen. Danach folgen Argumente mit denen Sie die Suchintervalle, den konkreten Algorithmus oder weitere Spezifika festlegen können. Für eine genauere Einführung bietet sich auch die Lektüre der Vignette für das allgemein gehaltene Paket `optimx` ([Nash and Varadhan, 2011](#)) an, die [hier](#) abgerufen werden kann.

Table 6.2: Generell anwendbare Optimierungsfunktionen.

Art	Optimierungsfunktion	Paket
Allgemein (eindimensional)	<code>optimize()</code>	<code>stats</code>
Allgemein (mehrdimensional)	<code>optimr()</code>	<code>optimx</code>
LP	<code>lp()</code>	<code>lpSolve</code>

<sup>5</sup>Es gibt auch zwei Optimierungsfunktionen in `base`, allerdings sind diese mittlerweile ein wenig in die Jahre gekommen. Es wird daher empfohlen, anstatt `optim()` und `optimize()` die Funktion `optimx::optimr()` zu verwenden, die größtenteils aber auch die gleiche Syntax verwendet. Eine Übersicht über die meisten verfügbaren Funktionen finden Sie [hier](#).

Art	Optimierungsfunktion	Paket
QP	<code>solve.QP()</code>	<code>quadprog</code>
NLP	<code>optimize()</code>	<code>optimize</code>
NLP	<code>optimx()</code>	<code>optimx</code>

Im Folgenden wollen wir anhand einiger einfacher Beispiele sehen wie Sie Optimierungsprobleme in R lösen können. Für eine tiefergehende Auseinandersetzung verweisen wir auf die entsprechenden spezialisierten Einführungen.

Betrachten wir die folgende Zielfunktion:

$$f(x) = 8x^2 + 2.5x^3 - 4.25x^4 + 2 \quad (6.17)$$

In R:

```
f_1 <- function(x) 8*x^2 + 2.5*x**3 - 4.25*x**4 + 2
```

Die Funktion ist in Abbildung 6.6 dargestellt. Wie man sieht verfügt sie über ein lokales Maximum bei  $x_a = -0.77$ , ein lokales Minimum bei  $x_b = 0$  und ein globales Maximum bei  $x_c = 1.22$ .

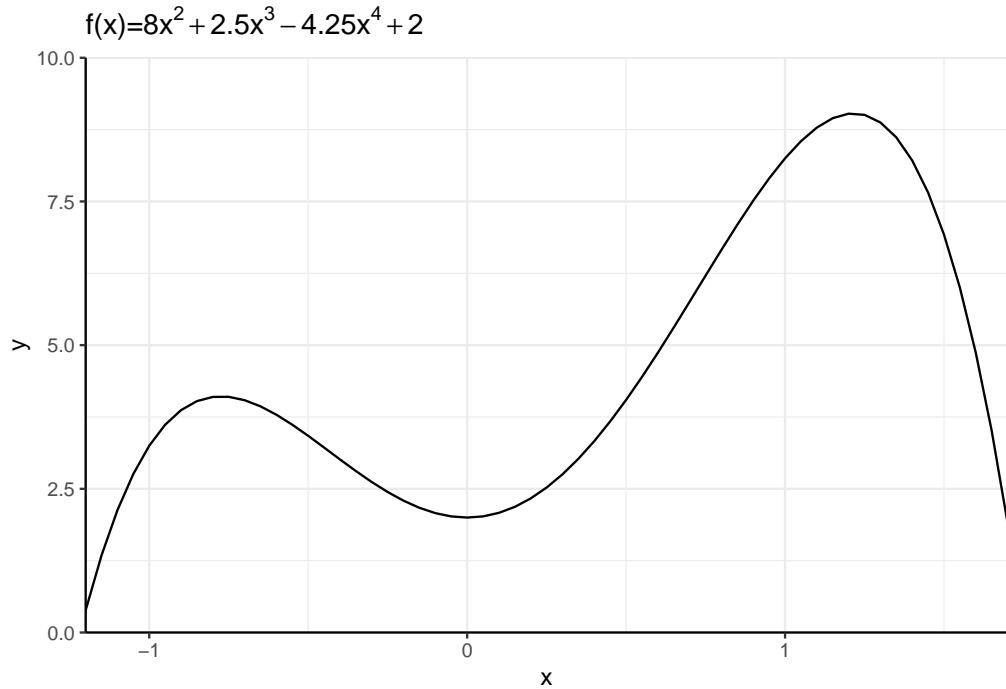


Figure 6.6: Graph der zu optimierenden Funktion.

Da es sich hier um ein eindimensionales Problem handelt, können wir die allgemeine Funktion `optimize()` verwenden. Wir übergeben als Argument `f` die zu optimierende Funktion und geben über `interval` das Intervall an, in dem nach einem Minimum (oder Maximum) gesucht werden soll:

```
opt_obj <- optimize(f = f_1, interval = c(-1.25, 1.75))
opt_obj
```

```
#> $minimum
#> [1] -7.54766e-06
#>
#> $objective
#> [1] 2
```

Das Ergebnis ist eine Liste mit zwei Elementen. Dem x-Wert des gesuchten Minimums:<sup>6</sup>

```
opt_obj[["minimum"]]
```

```
#> [1] -7.54766e-06
```

Und dem dazugehörigen Funktionswert:

```
opt_obj[["objective"]]
```

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

Falls wir ein Maximum suchen setzen wir `maximum=TRUE`:

```
opt_obj_max <- optimize(
  f = f_1, interval = c(-1.25, 1.75), maximum = TRUE)
opt_obj_max
```

```
#> $maximum
#> [1] 1.215492
#>
#> $objective
#> [1] 9.032067
```

Falls wir den Suchbereich entsprechend einschränken finden wir das lokale Maximum auf der linken Seite:

```
opt_obj_max <- optimize(
  f = f_1, interval = c(-1.25, 0), maximum = TRUE)
opt_obj_max

#> $maximum
#> [1] -0.7743199
#>
#> $objective
#> [1] 4.108106
```

Wir sind übrigens nicht auf eindimensionale Funktionen beschränkt. Wir können z.B. auch die folgende Zielfunktion optimieren:

$$f(x, y) = (a - x)^2 + b(y - x^2)^2$$

```
f_2 <- function(x, a=1, b=100){
  (a - x[1])**2 + b*(x[2]-x[1]**2)**2
}
```

---

<sup>6</sup>Minimale Rundungsfehler sind bei solchen numerischen Verfahren normal, daher wird Ihnen in diesem Fall nicht ‘exakt’ Null als Ergebnis angezeigt.

Bei dieser Funktion handelt es sich um die in der Optimierung sehr häufig als Benchmark verwendete [Rosenbrock Funktion](#). Grafisch können wir solche Funktionen mit Hilfe einer *Heatmap* darstellen, wobei wir in unserer Visualisierung in Abbildung 6.7 annehmen, dass  $a = 1$  und  $b = 100$ . In einer Heatmap geben die beiden Achsen die Kombination der  $x$  und  $y$ -Werte an, der resultierende Funktionswert wird über die Farbe repräsentiert.

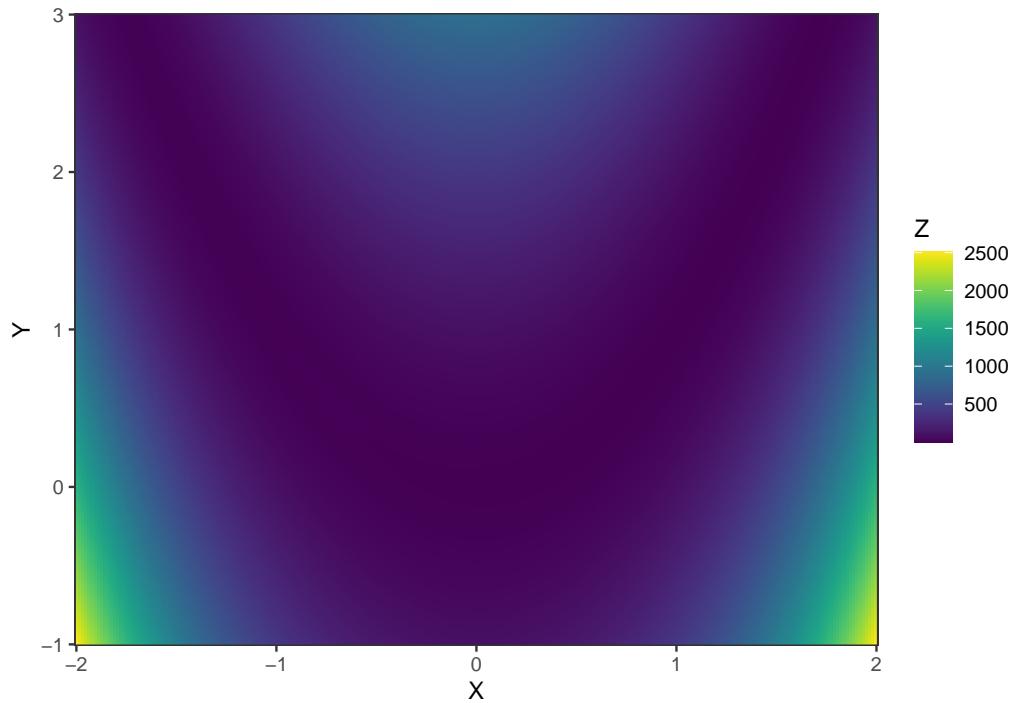


Figure 6.7: Darstellung einer Funktion als Heatmap

Da es sich jetzt um ein mehrdimensionales Problem handelt verwenden wir die Funktion `optimx::optimr()` anstatt von `optimize()`. Die Handhabung ist aber sehr ähnlich. Als erstes Argument übergeben wir `par` unsere ersten Vermutungen für das Extremum, also die Werte, mit der die Funktion ihre Suche beginnen soll. Danach als zweites Argument `fn` die zu optimierende Funktion. Falls diese Funktion noch weitere Argumente akzeptiert können wir die hier auch einfach hinzufügen. Für unseren Fall haben wir also:

```
opt_objekt <- optimx::optimr(
  par = c(1, 1),
  fn = f_2
)
```

Zunächst schauen wir ob der Algorithmus erfolgreich einen Extremwert gefunden hat. Bei erfolgreicher Suche hat der Listeneintrag `convergence` den Wert 0:

```
opt_objekt[["convergence"]] == 0
```

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

Die optimalen Argumente erhalten wir über den Listeneintrag `par`:

```
opt_objekt[["par"]]
```

```
#> [1] 1 1
```

Und den Wert der Zielfunktion im Extremum über den Listeneintrag `value`:

```
opt_objekt[["value"]]
```

```
#> [1] 0
```

Wenn wir `optimx::optimr()` übrigens zur Maximierung einsetzen wollen müssen wir nichts weiter tun als dem Argument `control` eine Liste mit dem Eintrag `fnscale=-1` zu übergeben:

```
opt_objekt <- optimx::optimr(
  par = c(1, 1),
  fn = f_2, control = list(fnscale=-1)
)
```

```
opt_objekt$convergence == 0
```

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

```
opt_objekt$par
```

```
#> [1] 3.661667e+76 -3.087043e+76
```

```
opt_objekt$value
```

```
#> [1] 1.797693e+308
```

### 6.2.6 Anwendungsbeispiel

Als Anwendungsbeispiel betrachten wir das klassische keynesianische Modell. Am geläufigsten ist dabei folgende Formulierung:

$$Y = C + I + G \quad (6.18)$$

$$C = c_0 + c_1 Y \quad (6.19)$$

In dem Modell geht man davon aus, dass sich die gesamtwirtschaftliche Güternachfrage  $Y$  aus dem Konsum  $C$ , den Investitionen  $I$  sowie den Staatsausgaben  $G$  ergibt. Die Konsumfunktion selbst wird als lineare Funktion modelliert, wobei  $c_0$  den autonomen Konsum (also den vom Einkommen unabhängigen Konsum) und  $c_1$  die marginale Konsumquote beschreibt.

Wir können nun die Notation leicht um  $T$  als die Steuerlast erweitern:

$$Y = \frac{c_0 + I + G}{1 - c_1(1 - T)} \quad (6.20)$$

Wenn wir nun wissen wollen wie  $Y$  auf eine Änderung der Staatsausgaben reagiert können wir diese Formel nach  $G$  ableiten. Dazu müssten wir gleich mehrere Regeln, die wir oben kennen gelernt haben, anwenden.

Aber natürlich können wir das Ganze ganz einfach in R lösen. Um die Ableitung herzuleiten verwenden wir dabei einfach wieder die Funktion `D()`:

```
keynes_model <- expression(Y=(c_0 + I + G) / (1 - c_1*(1-T)))
D(expr = keynes_model, name = "G")
#> 1/(1 - c_1 * (1 - T))
```

Es gilt also:

$$\frac{\partial Y}{\partial G} = \frac{1}{1 - c_1(1 - T)} \quad (6.21)$$

Nehmen wir einmal an die marginale Konsumquote  $c_1$  läge bei 20% und der Steuersatz  $T$  bei 25%. Eine Erhöhung der Staatsausgaben würde dann  $Y$  über den Multiplikator  $\frac{1}{1-0.2(1-0.25)} = 1.176471$  erhöhen.

Alternativ können wir das Ergebnis natürlich analytisch unter Zuhilfenahme der oben eingeführten Ableitungsregeln herleiten.

## 6.3 Lineare Algebra

Ebenfalls sehr häufig werden Sie mit Matrizen und den dazugehörigen Rechenoperationen ('Matrizenalgebra' genannt) in Kontakt kommen. Das Ziel dieses Abschnitts ist keine abschließende Einführung in Matrizen und Matrizenalgebra, sondern soll dazu dienen, einen groben Überblick über typische Rechenoperationen und deren Implementierung in R zu bekommen. Für eine ausführlichere Einführung verweisen wir auf [Wainwright and Chiang \(2005\)](#) oder [Aleskerov et al. \(2011\)](#).

Matrizen werden häufig im Kontext der *linearen Algebra* verwendet.<sup>7</sup> Zahlreiche sozioökonomische Konzepte bedienen sich der linearen Algebra, in der Matrizen häufig verwendet werden, um lineare Gleichungssysteme darzustellen. Die Matrixdarstellung ist dabei nicht nur kompakter, sie erlaubt es uns auch relativ leicht zu überprüfen ob das System konsistent und lösbar ist. Die folgenden zwei Beispiele machen dies hoffentlich deutlich.

### 6.3.1 Einführungsbeispiele

Das erste Beispiel bezieht sich wieder auf das oben eingeführte klassischen Keynesianische Modell:

$$Y = C + I + G \quad (6.22)$$

$$C = c_0 + c_1 Y \quad (6.23)$$

Nehmen wir nun an, die Staatsausgaben und Investitionen wären exogen bekannt. Dann kann dieses Modell äquivalent in Matrixform geschrieben werden:

$$Ax = d \quad (6.24)$$

---

<sup>7</sup>Das liegt daran, dass jede  $n \times k$ -Matrix  $A$ , also eine Matrix mit  $n$  Zeilen und  $k$  Spalten, als eine Funktion  $f(x) = Ax$  dargestellt werden kann, für die gilt:  $f : \mathbb{R}^n \rightarrow \mathbb{R}^k$ . Diese Funktion ist immer *linear*. Tatsächlich gilt, dass jede Funktion  $f$  nur dann linear ist, wenn es eine Matrix  $A$  gibt, für die gilt  $f(x) = Ax$ .

wobei  $A = \begin{pmatrix} 1 & -1 \\ -c_1 & 1 \end{pmatrix}$ ,  $x = \begin{pmatrix} Y \\ C \end{pmatrix}$  und  $d = \begin{pmatrix} I + G \\ c_0 \end{pmatrix}$ , wobei die beiden Unbekannten in diesem Fall das Einkommen  $Y$  und der Konsum  $C$  sind.

Matrizen helfen uns solche Gleichungssysteme komprimiert darzustellen und zu analysieren, insbesondere um zu testen ob es Werte für die freien Parameter - hier  $Y$  und  $C$  - gibt sodass das gesamte System konsistent ist. Wir sehen unten wie genau wir solche Systeme in R recht einfach lösen können.

Ein weiteres Beispiel wo wir - vielleicht auch häufig unbewusst - Methoden der linearen Algebra verwenden ist in der Ökonometrie. So wird das einfache lineare Regressionsmodell für  $n$  Beobachtungen und  $p$  erklärenden Variablen häufig folgendermaßen beschrieben (siehe Kapitel ??):

$$Y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \dots + \beta_p x_{ip} + \epsilon, i = 1, \dots, n \quad (6.25)$$

Da wir in der Praxis regelmäßig mehr als eine erklärende Variable verwenden (also  $p > 1$ ) werden Schätzgleichungen fast ausschließlich in Matrixform dargestellt, denn wir können explizit alle  $n$  Gleichungen untereinander schreiben:

$$\begin{aligned} Y_1 &= \beta_0 + \beta_1 x_{11} + \beta_2 x_{12} + \dots + \beta_p x_{1p} + e_1 \\ Y_2 &= \beta_0 + \beta_1 x_{21} + \beta_2 x_{22} + \dots + \beta_p x_{2p} + e_2 \\ &\vdots \\ Y_n &= \beta_0 + \beta_1 x_{n1} + \beta_2 x_{n2} + \dots + \beta_p x_{np} + e_n \end{aligned}$$

Und dann in Matrixform ausdrücken:

$$\begin{pmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_n \end{pmatrix} = \begin{pmatrix} 1 & x_{11} & x_{12} & \dots & x_{1p} \\ 1 & x_{21} & x_{22} & \dots & x_{2p} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & x_{n2} & \dots & x_{np} \end{pmatrix} \times \begin{pmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_n \end{pmatrix} + \begin{pmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_n \end{pmatrix} \quad (6.26)$$

Und Letzteres wie folgt schreiben:

$$Y = X\beta + \epsilon \quad (6.27)$$

Dementsprechend können wir auch den OLS-Schätzer in Matrixform darstellen, was ab Kapitel ?? auch die standardmäßige Darstellungsform sein wird. Dies erlaubt einfachere und allgemeinere Beweise, und ist vor allem für die algorithmische Implementierung sehr wichtig. Auch wenn wir uns mit diesen Details nicht notwendigerweise genau auseinandersetzen müssen, sollte die grundlegende Rolle der linearen Algebra doch nicht unterschätzt werden. Wir werden das Beispiel des OLS-Schätzers unten noch genauer besprechen. Zunächst beginnen wir mit einer allgemeinen Einführung in den Umgang mit Matrizen in R.

### 6.3.2 Einführung von Matrizen

Technisch handelt es sich bei Matrizen um zweidimensionale Objekte mit Zeilen und Spalten, bei denen es sich jeweils um atomare Vektoren handelt.

In R werden Matrizen mit der Funktion `matrix()` erstellt. Diese Funktion nimmt als erstes Argument die Elemente der Matrix und dann die Spezifikation der Anzahl von Zeilen (`nrow`) und/oder der Anzahl von Spalten (`ncol`):

```
m_1 <- matrix(11:20, nrow = 5)
m_1
```

```
#>      [,1] [,2]
#> [1,]    11   16
#> [2,]    12   17
#> [3,]    13   18
#> [4,]    14   19
#> [5,]    15   20
```

Wie können die Zeilen, Spalten und einzelne Werte folgendermaßen extrahieren und ggf. Ersetzungen vornehmen:

```
m_1[, 1] # Erste Spalte
```

```
#> [1] 11 12 13 14 15
```

```
m_1[1,] # Erste Zeile
```

```
#> [1] 11 16
```

```
m_1[2, 2] # Element [2,2]
```

```
#> [1] 17
```

Es gibt einige **besondere Matrizen**, die aufgrund ihrer speziellen Eigenschaften Eigennamen erhalten haben.

Eine Matrix mit der gleichen Anzahl von Zeilen und Spalten wird **quadratische Matrix** genannt.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \quad (6.28)$$

Die Elemente auf der ‘Diagonalen’ einer quadratischen  $n \times n$ -Matrix, also  $\{a_{ii}\}_{i=1}^n$ , werden die *Hauptdiagonale* dieser Matrix genannt.

Eine Matrix, die von Null verschiedene Einträge nur auf der Hauptdiagonale aufweist heißt **Diagonalmatrix**:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 4 \end{pmatrix} \quad (6.29)$$

Bei der **oberen Dreiecksmatrix** befinden sich von Null verschiedene Einträge ausschließlich auf oder über der Hauptdiagonale, bei der **unteren Dreiecksmatrix** ist dies genau umgekehrt. Hier ein Beispiel für eine untere Dreiecksmatrix:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 2 & 0 & 0 \\ 1 & 2 & 3 & 0 \\ 1 & 2 & 3 & 4 \end{pmatrix} \quad (6.30)$$

Bei der **Identitätsmatrix** (oder: ‘Einheitsmatrix’) handelt es sich um eine quadratische Matrix, die auf der Hauptdiagonalen nur 1er und neben der Haupdiagonalen nur 0er enthält. Sie wird mit  $\mathbb{I}_n$  bezeichnet, wobei  $n$  die Anzahl der Zeilen und Spalten angibt:

$$\mathbb{I}_4 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (6.31)$$

Wird eine beliebige Matrix mit einer passenden Identitätsmatrix multipliziert, ist das Ergebnis die ursprüngliche Matrix selbst, daher der Name. Wir können  $\mathbb{I}_n$  in R mit `diag(n)` direkt erstellen.

### 6.3.3 Grundregeln der Matrizenalgebra

Matrizenalgebra spielt in vielen statistischen Anwendungen eine wichtige Rolle. Sie funktioniert aber ein wenig anders als die ‘herkömmliche’ Algebra, mit denen die meisten von Ihnen schon vertraut sein werden. Zum Glück ist es in R sehr einfach die typischen Rechenoperationen für Matrizen zu implementieren. Im Folgenden werden wir die wichtigsten Rechenregeln für Matrizen kurz einführen und dabei die folgenden Beispielmatrizen verwenden:

$$A = \begin{pmatrix} 1 & 6 \\ 5 & 3 \end{pmatrix} \quad B = \begin{pmatrix} 0 & 2 \\ 4 & 8 \end{pmatrix} \quad (6.32)$$

```
matrix_a <- matrix(c(1,5,6,3), ncol = 2)
matrix_b <- matrix(c(0,4,2,8), ncol = 2)
```

#### Matrix-Transponierung

Die transponierte Matrix  $A'$  ergibt sich aus  $A$  indem die Spalten und Zeilen vertauscht werden. Im Folgenden ist unsere Beispielmatrix und ihre Transponierung dargestellt:

$$A = \begin{pmatrix} 1 & 6 \\ 5 & 3 \end{pmatrix} \quad A' = \begin{pmatrix} 1 & 5 \\ 6 & 3 \end{pmatrix} \quad (6.33)$$

In R können wir eine Matrix mit der Funktion `t()` transponieren:

```
matrix_a
#>      [,1] [,2]
#> [1,]    1    6
#> [2,]    5    3
t(matrix_a)
#>      [,1] [,2]
#> [1,]    1    5
#> [2,]    6    3
```

**Skalar-Addition**

$$4 + A = \begin{pmatrix} 4 + a_{11} & 4 + a_{21} \\ 4 + a_{12} & 4 + a_{22} \end{pmatrix} \quad (6.34)$$

In R:

```
4 + matrix_a
#>      [,1] [,2]
#> [1,]    5   10
#> [2,]    9   7
```

**Matrizen-Addition**

$$A + B = \begin{pmatrix} a_{11} + b_{11} & a_{21} + b_{21} \\ a_{12} + b_{12} & a_{22} + b_{22} \end{pmatrix} \quad (6.35)$$

```
matrix_a + matrix_b
```

```
#>      [,1] [,2]
#> [1,]    1    8
#> [2,]    9   11
```

**Skalar-Multiplikation**

$$2 \cdot A = \begin{pmatrix} 2 \cdot a_{11} & 2 \cdot a_{21} \\ 2 \cdot a_{12} & 2 \cdot a_{22} \end{pmatrix} \quad (6.36)$$

```
2*matrix_a
```

```
#>      [,1] [,2]
#> [1,]    2   12
#> [2,]   10    6
```

**Elementenweise Matrix Multiplikation (auch ‘Hadamard-Produkt’)**

$$A \odot B = \begin{pmatrix} a_{11} \cdot b_{11} & a_{21} \cdot b_{21} \\ a_{12} \cdot b_{12} & a_{22} \cdot b_{22} \end{pmatrix} \quad (6.37)$$

```
matrix_a * matrix_b
```

```
#>      [,1] [,2]
#> [1,]     0   12
#> [2,]    20   24
```

### Matrizen-Multiplikation

$$A \cdot B = \begin{pmatrix} a_{11} \cdot b_{11} + a_{12} \cdot b_{21} & a_{11} \cdot b_{21} + a_{12} \cdot b_{22} \\ a_{21} \cdot b_{11} + a_{22} \cdot b_{21} & a_{21} \cdot b_{12} + a_{22} \cdot b_{22} \end{pmatrix} \quad (6.38)$$

```
matrix_a %*% matrix_b
```

```
#>      [,1] [,2]
#> [1,]    24   50
#> [2,]    12   34
```

Wir wissen von oben auch, dass  $A\mathbb{I}_2 = A$ :

```
matrix_a
```

```
#>      [,1] [,2]
#> [1,]     1   6
#> [2,]     5   3
```

```
matrix_a %*% diag(2)
```

```
#>      [,1] [,2]
#> [1,]     1   6
#> [2,]     5   3
```

### Matrizen invertieren

Die Inverse einer Matrix  $A$ ,  $A^{-1}$ , ist definiert sodass gilt

$$AA^{-1} = I \quad (6.39)$$

Sie kann in R mit der Funktion `inv()` aus dem Paket `matlib`<sup>8</sup> identifiziert werden, wobei wir die Matrix als erstes Argument `X` an `inv()` übergeben:

```
inv(X = matrix_a)
```

```
#>      [,1]      [,2]
#> [1,] -0.1111111  0.22222222
```

---

<sup>8</sup>Alternativ können Sie auch die Funktion `solve()` aus `base` verwenden; hier ist das erste Argument `a` und der Output ist weniger informativ.

```
#> [2,] 0.1851852 -0.03703704
matrix_a %*% inv(matrix_a)
#>      [,1] [,2]
#> [1,] 1e+00 -2e-08
#> [2,] 2e-08  1e+00
```

Die minimalen Abweichungen sind auf maschinelle Rundungsfehler zurückzuführen und treten häufig auf.

Gerade die letzte Operation ist zentral um zu verstehen wie wir mit Hilfe der Matrizenalgebra lineare Gleichungssysteme wie oben beschrieben lösen können. Denn diese Gleichungssysteme können - wie in der Einleitung beschrieben - in die Form

$$Ax = b \quad (6.40)$$

gebracht werden. In Anwendungsfällen ist  $A$  eine Matrix mit Koeffizienten,  $x$  ein Vektor von unbekannten Variablen und  $b$  ein Vektor mit Konstanten. Entsprechend ist unser Interesse in der Identifikation eines Vektors  $x$  sodass die Gleichung konsistent ist und mindestens eine Lösung hat. Wenn wir die Gleichung gemäß der gerade beschriebenen Regeln umformen bekommen wir:

$$A^{-1}Ax = A^{-1}b \quad (6.41)$$

$$x = A^{-1}b \quad (6.42)$$

In der Matrzenschreibweise korrespondiert die Lösung eines solchen Systems also zur Invertierung der Matrix  $A$  - daher auch der Name der R-Funktion `solve()` aus dem Paket `base`.

Nehmen wir also einmal folgenden Fall an:  $A = \begin{pmatrix} 1 & 3 \\ -2 & 1 \end{pmatrix}$  und  $b = \begin{pmatrix} 9 \\ 4 \end{pmatrix}$ .

In diesem Fall können wir das Gleichungssystem in R lösen indem wir `Solve()` direkt die Matrix  $A$  (über das Argument `A`) und den Vektor  $b$  (über das Argument `b`) übergeben:

```
A <- matrix(c(1, -2, 3, 1), ncol = 2)
b <- matrix(c(9, -4), ncol = 1)
Solve(A = A, b = b)
```

```
#> x1      =  3
#> x2      =  2
```

Wir sehen also unmittelbar, dass das Gleichungssystem - und damit unser Modell - konsistent ist und eine eindeutige Lösung  $x = \begin{pmatrix} 3 \\ 2 \end{pmatrix}$  aufweist.<sup>9</sup> Dieses können wir folgendermaßen verifizieren:

```
x <- matrix(c(3, 2), ncol = 1)
A %*% x
```

```
#>      [,1]
```

<sup>9</sup>Wenn Sie die einzelnen Schritte zur Lösung nachverfolgen wollen, rufen Sie die Funktion mit dem Argument `verbose=TRUE` auf!

```
#> [1,]    9
#> [2,]   -4
```

Wie erwartet erhalten wir hier also wieder unseren ursprünglichen Wert für  $b$ .

Wenn allerdings  $A = \begin{pmatrix} -2 & 1 \\ -4 & 2 \end{pmatrix}$  und  $b = \begin{pmatrix} 3 \\ 2 \end{pmatrix}$ , dann würde Folgendes passieren:

```
A <- matrix(c(-2, -4, 1, 2), ncol = 2)
b <- matrix(c(3, 2), ncol = 1)
Solve(A = A, b = b)
```

```
#> x1 - 0.5*x2  =  -0.5
#>          0  =      2
```

Wir sehen also direkt, dass das System nicht lösbar wäre, denn das resultierende Gleichungssystem weist einen eindeutigen Widerspruch ( $0=2$ ) auf. Der Grund ist, dass die Matrix  $A$  *singulär* ist, das heißt sie besitzt keine Inverse. Das können Sie unmittelbar überprüfen:

```
inv(A)
```

```
#> Error in Inverse(X, tol = sqrt(.Machine$double.eps), ...): X is numerically singular
```

Wir können also nur über die Analyse der Matrix Schlussfolgerungen bezüglich des gesamten Gleichungssystems ziehen. Das ist in der Praxis, in dem die Gleichungssysteme ungleich größer und komplexer sind, von enormer Bedeutung.

Ein dritter möglicher Fall tritt ein wenn  $A = \begin{pmatrix} 1 & 3 \\ -2 & 1 \end{pmatrix}$  und  $b = \begin{pmatrix} 9 \\ 4 \end{pmatrix}$ :

```
A <- matrix(c(4, -2, -2, 1), ncol = 2)
b <- matrix(c(6, -3), ncol = 1)
Solve(A = A, b = b)
```

```
#> x1 - 0.5*x2  =  1.5
#>          0  =      0
```

In diesem Falle sehen wir keinen Widerspruch im Gleichungssystem, aber auch kein eindeutiges Ergebnis. Das Gleichungssystem hat also *unendlich viele* Lösungen.

Zur Vollständigkeit seien hier noch einmal die drei möglichen Ergebnisse einer solchen Matrizenanalyse kurz beschrieben:

1. Das Gleichungssystem hat *unendlich viele* Lösungen, wir können also auf Basis der Struktur keine genaue Vorhersage bezüglich der Parameter in  $x$  machen.
2. Das Gleichungssystem hat eine *eindeutige* Lösung, wir haben also ein konsistentes Modell, das eine eindeutige Vorhersage produziert.
3. Das Gleichungssystem hat *keine* Lösung, unser Modell ist also inkonsistent.

Im Folgenden werden wir uns das anhand der beiden Beispiele aus dem Abschnitt 6.3.1 genauer anschauen.

### 6.3.4 Anwendungsbeispiel 1: Das einfache Keynesianische Modell

In der Einleitung dieses Unterkapitels haben wir schon gesehen, dass wir das einfache Keynesianische Modell

$$Y = C + I + G \quad (6.43)$$

$$C = a + bY \quad (6.44)$$

auch in Matrizenbeschreibweise darstellen können:

$$Ax = d \quad (6.45)$$

$$\text{wobei } A = \begin{pmatrix} 1 & -1 \\ -b & 1 \end{pmatrix}, x = \begin{pmatrix} Y \\ C \end{pmatrix} \text{ und } d = \begin{pmatrix} I + G \\ a \end{pmatrix}.$$

Der Vorteil ist, dass wir unmittelbar überprüfen können ob das System für bestimmte Werte konsistent ist und eine eindeutige Lösung für  $Y$  und  $C$  besitzt.

Sind z.B. die Staatsausgaben mit  $G = 2$  und die Investitionen mit  $I = 2$  bekannt, und die marginale Konsumneigung mit  $b = 0.4$  und der einkommensunabhängige Konsum mit  $a = 1$  gegeben, können wir direkt überprüfen ob das System konsistent ist und, da  $x = \begin{pmatrix} Y \\ C \end{pmatrix}$  welche Werte für den Konsum und das Gesamteinkommen impliziert werden.

```
I_keynes <- 2
G_keynes <- 2
b_keynes <- 0.4
a_keynes <- 1

A_keynes <- matrix(c(1, -b_keynes, -1, 1), nrow = 2)
d_keynes <- matrix(c(I_keynes + G_keynes, a_keynes), ncol = 1)
Solve(A = A_keynes, b = d_keynes)

#> x1      =  8.33333333
#> x2      =  4.33333333
```

In diesem Fall sehen wir, dass das System konsistent ist und eine eindeutige Lösung für das Einkommen  $Y = 8\frac{1}{3}$  und den Konsum  $C = 4\frac{1}{3}$  impliziert.

### 6.3.5 Anwendungsbeispiel 2: OLS-Regression

Aus der Einleitung dieses Unterkapitels wissen wir, dass wir das lineare Regressionsmodell mit  $n$  Beobachtungen von  $p$  Variablen

$$\begin{aligned} Y_1 &= \beta_0 + \beta_1 x_{11} + \beta_2 x_{21} + \dots + \beta_p x_{1p} + \epsilon_1 \\ Y_2 &= \beta_0 + \beta_1 x_{21} + \beta_2 x_{22} + \dots + \beta_p x_{2p} + \epsilon_2 \\ &\vdots \\ Y_n &= \beta_0 + \beta_1 x_{n1} + \beta_2 x_{n2} + \dots + \beta_p x_{np} + \epsilon_n \end{aligned}$$

auch folgendermaßen schreiben können:

$$Y = X\beta + \epsilon \quad (6.46)$$

Wobei  $Y$  eine  $n \times 1$ -Matrix mit den Beobachtungen für die abhängige Variable,  $X$  eine  $n \times p$ -Matrix in der jede Spalte zu einem Vektor mit allen  $n$  Beobachtungen einer der  $p$  erklärenden Variablen korrespondiert.  $\epsilon$  schließlich ist die  $n \times 1$ -Matrix der Fehlerterme.

Nehmen wir folgenden Datensatz an:

```
#>           Auto Verbrauch PS Zylinder
#> 1: Ford Pantera L      15.8 264     8
#> 2: Ferrari Dino       19.7 175     6
#> 3: Maserati Bora       15.0 335     8
#> 4: Volvo 142E          21.4 109     4
```

Dies können wir schreiben als:

$$\begin{aligned} y_1 &= \beta_0 + \beta_1 x_{11} + \beta_2 x_{12} + \epsilon_1 \\ y_1 &= \beta_0 + \beta_1 x_{21} + \beta_2 x_{22} + \epsilon_2 \\ y_1 &= \beta_0 + \beta_1 x_{31} + \beta_2 x_{32} + \epsilon_3 \\ y_1 &= \beta_0 + \beta_1 x_{41} + \beta_2 x_{42} + \epsilon_4 \end{aligned} \quad (6.47)$$

und mit Zahlen:

$$\begin{aligned} 15.8 &= \beta_0 + \beta_1 264 + \beta_2 8 + \epsilon_1 \\ 19.7 &= \beta_0 + \beta_1 175 + \beta_2 6 + \epsilon_2 \\ 15.0 &= \beta_0 + \beta_1 335 + \beta_2 8 + \epsilon_3 \\ 21.4 &= \beta_0 + \beta_1 109 + \beta_2 4 + \epsilon_4 \end{aligned} \quad (6.48)$$

Und als Matrix:

$$\begin{pmatrix} 1 & 264 & 8 \\ 1 & 175 & 6 \\ 1 & 335 & 8 \\ 1 & 109 & 4 \end{pmatrix} \times \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{pmatrix} + \begin{pmatrix} \epsilon_1 \\ \epsilon_2 \\ \epsilon_3 \\ \epsilon_4 \end{pmatrix} = \begin{pmatrix} 15.8 \\ 19.7 \\ 15.0 \\ 21.4 \end{pmatrix}$$

Es gilt also, dass  $\hat{\beta} = \begin{pmatrix} \hat{\beta}_0 \\ \hat{\beta}_1 \\ \hat{\beta}_2 \end{pmatrix}$ ,  $X = \begin{pmatrix} 1 & 264 & 8 \\ 1 & 175 & 6 \\ 1 & 335 & 8 \\ 1 & 109 & 4 \end{pmatrix}$  und  $y = \begin{pmatrix} 15.8 \\ 19.7 \\ 15.0 \\ 21.4 \end{pmatrix}$ .

Es lässt sich allgemein zeigen, dass der gesuchte Schätzer  $\hat{\beta} = \begin{pmatrix} \hat{\beta}_0 \\ \hat{\beta}_1 \\ \hat{\beta}_2 \end{pmatrix}$  für das unbekannte  $\beta = \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{pmatrix}$  die Lösung des folgenden Gleichungssystems darstellt:<sup>10</sup>

$$\hat{\beta} = (X'X)^{-1} X'Y \quad (6.49)$$

Das können wir wiederum in R lösen:

```
ols_X <- matrix(c(1, 264, 8, 1, 175, 6, 1, 335, 8, 1, 109, 4),
                  ncol = 3, byrow = T)
ols_y <- matrix(c(15.8, 19.7, 15.0, 21.4), ncol = 1)

solve(t(ols_X) %*% ols_X) %*% t(ols_X) %*% ols_y

#> [1] 26.37086491
#> [2,] -0.01783627
#> [3,] -0.68592421
```

Oder direkt mit lm():

```
lm(Verbrauch ~ PS + Zylinder, data = ols_beispiel)

#>
#> Call:
#> lm(formula = Verbrauch ~ PS + Zylinder, data = ols_beispiel)
#>
#> Coefficients:
#> (Intercept)          PS      Zylinder
#>     26.37086       -0.01784      -0.68592
```

### 6.3.6 Optional: Herleitung des OLS-Schätzers

Mit dem bislang gewonnenen Verständnis von Matrizenalgebra ist es bereits möglich die Herleitung des OLS-Schätzers nachzuvollziehen. Diese Herleitung wird im Folgenden beschrieben.

Wir wissen bereits, dass die Residuen einer Schätzung gegeben sind durch:

$$e = Y - X\hat{\beta}$$

Wir können die Summe der quadrierten Residuen (RSS) in Matrixschreibweise schreiben als:

---

<sup>10</sup>Die genaue Herleitung finden Sie im nächsten (optionalen) Abschnitt.

$$e'e = \begin{pmatrix} e_1 & e_2 & \dots & e_n \end{pmatrix} \begin{pmatrix} e_1 \\ e_2 \\ \vdots \\ e_n \end{pmatrix} = \begin{pmatrix} e_1 \times e_1 & e_2 \times e_2 & \dots & e_n \times e_n \end{pmatrix} \quad (6.50)$$

Wir können dann schreiben:<sup>11</sup>

$$\begin{aligned} e'e &= (Y - X\hat{\beta})' (Y - X\hat{\beta}) \\ &= y'y - \hat{\beta}'X'y - y'X\hat{\beta} + \hat{\beta}'X'X\hat{\beta} \\ &= y'y - 2\hat{\beta}'X'y + \hat{\beta}'X'X\hat{\beta} \end{aligned}$$

Wir wollen diesen Ausdruck nun minimieren. Dazu leiten wir nach dem Vektor der zu schätzenden Koeffizienten  $\hat{\beta}$  ab:

$$\frac{\partial e'e}{\partial \hat{\beta}} = -2X'y + 2X'X\hat{\beta} = 0 \quad (6.51)$$

Diese Gleichung können wir nun umformen zu:

$$\begin{aligned} 2X'X\hat{\beta} &= 2X'Y \\ X'X\hat{\beta} &= X'Y \end{aligned}$$

Da gilt, dass  $(X'X)^{-1}(X'X) = I$  multiplizieren wir beide Seiten mit  $(X'X)^{-1}$ :<sup>12</sup>

$$\begin{aligned} (X'X)^{-1}X'X\hat{\beta} &= (X'X)^{-1}X'Y \\ \hat{\beta} &= (X'X)^{-1}(X'Y) \end{aligned} \quad (6.52)$$

Damit haben wir den Schätzer für  $\hat{\beta}$  hergeleitet.

### 6.3.7 Weiterführende Literatur

Es gibt im Internet zahlreiche gute Überblicksartikel zum Thema Matrizenalgebra in R, z.B. [hier](#) oder in größerem Umfang [hier](#). Auch das Angebot an Lehrbüchern ist sehr groß, für die ökonomischen Grundlagen bietet sich [Wainwright and Chiang \(2005\)](#) sehr gut an.

---

<sup>11</sup>Beachte dabei, dass  $Y'X\hat{\beta} = (Y'X\hat{\beta})' = \hat{\beta}'X'Y$ .

<sup>12</sup>Hier liegt übrigens auch der Grund für die OLS-Annahme, dass keine perfekte Multikollinearität besteht: denn in diesem Fall wäre eine Zeile der Matrix  $X$  eine lineare Kombination einer anderen Zeile und  $X$  wäre damit nicht mehr invertierbar, also  $X^{-1}$  würde nicht existieren und  $\hat{\beta}$  wäre nicht mehr definiert.

## 6.4 Analyse von Verteilungen

Fragen nach Verteilungen stehen im Zentrum vieler sozioökonomischer Arbeiten. Verteilung von Einkommen und Vermögen, sozialem, kulturellem oder physischem Kapital, Firmenproduktivitäten oder natürlichen Ressourcen - in vielen Bereichen geht es Verteilungen.

Gleichzeitig spielen Verteilungen in der technischen Literatur eine wichtige Rolle: in der Ökonometrie ist die Verteilung von Schätzern von zentraler Bedeutung, viele formale Konzepte setzen eine bestimmte Verteilung der Daten voraus und häufig bedarf es zur richtigen Wahl der quantitativen Methoden zumindest rudimentärer Kenntnis über die Verteilung die Daten.

Kurzum: Wissen über Verteilungen und deren Analyse ist für die sozioökonomische Forschungspraxis extrem hilfreich. Daher wollen wir uns im Folgenden mit verschiedenen Aspekten der Analyse von Verteilungen beschäftigen.

Wir steigen mit einer Erläuterung des (mathematischen) [Verteilungsbegriffs](#) ein und diskutieren den Zusammenhang zwischen Verteilungen und stochastischen Prozessen. Verteilungen sind nämlich immer dann zentral, wenn wir es mit probabilistischen Prozessen zu tun haben.

Als nächstes lernen wir [typische Kennzahlen](#) zur Beschreibung von Verteilungen kennen. Besonderes Augenmerk legen wir dabei auf Kennzahlen zur Streuung und Ungleichheit, wie die Standardabweichung oder den Gini Index.

Daraufhin lernen wir einige [grafische Methoden](#) kennen, um die wir die quantitativen Kennzahlen immer ergänzen sollten und schließen das Kapitel zuletzt mit einigen [abschließenden Bemerkungen](#) ab.

In diesem Abschnitt werden mehrere Konzepte aus der Stochastik vorausgesetzt. Wenn Sie sich damit noch unsicher fühlen empfiehlt sich vorher eine Lektüre von Kapitel 7.

### 6.4.1 Theoretische und empirische Verteilungen

Wenn wir über Verteilungen sprechen wird der Begriff (mindestens) in zwei verwandten aber unterschiedlichen Arten verwendet: im Sinne der **Verteilung einer Zufallsvariablen** und im Sinne einer **empirischen Beschreibung**.

Eine empirische Verteilung beschreiben wir in der Regel durch bestimmte Kennzahlen, wie den Mittelwert, die Standardabweichung oder den Gini-Index. Das erlaubt uns Informationen über die Daten in wenigen Zahlen zu kondensieren.<sup>13</sup>

Dennoch werden beide Perspektiven auch häufig kombiniert, vor allem wenn wir einen empirischen Datensatz mit einem parametrischen Wahrscheinlichkeitsmodell beschreiben wollen. Das bedeutet, dass wir die empirischen Daten als Realisierung einer theoretischen Zufallsvariablen (ZV) interpretieren und die für die theoretische ZV relevanten Parameter dann aus den Daten heraus schätzen.<sup>14</sup>

#### Anwendungsbeispiel

Stellen Sie sich vor Sie haben eine Stichprobe vor sich, welche die Verteilung in Abbildung 6.8 (linker Graph) aufweist.

Beachten Sie, dass die y-Achse die empirische Dichte der Beobachtungen auf der x-Achse angibt, wir haben hier also ein Maß für die relative Häufigkeit der Beobachtungen. Dies haben wir mit der Funktion `ggplot2::stat(density)` innerhalb von `ggplot2::geom_histogram()` erreicht.

Wenn wir die Daten so betrachten erscheint es naheliegend, sie als Realisierung einer Normalverteilung zu interpretieren: die Form ist grob glockenförmig und symmetrisch. Wir können diese Annahme plausibilisieren indem wir

---

<sup>13</sup>Wir sehen unten aber auch, dass solche Kennzahlen immer mit einer grafischen Darstellung kombiniert werden sollten.

<sup>14</sup>Ein "parametrisches Warhscheinlichkeitsmodell" meint dabei eine ZV mit bestimmten Parametern.

mit `ggplot2::geom_density()` die *empirische Dichtefunktion* der Verteilung schätzen und über die Daten legen, wie im rechten Graph der Abbildung 6.8.

```
Stichprobe <- ggplot2::ggplot(data = sample_data) +
  ggplot2::geom_histogram(
    mapping = aes(x=r, stat(density)),
    binwidth = 0.4) +
  ggplot2::scale_x_continuous(expand = c(0, 1)) +
  ggplot2::scale_y_continuous(expand = expansion(c(0, 0.05), c(0, 0))) +
  ggplot2::theme_bw() +
  theme(panel.border = element_blank(),
        axis.line = element_line())

Dichtefunktion <- ggplot2::ggplot(data = sample_data) +
  ggplot2::geom_histogram(
    mapping = aes(x=r, stat(density)),
    binwidth = 0.4, alpha=0.4) +
  coord_cartesian(xlim = c(-6, 12)) +
  ggplot2::stat_density(mapping = aes(x=r),
                        color="blue",
                        geom="line") +
  ggplot2::scale_y_continuous(expand = expansion(c(0, 0.05),
                                                c(0, 0))) +
  ggplot2::theme_bw() +
  theme(panel.border = element_blank(),
        axis.line = element_line())

ggpubr::ggarrange(Stichprobe, Dichtefunktion, ncol = 2)
```

Das bedeutet, dass wir unsere Daten mit Hilfe der Dichtefunktion (*probability density function* - PDF) der Normalverteilung beschreiben können. Die Formel an sich ist dabei weniger illustrativ, aber sie zeigt was wir mit einem *parametrischen* Wahrscheinlichkeitsmodell meinen:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \quad (6.53)$$

Wenn Sie die Formel genau anschauen finden sich darin zwei Parameter: ein Lageparameter  $\mu$  und ein Streuparameter  $\sigma^2$ . Das bedeutet, dass wir mit diesen beiden Werten die theoretische Normalverteilung vollständig charakterisieren können. Es wäre ja schön, wenn wir unsere Stichprobe oben ebenfalls mit solchen zwei Zahlen vollständig beschreiben könnten.

Das geht allerdings nicht. Unsere empirisch erhobenen Daten sind nie *komplett* identisch zu einer theoretischen Verteilung. Was wir daher machen können ist Folgendes: wir argumentieren, dass unsere Daten sinnvoll durch eine normalverteilte ZV *modelliert* werden können. Wir sagen dann, dass unsere Stichprobe *approximativ normalverteilt* ist. Dann müssen wir im nächsten Schritt nur noch die Werte für die beiden Parameter der Normalverteilung finden, sodass die Verteilung optimal zu unseren Daten passt. Das bedeutet wir ‘fitten’ die Verteilung zu unseren Daten.

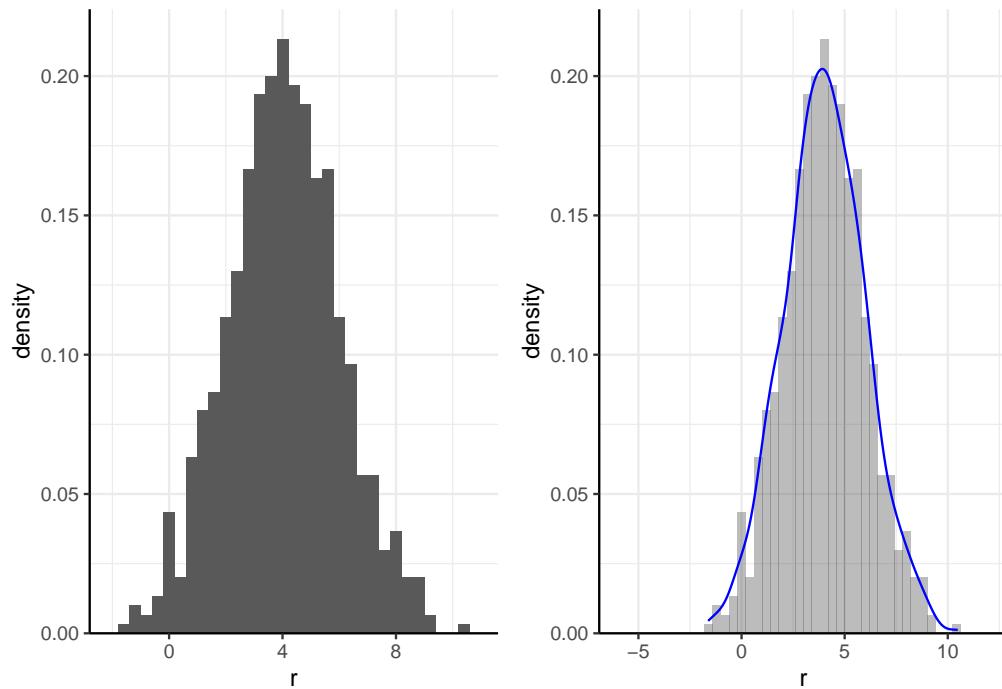


Figure 6.8: Stichprobe (linker Graph) und Stichprobe mit empirischer Dichtefunktion (rechter Graph)

Abbildung 6.9 verdeutlicht die Fits verschiedener Normalverteilungen.

Was damit gemeint ist verdeutlicht die folgende Darstellung:

Die Normalverteilung mit  $\mu = 4$  und  $\sigma^2 = 4$  passt zu den Daten recht gut. Aber wie identifizieren wir diese Werte? In der Praxis müssen diese Werte geschätzt werden. Dazu gibt es verschiedene Verfahren.

Die bekannteste Variante ist die *Maximum Likelihood* Schätzung. Das Verfahren wird später genauer beschrieben, hier illustrieren wir es mit unserem aktuellen Beispiel.

Die Grundidee der *Maximum Likelihood*-Schätzung ist simpel: wählen Sie die Parameter der Verteilung so, dass die beobachtete Stichprobe die am wahrscheinlichsten zu beobachtende Stichprobe ist. In unserem Falle: wählen Sie  $\mu = \mu^*$  und  $\sigma^2 = \sigma^{2*}$  so, dass  $\mathcal{N}(\mu^*, \sigma^{2*})$ , die Normalverteilung ist, bei der die Wahrscheinlichkeit unsere Stichprobe zu bekommen am größten ist.

Bedenken Sie, dass das nichts darüber aussagt *wie* wahrscheinlich das ist: wenn Sie eine unpassende Verteilung mit Maximum Likelihood fitten, bekommen Sie selbst für die besten Parameter einen schlechten Fit.

In unserem Fall wollen wir nun eine Normalverteilung zu unseren Daten fitten. Dazu verwenden wir die Funktion `fitdist()` aus dem Paket `fitdistrplus` (Delignette-Muller and Dutang, 2015). Dieser Funktion geben wir über das Argument `data` unsere Stichprobe und über das Argument `distr` das Kürzel für die Verteilungsklasse, die wir annehmen.<sup>15</sup>

```
fit_dist <- fitdistrplus::fitdist(data = sample_data$r,
                                    distr = "norm")
fit_dist[["estimate"]]

#>      mean        sd
#> 4.023254 1.967130
```

<sup>15</sup> Die bekanntesten Verteilungen werden in Kapitel 7 beschrieben. Die vollständige Liste der Verteilungskürzel in R finden Sie [hier](#).

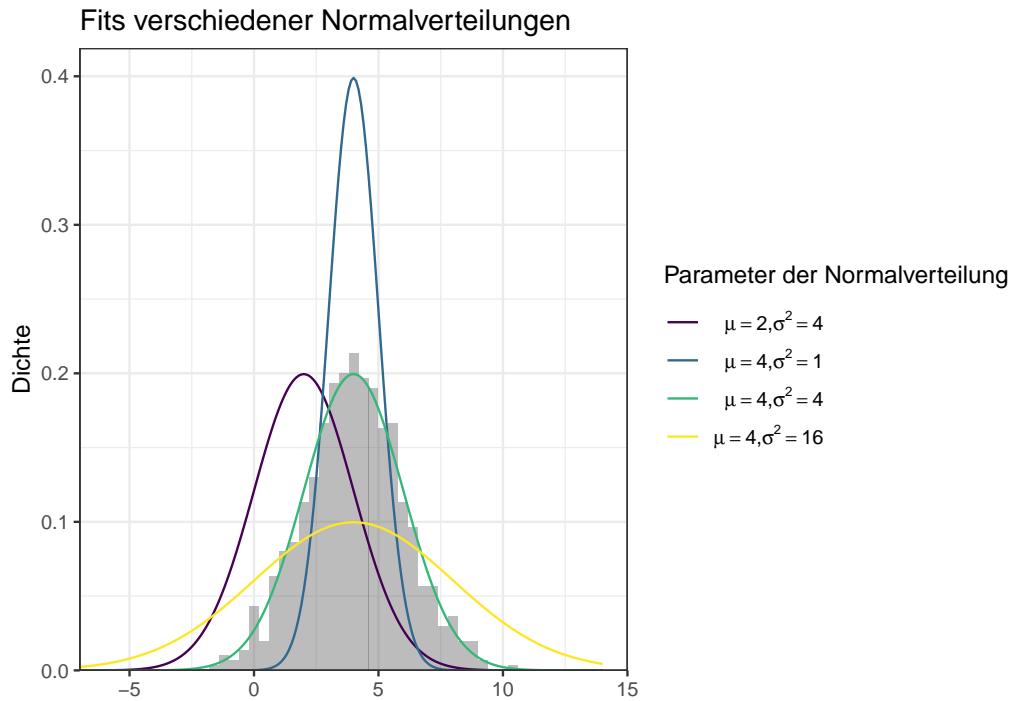


Figure 6.9: Beispiel zur Verdeutlichung von Fits verschiedener Normalverteilungen mit unterschiedlichen Parameterwerten

Wir sehen also, dass die optimale Parametrisierung zu  $\mu = 4.02$  und  $\sigma^2 = 1.967$  korrespondiert. Das passt gut zu unserem grafischen Resultat von oben, bei dem uns  $\mathcal{N}(4, 2)$  bereits als guter Fit ins Auge gesprungen ist.

Allerdings müssten Sie zusätzlich noch testen ob die Verteilungsannahme auch tatsächlich plausibel ist, wir testen also die Hypothese, dass die Daten aus einer  $\mathcal{N}(4, 2)$ -Verteilung gezogen wurden. Für den Fall der Normalverteilung können wir dies z.B. mit einem [Shapiro-Wilk-Test](#) machen.

Hier testen wir die  $H_0$ , dass die Daten tatsächlich durch eine Normalverteilung generiert wurden.<sup>16</sup>

```
shapiro_test <- shapiro.test(sample_data$r)
shapiro_test
```

```
#>
#> Shapiro-Wilk normality test
#>
#> data: sample_data$r
#> W = 0.99894, p-value = 0.9479
```

Da der  $p > 0.1$  können wir die Nullhypothese einer Normalverteilung nicht ablehnen und wir können nun ein gutes Bild unserer Daten vermitteln: wann immer Sie hören, dass ein bestimmter Datensatz approximativ gemäß  $\mathcal{N}(4, 2)$  verteilt ist, dann haben Sie ein sehr gutes Bild des Datensatzes erhalten.

Es gibt viele verschiedene Verteilungstests, je nach dem welche Verteilung Sie testen wollen. Dies ist ein komplexes Thema, das wir in diesem Kapitel nicht weitergehend behandeln. [Clauset et al. \(2009\)](#) ist ein sehr bekanntest Paper, das eine praktische Anleitung für den Fall der Pareto-Verteilung enthält, aber auch für andere Verteilungen

<sup>16</sup>Beachten Sie, dass ein solcher Test weniger gut geeignet ist, wenn Sie entscheiden wollen ob Ihre Daten normalverteilt ‘genug’ sind um bestimmte Methoden anzuwenden, die eine Normalverteilung voraussetzen. Dazu sollten Sie unbedingt auch grafische Methoden wie [QQ-Plots](#) verwenden. Für mehr Details schauen Sie mal in [diesen Blogartikel](#).

verwendet werden kann.<sup>17</sup> Ansonsten finden Sie [hier](#) oder [hier](#) praktische Anleitungen und Diskussionen.

### 6.4.2 Kennzahlen zur Beschreibung empirischer Verteilungen

Jede Beschreibung einer Verteilung mittels Kennzahlen sollte verschiedene Aspekte der Verteilung abdecken. Insbesondere sollten Aussagen zu **Lage**, zur **Streuung**, zur **Form** und zu möglichen Ausreißern und zu sonstigen **Besonderheiten** gemacht werden. Tabelle 6.3 listet die bekanntesten Kennzahlen in den jeweiligen Bereichen auf.

Table 6.3: Kennzahlen zur Beschreibung empirischer Verteilungen.

Kennzahl	Art	R-Funktion
Arithm. Mittel	Lage	<code>mean()</code>
Modus	Lage	<code>NA</code>
Median	Lage	<code>median()</code>
Quantile	Lage	<code>quantile()</code>
Varianz	Streuung	<code>var()</code>
Standardabweichung	Streuung	<code>sd()</code>
Variationskoeffizient	Streuung	<code>sd()/mean()</code>
IQR	Streuung	<code>IQR()</code>
Gini	Streuung	<code>ineq::Gini()</code>
Theil	Streuung	<code>ineq::Theil()</code>
Schiefe	Form	<code>moments::skewness()</code>
Steile	Form	<code>moments::kurtosis()</code>
Cook'sche Distanz	Sonst.	<code>cooks.distance()</code>

Für die folgenden Illustrationen nehmen wir an, dass wir es mit einem Datensatz mit  $N$  kontinuierlichen Beobachtungen  $x_1, x_2, \dots, x_n$  zu tun haben. Als Beispiel dient uns der Datensatz zu ökonomischen Journalen aus [Kleiber and Zeileis \(2008\)](#):<sup>18</sup>

```
#> Kuerzel                               Titel
#> 1: APEL                                Asian-Pacific Economic Literature
#> 2: SAJoEH                               South African Journal of Economic History
#> 3: CE                                    Computational Economics
#> 4: MEPiTE MOCT-MOST Economic Policy in Transitional Economics
#> 5: JoSE                                  Journal of Socio-Economics
#> 6: LabEc                                Labour Economics
#>
#>          Verlag Society Preis Seitenanzahl Buchstaben_pS Zitationen
#> 1:       Blackwell    no   123      440      3822       21
#> 2: So Afr ec history assn   no    20      309      1782       22
#> 3:       Kluwer     no   443      567      2924       22
#> 4:       Kluwer     no   276      520      3234       22
#> 5:    Elsevier    no   295      791      3024       24
#> 6:    Elsevier    no   344      609      2967       24
```

<sup>17</sup>Eine frei zugängliche Version des Papers findet sich [hier](#).

<sup>18</sup>Dieser Datensatz enthält Informationen über Preise, Seiten, Zitationen und Abonennten von 180 Journalen aus der Ökonomik im Jahr 2004. Bei den hier verwendeten Daten handelt es sich um eine Übersetzung des Datensatzes `Journals` aus dem Paket `AER` ([Kleiber and Zeileis, 2008](#)).

```
#>   Gruendung Abonnenten      Bereich
#> 1:    1986          14      General
#> 2:    1986          59 Economic History
#> 3:    1987          17 Specialized
#> 4:    1991           2 Area Studies
#> 5:    1972          96 Interdisciplinary
#> 6:    1994          15      Labor
```

### Kennzahlen zur Lage der Verteilung

Die bekannteste Maßzahl zur Lage einer Verteilung ist das **arithmetische Mittel**. Es ist anwendbar wenn wir es mit kontinuierlichen und mindestens intervall-skalierten Daten zu tun haben und ist definiert als:

$$\bar{x} = \frac{1}{N} \sum_{i=1}^N x_i$$

In R wird das arithmetische Mittel mit der Funktion `mean()` berechnet:

```
avg_preis <- mean(journal_daten[["Preis"]])
avg_preis

#> [1] 417.7222
```

Der durchschnittliche Preis der Journale ist also 417.72 Dollar.

Das arithmetische Mittel ist sehr anfällig gegenüber Ausreißern. Ein robusteres Maß ist der Median: er ist definiert als der Wert  $x_{0.5}$  bei dem 50% der Daten größer und 50% der Daten kleiner sind als  $x_{0.5}$ , genauer:

$$x_{0.5} = \begin{cases} \frac{1}{2} (x_{0.5 \cdot n} + x_{0.5 \cdot n + 1}) & \text{wenn } 0.5 \cdot n \text{ ganzzahlig} \\ \frac{1}{2} x_{\lfloor 0.5 \cdot n + 1 \rfloor} & \text{wenn } 0.5 \cdot n \text{ nicht ganzzahlig} \end{cases} \quad (6.54)$$

wobei wir annehmen, dass die Werte der Verteilung ihrer Größe nach geordnet sind, also  $(x_1 \leq x_2 \leq x_3 \leq \dots \leq x_n)$  und  $\lfloor x \rfloor$  die *Abrundungsfunktion* bezeichnet.<sup>19</sup>

In R wird der Median mit der Funktion `median()` berechnet:

```
med_preis <- median(journal_daten[["Preis"]])
med_preis

#> [1] 282
```

Da es insgesamt 180 Journale gibt gilt, dass 90 Journale teurer und 90 Journale billiger als 282 Dollar sind.

Die Idee des Medians kann über den Begriff der **Quantile** verallgemeinert werden. Wir sprechen bei dem  $\alpha$ -**Quantil** einer Verteilung von dem Wert, bei dem  $\alpha \cdot 100\%$  der Datenwerte kleiner und  $(1 - \alpha) \cdot 100\%$  der Datenwerte größer sind. Genauer:

---

<sup>19</sup>Eine Abrundungsfunktion runden eine Dezimalzahl auf die nächst-kleinere ganze Zahl ab. So ist z.B.  $\lfloor 1.9 \rfloor = 1$  und  $\lfloor 1.2 \rfloor = 1$ .

$$x_\alpha = \begin{cases} \frac{1}{2}(x_{\alpha \cdot n} + x_{\alpha \cdot n + 1}) & \text{wenn } \alpha \cdot n \text{ ganzzahlig} \\ \frac{1}{2}x_{\lfloor \alpha \cdot n + 1 \rfloor} & \text{wenn } \alpha \cdot n \text{ nicht ganzzahlig} \end{cases} \quad (6.55)$$

In R können wir Quantile einfach mit der Funktion `quantile()` berechnen. Diese Funktion akzeptiert als erstes Argument einen Vektor von Daten und als zweites Argument ein oder mehrere Werte für  $\alpha$ :

```
quantile(journal_daten[["Preis"]], c(0.25, 0.5, 0.75))
```

```
#>    25%    50%    75%
#> 134.50 282.00 540.75
```

Wie wir hier sehen ist der Median gleich dem 50%-Quantil.

Eine sehr flexible Kennzahl für die Lage einer Verteilung ist der **Modus**. Er bezeichnet den Wert, der am häufigsten in den Daten vorkommt. Daher ist der Modus auch schon für nominal-skalierte Daten verfügbar.

In R gibt es aber leider keine Funktion, die den Modus direkt berechnet. Vielleicht erinnern Sie sich aber, dass wir mit der Funktion `table()` eine Häufigkeitstabelle ausgeben können. Daher bekommen wir den Modus über folgenden Umweg:<sup>20</sup>

```
names(table(journal_daten[["Preis"]]))
) [table(journal_daten[["Preis"]]) == max(table(journal_daten[["Preis"]]))]
```

```
#> [1] "90"
```

### Kennzahlen zur Streuung einer Verteilung

Von besonderem Interesse in der sozioökonomischen Forschung ist die Analyse von Ungleichheiten. Dies bedeutet, dass Kennzahlen zur Beschreibung der *Streuung* von Verteilungen von besonderer praktischer Bedeutung sind.

Die am weitesten verbreiteten Streuungsmaße sind die **Varianz**  $Var$  und ihre Quadratwurzel, die **Standardabweichung**,  $s$ :

$$s_x = \sqrt{Var(x)} = \sqrt{\frac{1}{N-1} \sum_{i=1}^N (x_i - \bar{x})^2} \quad (6.56)$$

Dabei ist zu beachten, dass die empirische Standardabweichung oft einfacher zu interpretieren ist, da sie in den gleichen Einheiten gemessen wird wie die Daten der Stichprobe. Der **Variationskoeffizient** ist eine einheitslose Variante und ist als Quotient der empirische Standardabweichung und dem arithmetischen Mittel definiert:

$$v_x = \frac{s_x}{\bar{x}} \quad (6.57)$$

In R können die drei Maße folgendermaßen berechnet werden:

---

<sup>20</sup>Es gibt natürlich noch viele andere Möglichkeiten, siehe z.B. [hier](#).

```
var_preis <- var(journal_daten[["Preis"]])
var_preis

#> [1] 148868.3

sd_preis <- sd(journal_daten[["Preis"]])
sd_preis

#> [1] 385.8346

varcoef_preis <- sd(journal_daten[["Preis"]]) / mean(journal_daten[["Preis"]])
varcoef_preis

#> [1] 0.9236631
```

Ein ebenfalls häufig verwendetes Streuungsmaß ist der **Interquartilsabstand** (\*inter-quantile-range, IQR), welcher als die Differenz zwischen dem 25%– und 75%–Quantil definiert ist:

$$IQR = x_{0.75} - x_{0.25}$$

Hierbei handelt es sich also um das Intervall, das die ‘mittlere Hälfte’ der Verteilung umfasst. In R können wir den IQR mit der Funktion `IQR` berechnen:

```
IQR(journal_daten[["Preis"]])
```

```
#> [1] 406.25
```

Ein weit verbreitetes Maß zur Messung der Streuung ist der **Gini-Index**. Dabei handelt es sich um ein relatives Verteilungsmaß, welches auf das Intervall (0, 1) normiert wird und den Wert 0 im Falle einer kompletten Gleichverteilung und den Wert 1 im Falle eine kompletten Konzentration, d.h. dem Fall, dass ein Beobachtungssubjekt alles und alle anderen nichts besitzen.

In R können wir den Gini-Index z.B. mit der Funktion `Gini()` aus dem Paket `ineq` (Zeileis, 2014) berechnen, wobei wir hier die Korrektur für Stichproben verwenden müssen indem wir das Argument `corr = TRUE` setzen:

```
test_data_equality <- rep(0.5, 5)
test_data_inequality <- c(rep(0, 4), 1)
ineq::Gini(test_data_equality, corr = T)

#> [1] 0

ineq::Gini(test_data_inequality, corr = T)

#> [1] 1
```

Um die Besonderheiten des Gini's zu verstehen wollen wir uns genauer mit der Berechnung des Indexes vertraut machen. Der Gini-Index ist eng mit dem Konzept der **Lorenz-Kurve** verknüpft.

Grafisch gesprochen resultiert die Lorenz-Kurve wenn wir auf der x-Achse den Anteil der Beobachtungssubjekte und auf der y-Achse ihren Anteil an den relevanten Ressource abbilden. Definieren wir  $p$  als den Anteil an der Population und  $q = \mathcal{L}(p)$  als den Anteil an der Ressource, der von  $p\%$  der Population gehalten wird. Daraus resultiert, dass wir bei völliger Gleichverteilung eine Gerade sehen würden, da  $p\%$  der Population auch  $q = p = \mathcal{L}(p)\%$  der Ressource halten würden. Die Lorenz-Kurve visualisiert nun die *Abweichung* von diesem idealtypischen Fall in dem  $p = q$ .

Dies wird in Abbildung 6.10 deutlich, in der zwei mögliche Lorenz-Kurven dem hypothetischen Fall der perfekten Gleichverteilung gegenübergestellt werden:

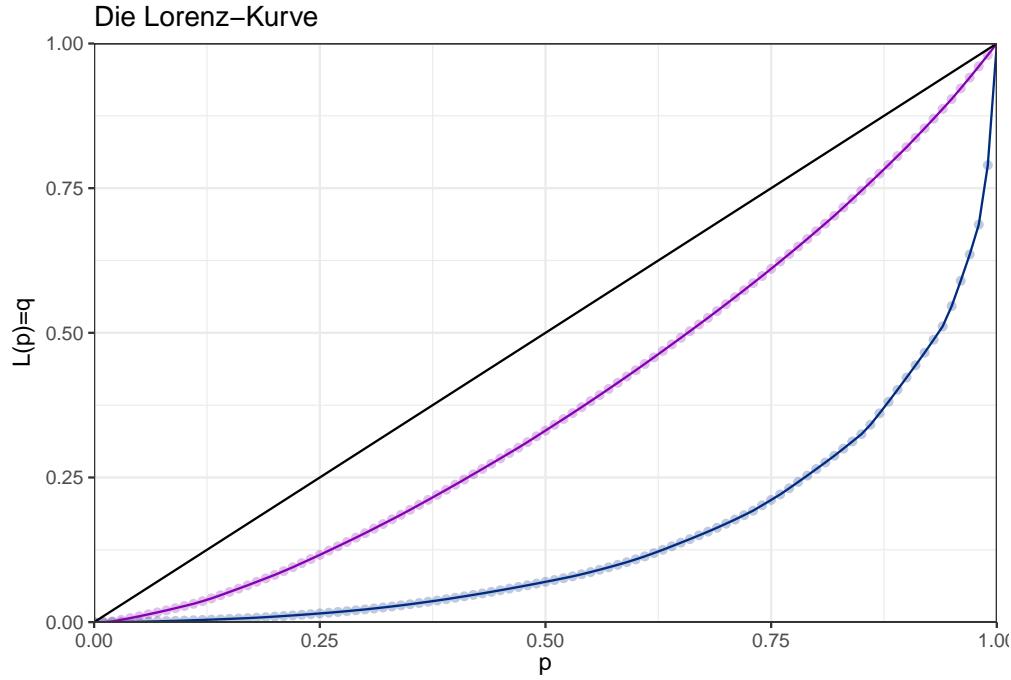


Figure 6.10: Vergleich zweier Lorenz-Kurven

Der Gini-Index  $\mathcal{G}$  misst diese Abweichung über die normierte Distanz zwischen  $p$  und  $q$  indem er einfach das Integral von  $p - \mathcal{L}(p)$  berechnet. Da die Lorenz-Kurve innerhalb eines  $1 \times 1$ -Quadrats definiert ist mutliplizieren wir das Integral mit 2 um die Normierung zwischen 0 und 1 zu erreichen, sonst wäre das Maximum des Gini-Indices 0.5 (da über der 45-Grad Linie per definitionem keine Kurve verlaufen kann):

$$\mathcal{G} = 2 \cdot \int_0^1 (p - \mathcal{L}(p)) dp = 1 - 2 \cdot \int_0^1 (\mathcal{L}(p)) dp \quad (6.58)$$

Der Gini-Index ist ein recht hilfreiches Maß für Ungleichverteilung wenn wir es mit *symmetrischen* Verteilungen zu tun haben, wie die lila Kurve in der Abbildung oben. Es ist jedoch ein schwierigeres Maß sobald eine *assymmetrische* Verteilung vorliegt, wie bei der blauen Kurve oben. In letzterem Fall werden wir möglicherweise die gleichen Ginis für recht unterschiedliche Verteilungen erhalten. Da Vermögens- und Einkommensverteilungen in der Regel immer asymmetrisch sind stellt das durchaus eine Herausforderung für den Gini dar und man sollte andere Ungleichheitsmaße wie den Atkinson-Index oder den Zanardi-Index in Betracht ziehen.

Der Gini-Index reagiert relativ schwach auf Änderungen an den Extremen der Ressourcenverteilung. Wenn diese Änderungen von besonderem Interesse sind bietet sich die Verwendung des [Theil-Index](#) an. Er ist leider nicht so einfach zu interpretieren wie der Gini und eignet sich daher vor allem für Vergleiche über die Zeit.<sup>7</sup> [Der Theil-Index besitzt noch weitere attraktivere Eigenschaften. Insbesondere können die Beiträge von Ungleichheiten innerhalb verschiedener Subgruppen und die Ungleichheiten zwischen Gruppen als solchen aus dem Index abgeleitet werden.

Die Definition ist folgendermaßen:

$$\mathcal{T} = \frac{1}{N} \sum_{i=1}^N \frac{x_i}{\bar{x}} \ln \frac{x_i}{\bar{x}} \quad (6.59)$$

wobei  $N$  die Anzahl der Personen,  $x_i$  die Ressourcenausstattung von Person  $i$  und  $\bar{x}$  das arithmetische Mittel der Ressourcenausstattung ist.

In R können wir den Theil Index mit der Funktion `Theil()` aus dem Paket `ineq` berechnen:

```
dist_expl <- rpareto(100, 3, 2.1)
ineq::Theil(dist_expl)

#> [1] 1.290735
```

Welches Verteilungsmaß für den jeweiligen Anwendungsfall am besten geeignet ist hängt auch von der Art der zugrundeliegenden Verteilung ab. So wird zwar häufig die Varianz als Streuungsmaß verwendet, wenn es sich bei der zu analysierenden Verteilung allerdings um eine bei Einkommen sehr häufig vorkommende Pareto-Verteilung handelt ist die Verwendung dieses Maßes ziemlich irreführend, da die Varianz für diese Verteilungen in vielen Fällen nicht sinnvoll definiert werden kann und wir mit der Formel für die Varianz indirekt unsere Stichprobengröße messen (Yang et al., 2019). Das richtige Maß hängt also immer von unseren theoretischen Vorüberlegungen zur zugrundeliegenden Verteilung und unserem konkreten Erkenntnisinteresse ab.

In diesem Sinne ist vor allem die weite Verbreitung des Gini-Indexes als *dem* Verteilungsmaß schlechthin durchaus kritisch zu sehen. So reagiert der Gini-Index vor allem auf Änderungen in den mittleren Bereichen der Verteilung und weniger auf Änderungen an den Rändern. Wer Effekte von wachsender Vermögenskonzentration bei den reichsten Individuen messen möchte sollte also lieber ein anderes Maß verwenden. Sein Nutzen ist insofern auch von der zugrundeliegenden Forschungsfrage abhängig. Das gilt natürlich auch für alle anderen Indices. So eignet sich der Theil-Index vor allem bei der Analyse von Änderungen über die Zeit in der gleichen Gruppe, da er nicht normiert ist. Er reagiert deutlich besser auf Änderungen an den Extremen als der Gini-Index.

Für eine gute kritische Auseinandersetzung mit dem Gini-Index und einen konstruktiven Gegenvorschlag siehe z.B. Clementi et al. (2019).

Zahlreiche gängige Verteilungsmaße sind in dem Paket `ineq` von Zeileis (2014) implementiert.

### Uni- und Multimodale Verteilungen

Die Unterscheidung zwischen uni- und multimodalen Verteilungen ist wichtig, weil viele Kennzahlen, wie die *Schiefe* oder *Steile* einer Verteilung (siehe unten) nur für unimodale Verteilungen intuitiv interpretiert werden können.

Ganz strikt genommen sprechen wir von einer **unimodalen** oder **eingipfligen** Verteilung wenn Sie nur einen Gipfel hat, also nur einen Modus Ansonsten sprechen wir von einer **multimodalen** oder **mehrgipfligen** (oder genauer *zweigipfligen*, *dreigipfligen*, ...) Verteilung.

In der Praxis haben viele Funktionen einen eindeutigen Modus, besitzen aber mehrere andere lokale Optima, also kleinere "Gipfel", sodass wir in der Regel von einer multimodalen Verteilung sprechen sobald es mehrere lokale Maxima gibt. Beispiele einer solch multimodalen Verteilung sind in Abbildung 6.11 dargestellt.

### Kennzahlen zur Form der Verteilung

Um die Form einer Verteilung besser zu beschreiben verwendet man häufig die **Schiefe** und **Steile** (auch: Kurtosis) einer Verteilung. Beide Kennzahlen sind zunächst einmal nur für *eingipflige/unimodale* Verteilungen sinnvoll.

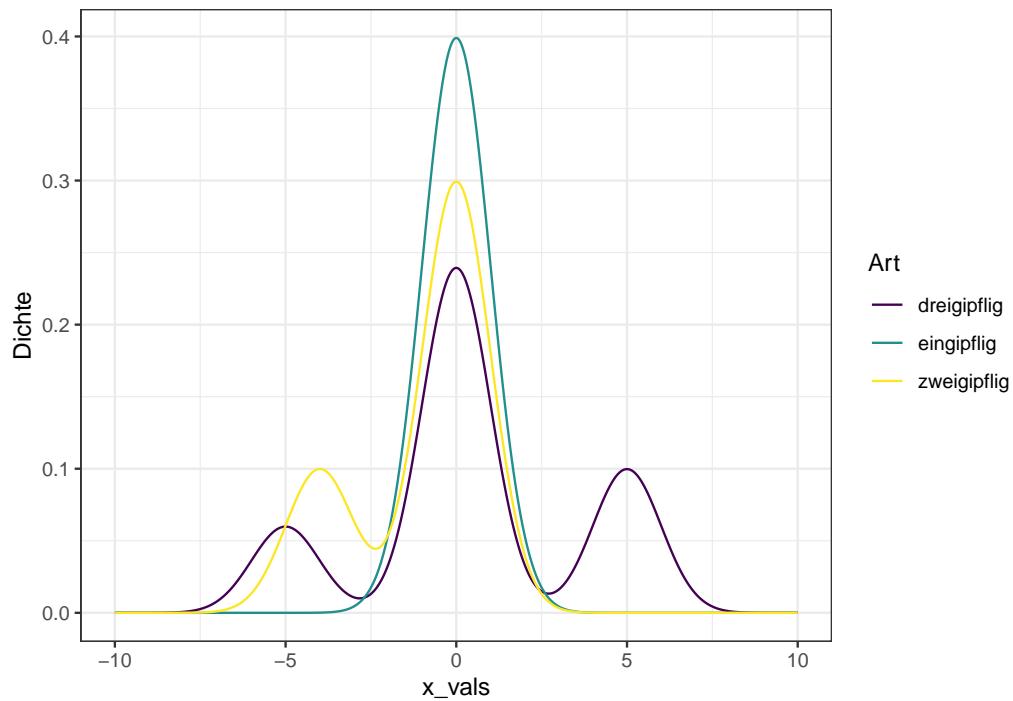


Figure 6.11: Beispiele multimodaler Verteilungen

Die Schiefe einer empirischen Verteilung ist definiert als:

$$\gamma_x = \frac{1}{n} \sum_{i=1}^n \left( \frac{x_i - \bar{x}}{s} \right)^3$$

wobei wir für die Schätzung wieder für die Reduktion der Freiheitsgrade korrigieren müssen, sodass die praktische Schätzfunktion gegeben ist durch:

$$\hat{\gamma}_x = \frac{1}{(n-1)(n-2)} \sum_{i=1}^n \left( \frac{x_i - \bar{x}}{s} \right)^3$$

Hieraus ableiten können wir den Begriff der **Symmetrie** einer Verteilung. Wir nennen eine Verteilung *symmetrisch* wenn  $\gamma_x = 0$ , **links-schief** (oder *rechts-steil*) wenn  $\gamma_x < 0$  und **rechts-schief** (oder *links-steil*) wenn  $\gamma_x > 0$ .

Woher diese Begriffe kommen können wir uns am besten mit Hilfe von Abbildung 6.12 verdeutlichen.

In R können wir die Schiefe einer Verteilung mit der Funktion `skewness()` aus dem Paket `moments` (Komsta and Novomestky, 2015) berechnen:

```
moments::skewness(journal_daten[["Preis"]])
```

```
#> [1] 1.691223
```

Wir würden hier also von einer *rechts-schiefen* Verteilung der Preise sprechen. Das sehen wir hier auch grafisch in Abbildung 6.13.

Die **Steile** (auch: Kurtosis)  $\omega_x$  einer Verteilung gibt ihre ‘Spitzgipfligkeit’ an. Je größer  $\omega_x$  desto ‘schmäler’ wird die Verteilung und desto weniger extreme Werte hat sie. Die Steile ist folgendermaßen definiert:

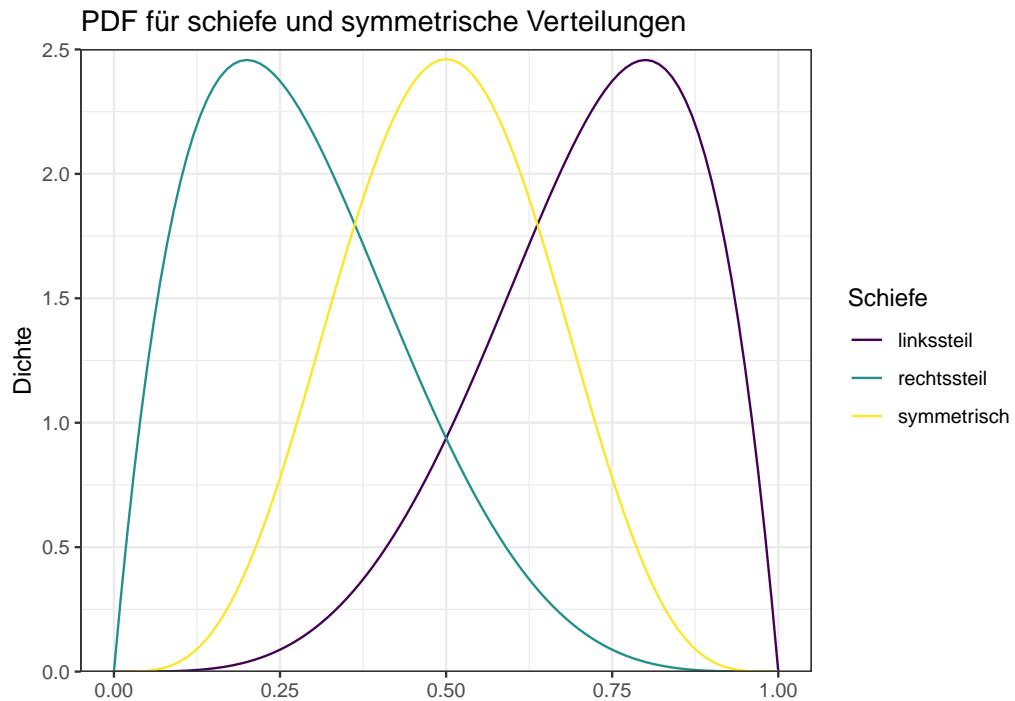


Figure 6.12: Beispiele für Verteilungen mit unterschiedlichen Schiefen

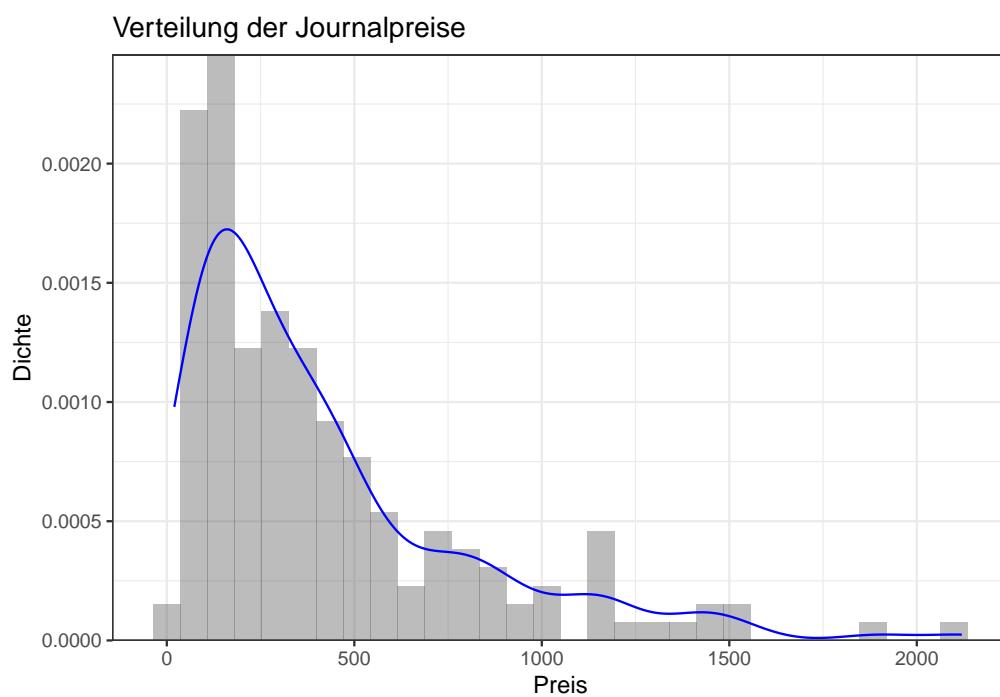


Figure 6.13: Rechts-schiefe Verteilung der Journalpreise

$$\omega_x = \frac{1}{n} \sum_{i=1}^n \left( \frac{x_i - \bar{x}}{s_x} \right)^4$$

Wie bei der Schiefe müssen wir für die Schätzung wieder für die Reduktion der Freiheitsgrade korrigieren, sodass die praktische Schätzfunktion gegeben ist durch:

$$\hat{\omega}_x = \frac{1}{(n-1)(n-2)} \sum_{i=1}^n \left( \frac{x_i - \bar{x}}{s_x} \right)^4$$

Wir können die Kurtosis einer Verteilung mit der Funktion `kurtosis()` aus dem Paket `moments` berechnen:

```
moments::kurtosis(journal_daten[["Preis"]])
```

```
#> [1] 5.992058
```

Da der Wert der Kurtosis  $\omega_x$  an sich nicht leicht zu interpretieren ist wird er häufig mit dem einer Standardnormalverteilung verglichen. Da deren Wert *per definitionem* 3 beträgt wird die *Exzess-Kurtosis* mit  $\tilde{\omega}_x = \omega_x - 3$  berechnet. Wir sprechen von einer *steilgipfligen* ('leptokurtischen') Verteilung wenn  $\tilde{\omega}_x > 0$  und von einer *flachgipfligen* ('platykurtischen') Verteilung wenn  $\tilde{\omega}_x < 0$ . Für den Fall der Preisverteilung von Journalen haben wir es also mit einer steilgipfligen Verteilung zu tun, d.h. die Verteilung der Journalpreise ist 'schmäler' als eine Normalverteilung.

Zur Verdeutlichung des Konzepts bietet Abbildung 6.14 ein grafisches Beispiel.

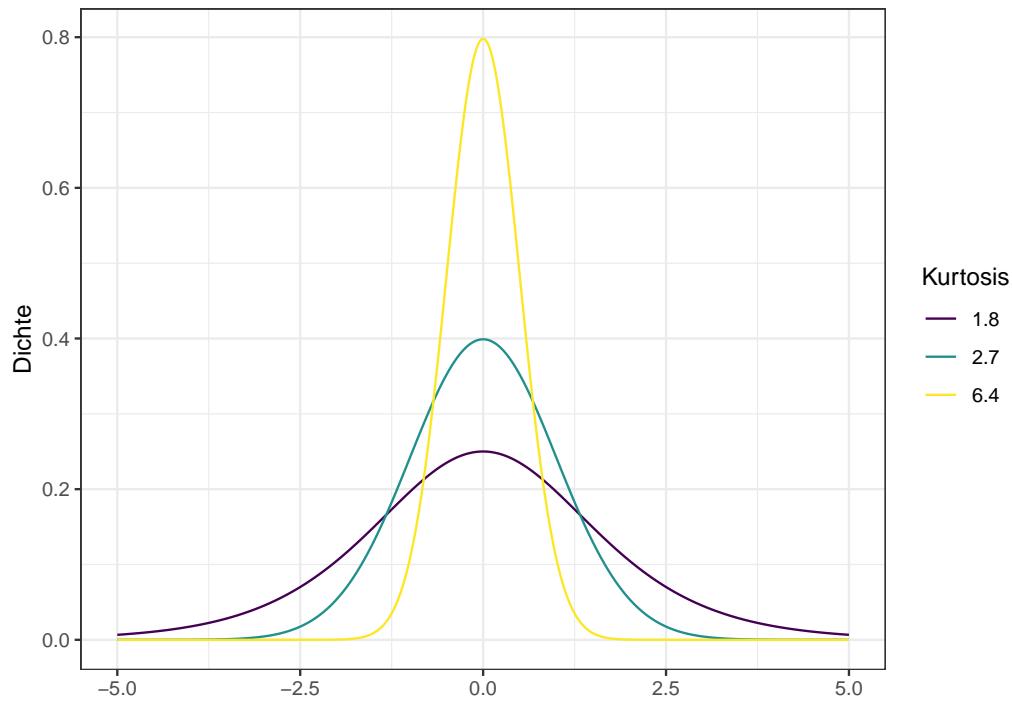


Figure 6.14: Beispiel von Verteilungen mit verschiedener Kurtosis

### Ausreißer und Schwanz-Eigenschaften

Ausreißer können einen großen Effekt auf Ihre Ergebnisse haben. Erinnern Sie sich daran, dass der Mittelwert eines Datensatzes sehr anfällig für Ausreißer, also besonders große oder kleine Werte, ist. Gleichermaßen gilt für viele andere Maße.

Insofern stellen sich zwei wichtige Fragen: Erstens, was genau verstehen wir unter einem Ausreißer? Zweitens, wie sollten wir mit Ausreißern umgehen?

Im Kontext eines Boxplot wurde ein Ausreißer als ein Wert der außerhalb des Intervalls ( $x_{0.25} - IQR \cdot 1.5, x_{0.75} + IQR \cdot 1.5$ ) liegt definiert. Dies führt häufig zu einer zu recht restriktiven Definition von Ausreißern, ist aber ein guter erster Schritt. Wir können die Ausreißer hier einfach identifizieren indem wir den Datensatz entsprechend filtern, z.B.:

```
IQR_Grenzen <- quantile(journal_daten[["Preis"]], c(0.25, 0.75))
untere_grenze <- IQR_Grenzen["25%"] - 1.5*IQR(journal_daten[["Preis"]])
obere_grenze <- IQR_Grenzen["75%"] + 1.5*IQR(journal_daten[["Preis"]])

outlier_teuer <- journal_daten %>%
  dplyr::filter(Preis > obere_grenze)

outlier_billig <- journal_daten %>%
  dplyr::filter(Preis < untere_grenze)

dplyr::select(outlier_teuer, Titel, Preis)
```

```
#>                               Titel Preis
#> 1:                      Ecological Economics 1170
#> 2:                      Applied Economics 2120
#> 3:          Journal of Banking and Finance 1539
#> 4: Journal of Economic Behavior & Organization 1154
#> 5:                      Research Policy 1234
#> 6:                      Economics Letters 1492
#> 7:          European Economic Review 1154
#> 8:          World Development 1450
#> 9:          Journal of Public Economics 1431
#> 10:          Journal of Econometrics 1893
#> 11:          Journal of Economic Theory 1400
#> 12:          Journal of Financial Economics 1339
```

```
dplyr::select(outlier_billig, Titel, Preis)
```

```
#> Empty data.table (0 rows and 2 cols): Titel,Preis
```

Wir sehen hier, dass es nur Ausreißer nach oben, also nur besonders teure Journale gibt. Nun können/müssen wir uns für diese Fälle überlegen wie wir mit den Ausreißern umgehen wollen.

Manche Ausreißer sind die Folge von Messfehlern oder Fehlern in der Datenaufbereitung. Idealerweise würden wir solche Ausreißer aus dem Datensatz entfernen wollen.

Andere Ausreißer sind dagegen einfach besonders interessante Datenpunkte, die *auf gar keinen Fall* aus dem Datensatz entfernt werden sollten. So hat Luxenburg im Vergleich zu anderen Europäischen Ländern ein wahnsinnig hohes Einkommensniveau, aber das bedeutet nicht, dass wir Luxenburg aus allen Analysen herausnehmen sollten. Im Bereich der Finanzmarktanalyse sind extreme Preisausschläge häufig gerade besonders relevant. Sie dürfen auf gar keinen Fall ausgeschlossen werden, denn häufig sind sie Ausgangspunkt von Krisen.

Häufig werden solche Ausreißer eliminiert da die Daten ohne sie leicht durch eine Normalverteilung approximiert werden können. Rechnet man mit diesen Modellen unterschätzt man aber per definitionem die Wahrscheinlichkeit für Extremwerte in der Zukunft. Daher ist die beste Vorgehensweise, sich Ausreißer explizit anzuschauen, indem wir den Datensatz nach extremen Werten (oder Werten mit einer hohen Cook'schen Distanz) filtern und dann selbst entscheiden ob diese Werte eher Resultat eines Messfehlers oder ein besonders interessanter Wert sind. Es gilt jedoch: im Zweifel sollten die Datenpunkte immer im Datensatz gelassen werden. Ein Ausreißer darf nur eliminiert werden wenn es *wirklich sehr gute Gründe* dafür gibt.

Im Falle der Journale ist es fraglich ob es wirklich gute Gründe gibt, diese 12 Journale als Ausreißer zu eliminieren. Im vorliegenden Fall spricht wenig dafür und wir sollten uns eher überlegen wie diese besondere Stellung der Journale erklärt werden kann, z.B. über ihre Popularität.

In diesem Kontext macht es auch Sinn die Kategorie der *endlastigen* oder der *heavy-tailed* Verteilungen einzuführen. Darunter verstehen wir Verteilungen, die besonders viele Extremwerte aufweisen - oder technisch: deren Dichte sub-exponentiell abfällt, deren Extremevents also wahrscheinlicher sind als bei der Exponentialverteilung.

Einkommens- und Vermögensverteilungen sind in der Regel *heavy-tailed*: es gibt zwar sehr viele Menschen mit geringen, und nur wenige mit sehr hohen Einkommen, aber mehr Menschen mit hohen Einkommen als wir es bei einer Exponentialverteilung erwarten würden.

Eine alternative Definition von Ausreißern im Kontext der Regressionsanalyse ist die Berechnung der ‘Cook’schen Distanz’ für jeden Beobachtungswert. Die ‘Cook’sche Distanz’ wird immer im Hinblick auf ein bestimmtes Regressionsmodell berechnet und gibt den Einfluss einer jeden Variable auf das Endergebnis an. Dann kann man sich die einflussreichsten Variablen genauer anschauen und sich fragen wie mit diesen Datenpunkten umzugehen ist.

Die Grundidee der ‘Cook’schen Distanz’ ist für jede Beobachtung das Regressionsergebnis zu vergleichen mit dem hypothetischen Fall, dass diese Beobachtung ausgelassen worden wäre.

Wir können für ein bestimmtes Regressionmodell die Cook’sche Distanz mit der Funktion `cooks.distance()` berechnen. Zum Zwecke der Illustration regressieren wir in dem Journaldatensatz die Variable ‘Preis’ auf die Variablen ‘Seitenanzahl’ und ‘Zitationen’:

```
reg_objekt <- lm(Preis ~ Seitenanzahl + Zitationen,
                  data = journal_daten)
distanzen <- cooks.distance(reg_objekt)
```

Ab wann eine Beobachtung als Ausreißer im Sinne von der Cook’schen Distanz gilt ist nicht klar zu definieren. Als Daumenregel hat sich die Grenze  $\frac{4}{n-k-1}$  etabliert, aber in der Praxis ergibt es immer Sinn einfach die Werte mit der größten Distanz genauer anzuschauen. Diese lassen sich aus Abbildung ??.

```
#> [1] "Managerial and Decision Econ"    "Applied Economics"
#> [3] "Journal of Banking and Finance" "Economics Letters"
#> [5] "World Development"                 "Journal of Public Economics"
#> [7] "Journal of Economic Literature"   "Journal of Econometrics"
#> [9] "Journal of Economic Theory"       "Economic Journal"
#> [11] "Journal of Financial Economics" "Journal of Finance"
#> [13] "Econometrica"
```

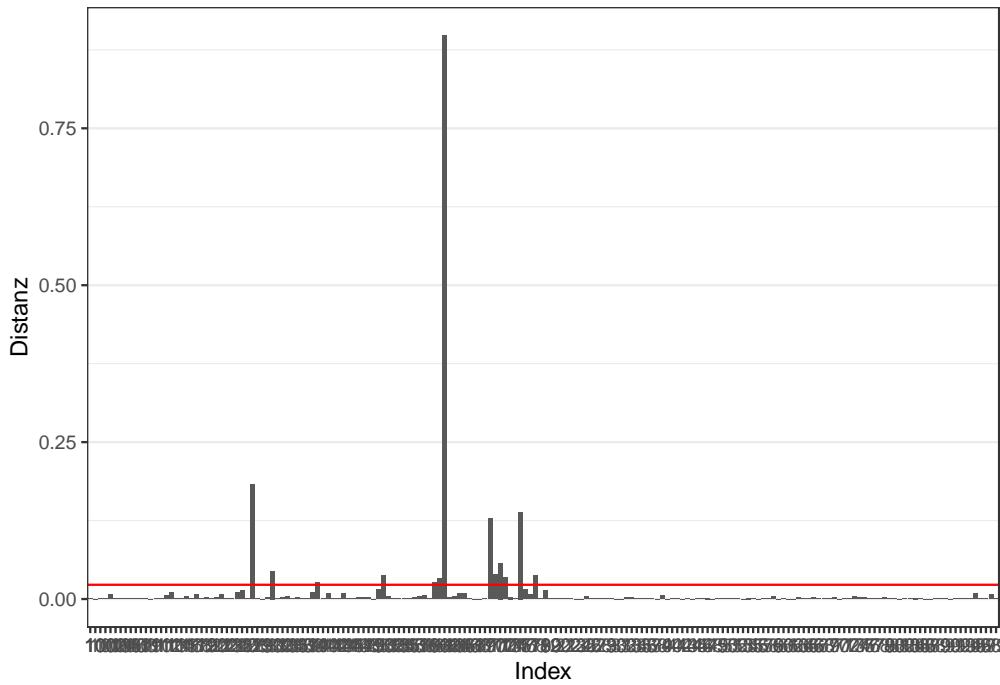


Figure 6.15: Betrachtung der Ausreißer nach der Cook'schen Distanz

### 6.4.3 Grafische Komplemente zu klassischen Kennzahlen

Ein hilfreiches Mittel zur Beschreibung von Verteilungen ist der **Boxplot**. Bei dem Boxplot handelt es sich um eine grafischen Zusammenfassung einiger zentraler deskriptiver Kennzahlen, wie in Abbildung 6.16 zu sehen ist.

```
ggplot2::ggplot(data = wb_data,
  mapping = aes(x=region, y=Lebenserwartung)
) +
  ggplot2::geom_boxplot() +
  ggplot2::theme_bw() +
  ggplot2::labs(title = "Lebenserwartungen in den Weltregionen",
  caption = "Quelle: Weltbank") +
  ggplot2::theme(
  axis.title.x = element_blank(),
  axis.text.x = element_text(angle = 10, vjust = 0.65))
```

Im Boxplot werden mehrere relevante Kennzahlen zusammengefasst. Eine schöne Übersicht bietet Abbildung 6.17:

Die Box in der Mitte des Boxplots repräsentiert die IQR der Daten, der Median ist mit einem Strich innerhalb der Box dargestellt. Die Striche an der Box repräsentieren dann das Intervall bis zum größten bzw. kleinsten Wert, der nicht weiter als  $1.5 \cdot IQR$  vom Median entfernt ist. Ausreißer, hier definiert als Werte außerhalb dieses Intervalls, werden dann durch einzelne Punkte visualisiert. Selbstverständlich können Sie das Aussehen noch weiter an Ihre Präferenzen anpassen. Die Parameter dazu sind in der Hilfefunktion beschrieben. Sehr gute Anleitungen finden sich zudem [hier](#) und [hier](#).

In [diesem Post](#) werden auch die Nachteile dieser Visualisierungsform sehr gut beschrieben. Der größte Nachteil liegt zweifellos im Verstecken der eigentlichen Verteilung 'hinter der Box'. Es ist nicht klar, ob sich ein Großteil der Daten am oberen oder unteren Teil befindet oder ob die Daten eher gleichverteilt sind. Eine einfache Lösung für

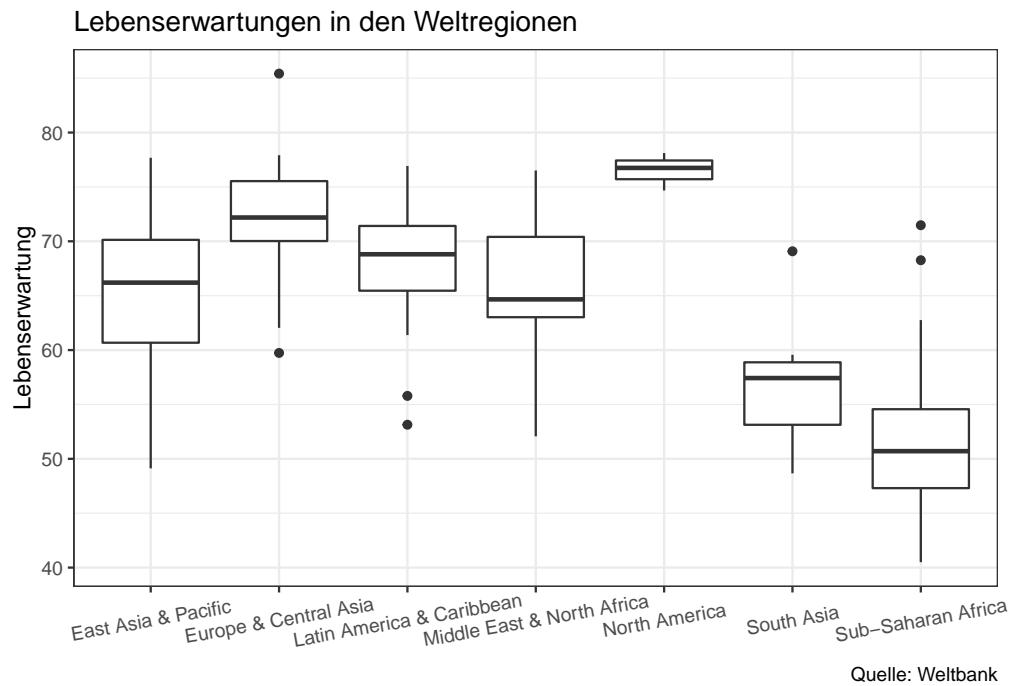


Figure 6.16: Darstellung der Lebenserwartungen in den Weltregionen anhand eines Boxplots

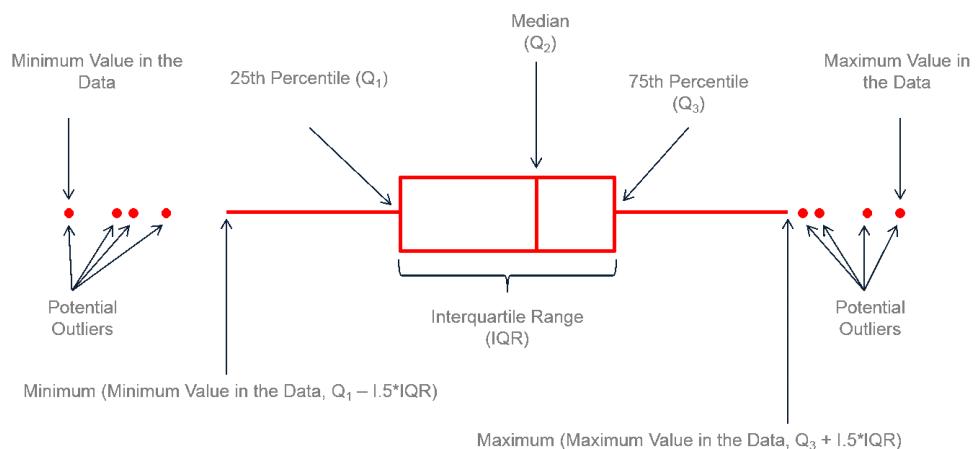


Figure 6.17: Boxplot Anatomie. Quelle: <https://www.leansigmacorporation.com/box-plot-with-minitab/>

kleinere Datensätze liegt in der Ergänzung der einzelnen Beobachtungen durch `boxplot_jitter`, wobei sie hier die Transparenz durch `alpha=0.25` anpassen sollten, und den Boxplot zur besseren Lesbarkeit über die Beobachtungen plotten sollten. Für größere Datensätzen können Sie einfach einen `Violinenplot` verwenden, wie in Abbildung 6.18 gezeigt.

```
boxplot_classic <- ggplot2::ggplot(data = wb_data,
  mapping = aes(x=region, y=Lebenserwartung)
) +
  ggplot2::geom_boxplot() +
  ggplot2::theme_bw() +
  ggplot2::labs(title = "Klassische Darstellung") +
  ggplot2::theme(axis.title.x = element_blank(),
  axis.text.x = element_text(angle = 90, hjust = 1))

boxplot_jitter <- ggplot2::ggplot(data = wb_data,
  mapping = aes(x=region, y=Lebenserwartung)
) +
  ggplot2::geom_boxplot() +
  ggplot2::geom_jitter(alpha=0.25) +
  ggplot2::theme_bw() +
  ggplot2::labs(title = "Klassische Darstellung mit jitter") +
  ggplot2::theme(axis.title.x = element_blank(),
  axis.text.x = element_text(angle = 90, hjust = 1))

violin_plot <- ggplot2::ggplot(data = wb_data,
  mapping = aes(x=region, y=Lebenserwartung)
) +
  ggplot2::geom_violin() +
  ggplot2::theme_bw() +
  ggplot2::labs(title = "Violinen-Plot") +
  ggplot2::theme(axis.title.x = element_blank(),
  axis.text.x = element_text(angle = 90, hjust = 1))

ggpubr::ggarrange(boxplot_classic, boxplot_jitter, violin_plot, ncol = 3)
```

Es ist jedoch immer wichtig eine Verteilung nicht nur mit Kennzahlen, sondern auch grafisch zu beschreiben. Dies wurde erstmals durch Anscombe (1973) durch sein “Anscombe’s Quartett” illustriert. Dabei handelt es sich um vier Datensätze, die alle (fast exakt) gleiche deskriptive Statistiken aufweisen, jedoch offensichtlich sehr unterschiedlich sind. Diese offensichtlichen Unterschiede werden aber nur durch grafische Inspektion deutlich.

Der Datensatz ist in jeder R Installation vorhanden:

```
data("anscombe")
head(anscombe)

#>   x1  x2  x3  x4    y1    y2    y3    y4
#> 1 10  10  10   8 8.04 9.14  7.46  6.58
#> 2  8   8   8   8 6.95 8.14  6.77  5.76
```

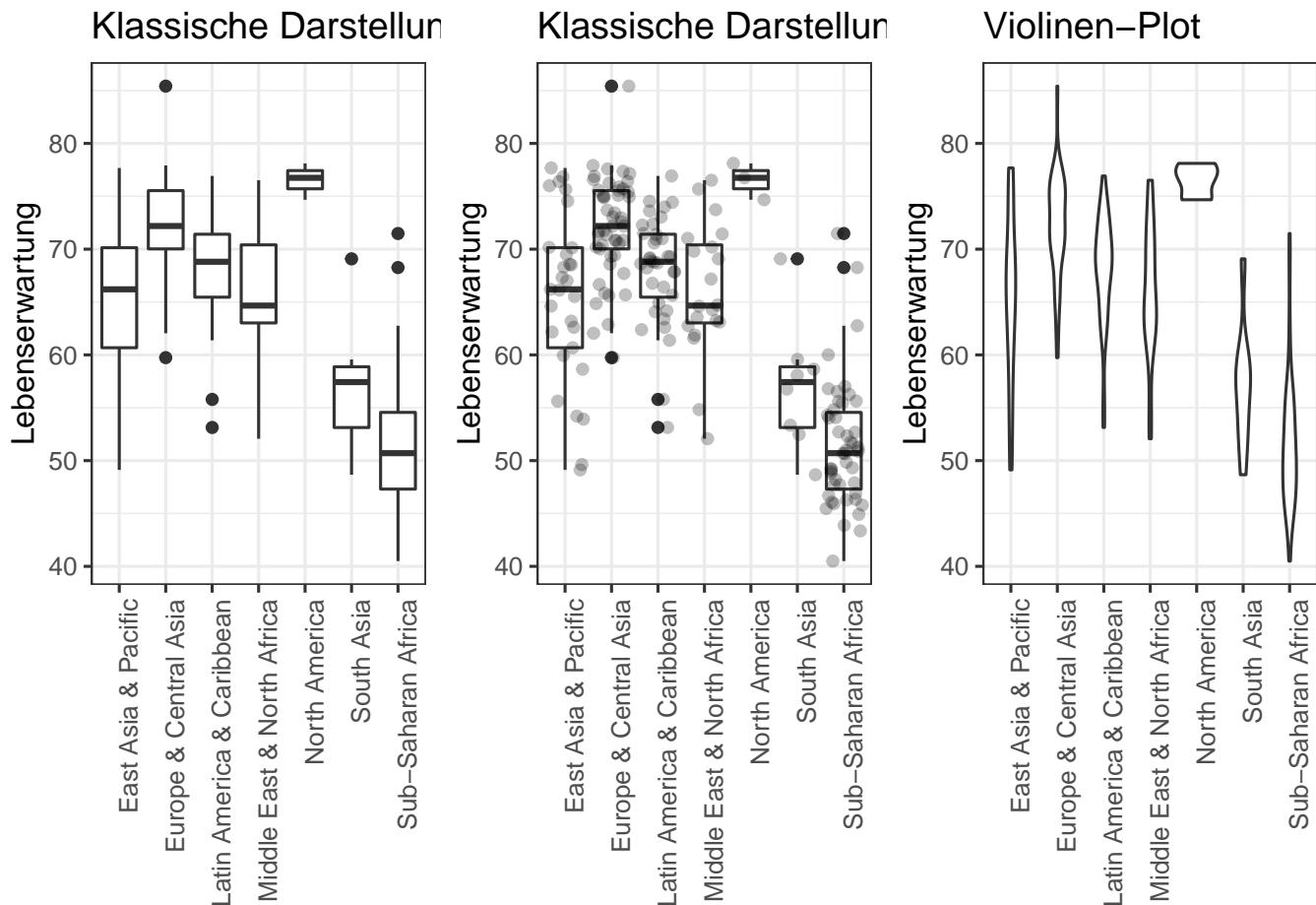


Figure 6.18: Vergleich von klassischen Darstellungen und dem Violinenplot

```
#> 3 13 13 13  8 7.58 8.74 12.74 7.71
#> 4   9   9   9  8 8.81 8.77  7.11 8.84
#> 5 11 11 11  8 8.33 9.26  7.81 8.47
#> 6 14 14 14  8 9.96 8.10  8.84 7.04
```

Tabelle 6.4 gibt die Werte der quantitativen Kennzahlen an.

Table 6.4: Werte der quantitativen Kennzahlen bei Anscombe

Kennzahl	Wert
Mittelwert von $x$	9
Mittelwert von $y$	7.5
Varianz von $x$	11
Varianz von $y$	4.13
Korrelation zw. $x$ und $y$	0.82

Nur die grafische Inspektion zeigt, wie unterschiedlich die Verteilungen tatsächlich sind - siehe Abbildung 6.19.

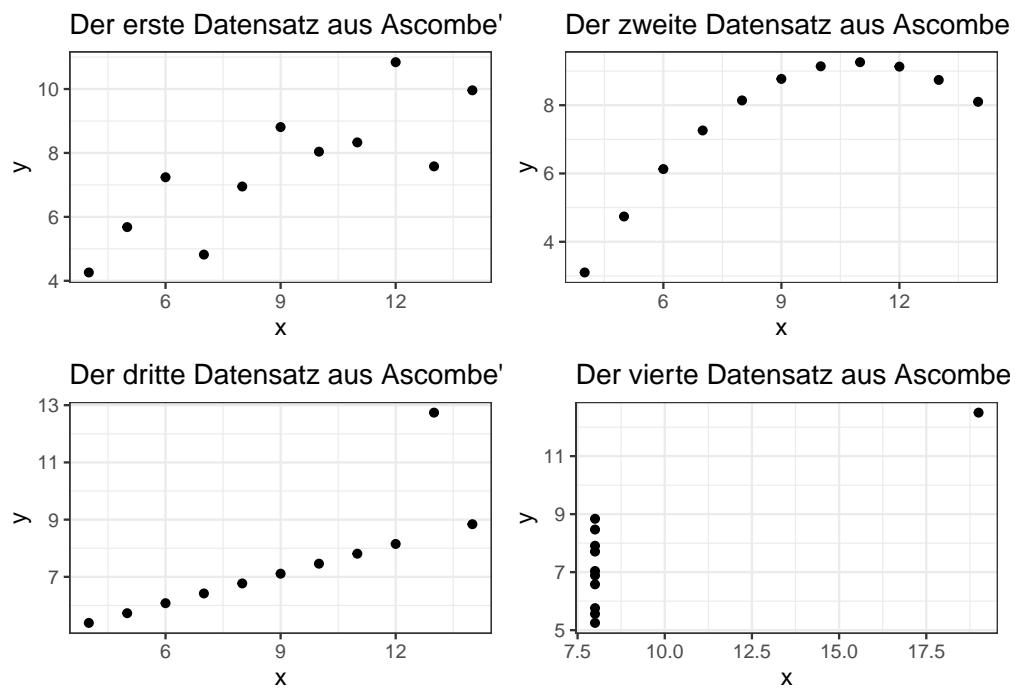


Figure 6.19: Graphischer Vergleich der vier Verteilungen

Damit zeigt sich, dass jede gute Beschreibung einer Verteilung sowohl aus quantitativen als auch grafischen Teilen bestehen sollte.<sup>21</sup>

<sup>21</sup>Interessanterweise ist bis heute nicht bekannt wie Anscombe (1973) seinen Datensatz erstellt hat. Für neuere Sammlungen von Datensätzen, die das gleiche Phänomen illustrieren siehe z.B. Chatterjee and Firat (2007) oder Matejka and Fitzmaurice (2017). Eine sehr schöne Illustration der Idee findet sich auch auf dieser Homepage, die vom Autor von Matejka and Fitzmaurice (2017) gestaltet wurde.

#### 6.4.4 Abschließende Bemerkungen

Es ist wichtig, dass wir uns mit der Verteilung unserer Daten nicht nur theoretisch, sondern auch empirisch und praktisch auseinandersetzen. Für viele Verteilungen sind z.B. bestimmte Kennzahlen nicht definiert. So hat zum Beispiel die bei Vermögens- und Einkommensverteilungen häufig zu beobachtende Pareto-Verteilungen häufig keinen wohldefinierten Mittelwert und keine wohldefinierte Varianz. Daher haben aus Stichproben geschätzte Kennzahlen, die sich dieser Konzepte bedienen, keine wirkliche Aussagekraft. Eine exzellente Beschreibung der Probleme, möglicher Alternativen und ein gutes Anwendungsbeispiel findet sich z.B. in [Yang et al. \(2019\)](#).



# Chapter 7

## Grundlagen der Wahrscheinlichkeitstheorie

In diesem Kapitel werden einige grundlegende Konzepte der Wahrscheinlichkeitstheorie eingeführt, bzw. wiederholt. Die zentralen Themen auf die wir uns fokussieren werden sind dabei:

- Der Zusammenhang zwischen Wahrscheinlichkeitstheorie und Statistik
- Grundbegriffe der Wahrscheinlichkeitstheorie und Statistik
- Zufallsvariablen
- Diskrete und stetige Verteilungen einzelner und mehrerer Zufallsvariablen

Grundkonzepte der deskriptiven und schließenden Statistik (insb. Parameterschätzung, Hypothesentests und die Berechnung von Konfidenzintervallen) werden in den beiden folgenden Kapiteln zur deskriptiven und schließenden Statistik (siehe Kapitel ?? und Kapitel ?? behandelt.

### Verwendete Pakete

```
library(here)
library(tidyverse)
library(ggpubr)
library(latex2exp)
library(data.table)
library(viridis)
```

### 7.1 Einleitung: Wahrscheinlichkeitstheorie und Statistik

Statistik und Wahrscheinlichkeitstheorie sind untrennbar miteinander verbunden. In der Wahrscheinlichkeitstheorie beschäftigt man sich mit Modellen von Zufallsprozessen, also Prozessen, deren Ausgang nicht exakt vorhersehbar ist. Häufig spricht man von *Zufallsexperimenten*.

Die Wahrscheinlichkeitstheorie entwickelt dabei Modelle, welche diese Zufallsexperimente und deren mögliche Ausgänge beschreiben und dabei den möglichen Ausgängen Wahrscheinlichkeiten zuordnen. Diese Modelle werden *Wahrscheinlichkeitsmodelle* genannt.

In der Statistik versuchen wir anhand von beobachteten Daten herauszufinden, welches Wahrscheinlichkeitsmodell gut geeignet ist, den die Daten generierenden Prozess (*data generating process* - DGP) zu beschreiben. Das ist der Grund warum man für Statistik auch immer Kenntnisse der Wahrscheinlichkeitstheorie braucht.

Kurz gesagt: in der Wahrscheinlichkeitstheorie wollen wir mit Hilfe von Wahrscheinlichkeitsmodellen Daten vorhersagen, in der Statistik mit Hilfe bekannter Daten Rückschlüsse auf die zugrundeliegenden Wahrscheinlichkeitsmodelle ziehen.

## 7.2 Grundbegriffe der Wahrscheinlichkeitstheorie

### 7.2.1 Wahrscheinlichkeitstheoretische Modelle

Ein wahrscheinlichkeitstheoretisches Modell besteht *immer* aus den folgenden drei Komponenten:

**Ergebnisraum:** diese Menge  $\Omega$  enthält alle möglichen Ergebnisse des modellierten Zufallsexperiments. Das einzelne Ergebnis bezeichnen wir mit  $\omega$ .

**Beispiel:** Handelt es sich bei dem Zufallsexperiment um das Werfen eines normalen sechseitigen Würfels, so gilt  $\Omega = \{1, 2, 3, 4, 5, 6\}$ . Wenn der Würfel gefallen ist, bezeichnen wir die oben liegende Zahl als das Ergebnis  $\omega$  des Würfelwurfs, wobei hier gilt  $\omega_1 = \text{"Der Würfel zeigt 1", u.s.w.}$

**Ereignisse:** unter Ereignissen  $A, B, C, \dots$  verstehen wir die Teilmengen des Ergebnisraums. Ein Ereignis enthält ein oder mehrere Elemente des Ergebnisraums. Enthält ein Ereignis genau ein Element, sprechen wir von einem *Elementarereignis*.

**Beispiel:** “Es wird eine gerade Zahl gewürfelt” ist ein mögliches Ereignis im oben beschriebenen Zufallsexperiment. Das Ereignis - nennen wir es hier  $A$  - tritt ein, wenn ein Würfelwurf mit dem Ergebnis “2”, “4” oder “6” endet. Also:  $A = \{\omega_2, \omega_4, \omega_6\}$ . Das Ereignis  $B$  “Es wird eine 2 gewürfelt” tritt nur ein, wenn das Ergebnis des Würfelwurfs eine 2 ist:  $B = \{\omega_2\}$ . Entsprechend nennen wir es ein *Elementarereignis*.

Da es sich bei Ereignissen um Mengen handelt können wir die typischen mengentheoretischen Konzepte wie ‘Vereinigung’, ‘Differenz’ oder ‘Komplement’ zu ihrer Beschreibung verwenden. Diese sind in Tabelle 7.1 zusammengefasst:

Table 7.1: Mengentheoretische Konzepte und ihre Übersetzungen.

Konzept	Symbol	Übersetzung
Schnittmenge	$A \cap B$	$A$ und $B$
Vereinigung	$A \cup B$	$A$ und/oder $B$
Komplement	$A^c$	Nicht $A$
Differenz	$A \setminus B = A \cap B^c$	$A$ ohne $B$

**Wahrscheinlichkeiten:** jedem *Ereignis*  $A$  wird eine Wahrscheinlichkeit  $\mathbb{P}(A)$  zugeordnet. Wahrscheinlichkeiten können aber nicht beliebige Zahlen sein. Vielmehr müssen sie im Einklang mit den drei *Axiomen von Kolmogorow* stehen:

1. Für jedes Ereignis  $A$  gilt:  $0 \leq \mathbb{P}(A) \leq 1$
2. Das sichere Ereignis  $\Omega$  umfasst den ganzen Ergebnisraum und es gilt entsprechend  $\mathbb{P}(\Omega) = 1$ .
3. Es gilt:  $\mathbb{P}(A \cup B) = \mathbb{P}(A) + \mathbb{P}(B)$  falls  $A \cap B = \emptyset$ , also wenn sich  $A$  und  $B$  gegenseitig ausschließen.

Aus diesen Axiomen lassen sich eine ganze Menge Sätze heraus ableiten, auf die wir im Folgenden nicht weiter eingehen wollen. Die Grundidee ist aber, bestimmten Ereignissen von Anfang an bestimmte Wahrscheinlichkeiten zuzuordnen, und die Wahrscheinlichkeiten für andere Ereignisse dann aus den eben beschriebenen Regeln abzuleiten.

Je nach Art des Ergebnisraums  $\Omega$  unterscheiden wir zwei grundsätzlich verschiedene Arten von Wahrscheinlichkeitsmodellen: ist  $\Omega$  **abzählbar** handelt es sich um ein **diskretes Wahrscheinlichkeitsmodell**. Der Würfelwurf oder ein Münzwurf sind hierfür Beispiele: die Menge der möglichen Ergebnisse ist hier klar abzählbar.<sup>1</sup>

Ist  $\Omega$  **nicht abzählbar** handelt es sich dagegen um ein **stetiges Wahrscheinlichkeitsmodell**. Ein Beispiel hierfür wäre das Fallenlassen von Steinen und die Messung der Falldauer. Die einzelnen Ereignisse wären dann die Falldauer und da wir die Zeiten in immer kleineren Intervallen angeben können, es also allein zwischen den Messungen “1 Sekunde” und “2 Sekunden” unendlich viele Zwischenschritte gibt, würde gelten, dass  $\Omega = \mathbb{R}^+$ . Dabei handelt es sich um eine nicht abzählbare Menge.

Welches Modell für den konkreten Anwendungsfall vorzuziehen ist, muss auf Basis von theoretischen Überlegungen entschieden werden.

### 7.2.2 Stochastische Unabhängigkeit

Von Interesse ist häufig aus den Wahrscheinlichkeiten für zwei Ereignisse,  $A$  und  $B$ , die Wahrscheinlichkeit für  $A \cap B$ , also die Wahrscheinlichkeit, dass beide Ereignisse auftreten, zu berechnen. Leider ist das nur im Spezialfall der **stochastischen Unabhängigkeit** ohne Probleme möglich. Stochastische Unabhängigkeit kann immer dann sinnvollerweise angenommen werden, wenn zwischen den beteiligten Ereignissen kein kausaler Zusammenhang besteht. In diesem Fall gilt dann:

$$\mathbb{P}(A \cap B) = \mathbb{P}(A) \cdot \mathbb{P}(B)$$

**Beispiel für stochastische Unabhängigkeit:** Es ist plausibel anzunehmen, dass es keinen kausalen Zusammenhang zwischen zwei aufeinanderfolgenden Münzwürfen gibt. Entsprechend sind die Ereignisse  $A$ : “Zahl im ersten Wurf” und  $B$ : “Kopf im zweiten Wurf” stochastisch unabhängig und  $\mathbb{P}(A \cap B) = \mathbb{P}(A) \cdot \mathbb{P}(B) = \frac{1}{4}$ .

**Beispiel für stochastische Abhängigkeit:** Ein anderer Fall liegt vor, wenn wir die Ereignisse  $C$ : “Die Summe beider Würfe ist 6” und  $D$ : “Der erste Wurf zeigt eine 2.” betrachten. Hier ist offensichtlich, dass ein kausaler Zusammenhang zwischen den beiden Würfen und den Ereignissen besteht. Es gilt:  $\mathbb{P}(C \cap D) = \mathbb{P}(\{2, 4\}) = \frac{1}{36}$ . Würden wir die Wahrscheinlichkeiten einfach multiplizieren erhalten wir allerdings  $\mathbb{P}(C) \cdot \mathbb{P}(D) = \frac{5}{36} \cdot \frac{1}{6} = \frac{5}{216}$ , wobei  $\mathbb{P}(C) = \frac{5}{36}$ .

### 7.2.3 Bedingte Wahrscheinlichkeiten

Ein weiteres wichtiges Konzept ist das der **bedingten Wahrscheinlichkeit**: die bedingte Wahrscheinlichkeit von  $A$  gegeben  $B$ ,  $\mathbb{P}(A|B)$ , bezeichnet die Wahrscheinlichkeit für  $A$ , wenn wir wissen, dass  $B$  bereits eingetreten ist.

Es gilt dabei:<sup>2</sup>

$$\mathbb{P}(A|B) = \frac{\mathbb{P}(A \cap B)}{\mathbb{P}(B)}$$

---

<sup>1</sup>Wir nennen eine Menge abzählbar wenn sie mit Hilfe der ganzen Zahlen  $\mathbb{N}$  indiziert werden kann. Das bedeutet, dass auch unendlich große Mengen als abzählbar gelten können.

<sup>2</sup>An der Formel wird noch einmal deutlich, dass wenn  $A$  und  $B$  stochastisch unabhängig sind wir nichts von  $B$  über  $A$  und umgekehrt lernen können, also gilt:  $\mathbb{P}(A|B) = \mathbb{P}(A)$  und  $\mathbb{P}(B|A) = \mathbb{P}(B)$ .

**Beispiel:** Sei  $A$ : “Der Würfel zeigt eine 6” und  $B$ : “Der Würfelwurf zeigt eine gerade Zahl”. Wenn wir bereits wissen, dass  $B$  eingetreten ist, ist  $\mathbb{P}(A)$  nicht mehr  $\frac{1}{6}$ , weil wir ja wissen, dass 1, 3 und 5 nicht auftreten können. Vielmehr gilt  $\mathbb{P}(A|B) = \frac{1/6}{1/2} = \frac{1}{3}$ .

### 7.2.4 Der Satz von Bayes

Wenn wir aus der bedingten Wahrscheinlichkeit für  $A$  gegeben  $B$  die bedingte Wahrscheinlichkeit von  $B$  gegeben  $A$  berechnen wollen, also aus  $\mathbb{P}(A|B)$  den Ausdruck  $\mathbb{P}(B|A)$  ableiten möchten, dann müssen wir den **Satz von Bayes** anwenden - es gilt nämlich leider nicht notwendigerweise  $\mathbb{P}(A|B) = \mathbb{P}(B|A)$ . Vielmehr gilt nach dem *Satz von Bayes*:

$$\mathbb{P}(A|B) = \frac{\mathbb{P}(A \cap B)}{\mathbb{P}(B)} = \frac{\mathbb{P}(B|A)\mathbb{P}(A)}{\mathbb{P}(B)}$$

**Beispiel für die Anwendung von Bayes’ Theorem:** Nehmen wir an, Claudius fährt an 70% der Tage mit dem Fahrrad in die Universität. Immer wenn Claudius mit dem Fahrrad fährt, ist er zu 80% pünktlich. Insgesamt ist er an 60% der Tage pünktlich. Wenn er nun heute pünktlich gekommen ist, wie hoch ist dann die Wahrscheinlichkeit, dass er mit dem Fahrrad gekommen ist? Um diese Frage zu beantworten können wir den Satz von Bayes verwenden. Sei  $A$  : “Claudius kommt mit dem Fahrrad” und  $B$  : “Claudius ist pünktlich”. Dann gilt in jedem Falle  $\mathbb{P}(A) = 0.7$ ,  $\mathbb{P}(B) = 0.6$  sowie  $\mathbb{P}(B|A) = 0.8$ . Wir sind interessiert an  $\mathbb{P}(A|B)$ , also der Wahrscheinlichkeit, dass Claudius mit dem Fahrrad gefahren ist, wenn er pünktlich war. Das können wir nun mit der oben beschriebene Formel herausfinden:

$$\mathbb{P}(A|B) = \frac{\mathbb{P}(B|A)\mathbb{P}(A)}{\mathbb{P}(B)} = \frac{0.8 \cdot 0.7}{0.6} \approx 0.93$$

Die Wahrscheinlichkeit, dass Claudius mit dem Fahrrad gekommen ist beträgt also ca. 93 Prozent!

**Herleitung von Bayes’ Theorem aus den Formeln für bedingte Wahrscheinlichkeiten** Wir können Bayes’ Theorem aus den oben beschriebenen Formeln für bedingte Wahrscheinlichkeiten recht einfach herleiten. Wir wissen, dass

$$\mathbb{P}(A|B) = \frac{\mathbb{P}(A \cap B)}{\mathbb{P}(B)}$$

Für  $\mathbb{P}(A \cap B)$  im Zähler können wir äquivalent  $\frac{\mathbb{P}(A \cap B)}{\mathbb{P}(A)} \cdot \mathbb{P}(A)$  schreiben. Daraus ergibt sich für die Formel oben:

$$\mathbb{P}(A|B) = \frac{\frac{\mathbb{P}(A \cap B)}{\mathbb{P}(A)} \cdot \mathbb{P}(A)}{\mathbb{P}(B)}$$

Gleichzeitig gilt aber auch:

$$\mathbb{P}(B|A) = \frac{\mathbb{P}(A \cap B)}{\mathbb{P}(A)}$$

Das können wir nun im Zähler ersetzen und erhalten so den Satz von Bayes:

$$\mathbb{P}(A|B) = \frac{\mathbb{P}(B|A) \cdot \mathbb{P}(A)}{\mathbb{P}(B)}$$

### 7.2.5 Das Gesetz der totalen Wahrscheinlichkeiten

Wenn wir die Wahrscheinlichkeiten für mehrstufige Zufallsexperimente berechnen wollen müssen wir oft Wahrscheinlichkeiten von verschiedenen Ebenen „aggregieren“. Das machen wir mit dem **Gesetz der totalen Wahrscheinlichkeit**, das in Beweisen im Bereich der Stochastik sehr häufig verwendet wird. Formal besagt das *Gesetz der totalen Wahrscheinlichkeit* folgendes: seien  $A_1, \dots, A_k$  Ereignisse, die sich nicht überschneiden und gemeinsam den kompletten Ereignisraum  $\Omega$  abdecken, dann gilt:

$$\mathbb{P}(B) = \sum_{i=1}^k \mathbb{P}(B|A_i)\mathbb{P}(A_i)$$

Das sieht natürlich erst einmal sperrig aus, wie so oft ist es aber eigentlich ganz einfach. Das folgende Beispiel illustriert das.

**Beispiel für die Anwendung vom Gesetz der totalen Wahrscheinlichkeit:** Wir haben eine Urne mit drei weißen und sieben schwarzen Kugeln. Wir ziehen zweimal eine Kugel ohne sie dabei zurückzulegen - wir haben es also mit einem zweistufigen Zufallsexperiment zu tun. Wie hoch ist nun die Wahrscheinlichkeit, dass wir genau eine schwarze Kugel gezogen haben? Sei  $B$ : „Eine schwarze Kugel wird gezogen“ und  $A$ : „Eine weiße Kugel wird gezogen“. Wir addieren nun die Wahrscheinlichkeiten für aller Ergebnisse, die in Kombination zu unserem Gesamtereignis führen, also die Wahrscheinlichkeit erst eine weiße und dann eine schwarze und die Wahrscheinlichkeit erst eine schwarze und dann eine weiße Kugel zu ziehen:

$$\mathbb{P}(B = 1) = \frac{7}{9} \cdot \frac{3}{10} + \frac{3}{9} \cdot \frac{7}{10} = \frac{42}{90}$$

wobei  $\mathbb{P}(B|A) = \frac{7}{9} \cdot \frac{3}{10}$  und  $\mathbb{P}(A|B) = \frac{3}{9} \cdot \frac{7}{10}$ . Die Wahrscheinlichkeit genau eine schwarze Kugel zu ziehen liegt also bei ca. 45.5 Prozent.

## 7.3 Diskrete Wahrscheinlichkeitsmodelle

Wenn wir die Wahrscheinlichkeit für das Eintreten eines Ereignisses  $A$  erfahren möchten können wir im Falle eines diskreten Ergebnisraums einfach die Eintrittswahrscheinlichkeiten für alle Ergebnisse, die zu  $A$  gehören, aufsummieren:

$$\mathbb{P}(A) = \sum_{\omega \in A} \mathbb{P}(\{\omega\})$$

**Beispiel:** Beim Werfen eines sechseitigen Würfels ist die Wahrscheinlichkeit für das Ereignis „Es wird eine gerade Zahl gewürfelt“:  $\mathbb{P}(2) + \mathbb{P}(4) + \mathbb{P}(6) = \frac{1}{6} + \frac{1}{6} + \frac{1}{6} = \frac{1}{2}$ .

### 7.3.1 Diskrete Zufallsvariablen

Bei Zufallsvariablen (ZV) handelt es sich um besondere *Funktionen*. Die Definitionsmenge einer Zufallsvariable ist immer der zugehörige Ergebnisraum  $\Omega$ , die Zielmenge ist i.d.R.  $\mathbb{R}$ , sodass gilt:

$$X : \Omega \rightarrow \mathbb{R}, \omega \mapsto X(\omega)$$

Im Kontext von ZV sprechen wir häufig nicht von dem zugrundeliegenden Ergebnisraum  $\Omega$ , sondern - inhaltlich äquivalent - vom *Wertebereich von  $X$* , bezeichnet als  $W_X$ . Produkte und Summen von ZV sind selbst wieder Zufallsvariablen. Man addiert bzw. multipliziert ZV indem man ihre Werte addiert bzw. multipliziert.

In der Regel bezeichnen wir Zufallsvariablen mit Großbuchstaben und die konkrete Realisation einer ZV mit einem Kleinbuchstaben, sodass  $\mathbb{P}(X = x)$  die Wahrscheinlichkeit angibt, dass die ZV  $X$  den konkreten Wert  $x$  annimmt. Bei  $x$  sprechen wir von einer *Realisierung* der ZV  $X$ . Wir nehmen für die weitere Notation an, dass  $W_X = \{x_1, x_2, \dots, x_K\}$  und bezeichnen das einzelne Element mit  $x_k$  mit  $1 \leq k \leq K$ .

Dies bedeutet streng genommen, dass die ZV selbst nicht als zufällig definiert wird. Zufällig ist nur der Input  $\omega$  der entsprechenden Funktion  $X : \Omega \rightarrow X(\omega)$ , also z.B. ein Würfelwurf. Der funktionale Zusammenhang zwischen Funktionswert  $X(\omega)$  und dem Input  $\omega$  ist hingegen eindeutig und deterministisch.

Das impliziert, dass wenn ein Zufallsexperiment zweimal das gleiche Ergebnis  $\omega$  hat, auch der Wert  $X(\omega)$  der gleiche ist.

Das mag im Moment ein wenig nach ‘Pfennigfuchserei’ aussehen, die Unterscheidung zwischen dem nicht-zufälligem funktionalen Zusammenhang, aber einem zufälligen Input bei ZV ist wichtig, um den Sinn in vielen fortgeschrittenen Beiträgen im Bereich der Ökonometrie zu sehen.

Im Falle von diskreten ZV können wir eine Liste erstellen, die für alle möglichen Werte  $x_k \in W_X$  die jeweilige Wahrscheinlichkeit  $\mathbb{P}(X = x_k)$  angibt.<sup>3</sup> Diese Liste nennen wir **Wahrscheinlichkeitsverteilung** von  $X$  und sie wird häufig visuell dargestellt. Um diese Liste zu erstellen verwenden wir die zu  $X$  gehörende **Wahrscheinlichkeitsfunktion** (*Probability Mass Function*, PMF),  $p(x_k)$ , die uns für jedes Ergebnis die zugehörige Wahrscheinlichkeit gibt.<sup>4</sup>

$$p(x_k) = \mathbb{P}(X = x_k)$$

Ebenfalls häufig verwendet wird die **kumulierte Wahrscheinlichkeitsfunktion** (*cumulative distribution function*, CDF):

$$F_X(a) = \mathbb{P}(X \leq x_k)$$

Die CDF einer diskrete ZV  $X$  gibt also die Wahrscheinlichkeit an, dass  $X$  sich als ein Wert kleiner gleich einem Schwellenwert realisiert. Daher auch der Name: sie kumuliert die Eintrittswahrscheinlichkeiten aller Events zwischen  $-\infty$  und  $a$ . Für eine solche Funktion gilt wie für die PMF, dass  $0 \leq F_X(a) \leq 1$ . Zudem handelt es sich bei der CDF um eine wachsende Funktion ( $F_X(a) \leq F(b) \leftrightarrow a \leq b$ ) und es gilt, dass  $\lim_{a \rightarrow \infty} F_X(a) = 1$  sowie  $\lim_{a \rightarrow -\infty} F_X(a) = 0$ , d.h. für sehr große Werte von  $a$  geht  $F_X$  gegen 1 und für sehr kleine Werte gegen 0.

Wenn wir eine ZV analysieren tun wir dies in der Regel durch eine Analyse ihrer Wahrscheinlichkeitsverteilung. Zur genaueren Beschreibung einer ZV wird entsprechend häufig einfach die Wahrscheinlichkeitsfunktion angegeben.

Im Folgenden wollen wir einige häufig auftretende Wahrscheinlichkeitsverteilungen kurz einführen. Am Ende des Abschnitts findet sich dann ein tabellarischer Überblick. Doch vorher wollen wir uns noch mit den wichtigsten **Kennzahlen einer Verteilung** vertraut machen. Denn wie Sie sich vorstellen können sind Wahrscheinlichkeitsverteilungen als Listen, die alle möglichen Realisierungen einer ZV enthalten ziemlich umständlich zu handhaben. Da-

<sup>3</sup>Aus den *Kolmogorow Axiomen* oben ergibt sich, dass die Summe all dieser Wahrscheinlichkeiten 1 ergeben muss:  $\sum_{k \geq 1} \mathbb{P}(X = x_k) = 1$ .

<sup>4</sup>Zu jeder Wahrscheinlichkeitsverteilung gibt es eine eindeutige Wahrscheinlichkeitsfunktion und jede Wahrscheinlichkeitsfunktion definiert umgekehrt eine eindeutig bestimmte diskrete Wahrscheinlichkeitsverteilung.

her beschreiben wir Wahrscheinlichkeitsverteilungen nicht indem wir eine Liste beschreiben, sondern indem wir bestimmte Kennzahlen zu ihrer Beschreibung verwenden. Die wichtigsten Kennzahlen einer ZV  $X$  sind der **Erwartungswert**  $\mathbb{E}(x)$  als *Lageparameter* und die **Standardabweichung**  $\sigma(X)$  als *Streuungsmaß*.

Der Erwartungswert ist definitiert als die nach ihrer Wahrscheinlichkeit gewichtete Summe aller Elemente im Wertebereich von  $X$  und gibt damit die mittlere Lage der Wahrscheinlichkeitsverteilung an. Wenn  $W_X$  der Wertebereich von  $X$  ist, dann gilt:

$$\mathbb{E}(x) = \mu_X = \sum_{x_k \in W_X} p(x_k)x_k$$

**Beispiel:** Der Erwartungswert einer ZV  $X$ , die das Werfen eines fairen Würfels beschreibt ist:  $\mathbb{E}(X) = \sum_{k=1}^6 k \cdot \frac{1}{6} = 3.5$ .

Wie wir im Kapitel ?? sehen werden, wird der Erwartungswert in der empirischen Praxis häufig über den Mittelwert einer Stichprobe identifiziert.

Ein gängiges Maß für die Streuung einer Verteilung  $X$  ist die Varianz  $Var(X)$  oder ihre Quadratwurzel, die Standardabweichung,  $\sigma(X) = \sqrt{Var(X)}$ . Letztere wird häufiger verwendet, weil sie die gleiche Einheit hat wie  $X$ :

$$Var(X) = \sum_{x_k \in W_X} [x_k - \mathbb{E}(X)]^2 p(x_k)$$

**Beispiel:** Die Standardabweichung einer ZV  $X$ , die das Werfen eines fairen Würfels beschreibt ist:  
 $\sigma_X = \sqrt{\sum_k [x_k - \mathbb{E}(X)]^2 p(x_k)} = \sqrt{5.83} \approx 2.414$ .

Im Folgenden wollen wir uns einige der am häufigsten verwendeten ZV und ihre Verteilungen genauer ansehen. Am Ende der Beschreibung jeder Funktion folgt ein Beispiel für eine Anwendung. Wenn Ihnen die theoretischen Ausführungen am Anfang etwas kryptisch erscheinen, empfiehlt es sich vielleicht erst einmal das Anwendungsbeispiel anzusehen.

### 7.3.2 Beispiel: die Binomial-Verteilung

Die vielleicht bekannteste diskrete Wahrscheinlichkeitsverteilung ist die Binomialverteilung  $\mathcal{B}(n, p)$ . Mit ihr modelliert man Zufallsexperimente, die aus einer Reihe von Aktionen bestehen, die entweder zum ‘Erfolg’ oder ‘Misserfolg’ führen.

Die Binomialverteilung ist eine Verteilung mit zwei **Parametern**. Parameter sind Werte, welche die Struktur der Verteilung bestimmen. In der Statistik sind wir häufig daran interessiert, die Parameter einer Verteilung zu bestimmen. Im Falle der Binomialverteilung gibt es die folgenden zwei Parameter:  $p$  gibt die Erfolgswahrscheinlichkeit einer einzelnen Aktion an (und es muss daher gelten  $p \in [0, 1]$ ) und  $n$  gibt die Anzahl der Aktionen an. Daher auch die Kurzschreibweise  $\mathcal{B}(n, p)$ .

**Beispiel:** Wenn wir eine faire Münze zehn Mal werfen, können wir das mit einer Binomialverteilung mit  $p = 0.5$  und  $n = 10$  modellieren.

Die *Wahrscheinlichkeitsfunktion*  $p(x)$  der Binomialverteilung ist die Folgende, wobei  $x$  die Anzahl der Erfolge darstellt:

$$\mathbb{P}(X = x) = p(x) = \binom{n}{x} p^x (1-p)^{n-x}$$

Dies ergibt sich aus den grundlegenden Wahrscheinlichkeitsgesetzen:  $\binom{n}{x}$  ist der **Binomialkoeffizient** und gibt uns die Anzahl der Möglichkeiten wie man bei  $n$  Versuchen  $x$  Erfolge erzielen kann. Dies multiplizieren wir mit der Wahrscheinlichkeit  $x$ -mal einen Erfolg zu erzielen und  $n - x$ -mal einen Misserfolg zu erzielen.

Wenn die ZV  $X$  einer Binomialverteilung mit bestimmten Parametern  $p$  und  $n$  folgt, dann schreiben wir  $P \propto \mathcal{B}(n, p)$  und es gilt, dass  $\mathbb{E}(X) = np$  und  $\sigma(X) = \sqrt{np(1-p)}$ .<sup>5</sup>

In Abbildung 7.1 sehen wir eine Darstellung der Wahrscheinlichkeitsverteilung der Binomialverteilung für verschiedene Parameterwerte.

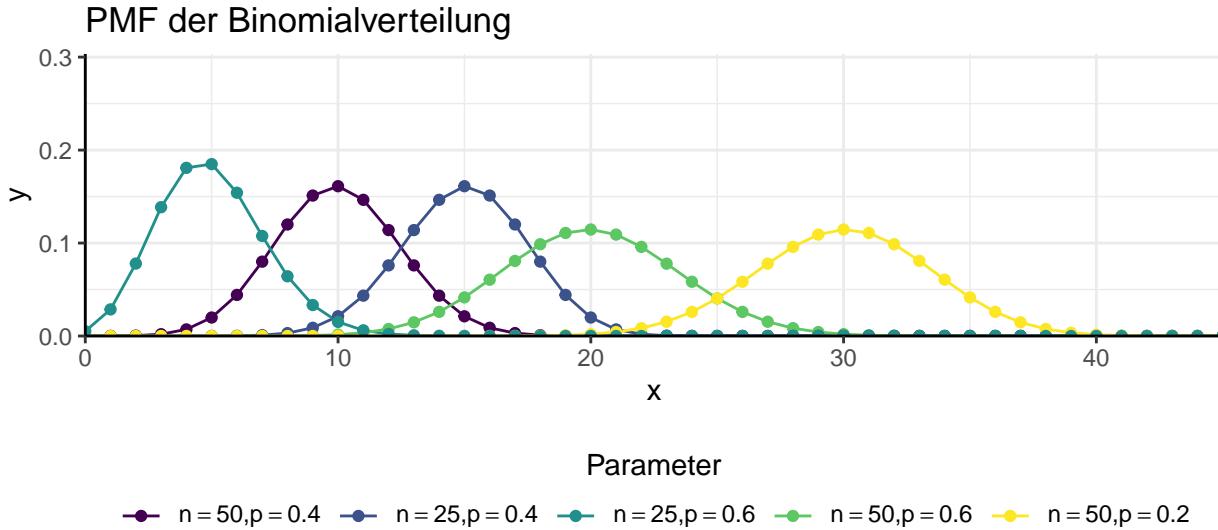


Figure 7.1: Wahrscheinlichkeitsverteilung der Binomialverteilung für verschiedene Parameterwerte.

R stellt uns einige nützliche Funktionen bereit, mit denen wir typische Rechenaufgaben einfach lösen können:

Möchten wir die Wahrscheinlichkeit berechnen, genau  $x$  Erfolge zu beobachten, also  $\mathbb{P}(X = x)$  geht das mit der Funktion `dbinom()`. Die notwendigen Argumente sind `x` für den interessierenden  $x$ -Wert, `size` für den Parameter  $n$  und `prob` für den Parameter  $p$ :

```
dbinom(x = 10, size = 50, prob = 0.25)
```

```
## [1] 0.09851841
```

Das bedeutet, wenn  $X \sim B(50, 0.25)$ , dann:  $\mathbb{P}(X = 10) = 0.09852$ . Dies ist in Abbildung 7.2 illustriert.

Natürlich können wir an die Funktion auch einen atomaren Vektor als erstes Argument übergeben:

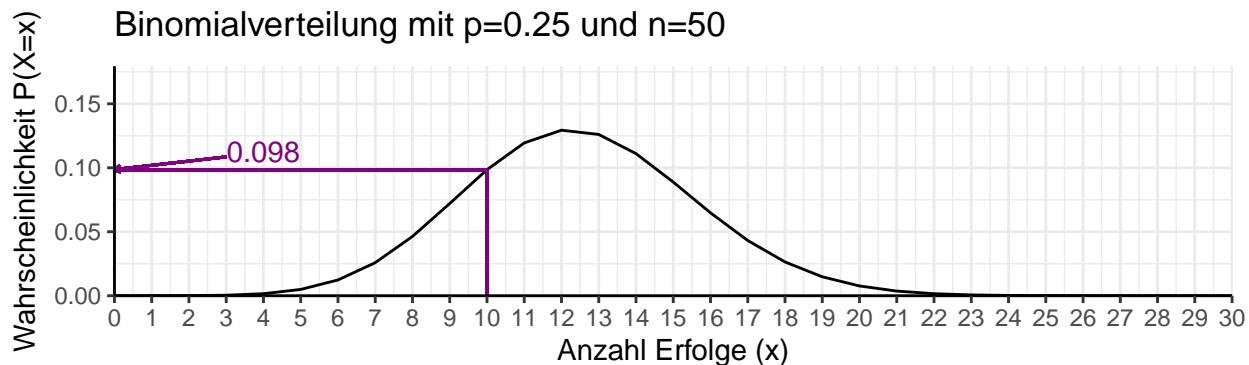
```
dbinom(x = 5:10, size = 50, prob = 0.25)
```

```
## [1] 0.004937859 0.012344647 0.025864974 0.046341412 0.072086641 0.098518410
```

Häufig sind wir auch an der **kumulierten Wahrscheinlichkeitsfunktion** interessiert. Während uns die Wahrscheinlichkeitsfunktion die Wahrscheinlichkeit für genau  $x$  Erfolge angibt, also  $\mathbb{P}(X = x)$ , gibt uns die *kumulierte* Wahrscheinlichkeitsfunktion die Wahrscheinlichkeit für  $x$  oder weniger Erfolge, also  $\mathbb{P}(X \leq x)$ .

Die entsprechenden Werte für die kumulierten Wahrscheinlichkeitsfunktion erhalten wir mit der Funktion `pbinom()`, welche quasi die gleichen Argumente benötigt wie `dbinom()`. Nur gibt es anstatt des Parameters `x` jetzt einen Parameter `q`:

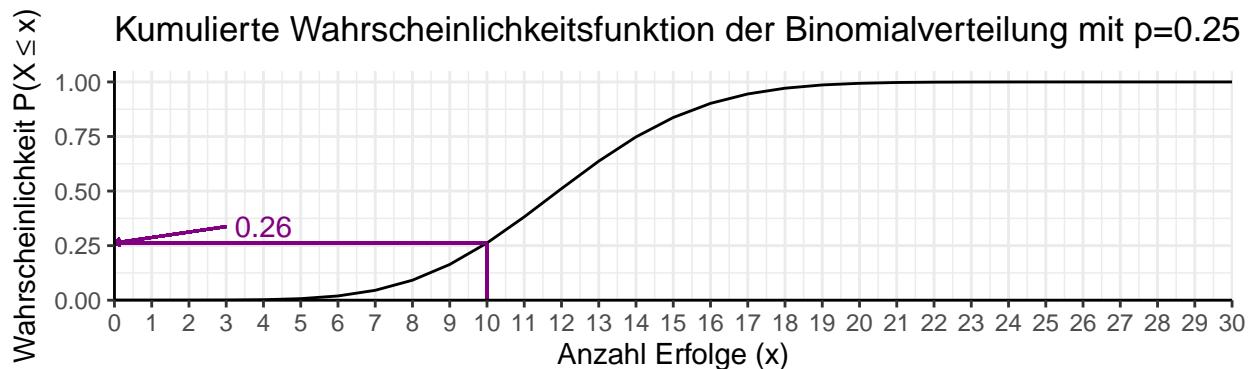
<sup>5</sup>Die Herleitung finden Sie im Statistikbuch Ihres Vertrauens oder auf [Wikipedia](#).

Figure 7.2: Beispiel einer Binomialverteilung mit  $p=0.25$  und  $n=50$ .

```
pbinom(q = 10, size = 50, prob = 0.25)
```

```
## [1] 0.2622023
```

Die Wahrscheinlichkeit 10 oder weniger Erfolge bei 10 Versuchen und einer Erfolgswahrscheinlichkeit von 25% zu erzielen beträgt also 26.2%. Dies ist auch in Abbildung 7.3 ersichtlich.

Figure 7.3: Kumulative Wahrscheinlichkeitsfunktion mit  $p=0.25$  und  $n=50$ 

Schlussendlich haben wir die Funktion `qbinom()`, welche als ersten Input eine Wahrscheinlichkeit  $p$  akzeptiert und dann den kleinsten Wert  $x$  findet, für den gilt, dass  $\mathbb{P}(X = x) \geq p$ .

Wenn wir also wissen möchten wie viele Erfolge mit einer Wahrscheinlichkeit von 50% mindestens zu erwarten sind, dann schreiben wir:

```
qbinom(p = 0.5, size = 50, prob = 0.25)
```

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

Es gilt also:  $\mathbb{P}(X = 12) \geq p$ .

Abbildung 7.4 verdeutlicht dies grafisch.

Möchten wir schließlich eine bestimmte Menge an **Realisierungen** aus einer Binomialverteilung ziehen geht das mit der Funktion `rbinom()`, welche auch wieder drei Argumente verlangt: `n` für die Anzahl der zu ziehenden Realisierungen, sowie `size` und `prob` als da Parameter  $n$  und  $p$  der Binomialverteilung:

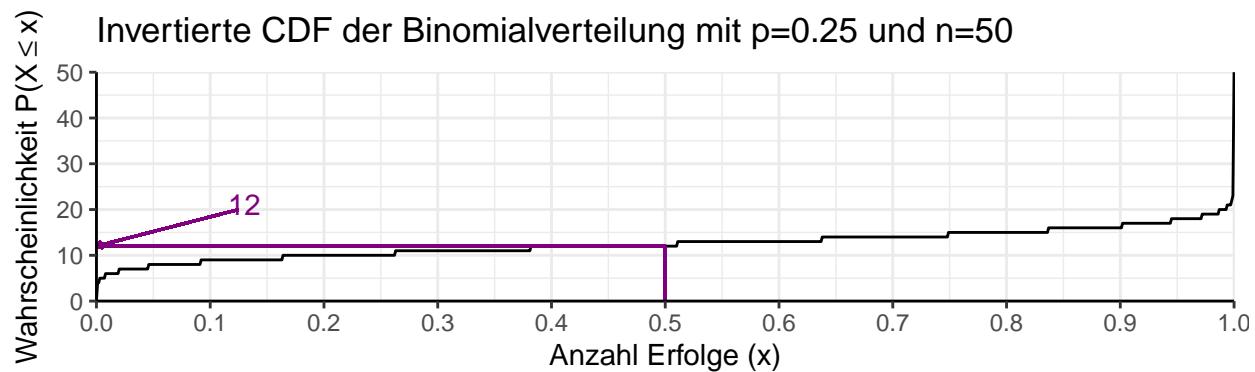
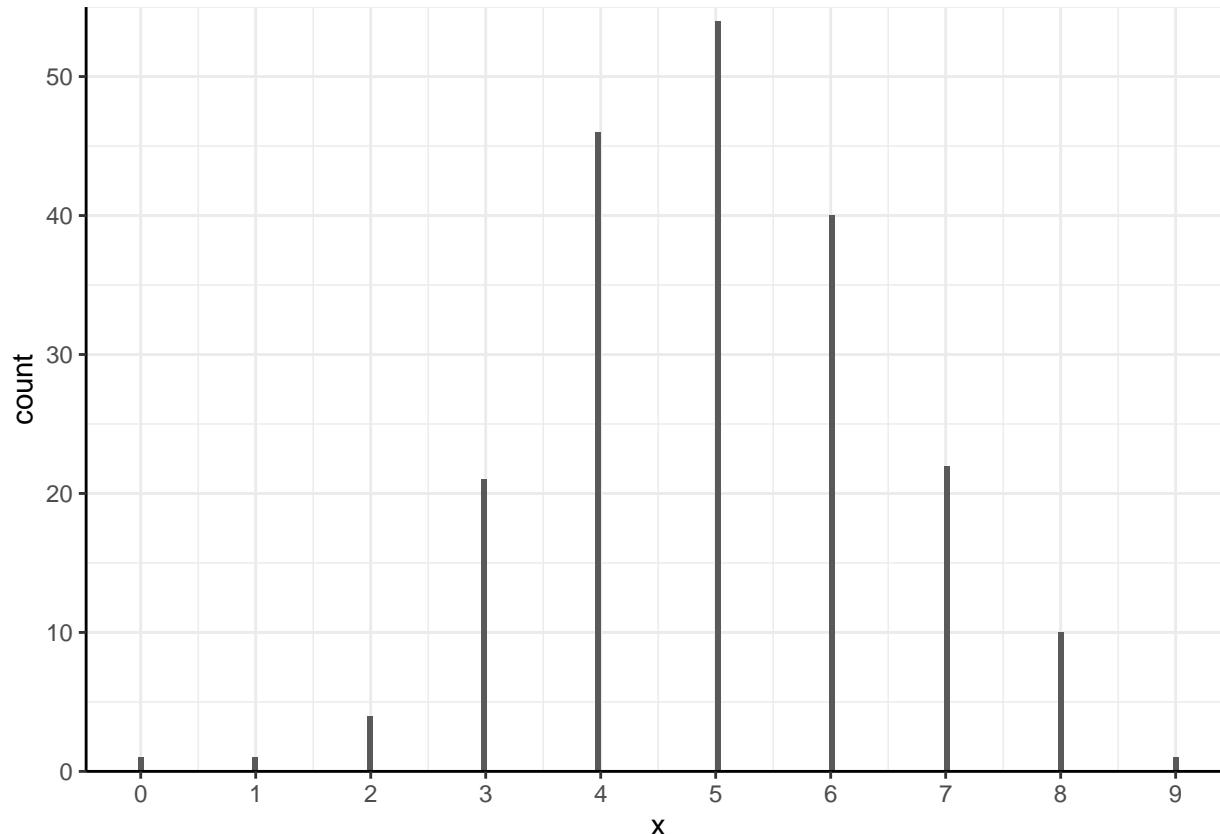


Figure 7.4: Graph der invertierten kumulierten Wahrscheinlichkeitsfunktion der Binomialverteilung mit  $p = 0.25$  und  $n = 50$

```
sample_binom <- rbinom(n = 5, size = 10, prob = 0.4)
sample_binom
```

```
## [1] 2 2 3 5 6
```

**Anwendungsbeispiel Binomialverteilung:** Unser Zufallsexperiment besteht aus dem zehnmaligen Werfen einer fairen Münze. Unter ‘Erfolg’ verstehen wir das Werfen von ‘Zahl’. Nehmen wir an, wir führen das Zufallsexperiment 10 Mal durch, werfen also insgesamt 100 Mal die Münze und schreiben jeweils auf, wie häufig wir dabei einen Erfolg verbuchen konnten. Wenn wir unsere Ergebnisse aufmalen, indem wir auf der x-Achse die Anzahl der Erfolge, und auf der y-Achse die Anzahl der Experimente mit genau dieser Anzahl an Erfolgen festhalten, erhalten wir ein Histogramm, das ungefähr so aussieht wie in Abbildung ??.



Aus der Logik der Konstruktion des Zufallsexperiments und der Inspektion unserer Daten können wir schließen, dass die Binomialverteilung eine sinnvolle Beschreibung des Zufallsexperiments und der daraus entstandenen Stichprobe von 100 Münzwurfergebnissen ist. Da wir eine faire Münze geworfen haben macht es Sinn für die Binomialverteilung  $p = 0.5$  anzunehmen, und da wir in jedem einzelnen Experiment die Münze 10 Mal geworfen haben für  $n = 10$ . Wenn wir die mit  $n = 10$  und  $p = 0.5$  parametrisierte theoretische Binomialverteilung nehmen und ihre theoretische Verteilungsfunktion über die Aufzeichnungen unserer Ergebnisse legen, können wir uns in dieser Vermutung bestärkt fühlen, wie in Abbildung 7.5 ersichtlich ist.

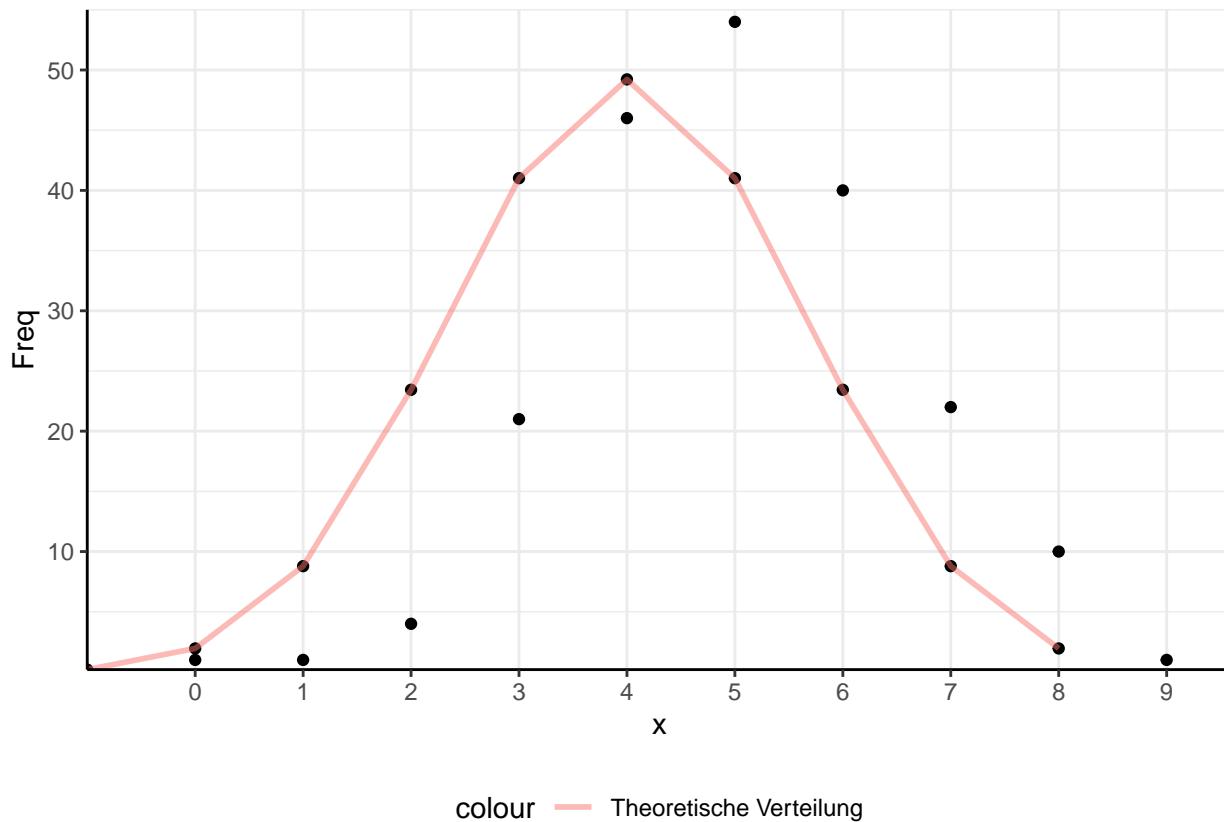


Figure 7.5: Vergleich der empirischen Stichprobe und der parametrisierten theoretischen Binomialverteilungsfunktion

### 7.3.3 Beispiel: die Poisson-Verteilung

Bei der Poisson-Verteilung handelt es sich um die Standardverteilung für unbeschränkte Zähldaten, also diskrete Daten, die kein natürliches Maximum haben.

Es handelt sich dabei zudem um eine **ein-parametrische** Funktion, deren einziger Parameter  $\lambda > 0$  ist.  $\lambda$  wird häufig als die mittlere Ereignishäufigkeit interpretiert und ist **zgleich Erwartungswert als auch Varianz** der Verteilung:  $\mathbb{E}(P_\lambda) = \text{Var}(P_\lambda) = \lambda$ .

Ihre Definitionsmenge ist  $\mathbb{N}$ , also alle natürlichen Zahlen - daher ist sie im Gegensatz zur Binomialverteilung geeignet, wenn die Definitionsmenge der Verteilung keine natürliche Grenze hat.

Die **Wahrscheinlichkeitsfunktion** der Poisson-Verteilung hat die folgende Form:

$$p_\lambda(x) = \frac{\lambda^x}{x!} e^{-\lambda}$$

Abbildung 7.6 zeigt wie sich die Wahrscheinlichkeitsfunktion für unterschiedliche Werte von  $\lambda$  manifestiert.

### Die Poisson–Verteilung für verschiedene $\lambda$

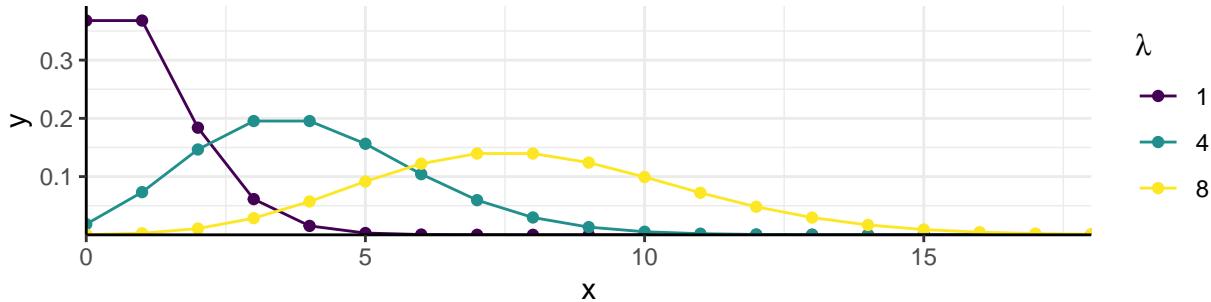


Figure 7.6: Poisson-Verteilung für verschiedene Parameter.

Wir können die Verteilung mit sehr ähnlichen Funktionen wie bei der Binomialverteilung analysieren. Nur die Parameter müssen entsprechend angepasst werden, da es bei der Poisson-Verteilung jetzt nur noch einen Parameter (`lambda`) gibt.

Möchten wir die Wahrscheinlichkeit berechnen, genau  $x$  Erfolge zu beobachten, also  $\mathbb{P}(X = x)$  geht das mit der Funktion `dpois()`. Das einzige notwendige Argument ist `lambda` (siehe zudem Abbildung 7.7):

```
dpois(5, lambda = 4)
```

```
## [1] 0.1562935
```

### Die Poisson–Verteilung für $\lambda = 4$

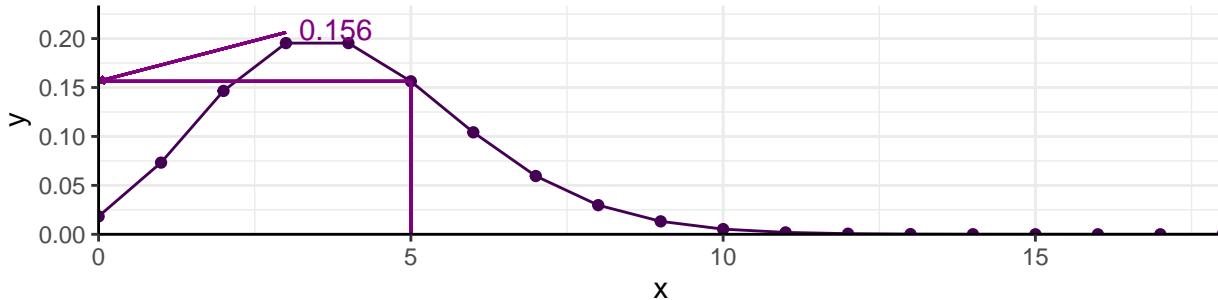


Figure 7.7: Poisson-Verteilung mit ausgewählten Parameterwerten.

Informationen über die CDF erhalten wir über die Funktion `ppois()`, die zwei Argumente, `q` und `lambda`, annimmt. Grafisch dargestellt ist dies in Abbildung 7.8.

Mit der Funktion `qpois()` finden wir für eine Wahrscheinlichkeit `p` den kleinsten Wert  $x$ , für den gilt, dass  $\mathbb{P}(X = x) \geq p$ .

Wenn wir also wissen möchten wie viele Erfolge mit einer Wahrscheinlichkeit von 50% mindestens zu erwarten sind, dann schreiben wir:

```
qpois(p = 0.5, lambda = 4)
```

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

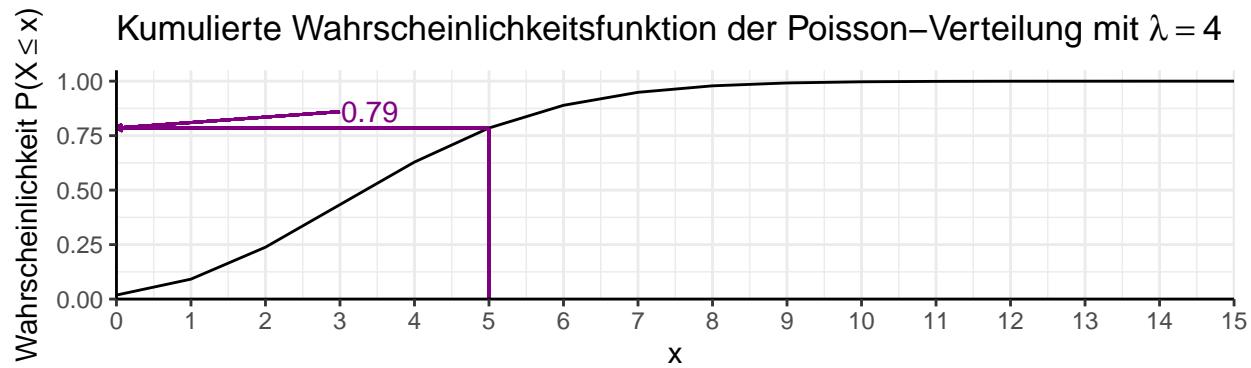


Figure 7.8: Kumulierte Wahrscheinlichkeitsfunktion der Poisson-Verteilung mit  $\lambda = 4$

Es gilt also:  $\mathbb{P}(X = 4) \geq 0.5$ .

Wir können dies erneut grafisch verdeutlichen, wie in Abbildung 7.9 dargestellt.

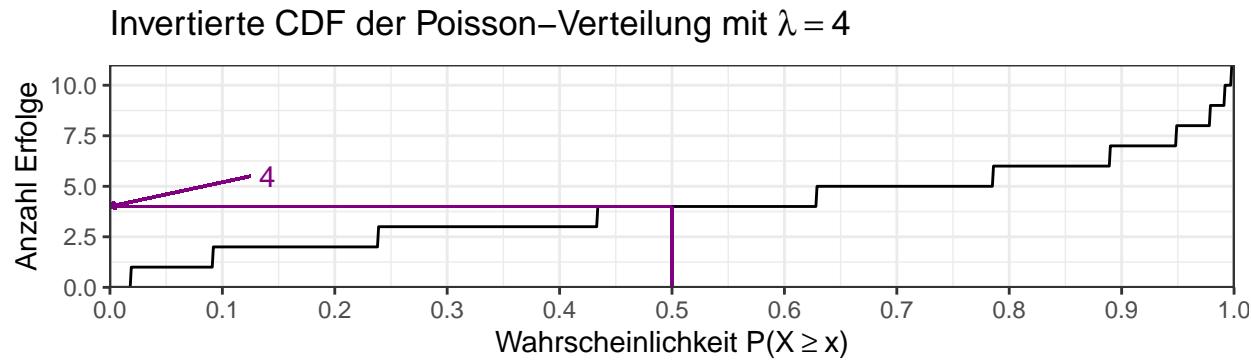


Figure 7.9: Invertierte CDF der Poisson-Verteilung mit  $\lambda = 4$

Möchten wir schließlich eine bestimmte Menge an **Realisierungen** der ZV aus einer Poisson-Verteilung ziehen geht das mit `rpois()`, welches zwei notwendige Argumente annimmt: `n` für die Anzahl der Realisierungen und `lambda` für den Parameter  $\lambda$ :

```
pois_sample <- rpois(n = 5, lambda = 4)
pois_sample
```

```
## [1] 3 8 4 4 3
```

### 7.3.4 Hinweise zu diskreten Wahrscheinlichkeitsverteilungen

Wie Sie vielleicht bereits bemerkt haben sind die R Befehle für verschiedene Verteilungen alle gleich aufgebaut. Wenn \* für die Abkürzung einer bestimmten Verteilung steht, können wir mit der Funktion `d*`() die Werte der Wahrscheinlichkeitsverteilung, mit `p*`() die Werte der kumulierten Wahrscheinlichkeitsverteilung und mit `q*`() die der Quantilsfunktion berechnen. Mit `r*`() werden Realisierungen von Zufallszahlen generiert. Für das Beispiel der Binomialverteilung, welcher die Abkürzung `binom` zugewiesen wurde, heißen die Funktionen entsprechend `dbinom()`, `pbinom()`, `qbinom()` und `rbinom()`.

Tabelle 7.2 gibt einen Überblick über gängige Abkürzungen und die Parameter der oben besprochenen diskreten Verteilungen.

Table 7.2: Überblick der besprochenen diskreten Verteilungen.

Verteilung	Abkürzung	Parameter
Binomialverteilung	<code>binom</code>	<code>size, prob</code>
Poisson-Verteilung	<code>pois</code>	<code>lambda</code>

## 7.4 Stetige Wahrscheinlichkeitsmodelle

### 7.4.1 Stetige ZV

In vorangegangen Abschnitt haben wir uns mit diskreten Wahrscheinlichkeitsmodellen beschäftigt. Die diesen Modellen zugrundeliegenden ZV hatten einen abzählbaren Wertebereich. Häufig interessieren wir uns aber für ZV mit einem nicht abzählbaren Wertebereich, z.B.  $\mathbb{R}$  oder  $[0, 1]$ .<sup>6</sup>

Bei stetigen Wahrscheinlichkeitsmodellen liegen zwischen zwei Punkten unendlich viele Punkte. Das hat bedeutende Implikationen für die Angabe von Wahrscheinlichkeiten. Im Gegensatz zu diskreten Wahrscheinlichkeitsmodellen hat demnach jeder einzelne Punkt im Wertebereich der ZV die Wahrscheinlichkeit 0:

$$\mathbb{P}(X = x_k) = 0 \quad \forall x_k \in W_X$$

wobei  $W_X$  für den Wertebereich von ZV  $X$  steht.

Als Lösung werden Wahrscheinlichkeiten bei stetigen ZV nicht als Punktwahrscheinlichkeiten, sondern als Intervallwahrscheinlichkeiten angeben. Aus  $\mathbb{P}(X = x)$  im diskreten Fall wird im stetigen Fall also:

$$\mathbb{P}(a < X \leq b) = \int_a^b f(x)dx, \quad a < b$$

Entsprechend wird für stetige ZV eine etwas andere Notation als für diskrete ZV verwendet, wobei das Prinzip gleich bleibt. Zudem werden Sie merken, dass im stetigen Fall anstatt Summen immer Integrale verwendet werden. Informell kann man ja auch sagen, dass ein Integral nichts anderes ist als eine Summe über stetige Werte.

Wo wir bei diskreten ZV eine Wahrscheinlichkeitsfunktion (PMF) verwendet haben verwenden wir nun eine **Wahrscheinlichkeitsdichte** (*probability density function*, PDF). Aus der PMF  $p_X(x) = \mathbb{P}(X = x_k)$  im diskreten Fall wird nun also die PDF  $f_X(x)$ , für die gilt, dass  $\mathbb{P}(a < X \leq b) = \int_a^b f(x)dx$  für den stetigen Fall.

Für die PDF gilt äquivalent zum diskreten Fall, dass  $f_X(x) \geq 0$  (keine negativen Werte) und  $\int_{-\infty}^{\infty} f(x)dx = 1$  (das Integral (die ‘stetige Summe’) über den ganzen Wertebereich ergibt 1). Allerdings gibt es einen wichtigen Unterschied: im Gegensatz zur PMF  $p_X(x)$  gibt die PDF  $f_X(x)$  *keine* Wahrscheinlichkeiten an - daher auch nur die Restriktion  $f_X(x) \geq 0$  und *nicht*  $1 \geq f_X(x) \geq 0$ . Wenn wir von Wahrscheinlichkeiten reden wollen, müssen wir die PDF integrieren:

$$\mathbb{P}((a, b]) = \int_a^b f(x)dx$$

da wir im stetigen Fall für einzelne Punkte keine von Null verschiedenen Wahrscheinlichkeiten haben.

---

<sup>6</sup>Die Intervallschreibweise  $[0, 1]$  ist potenziell verwirrend. Es gilt:  $[a, b] = \{x \in \mathbb{R} | a \leq x \leq b\}$  (geschlossenes Intervall),  $(a, b) = \{x \in \mathbb{R} | a < x < b\}$  (offenes Intervall),  $(a, b] = \{x \in \mathbb{R} | a < x \leq b\}$  (linksöffnetes Intervall) und  $(a, b) = \{x \in \mathbb{R} | a \leq x < b\}$  (rechtsöffnetes Intervall).

Wen übrigens die unterschiedlichen Bezeichnungen “probability *mass*” und “probablity *density*” irritieren: tatsächlich ist die Verwendung dieser Bezeichnungen ganz analog zu den physikalischen Pendants “Masse” und “Dichte” in der Physik: wenn wir die Masse für einzelne Teile einer Stange haben bekommen wir die Gesamtmasse indem wir die Masse der Teile addieren - das ist genauso wie bei der PMF: wir bekommen die Gesamtwahrscheinlichkeit indem wir die Wahrscheinlichkeiten für einzelne Events addieren und die Einzelgewichte indem wir uns die Masse der einzelnen Teile ansehen. Wenn wir jetzt eine Stange haben, die unterschiedlich dicht ist, bekommen wir die gesamte Masse indem wir über die Dichte integrieren. Wir können die Stange auch in kleinere Teile schneiden und deren Masse addieren. Wenn die Teile unendlich klein werden kommen wir zum gleichen Ergebnis wie bei der Integration.

Die **kumulative Verteilungsfunktion** (CDF) ist im stetigen Fall genauso definiert wie im diskreten Fall:  $F_X(x) = \mathbb{P}(X \leq x)$ , wobei immer gilt:

$$\mathbb{P}(a < X \leq b) = F(b) - F(a) = \int_a^b f(x)dx$$

Man sieht hier, dass die Dichtefunktion (PDF) einer ZV die Ableitung ihrer kumulative Verteilungsfunktion (CDF) ist:

$$F'_X(x) = f_X(x)$$

Wie oben beschrieben können wir die Werte an einzelnen Punkten der PDF nicht als *absolute* Wahrscheinlichkeiten interpretieren, da die Wahrscheinlichkeit für einzelne Punkte immer gleich 0 ist und die PDF auch Werte größer 1 annehmen kann. Wir können aber die Werte der PDF an zwei oder mehr Punkten vergleichen um die *relative* Wahrscheinlichkeit der einzelnen Punkte zu bekommen.

Wie bei den diskreten ZV beschreiben wir eine ZV mit Hilfe von bestimmten Kennzahlen, wie dem **Erwartungswert**, der **Varianz** und den **Quantilen**. Diese sind quasi äquivalent zum diskreten Fall definiert, nur eben über Integrale (wir vergleichen alle folgenden Definitionen mit ihrem diskreten Pendant am Ende des Abschnitts). Für den Erwartungswert der ZV  $X$  gilt somit:

$$\mathbb{E}(X) = \int_{-\infty}^{\infty} xf(x)dx$$

Für die Varianz und die Standardabweichung entsprechend:

$$Var(X) = \mathbb{E}(X - \mathbb{E}(X))^2 = \int_{-\infty}^{\infty} (x - \mathbb{E}(X))^2 f(x)dx$$

$$\sigma_X = \sqrt{Var(X)}$$

Und, schlussendlich, gilt für das  $\alpha$ -Quantil  $q(\alpha)$ :

$$\mathbb{P}(X \leq q(\alpha)) = \alpha$$

In Abbildung 7.10 werden das 0.25 und 0.5-Quantil visuell dargestellt.

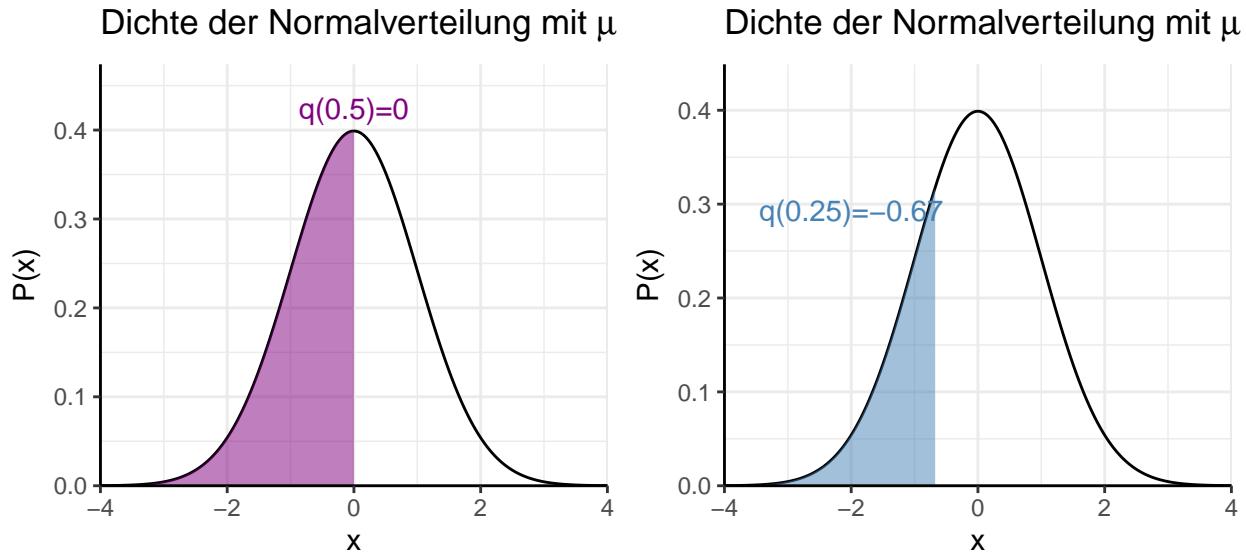


Figure 7.10: Vergleich des 0.25- und 0.5-Quantils

Tabelle 7.3 vergleicht noch einmal die Definitionen der Kennzahlen und charakteristischer Verteilungen für den stetigen und diskreten Fall.

Table 7.3: Vergleich der Kennzahlen charakteristischer Verteilungen im stetigen und im diskreten Fall.

Bezeichnung	Diskreter Fall	Stetiger Fall
Erwartungswert	$\mathbb{E}(x) = \sum_{x \in W_x} \mathbb{P}(X = x)x$	$\mathbb{E}(X) = \int_{-\infty}^{\infty} xf(x)dx$
Varianz	$Var(X) = \sum_{x \in W_x} [x - \mathbb{E}(X)]^2 \mathbb{P}(X = x)x$	$Var(X) = \mathbb{E}(X - \mathbb{E}(X))^2$
Standardabweichung	$\sqrt{Var(X)}$	$\sqrt{Var(X)}$
$\alpha$ -Quantil	$\mathbb{P}(X \leq q(\alpha)) = \alpha$	$\mathbb{P}(X \leq q(\alpha)) = \alpha$
Dichtefunktion (PDF)	NA	$f_X(x) = \mathbb{P}([a, b]) = \int_a^b f(x)dx$
Wahrsch's-funktion (PMF)	$p_X(x_k) = \mathbb{P}(X = x_k)$	NA
Kumulierte Verteilungsfunktion (CDF)	$\mathbb{P}(X \leq x)$	$F(x) = \mathbb{P}(X \leq x)$

Analog zum diskreten Fall wollen wir uns nun die am häufigsten vorkommenden stetigen Verteilungen noch einmal genauer anschauen. Vorher wollen wir jedoch die oben eingeführten Konzepte (statistische Unabhängigkeit, bedingte Wahrscheinlichkeiten, etc.) noch für den stetigen Fall formulieren - am Prinzip ändert sich hier nichts, nur an der Notation.

#### 7.4.2 Beispiel: die Uniformverteilung

Die Uniformverteilung kann auch mit einem beliebigen Intervall  $[a, b]$  mit  $a < b$  definiert werden und ist dadurch gekennzeichnet, dass die Dichte über  $[a, b]$  vollkommen konstant ist. Ihre einzigen Parameter sind die Grenzen des Intervalls,  $a$  und  $b$ .

Die kontinuierliche Verteilung hat die Dichte alle Werte zwischen  $a$  und  $b$ , die mit  $x$  bezeichnet werden, gleich Null ist.

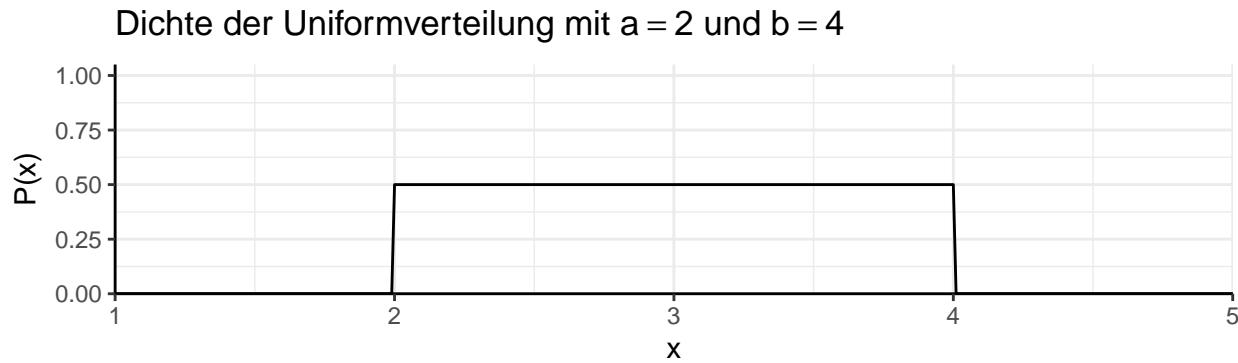


Figure 7.11: Dichte der Uniformverteilung mit  $a = 2$  und  $b = 4$

```
dunif(seq(2, 3, 0.1), min = 0, max = 4)
```

```
## [1] 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25
```

Wie wir sehen erhalten wir hier immer den gleichen Wert  $\frac{1}{b-a}$ , was die zentrale Eigenschaft der Uniformverteilung ist. Hier wird auch deutlich, dass dieser Wert die *relative* Wahrscheinlichkeit angibt, da die absolute Wahrscheinlichkeit für jeden einzelnen Wert wie oben beschrieben bei stetigen ZV 0 ist.

Die CDF berechnen wir entsprechend mit `punif()`. Wenn  $X \sim U(0, 4)$  erhalten wir  $\mathbb{P}(X \leq 3)$  entsprechend mit:

```
punif(0.8, min = 0, max = 4)
```

```
## [1] 0.2
```

Abbildung 7.12 zeigt dies grafisch.

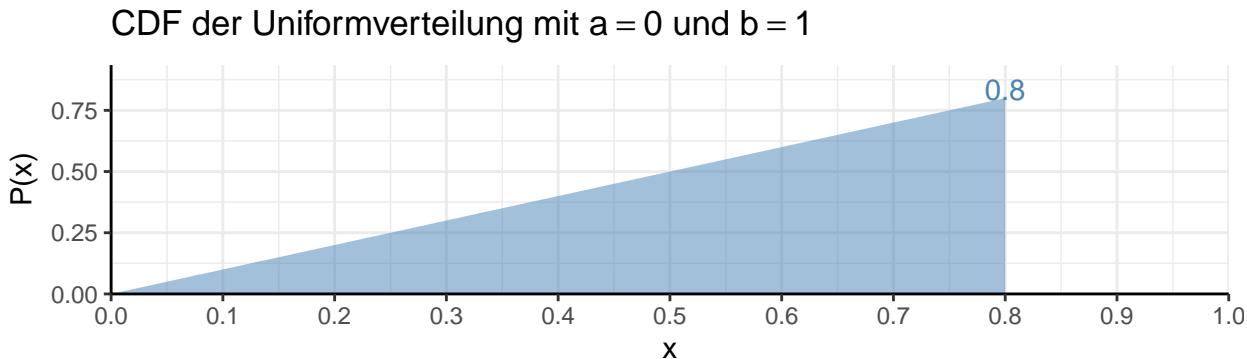


Figure 7.12: CDF der Uniformverteilung mit  $a = 0$  und  $b = 1$

Auch ansonsten können wir die Syntax der diskreten Verteilungen mehr oder weniger übernehmen: `qunif()` akzeptiert die gleichen Parameter wie `punif()` und gibt uns Werte der inversen CDF. `runif()` kann verwendet werden um Realisierungen einer uniform verteilten ZV zu generieren:

```
uniform_sample <- runif(5, min = 0, max = 4)
uniform_sample
```

```
## [1] 3.5209862 1.4563675 1.1529571 0.6825809 0.6886870
```

### 7.4.3 Beispiel: die Normalverteilung

Die wahrscheinlich bekannteste stetige Verteilung ist die Normalverteilung. Das liegt nicht nur daran, dass viele natürliche Phänomene als die Realisierung einer normalverteilten ZV modelliert werden können, sondern auch weil es sich mit der Normalverteilung in der Regel sehr einfach rechnen lässt. Sie ist also häufig auch einfach eine bequeme Annahme.

Bei der Normalverteilung handelt es sich um eine **zwei-parametrische** Verteilung über den Wertebereich  $W_X = \mathbb{R}$ . Die beiden Parameter sind  $\mu$  und  $\sigma^2$ , welche unmittelbar als Erwartungswert ( $\mathbb{E}(X) = \mu$ ) und Varianz ( $Var(X) = \sigma^2$ ) gelten. Wir schreiben  $X \sim \mathcal{N}(\mu, \sigma^2)$  wenn für die PDF von  $X$  gilt:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Unter der **Standard-Normalverteilung** verstehen wir eine Normalverteilung mit den Paramtern  $\mu = 0$  und  $\sigma = 1$ .<sup>7</sup> Sie verfügt über die deutlich vereinfachte PDF:

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

Die CDF der Normalverteilung ist analytisch nicht einfach darzustellen, die Werte können in R aber leicht über die Funktion `pnorm` (s.u.) abgerufen werden.

In Abbildung 7.13 sind die PDF und CDF für exemplarische Parameterkombinationen dargestellt.

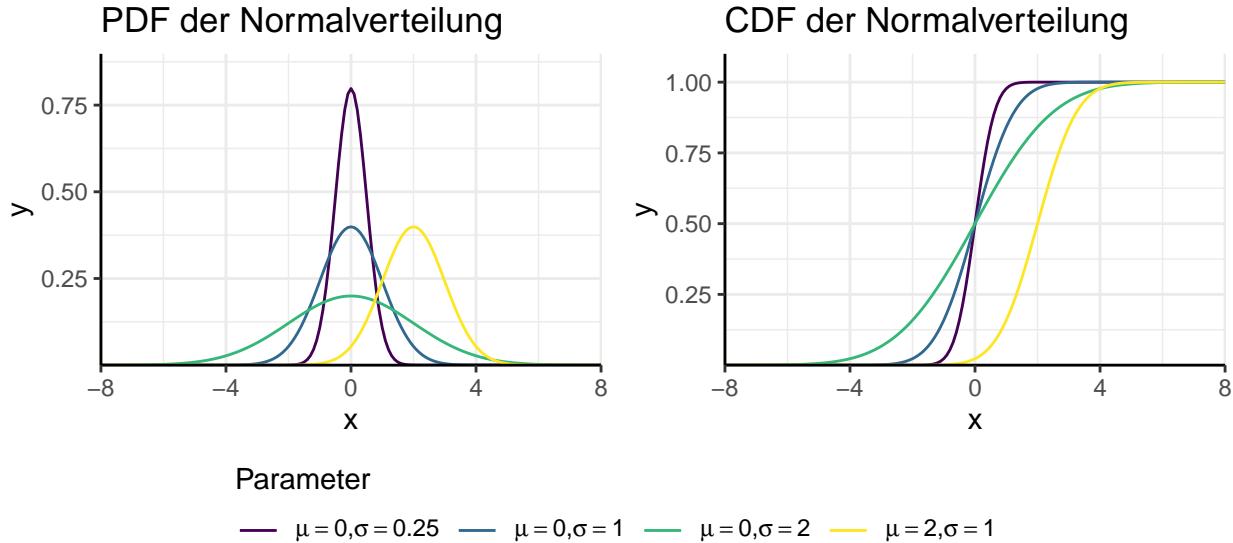


Figure 7.13: Beispielhafter Vergleich einer PDF und CDF bei Normalverteilung

Die Abkürzung in R ist `norm`. Alle Funktionen nehmen die Parameter  $\mu$  und  $\sigma$  (nicht  $\sigma^2$ ) über `mean` und `sd` als notwendige Argumente. Ansonsten ist die Verwendung äquivalent zu den vorherigen Beispielen:

```
dnorm(c(0.5, 0.75), mean = 1, sd = 2) # relative Wahrscheinlichkeiten über PDF
```

```
## [1] 0.1933341 0.1979188
```

<sup>7</sup>Viele Tabellen mit bestimmten Kennzahlen der Normalverteilung beziehen sich auf die Standard-Normalverteilung. Wenn man diese Werte verwenden will, muss man die tatsächlich verwendete Stichprobe ggf. erst [z-transformieren](#). Unter Letzterem versteht man die *Normalisierung* einer ZV sodass sie den Erwartungswert 0 und die Varianz 1 besitzt. Dies geht i.d.R. für jede ZV  $X$  recht einfach über die Formel  $Z = \frac{X-\mu}{\sigma}$ , wobei  $Z$  die standardisierte ZV,  $\mu$  den Erwartungswert und  $\sigma$  die Standardabweichung von  $X$  bezeichnet

```

pnorm(c(0.5, 0.75), mean = 1, sd = 2) # Werte der CDF
## [1] 0.4012937 0.4502618

qnorm(c(0.5, 0.75), mean = 1, sd = 2) # Werte der I-CDF
## [1] 1.000000 2.34898

norm_sample <- rnorm(5, mean = 1, sd = 2) # 5 Realisierungen der ZV
norm_sample
## [1] 0.9099446 -0.5698089 -2.3358839 0.2395470 2.8379932

```

Beispiel zum Zusammenhang `dnorm()` und `qnorm()`

#### 7.4.4 Beispiel: die Exponentialverteilung

Sehr häufig wird uns auch die Exponentialverteilung begegnen. Außerhalb der Ökonomik wird sie v.a. zur Modellierung von Zerfallsprozessen oder Wartezeiten verwendet, in der Ökonomik spielt sie in der Wachstumstheorie eine zentrale Rolle. Es handelt sich bei der Exponentialverteilung um eine **ein-parametrigie** Verteilung mit Parameter  $\lambda \in \mathbb{R}^+$  und mit dem Wertebereich  $W_X = [0, \infty]$ .

Die PDF der Exponentialverteilung ist:

$$f(x) = \begin{cases} 0 & x < 0 \\ \lambda e^{-\lambda x} & x \geq 0 \end{cases}$$

wobei  $e$  die **Eulersche Zahl** ist. Die CDF ist entsprechend:

$$F(x) = \begin{cases} 0 & x < 0 \\ 1 - e^{-\lambda x} & x \geq 0 \end{cases}$$

Beide Verteilungen sind in Abbildung 7.14 dargestellt.

Der Erwartungswert und die Varianz sind für die Exponentialverteilung äquivalent und hängen ausschließlich von  $\lambda$  ab:  $\mathbb{E}(X) = \sigma_X = \frac{1}{\lambda}$ .

Die Abkürzung in R ist `exp`. Alle Funktionen nehmen den Parameter  $\lambda$  über das Argument `rate` an:

```

dexp(c(0.5, 0.75), rate = 1) # relative Wahrscheinlichkeiten über PDF
## [1] 0.6065307 0.4723666

pexp(c(0.5, 0.75), rate = 1) # Werte der CDF
## [1] 0.3934693 0.5276334

qexp(c(0.5, 0.75), rate = 1) # Werte der I-CDF
## [1] 0.6931472 1.3862944

exp_sample <- rexp(5, rate = 1) # 5 Realisierungen der ZV
exp_sample

```

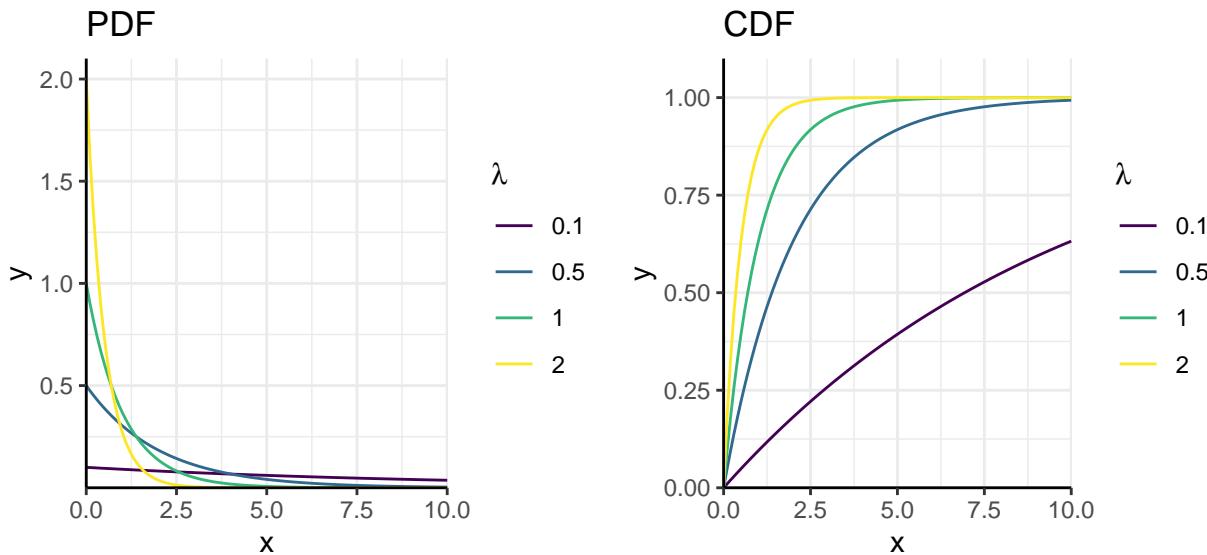


Figure 7.14: Beispielhafter Vergleich einer PDF und CDF bei Exponentialverteilung

```
## [1] 0.8232605 0.4757590 3.4635949 1.2740277 1.0814852
```

Es gibt übrigens einen [wichtigen Zusammenhang](#) zwischen der stetigen Exponential- und der diskreten Poisson-Verteilung.

## 7.5 Zusammenfassung Wahrscheinlichkeitsmodelle für einzelne ZV

Tabelle 7.4 fasst noch einmal alle Wahrscheinlichkeitsmodelle zusammen, die wir bislang betrachtet haben.

Table 7.4: Überblick der Verteilungen und ihrer Parameter.

Verteilung	Art	Abkürzung	Parameter
Binomialverteilung	Diskret	<code>binom</code>	<code>size, prob</code>
Poisson-Verteilung	Diskret	<code>pois</code>	<code>lambda</code>
Uniform-Verteilung	Kontinuierlich	<code>unif</code>	<code>min, max</code>
Normalverteilung	Kontinuierlich	<code>norm</code>	<code>mean, sd</code>
Exponential-Verteilung	Kontinuierlich	<code>exp</code>	<code>rate</code>

In der statistischen Praxis sind das die Modelle, die wir verwenden, um die DGP (*data generating processes*) zu beschreiben - also die Prozesse, welche die Daten, die wir in unserer Forschung verwenden, generiert haben.

Deswegen sprechen Statistiker\*innen auch häufig von *Populationsmodellen*. Am besten stellt man es sich mit Hilfe der `r*()` Funktionen vor: man nimmt an, dass es einen DGP gibt, und dass unsere Daten der Output der `r*()`-Funktion zum Ziehen von Realisierungen sind. Mit dem Begriff des Populationsmodells macht man dabei deutlich, dass unsere Stichprobe nur eine Stichprobe darstellt - und nicht die gesamte Population aller möglichen Realisierungen des DGP.

Nun wird auch deutlich, warum Kenntnisse in der Wahrscheinlichkeitsrechnung so wichtig sind: wenn wir statistisch mit Daten arbeiten, dann versuchen wir in der Regel über die Daten Rückschlüsse auf den DGP zu schließen. Dafür müssen wir zunächst einmal eine grobe Struktur für den DGP annehmen, und dafür brauchen wir Kenntnisse in der Wahrscheinlichkeitsrechnung und für den entsprechenden Anwendungsfall konkrete Vorannahmen. Dann können wir, gegeben unsere Daten, unsere Beschreibung des DGP verfeinern.

Das bedeutet, dass wir für den DGP ein bestimmtes Wahrscheinlichkeitsmodell annehmen und dann auf Basis unserer Daten die Parameter für dieses Modell schätzen. Dieses Vorgehen nennen wir *parametrisch*, weil wir hier

Table 7.5: Die gemeinsame Verteilung für das Werfen zweier Würfel.

$X/Y$	1	2	3	4	5	6
1	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$
2	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$
3	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$
4	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$
5	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$
6	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$

ZV sind in der Praxis häufig besonders relevant.

Die Verteilungen, die wir bislang kennengelernt haben beschreiben alle die Verteilung einer einzelnen ZV. Anhand der Konzepte der bedingten Wahrscheinlichkeit und der statistischen Unabhängigkeit konnten wir ja schon erahnen, dass es häufig von besonderem Interesse ist, wie mehrere ZV miteinander interagieren. Häufig sind wir daran interessiert, die Verteilung dieser neuen ZV zu charakterisieren. Wir nennen die Verteilung einer ZV, die sich aus mehreren ZV ergibt eine **gemeinsame Verteilung**. Eine gemeinsame Verteilung gibt uns Informationen über die Wahrscheinlichkeit von Ereignissen, die von allen beteiligten ZV abhängen.

**Beispiel:** Ein Beispiel für eine praktisch sehr relevante ‘kombinierte’ ZV wäre die gemeinsame Verteilung von Luftverschmutzung und Atemwegserkrankungen. Sowohl der Grad an Luftverschmutzung als auch das Auftreten einer Atemwegserkrankung kann jeweils als isolierte ZV modelliert werden, aber von besonderem Interesse ist natürlich deren gemeinsame Verteilung, bzw. die bedingten Wahrscheinlichkeiten (also für einen bestimmten Grad an Luftverschmutzung eine Atemwegserkrankung zu bekommen).

### 7.6.1 Gemeinsame Verteilungen für diskrete ZV

Nehmen wir einmal an wir haben es mit zwei diskreten ZV Variablen,  $X$  und  $Y$ , zu tun.  $X$  kann dabei die Werte  $\{x_1, x_2, \dots, x_n\}$  und  $Y$  die Werte  $\{y_1, y_2, \dots, y_m\}$  annehmen. Die gemeinsame Verteilungsfunktion sollte nun Wahrscheinlichkeiten für alle möglichen Kombinationen  $\{(x_1, y_1), (x_1, y_2), \dots, (x_n, y_m)\}$  angeben. Wir sprechen hier also von einer gemeinsamen PMF  $p_{XY}(x_i, y_j)$  für die gilt:

$$p_{XY}(x_i, y_j) = \mathbb{P}(X = x_i, Y = y_j)$$

Eine solche gemeinsame PMF hat zwei Eigenschaften, ganz analog zur ‘normalen’ PMF:

1.  $0 \leq p_{XY}(x_i, y_j) \leq 1$ : die Wahrscheinlichkeiten für jede Kombination müssen zwischen 0 und 1 liegen;
2.  $\sum_{i=1}^n \sum_{j=1}^m p_{XY}(x_i, y_j) = 1$ : die Summe aller Einzelwahrscheinlichkeiten ist 1.

**Beispiel: das Werfen zweier Würfel** Da wir den Wurf eines einzelnen Würfels als diskrete ZV repräsentieren können, können wir den Wurf zweier Würfel als eine gemeinsame diskrete ZV repräsentieren. Grafisch können wir die Wahrscheinlichkeiten recht anschaulich in einer Tabelle abbilden, wobei die einzelnen Zellen jeweils die Werte der gemeinsamen PMF enthalten (siehe Abbildung 7.5).

$X$  und  $Y$  beschreiben dabei jeweils die ZV für den ersten und zweiten Würfel.

### 7.6.2 Gemeinsame Verteilungen für stetige ZV

Die Darstellung des stetigen Falles ist komplett äquivalent: nehmen wir an,  $X$  sei eine stetige ZV mit Wertebereich  $[a, b]$  und  $Y$  eine stetige ZV mit Wertebereich  $[c, d]$ , dann ist der Wertebereich der gemeinsamen PDF  $f_{XY}(x, y)$  gegeben durch  $[a, b] \times [c, d]$ . Auch für  $f_{XY}(x, y)$  gilt analog zum einfachen Fall:

1.  $p_{XY}(x_i, y_i) \geq 0$ : die Wahrscheinlichkeitsdichte für jede Kombination muss größer Null sein;
2.  $\int_c^d \int_a^b f_{XY}(x, y) dx dy = 1$ : das Integral über den gesamten Wertebereich ist 1.

Grafisch könnten wir die gemeinsame PDF als Quadrat darstellen (siehe Abbildung 7.15).

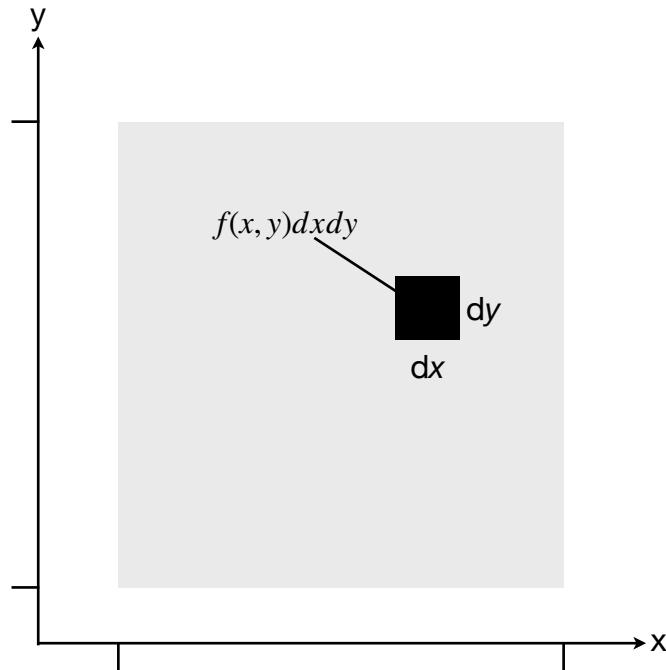


Figure 7.15: Gemeinsame Verteilung zweier stetiger ZV. Die Wahrscheinlichkeit eines Events korrespondiert zur Fläche.

### 7.6.3 Gemeinsame kumulative Verteilungen

Auch die kumulativen Verteilungen für mehrere ZV sind äquivalent zum einfachen Fall definiert. In der allgemeinen Schreibweise schreiben wir:

$$F_{XY}(x, y) = \mathbb{P}(X \leq x, Y \leq y).$$

Für den diskreten Bereich übersetzt sich das konkret in:

$$F_{XY}(x, y) = \sum_{x_i \leq x} \sum_{y_j \leq y} p(x, y)$$

Im kontinuierlichen Fall ist diese Funktion wieder für den Wertebereich  $[a, b] \times [c, d]$  definiert als:

$$F_{XY}(x, y) = \int_c^d \int_a^b f(x, y) dx dy$$

Table 7.6: Die gemeinsame Verteilung für das Werfen zweier Würfel.

$X/Y$	1	2	3	4	5	6	$p_X(x_i)$
1	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\sum_j p(1, y_j) = \frac{1}{6}$
2	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\sum_j p(2, y_j) = \frac{1}{6}$
3	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\sum_j p(3, y_j) = \frac{1}{6}$
4	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\sum_j p(4, y_j) = \frac{1}{6}$
5	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\sum_j p(5, y_j) = \frac{1}{6}$
6	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\frac{1}{36}$	$\sum_j p(6, y_j) = \frac{1}{6}$

Wenn wir aus der CDF die PDF herleiten wollen müssen wir die CDF nach beiden Variablen ableiten:

$$f(x, y) = \frac{\partial^2 F}{\partial x \partial y}(x, y)$$

Ansonsten sind die Eigenschaften der gemeinsamen CDF wieder vergleichbar zu denen der einfachen CDF (wachsend und für positive/negative Extremwerte von x und y geht der Wert gegen 0/1).

#### 7.6.4 Marginale Verteilungen

Häufig kennen wir die gemeinsame Verteilung von zwei oder mehr ZV und wollen aus dieser gemeinsamen Verteilung die Verteilungen der einzelnen ZV ableiten. Haben wir es z.B. mit einer gemeinsamen Verteilung  $p_{XY}(x, y)$  zu tun wollen wir häufig die separaten Verteilungen  $p_X(x)$  und  $p_Y(y)$  ableiten. Wir sprechen in diesem Fall von der Herleitung einer *marginalen* Verteilung von  $X$  bzw.  $Y$ . Im Ergebnis ist eine marginale Verteilung eine ‘ganz normale’ Verteilung, so wie wir sie vor diesem Abschnitt besprochen haben - der Zusatz ‘marginal’ ergibt sich nur daraus, dass sie aus einer gemeinsamen Verteilung abgeleitet wurde.

Im diskreten Fall erhalten wir die marginalen Verteilungen durch das Aufsummieren bei Konstanthaltung der anderen Variablen. Im Falle der gemeinsamen Verteilung  $p_{XY}(x, y)$  gilt dabei also:

$$p_X(x_i) = \sum_j p(x_i, y_j)$$

und

$$p_Y(y_i) = \sum_i p(x_i, y_i)$$

**Beispiel:** Für das oben beschriebene Beispiel des Werfens zweier Würfel können wir die marginale Verteilung des ersten Würfelwurfes (also von  $X$ ) auf die in Abbildung 7.6 dargestellte Art und Weise erhalten. Hier wird auch deutlich, wo der Name ‘marginal’ herkommt: wir betrachten die aufsummierten Wahrscheinlichkeiten ‘am Rand’. Die marginale Verteilung im stetigen Fall hat genau die gleiche Bedeutung wie im diskreten Fall.

### 7.6.5 Bedingte Verteilungen und bedinge Momente

Die bedingte Verteilung einer ZV beschreibt die Verteilung einer ZV für den Fall dass eine andere ZV auf einen bestimmten Realisationswert bedingt ist. Die Definition ist analog zur allgemeinen bedingten Wahrscheinlichkeit, die wir schon weiter oben eingeführt haben. Daher betrachten wir hier nur ein Beispiel und zwar den folgenden Zusammenhang zwischen  $X$  und  $Y$ , wobei gilt, dass  $X$  : “Es schneit” und  $Y$  : “Es ist kalt”. Dann wäre eine solche gemeinsame Verteilung plausibel:

	Kalt ( $X = 1$ )	Warm ( $X = 0$ )
Schnee ( $Y = 1$ )	0.15	0.07
Kein Schnee ( $Y = 0$ )	0.15	0.63

Um aus dieser gemeinsamen Verteilung die bedingte Verteilung von  $Y$  abzuleiten verwenden wir die bereits oben eingeführte Formel:

$$\mathbb{P}(A|B) = \frac{\mathbb{P}(A \cap B)}{\mathbb{P}(B)}$$

und passen sie für unseren Verteilungsfall an:

$$\mathbb{P}(X = x|Y = y) = \frac{\mathbb{P}(X = x, Y = y)}{\mathbb{P}(Y = y)}$$

Sind wir z.B. an der Wahrscheinlichkeit für Schnee interessiert, gegeben dass das Wetter kalt ist, dann ergibt sich:

$$\mathbb{P}(Y = 1|X = 1) = \frac{\mathbb{P}(X = 1, Y = 1)}{\mathbb{P}(X = 1)} = \frac{0.15}{0.3} = 0.5$$

Wir können nun die bereits oben beschriebene Eigenschaft der *Unabhängigkeit* auch noch einmal im Kontext von gemeinsamen Verteilungen formulieren: zwei ZV  $X$  und  $Y$  gelten als unabhängig, wenn die bedingte Verteilung von  $X$  gegeben  $Y$  nicht von  $Y$  abhängt, also gilt, dass  $\mathbb{P}(X = x|Y = y) = \mathbb{P}(X = x)$ .

Ganz analog zur Verteilung können wir bedingte Momente (wie den Erwartungswert oder die Varianz) formulieren. Besonders häufig verwendet wird dabei der *bedingte Erwartungswert*. Ein in diesem Kontext häufig gebrauchtes Konzept ist das *Gesetz der wiederholten Erwartungen (law of iterated expectations)*, das einen Zusammenhang zwischen dem Erwartungswert und dem bedingten Erwartungswert herstellt:

$$\mathbb{E}(X) = \mathbb{E}[\mathbb{E}(X|Y)]$$

Im diskreten Fall können wir das Konzept noch leichter verdeutlichen. Hier gilt:

$$\mathbb{E}(X) = \mathbb{E}[\mathbb{E}(X|Y)] = \sum_i \mathbb{E}(X|Y = y_i) \cdot \mathbb{P}(Y = y) \tag{7.1}$$

Für den Fall zweier Würfe ist das nichts anderes aus das Summieren der Zeile, wobei jedes Event mit der Eintrittwahrscheinlichkeit gewichtet wird.

$$\begin{aligned}\mathbb{E}(X) &= \mathbb{E}[\mathbb{E}(X|Y)] = \mathbb{E}(X|Y=1) \cdot \frac{1}{6} + \mathbb{E}(X|Y=2) \cdot \frac{1}{6} + \mathbb{E}(X|Y=3) \cdot \frac{1}{6} + \mathbb{E}(X|Y=4) \cdot \frac{1}{6} + \\ &\quad \mathbb{E}(X|Y=5) \cdot \frac{1}{6} + \mathbb{E}(X|Y=6) \cdot \frac{1}{6}\end{aligned}\tag{7.2}$$

Für den Fall, dass wir am Erwartungswert für eine 6 beim ersten Würfel interessiert sind wäre das also:

$$\begin{aligned}\mathbb{E}(X=6) &= \mathbb{E}[\mathbb{E}(X=6|Y)] = \mathbb{E}(X=6|Y=1) \cdot \frac{1}{6} + \mathbb{E}(X=6|Y=2) \cdot \frac{1}{6} + \mathbb{E}(X=6|Y=3) \cdot \frac{1}{6} \\ &\quad + \mathbb{E}(X=6|Y=4) \cdot \frac{1}{6} + \mathbb{E}(X=6|Y=5) \cdot \frac{1}{6} + \mathbb{E}(X=6|Y=6) \cdot \frac{1}{6} \\ &= \mathbb{E}[\mathbb{E}(X=6|Y)] = \frac{1}{6} \cdot \frac{1}{6} + \frac{1}{6} \cdot \frac{1}{6} = \frac{1}{6}\end{aligned}\tag{7.3}$$

Im Falle der Würfel liegt übrigens ein Beispiel von *mean independence* vor, denn in diesem Fall gilt:

$$\mathbb{E}(X|Y) = \mathbb{E}(X)$$

Im Falle des doppelten Wüfelwurfes gilt nämlich, dass  $\mathbb{E}(X=6|Y) = \mathbb{E}(X=6) = \frac{1}{6}$ .

Das Gesetz der wiederholten Erwartungen funktioniert auch bei abhängigen ZV. Schauen wir noch einmal auf das Beispiel mit der Kälte und dem Schnee und berechnen wir die Erwartung, dass es schneit:

$$\mathbb{E}(Y=1) = \mathbb{E}[\mathbb{E}(Y=1|X)] = 0.15 \cdot 0.3 + 0.07 \cdot 0.7 = 0.094$$

Dabei liegt hier *keine mean independence* vor, denn:  $\mathbb{E}(Y=1) = \mathbb{E}(X=1, Y=1) + \mathbb{E}(X=0, Y=1) = 0.22$ , aber  $\mathbb{E}(Y=1|X=0) = 0.07$  und  $\mathbb{E}(Y=1|X=1) = 0.15$ . Sie können sich leicht merken, dass für abhängige ZV nie, und für unabhängige ZV immer *mean independence* bevorliegt (der umgekehrte Fall gilt jedoch nicht!).

Wir werden später im Kontext der Regressionsanalyse noch sehr häufig auf dieses Gesetz und die Konzepte der marginalen und bedingten Verteilungen zurückkommen.



## Teil III

# Grundlagen der Regressionsanalyse in R



# Bibliography

- Aleskerov, F., Ersel, H., and Piontковski, D. (2011). *Linear Algebra for Economists*. Springer. ISBN: 978-3-642-20569-9, DOI: 10.1007/978-3-642-20570-5.
- Anscombe, F. J. (1973). Graphs in statistical analysis. *The American Statistician*, 27:17–21. DOI 10.2307/2682899.
- Arel-Bundock, V. (2019). *WDI: World Development Indicators (World Bank)*. R package version 2.6.0.
- Arel-Bundock, V., Enevoldsen, N., and Yetman, C. (2018). countrycode: An r package to convert country names and country codes. *Journal of Open Source Software*, 3(28):848.
- Bengtsson, H. (2019). *R.utils: Various Programming Utilities*. R package version 2.9.0.
- Chatterjee, S. and Firat, A. (2007). Generating data with identical statistics but dissimilar graphics. *The American Statistician*, 61(3):248–254. DOI 10.1198/000313007X220057.
- Clauset, A., Shalizi, C. R., and Newman, M. E. J. (2009). Power-Law Distributions in Empirical Data. *SIAM Review*, 51(4):661–703.
- Clementi, F., Gallegati, M., Gianmoena, L., Landini, S., and Stiglitz, J. E. (2019). Mis-measurement of inequality: a critical reflection and new insights. *Journal of Economic Interaction and Coordination*, 14(4):891–921.
- Delignette-Muller, M. L. and Dutang, C. (2015). fitdistrplus: An R package for fitting distributions. *Journal of Statistical Software*, 64(4):1–34.
- Dowle, M. and Srinivasan, A. (2019). *data.table: Extension of ‘data.frame’*. R package version 1.12.2.
- Friendly, M., Fox, J., and Chalmers, P. (2019). *matlib: Matrix Functions for Teaching and Learning Linear Algebra and Multivariate Statistics*. R package version 0.9.2.
- Gräßner, C., Heimberger, P., Kapeller, J., and Schütz, B. (2020). Is Europe disintegrating? Macroeconomic divergence, structural polarization, trade and fragility. *Cambridge Journal of Economics*, 44(3):647–669. DOI 10.1093/cje/bez059.
- Gräßner, C. (2019). *icaeDesign: Corporate design-like functions for the ICAE*. R package version 0.1.3.
- Hanson, G. H. (2012). The rise of middle kingdoms: Emerging economies in global trade. *Journal of Economic Perspectives*, 26(2):41–64.
- Herndon, T., Ash, M., and Pollin, R. (2013). Does high public debt consistently stifle economic growth? A critique of Reinhart and Rogoff. *Cambridge Journal of Economics*, 38(2):257–279. DOI 10.1093/cje/bet075.
- Holtz, Y. and Healy, C. (2020). From data to viz. <https://www.data-to-viz.com/>. Accessed: 2020-09-08.
- Kapeller, J., Gräßner, C., and Heimberger, P. (2019). *Wirtschaftliche Polarisierung in Europa: Ursachen und Handlungsoptionen*. Friedrich-Ebert Stiftung, Bonn. ISBN 978-3-96250-376-5.

- Kassambara, A. (2019). *ggnpubr: 'ggplot2' Based Publication Ready Plots*. R package version 0.2.1.
- Kleiber, C. and Zeileis, A. (2008). *Applied Econometrics with R*. Springer-Verlag, New York. ISBN 978-0-387-77316-2.
- Komsta, L. and Novomestky, F. (2015). *moments: Moments, cumulants, skewness, kurtosis and related tests*. R package version 0.14.
- Krämer, W. (2015). *So lügt man mit Statistik*. Campus Verlag, Frankfurt und New York. ISBN: 978-3593504599.
- Kuznets, S. (1934). *National Income, 1929-1932*. U.S. Government Printing Office, Washington D.C. Out of print; obtained from <https://fraser.stlouisfed.org/title/971>.
- Lepenies, P. (2016). *The Power of a Single Number*. Columbia University Press, New York, NY.
- Matejka, J. and Fitzmaurice, G. (2017). Same stats, different graphs: Generating datasets with varied appearance and identical statistics through simulated annealing. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 1290–1294, New York, NY. ACM. DOI 10.1145/3025453.3025912.
- Meschiari, S. (2015). *latex2exp: Use LaTeX Expressions in Plots*. R package version 0.4.0.
- Müller, K. (2017). *here: A Simpler Way to Find Your Files*. R package version 0.1.
- Nash, J. C. and Varadhan, R. (2011). Unifying optimization algorithms to aid software system users: optimx for R. *Journal of Statistical Software*, 43(9):1–14.
- OECD (2019). Share of employed who are managers, by sex. Accessed: November 15, 2019; Location: Social Protection and Well-being/Gender/Employment.
- Pebesma, E., Mailund, T., and Hiebert, J. (2016). Measurement units in R. *R Journal*, 8(2):486–494.
- R Core Team (2018). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.
- Schwabish, J. A. (2014). An economist’s guide to visualizing data. *Journal of Economic Perspectives*, 28(1):209–234. DOI 10.1257/jep.28.1.209.
- Slowikowski, K. (2019). *ggrepel: Automatically Position Non-Overlapping Text Labels with 'ggplot2'*. R package version 0.8.1.
- The Growth Lab at Harvard University (2019). International Trade Data (SITC, Rev. 2). File: country\_sitcproduct2digit\_year.
- Wainwright, K. and Chiang, A. (2005). *Fundamental Methods of Mathematical Economics*. McGraw-Hill. ISBN: 978-0071238236.
- Wickham, H. (2010). A layered grammar of graphics. *Journal of Computational and Graphical Statistics*, 19(1):3–28. DOI 10.1198/jcgs.2009.07098.
- Wickham, H. (2014). Tidy data. *The Journal of Statistical Software*, 59. DOI 10.18637/jss.v059.i10.
- Wickham, H. (2016). *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York.
- Wickham, H. (2018). *scales: Scale Functions for Visualization*. R package version 1.0.0.
- Wickham, H. (2019). *Advanced R*. Chapman and Hall/CRC, Boca Raton, Florida, 2nd edition. ISBN 978-0815384571.

- Wickham, H. and Bryan, J. (2019). *Advanced R*. O'Reilly Media, Sebastopol, CA, 2nd edition. ISBN 978-1491910597.
- Wickham, H., François, R., Henry, L., and Müller, K. (2019). *dplyr: A Grammar of Data Manipulation*. R package version 0.8.2.
- Wickham, H. and Henry, L. (2019). *tidy: Tidy Messy Data*. R package version 1.0.0.
- Wickham, H. and Miller, E. (2019). *haven: Import and Export 'SPSS', 'Stata' and 'SAS' Files*. R package version 2.1.0.
- Wilkinson, L. (1999). *The Grammar of Graphics*. Springer, New York. ISBN 978-1-4757-3100-2, DOI 10.1007/978-1-4757-3100-2.
- Yang, J., Heinrich, T., Winkler, J., Lafond, F., Koutroumpis, P., and Farmer, J. (2019). Measuring productivity dispersion: a parametric approach using the levy alpha-stable distribution. *INET Oxford Working Paper*, 2019-14.
- Zeileis, A. (2014). *ineq: Measuring Inequality, Concentration, and Poverty*. R package version 0.2-13.