

# DataScience1

## Grundlagen der Prognosemodellierung

Sebastian Sauer

2022-06-18 18:15:05

- Zu diesem Buch
  - 0.1 Was Sie hier lernen und wozu das gut ist
  - 0.2 Zitation
  - 0.3 Technische Details
- 1 Hinweise
  - 1.1 Lernziele
  - 1.2 Voraussetzungen
  - 1.3 Lernhilfen
    - 1.3.1 Software
    - 1.3.2 Videos
    - 1.3.3 Online-Zusammenarbeit
  - 1.4 Modulzeitplan
  - 1.5 Literatur
  - 1.6 FAQ
- 2 Prüfung
  - 2.1 Prüfungleistung
  - 2.2 tl;dr: Zusammenfassung
  - 2.3 Vorhersage
  - 2.4 Hauptziel: Genaue Prognose
  - 2.5 Zum Aufbau Ihrer Prognosedatei im CSV-Format
  - 2.6 Einzureichende Dateien
  - 2.7 Gliederung Ihrer Analyse
    - 2.7.1 Abschnitt Forschungsfrage und Hintergrund
    - 2.7.2 Vorbereitung
    - 2.7.3 Explorative Datenanalyse
    - 2.7.4 Modellierung
    - 2.7.5 Vorhersagen
  - 2.8 Tipps
    - 2.8.1 Tipps für eine gute Prognose
    - 2.8.2 Tipps zur Datenverarbeitung
    - 2.8.3 Tipps zum Aufbau des Analyseskripts
    - 2.8.4 Sonstiges
  - 2.9 Bewertung
    - 2.9.1 Kriterien
    - 2.9.2 Kennzahl der Modellgüte
    - 2.9.3 Notenstufen
    - 2.9.4 Bewertungsprozess
  - 2.10 Hinweise
  - 2.11 Formalia
  - 2.12 Ich brauche Hilfe!
    - 2.12.1 Wo finde ich Beispiele und Vorlagen?
    - 2.12.2 Materialsammlung
    - 2.12.3 Videos
  - 2.13 Plagiatskontrolle
- I Themen
- 3 Statistisches Lernen
  - 3.1 Lernsteuerung
    - 3.1.1 Vorbereitung
    - 3.1.2 Lernziele
    - 3.1.3 Literatur
    - 3.1.4 Hinweise
  - 3.2 Was ist Data Science?
  - 3.3 Was ist Machine Learning?
    - 3.3.1 Rule-based
    - 3.3.2 Data-based
  - 3.4 Modell vs. Algorithmus
    - 3.4.1 Modell
    - 3.4.2 Beispiel für einen ML-Algorithmus

- 3.5 Taxonomie
  - 3.5.1 Geleitetes Lernen
  - 3.5.2 Ungeleitetes Lernen
- 3.6 Ziele des ML
- 3.7 Über- vs. Unteranpassung
- 3.8 No free lunch
- 3.9 Bias-Varianz-Abwägung
- 3.10 Aufgaben
- 3.11 Vertiefung
- 4 R, zweiter Blick
  - 4.1 Lernsteuerung
    - 4.1.1 Vorbereitung
    - 4.1.2 Lernziele
    - 4.1.3 Literatur
  - 4.2 Objekttypen in R
    - 4.2.1 Überblick
    - 4.2.2 Taxonomie
    - 4.2.3 Indizieren
    - 4.2.4 Weiterführende Hinweise
    - 4.2.5 Indizieren mit dem Tidyverse
  - 4.3 Datensätze von lang nach breit umformatieren
  - 4.4 Funktionen
  - 4.5 Wiederholungen programmieren
    - 4.5.1 `across()`
    - 4.5.2 `map()`
    - 4.5.3 Weiterführende Hinweise
  - 4.6 Listenspalten
    - 4.6.1 Wozu Listenspalten?
    - 4.6.2 Beispiele für Listenspalten
    - 4.6.3 Programmieren mit dem Tidyverse
  - 4.7 R ist schwierig
  - 4.8 Aufgaben
  - 4.9 Vertiefung
- 5 tidymodels
  - 5.1 Lernsteuerung
    - 5.1.1 Vorbereitung
    - 5.1.2 Lernziele
    - 5.1.3 Literatur
  - 5.2 Daten
  - 5.3 Train- vs Test-Datensatz aufteilen
  - 5.4 Grundlagen der Modellierung mit tidymodels
    - 5.4.1 Modelle spezifizieren
    - 5.4.2 Modelle berechnen
    - 5.4.3 Vorhersagen
    - 5.4.4 Vorhersagen im Train-Datensatz
    - 5.4.5 Modellkoeffizienten im Train-Datensatz
    - 5.4.6 Parsnip RStudio add-in
  - 5.5 Workflows
    - 5.5.1 Konzept des Workflows in Tidymodels
    - 5.5.2 Einfaches Beispiel
    - 5.5.3 Vorhersage mit einem Workflow
    - 5.5.4 Modellgüte
    - 5.5.5 Vorhersage von Hand
  - 5.6 Rezepte zur Vorverarbeitung
    - 5.6.1 Was ist Rezept und wozu ist es gut?
    - 5.6.2 Workflows mit Rezepten
    - 5.6.3 Spaltenrollen
    - 5.6.4 Fazit
  - 5.7 Aufgaben
- 6 kNN
  - 6.1 Lernsteuerung
    - 6.1.1 Lernziele
    - 6.1.2 Literatur
  - 6.2 Überblick
  - 6.3 Intuitive Erklärung
  - 6.4 Krebsdiagnostik
  - 6.5 Berechnung der Nähe

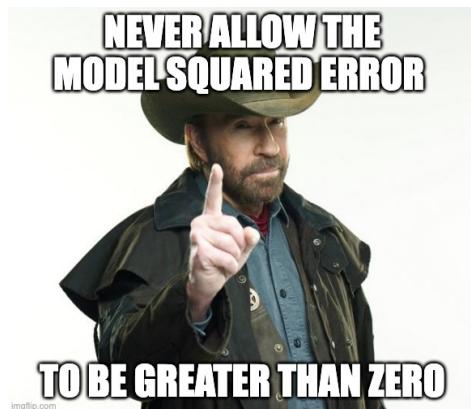
- 6.6 kNN mit Tidymodels
  - 6.6.1 Analog zu Timbers et al.
  - 6.6.2 Rezept definieren
  - 6.6.3 Modell definieren
  - 6.6.4 Workflow definieren
  - 6.6.5 Vorhersagen
- 6.7 Mit Train-Test-Aufteilung
  - 6.7.1 Rezept definieren
  - 6.7.2 Modell definieren
  - 6.7.3 Workflow definieren
  - 6.7.4 Vorhersagen
  - 6.7.5 Modellgüte
  - 6.7.6 Visualisierung
- 6.8 Kennzahlen der Klassifikation
- 6.9 Krebstest-Beispiel
- 6.10 Aufgaben
- 7 Resampling und Tuning
  - 7.1 Lernsteuerung
    - 7.1.1 Vorbereitung
    - 7.1.2 Lernziele
    - 7.1.3 Literatur
  - 7.2 Überblick
  - 7.3 tidymodels
    - 7.3.1 Datensatz aufteilen
    - 7.3.2 Rezept, Modell und Workflow definieren
  - 7.4 Resampling
  - 7.5 Illustration des Resampling
    - 7.5.1 Einfache v-fache Kreuzvalidierung
    - 7.5.2 Wiederholte Kreuzvalidierung
    - 7.5.3 Resampling passiert im Train-Sample
    - 7.5.4 Andere Illustrationen
  - 7.6 Gesetz der großen Zahl
  - 7.7 Über- und Unteranpassung an einem Beispiel
  - 7.8 CV in tidymodels
    - 7.8.1 CV definieren
    - 7.8.2 Resamples fitten
  - 7.9 Tuning
    - 7.9.1 Tuning auszeichnen
    - 7.9.2 Grid Search vs. Iterative Search
  - 7.10 Tuning mit Tidymodels
    - 7.10.1 Datenabhängige Tuningparameter
    - 7.10.2 Modelle mit Tuning berechnen
    - 7.10.3 Vorhersage im Test-Sample
  - 7.11 Aufgaben
  - 7.12 Vertiefung
- 8 Logistische Regression
  - 8.1 Lernsteuerung
    - 8.1.1 Vorbereitung
    - 8.1.2 Lernziele
    - 8.1.3 Literatur
  - 8.2 Intuitive Erklärung
  - 8.3 Profil
  - 8.4 Warum nicht die lineare Regression verwenden?
    - 8.4.1 Lineare Modelle running wild
    - 8.4.2 Wir müssen die Regressionsgerade umbiegen
    - 8.4.3 Verallgemeinerte lineare Modelle zur Rettung
  - 8.5 Der Logit-Link
  - 8.6 Aber warum?
    - 8.6.1 tidymodels, m83
  - 8.7 lm83, glm
  - 8.8 m83, tidymodels
    - 8.8.1 Wahrscheinlichkeit in Odds
    - 8.8.2 Von Odds zu Log-Odds
  - 8.9 Inverser Logit
  - 8.10 Vom Logit zur Klasse
    - 8.10.1 Grenzwert wechseln
  - 8.11 Logit und Inverser Logit

- 8.11.1 Logit
- 8.11.2 Inv-Logit
- 8.12 Logistische Regression im Überblick
  - 8.12.1 Die Koeffizienten sind schwer zu interpretieren
  - 8.12.2 Logits vs. Wahrscheinlichkeiten
- 8.13 Aufgaben
- 8.14 Vertiefung
- 9 Entscheidungsbäume
  - 9.1 Vorbereitung
  - 9.2 Lernsteuerung
  - 9.3 Anatomie eines Baumes
  - 9.4 Bäume als Regelmassen rekursiver Partitionierung
  - 9.5 Klassifikation
  - 9.6 Gini als Optimierungskriterium
  - 9.7 Metrische Prädiktoren
  - 9.8 Regressionsbäume
  - 9.9 Baum beschneiden
  - 9.10 Das Rechteck schlägt zurück
  - 9.11 Tidymodels
    - 9.11.1 Initiale Datenaufteilung
    - 9.11.2 Kreuzvalidierung definieren
    - 9.11.3 Rezept definieren
    - 9.11.4 Modell definieren
    - 9.11.5 Workflow definieren
    - 9.11.6 Modell tunen und berechnen
    - 9.11.7 Modellgüte evaluieren
    - 9.11.8 Bestes Modell auswählen
    - 9.11.9 Final Fit
    - 9.11.10 Nur zum Spaß: Vergleich mit linearem Modell
- 10 Ensemble Lerner
  - 10.1 Lernsteuerung
    - 10.1.1 Lernziele
    - 10.1.2 Literatur
    - 10.1.3 Hinweise
  - 10.2 Vorbereitung
  - 10.3 Hinweise zur Literatur
  - 10.4 Wir brauchen einen Wald
  - 10.5 Was ist ein Ensemble-Lerner?
  - 10.6 Bagging
    - 10.6.1 Bootstrapping
  - 10.7 Bagging-Algorithmus
    - 10.7.1 Variablenrelevanz
    - 10.7.2 Out of Bag Vorhersagen
  - 10.8 Random Forests
  - 10.9 Boosting
    - 10.9.1 AdaBoost
    - 10.9.2 XGBoost
  - 10.10 Tidymodels
    - 10.10.1 Datensatz Churn
    - 10.10.2 Data Splitting und CV
    - 10.10.3 Feature Engineering
    - 10.10.4 Modelle
    - 10.10.5 Workflows
    - 10.10.6 Modelle berechnen mit Tuning, einzeln
    - 10.10.7 Workflow-Set tunen
    - 10.10.8 Ergebnisse im Train-Set
    - 10.10.9 Bestes Modell
    - 10.10.10 Finalisieren
    - 10.10.11 Last Fit
    - 10.10.12 Variablenrelevanz
    - 10.10.13 ROC-Curve
  - 10.11 Aufgaben
  - 10.12 Aufgaben
  - 10.13 Vertiefung
- 11 Regularisierte Modelle
  - 11.1 Lernsteuerung
    - 11.1.1 Lernziele

- 11.1.2 Literatur
- 11.1.3 Hinweise
- 11.2 Vorbereitung
- 11.3 Regularisierung
  - 11.3.1 Was ist Regularisierung?
  - 11.3.2 Ähnliche Verfahren
  - 11.3.3 Normale Regression (OLS)
- 11.4 Ridge Regression, L2
  - 11.4.1 Strafterm
  - 11.4.2 Standardisierung
- 11.5 Lasso, L1
  - 11.5.1 Strafterm
  - 11.5.2 Variablenelektion
- 11.6 L1 vs. L2
  - 11.6.1 Wer ist stärker?
  - 11.6.2 Elastic Net als Kompromiss
- 11.7 Aufgaben
- 12 Kaggle
  - 12.1 Vorbereitung
    - 12.1.1 Lernsteuerung
    - 12.1.2 Lernziele
    - 12.1.3 Hinweise
    - 12.1.4 R-Pakete
  - 12.2 Was ist Kaggle?
  - 12.3 Fallstudie TMDB
    - 12.3.1 Aufgabe
    - 12.3.2 Hinweise
    - 12.3.3 Daten
    - 12.3.4 Train-Set verschlanken
    - 12.3.5 Datensatz kennenlernen
    - 12.3.6 Fehlende Werte prüfen
  - 12.4 Rezept
    - 12.4.1 Rezept definieren
    - 12.4.2 Check das Rezept
    - 12.4.3 Check Test-Sample
  - 12.5 Kreuzvalidierung
  - 12.6 Modelle
    - 12.6.1 Baum
    - 12.6.2 Random Forest
    - 12.6.3 XGBoost
    - 12.6.4 LM
  - 12.7 Workflows
  - 12.8 Fitten und tunen
  - 12.9 Finalisieren
    - 12.9.1 Welcher Algorithmus schneidet am besten ab?
    - 12.9.2 Final Fit
  - 12.10 Submission
    - 12.10.1 Submission vorbereiten
    - 12.10.2 Kaggle Score
  - 12.11 Aufgaben
  - 12.12 Vertiefung
- 13 Der rote Faden
  - 13.0.1 Lernziele
  - 13.0.2 Literatur
  - 13.1 Aussichtspunkt 1: Blick vom hohen Berg
  - 13.2 Aussichtspunkt 2: Blick in den Hof der Handwerker
    - 13.2.1 Ein maximale einfaches Werkstück mit Tidymodels
    - 13.2.2 Ein immer noch recht einfaches Werkstück mit Tidymodels
  - 13.3 Aussichtspunkt 3: Der Nebelberg (Quiz)
  - 13.4 Aussichtspunkt 4: Der Exerzitien-Park
  - 13.5 Aussichtspunkt 5: In der Bibliothek
  - 13.6 Krafttraining
  - 13.7 Aufgaben
  - 13.8 Vertiefung
- 14 Fallstudien
  - 14.0.1 Lernziele
  - 14.0.2 Literatur

- 14.1 Fallstudien zur explorativen Datenanalyse
- 14.2 Fallstudien zu linearen Modellen
- 14.3 Fallstudien zum maschinellen Lernen mit Tidymodels
- 14.4 Aufgaben
- 14.5 Vertiefung
- References

## Zu diesem Buch



(<https://imgflip.com/i/689g8g>)

from Imgflip Meme Generator (<https://imgflip.com/memegenerator>)

## 0.1 Was Sie hier lernen und wozu das gut ist

Alle Welt spricht von Big Data, aber ohne die Analyse sind die großen Daten nur großes Rauschen. Was letztlich interessiert, sind die Erkenntnisse, die Einblicke, nicht die Daten an sich. Dabei ist es egal, ob die Daten groß oder klein sind. Natürlich erlauben die heutigen Datenmengen im Verbund mit leistungsfähigen Rechnern und neuen Analysemethoden ein Verständnis, das vor Kurzem noch nicht möglich war. Und wir stehen erst am Anfang dieser Entwicklung. Vielleicht handelt es sich bei diesem Feld um eines der dynamischsten Fachgebiete der heutigen Zeit. Sie sind dabei: Sie lernen einiges Handwerkszeugs des “Datenwissenschaftlers”. Wir konzentrieren uns auf das vielleicht bekannteste Teilgebiet: Ereignisse vorhersagen auf Basis von hoch strukturierten Daten und geeigneter Algorithmen und Verfahren. Nach diesem Kurs sollten Sie in der Lage sein, typisches Gebabbel des Fachgebiet mit Lässigkeit mitzumachen. Ach ja, und mit einem Erfolg Vorhersagemodelle entwickeln.

## 0.2 Zitation

Nutzen Sie diese DOI, um dieses Buch zu zitieren: DOI [10.5281/zenodo.6602660](https://doi.org/10.5281/zenodo.6602660) (<https://zenodo.org/badge/latestdoi/461950782>)

## 0.3 Technische Details

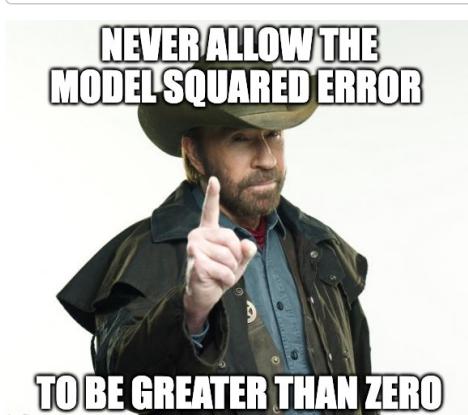
- Diese Version des Buches wurde erstellt am: 2022-06-18 18:15:06
- Die URL zu diesem Buch lautet <https://sebastiansauer.github.io/datascience1/> (<https://sebastiansauer.github.io/datascience1/>) und ist bei GitHub Pages (<https://pages.github.com/>) gehostet.
- Lesen Sie sich die folgenden Informationen bitte gut durch: Hinweise (<https://sebastiansauer.github.io/fopra/Interna/Hinweise.html>)
- Den Quellcode finden Sie in diesem Github-Repo (<https://github.com/sebastiansauer/datascience1>).
- Sie haben Feedback, Fehlerhinweise oder Wünsche zur Weiterentwicklung? Am besten stellen Sie hier (<https://github.com/sebastiansauer/datascience1/issues>) einen Issue ein.
- Dieses Projekt steht unter der MIT-Lizenz (<https://github.com/sebastiansauer/datascience1/blob/main/LICENSE>).
- Dieses Buch wurde in RStudio (<http://www.rstudio.com/ide/>) mit Hilfe von bookdown (<http://bookdown.org/>) geschrieben.
- Diese Version des Buches wurde mit der R-Version R version 4.2.0 (2022-04-22) und den folgenden Paketen erstellt:

package	version	source
bookdown	0.26	CRAN (R 4.2.0)
broom	0.8.0	CRAN (R 4.2.0)
corrr	NA	NA
dials	0.1.1	CRAN (R 4.2.0)
downlit	0.4.0	CRAN (R 4.2.0)
dplyr	1.0.9	CRAN (R 4.2.0)

package	version	source
ggplot2	3.3.6	CRAN (R 4.2.0)
glmnet	NA	NA
infer	1.0.2	CRAN (R 4.2.0)
ISLR	NA	NA
kknn	1.3.1	CRAN (R 4.2.0)
klaR	NA	NA
MASS	7.3-56	CRAN (R 4.2.0)
modeldata	0.1.1	CRAN (R 4.2.0)
parsnip	0.2.1	CRAN (R 4.2.0)
patchwork	1.1.1	CRAN (R 4.2.0)
purrr	0.3.4	CRAN (R 4.2.0)
randomForest	NA	NA
ranger	0.13.1	CRAN (R 4.2.0)
readr	2.1.2	CRAN (R 4.2.0)
rsample	0.1.1	CRAN (R 4.2.0)
rstatix	NA	NA
tibble	3.1.7	CRAN (R 4.2.0)
tidymodels	0.2.0	CRAN (R 4.2.0)
tidyverse	1.3.1	CRAN (R 4.2.0)
tune	0.2.0	CRAN (R 4.2.0)
vip	0.3.2	CRAN (R 4.2.0)
workflows	0.2.6	CRAN (R 4.2.0)
workflowsets	0.2.1	CRAN (R 4.2.0)
xgboost	1.6.0.1	CRAN (R 4.2.0)
yardstick	1.0.0	CRAN (R 4.2.0)

# 1 Hinweise

```
knitr::opts_chunk$set(cache = FALSE)
```



(<https://imgflip.com/i/689g8g>)

from Imgflip Meme Generator (<https://imgflip.com/memegenerator>)

## 1.1 Lernziele

Nach diesem Kurs sollten Sie

- grundlegende Konzepte des statistischen Lernens verstehen und mit R anwenden können
- gängige Prognose-Algorithmen kennen, in Grundzügen verstehen und mit R anwenden können
- die Güte und Grenze von Prognosemodellen einschätzen können

## 1.2 Voraussetzungen

Um von diesem Kurs am besten zu profitieren, sollten Sie folgendes Wissen mitbringen:

- grundlegende Kenntnisse im Umgang mit R, möglichst auch mit dem tidyverse
- grundlegende Kenntnisse der deskriptiven Statistik
- grundlegende Kenntnis der Regressionsanalyse

## 1.3 Lernhilfen

### 1.3.1 Software

- Installieren Sie R und seine Freunde (<https://data-se.netlify.app/2021/11/30/installation-von-r-und-seiner-freunde/>).
- Installieren Sie die folgende R-Pakete:
  - tidyverse
  - tidymodels
  - weitere Pakete werden im Unterricht bekannt gegeben (es schadet aber nichts, jetzt schon Pakete nach eigenem Ermessen zu installieren)
- R Syntax aus dem Unterricht (<https://github.com/sebastiansauer/Lehre>) findet sich im Github-Repo bzw. Ordner zum jeweiligen Semester.

### 1.3.2 Videos

- Playlist zu den Themen (<https://youtube.com/playlist?list=PLRR4REmBgpIGv1e4hZ8aslL3qVBe5LcKp>)
- Auf dem YouTube-Kanal des Autors (<https://www.youtube.com/channel/UChvdtj8maE7g-SOCh4aDB9g>) finden sich eine Reihe von Videos mit Bezug zum Inhalt dieses Buches.

### 1.3.3 Online-Zusammenarbeit

Hier finden Sie einige Werkzeuge, die das Online-Zusammenarbeiten vereinfachen:

- Frag-Jetzt-Raum zum anonymen Fragen stellen während des Unterrichts (<https://frag.jetzt/home>). Der Keycode wird Ihnen vom Dozenten bereitgestellt.
- Padlet (<https://de.padlet.com/>) zum einfachen (und anonymen) Hochladen von Arbeitsergebnissen der Studenten im Unterricht. Wir nutzen es als eine Art Pinwand zum Sammeln von Arbeitsbeiträgen. Die Zugangsdaten stellt Ihnen der Dozent bereit.

## 1.4 Modulzeitplan

KW	Terminhinweise	Kurswoche	Titel_Link
11	Lehrbeginn am Dienstag	1	Statistisches Lernen ( <a href="https://sebastiansauer.github.io/datascience1/statistisches-lernen.html">https://sebastiansauer.github.io/datascience1/statistisches-lernen.html</a> )
12	NA	2	R, zweiter Blick ( <a href="https://sebastiansauer.github.io/datascience1/r-zweiter-blick.html">https://sebastiansauer.github.io/datascience1/r-zweiter-blick.html</a> )
13	NA	3	R, zweiter Blick 2 ( <a href="https://sebastiansauer.github.io/datascience1/r-zweiter-blick-2.html">https://sebastiansauer.github.io/datascience1/r-zweiter-blick-2.html</a> )
14	NA	4	tidymodels ( <a href="https://sebastiansauer.github.io/datascience1/tidymodels.html">https://sebastiansauer.github.io/datascience1/tidymodels.html</a> )
15	NA	5	kNN ( <a href="https://sebastiansauer.github.io/datascience1/knn.html">https://sebastiansauer.github.io/datascience1/knn.html</a> )
16	NA	6	Wiederholung ( <a href="https://sebastiansauer.github.io/datascience1/wiederholung.html">https://sebastiansauer.github.io/datascience1/wiederholung.html</a> )
17	NA	7	Resampling und Tuning ( <a href="https://sebastiansauer.github.io/datascience1/resampling-und-tuning.html">https://sebastiansauer.github.io/datascience1/resampling-und-tuning.html</a> )
18	NA	8	Logistische Regression ( <a href="https://sebastiansauer.github.io/datascience1/logistische-regression.html">https://sebastiansauer.github.io/datascience1/logistische-regression.html</a> )
19	NA	9	Entscheidungsbäume ( <a href="https://sebastiansauer.github.io/datascience1/entscheidungsbaeume.html">https://sebastiansauer.github.io/datascience1/entscheidungsbaeume.html</a> )
20	NA	10	Ensemble-Lerner ( <a href="https://sebastiansauer.github.io/datascience1/ensemble-lerner.html">https://sebastiansauer.github.io/datascience1/ensemble-lerner.html</a> )
21	nächste Woche ist Projektwoche	11	Regularisierung ( <a href="https://sebastiansauer.github.io/datascience1/regularisierung.html">https://sebastiansauer.github.io/datascience1/regularisierung.html</a> )

KW	Terminhinweise	Kurswoche	Titel_Link
22	Projektwoche, kein regulärer Unterricht	11	Blockwoche: kein Unterricht in dieser Woche ( <a href="https://sebastiansauer.github.io/datascience1/blockwoche:-kein-unterricht-in-dieser-woche.html">https://sebastiansauer.github.io/datascience1/blockwoche:-kein-unterricht-in-dieser-woche.html</a> )
23	Pfingsten, keine Vorlesung. Die Übung findet NUR ONLINE statt.	12	Kaggle ( <a href="https://sebastiansauer.github.io/datascience1/kaggle.html">https://sebastiansauer.github.io/datascience1/kaggle.html</a> )
24	Fronleichnam; die Übung WIRD VERSCHOBEN (Termin im Juli)	13	Der rote Faden ( <a href="https://sebastiansauer.github.io/datascience1/der-rote-faden.html">https://sebastiansauer.github.io/datascience1/der-rote-faden.html</a> )
25	vorletzte Unterrichtswoche	14	Fallstudien ( <a href="https://sebastiansauer.github.io/datascience1/fallstudien.html">https://sebastiansauer.github.io/datascience1/fallstudien.html</a> )
26	Letzte Unterrichtswoche	15	Dimensionsreduktion ( <a href="https://sebastiansauer.github.io/datascience1/dimensionsreduktion.html">https://sebastiansauer.github.io/datascience1/dimensionsreduktion.html</a> )

## 1.5 Literatur

Zentrale Kursliteratur für die theoretischen Konzepte ist Rhys (2020). Bitte prüfen Sie, ob das Buch in einer Bibliothek verfügbar ist. Die praktische Umsetzung in R basiert auf Silge and Kuhn (2022) (dem “Tidymodels-Konzept”); das Buch ist frei online verfügbar.

Eine gute Ergänzung ist das Lehrbuch von Timbers, Campbell, and Lee (2022), welches grundlegende Data-Science-Konzepte erläutert und mit tidymodels umsetzt.

James et al. (2021) haben ein weithin renommiertes und sehr bekanntes Buch verfasst. Es ist allerdings etwas anspruchsvoller als Rhys (2020), daher steht es nicht im Fokus dieses Kurses, aber einige Schwenker zu Inhalten von James et al. (2021) gibt es. Schauen Sie mal rein, das Buch ist gut!

In einigen Punkten ist weiterhin Sauer (2019) hilfreich; das Buch ist über SpringerLink in Ihrer Hochschul-Bibliothek verfügbar. Eine gute Ergänzung ist das “Lab-Buch” von Hvitfeldt (2022). In dem Buch wird das Lehrbuch James et al. (2021) in Tidymodels-Konzepte übersetzt; durchaus nett!

## 1.6 FAQ

- *Folien*
  - Frage: Gibt es ein Folienskript?
  - Antwort: Wo es einfache, gute Literatur gibt, gibt es kein Skript. Wo es keine gute oder keine einfach zugängliche Literatur gibt, dort gibt es ein Skript.
- *Englisch*
  - Ist die Literatur auf Englisch?
  - Ja. Allerdings ist die Literatur gut zugänglich. Das Englisch ist nicht schwer. Bedenken Sie: Englisch ist die lingua franca in Wissenschaft und Wirtschaft. Ein solides Verständnis englischer (geschriebener) Sprache ist für eine gute Ausbildung unerlässlich. Zu dem sollte die Kursliteratur fachlich passende und gute Bücher umfassen; oft sind das englische Titel.
- *Anstrengend*
  - Ist der Kurs sehr anstrengend, aufwändig?
  - Der Kurs hat ein mittleres Anspruchsniveau.
- *Mathe*
  - Muss man ein Mathe-Crack sein, um eine gute Note zu erreichen?
  - Nein. Mathe steht nicht im Vordergrund. Schauen Sie sich die Literatur an, sie werden wenig Mathe darin finden.
- *Prüfungsliteratur*
  - Welche Literatur ist prüfungsrelevant?
  - Die Prüfung ist angewandt, z.B. ein Prognosewettbewerb. Es wird keine Klausur geben, in der reines Wissen abgefragt wird.
- *Nur R?*
  - Wird nur R in dem Kurs gelehrt? Andere Programmiersprachen sind doch auch wichtig.
  - In der Datenanalyse gibt es zwei zentrale Programmiersprachen, R und Python. Beide sind gut und beide werden viel verwendet. In einer Grundausbildung sollte man sich auf eine Sprache begrenzen, da sonst den Sprachen zu viel Zeit eingeräumt werden muss. Wichtiger als eine zweite Programmiersprache zu lernen, mit der man nicht viel mehr kann als mit der ersten, ist es, die Inhalte des Fachs zu lernen.

## 2 Prüfung

### 2.1 Prüfungsteil

Die Prüfung besteht aus einem Prognosewettbewerb.

### 2.2 tl;dr: Zusammenfassung

Vorhersagen sind eine praktische Sache, zumindest wenn Sie stimmen. Wenn Sie den DAX-Stand von morgen genau vorhersagen können, rufen Sie mich bitte sofort an. Genau das ist Ihre Aufgabe in dieser Prüfungsleistung: Sie sollen Werte vorhersagen.

Etwas konkreter: Stellen Sie sich ein paar Studenten vor. Von allen wissen Sie, wie lange die Person für die Statistikklausur gelernt hat. Außerdem wissen Sie die Motivation jeder Person und vielleicht noch ein paar noten-relevante Infos. Und Sie wissen die Note jeder Person in der Statistikklausur. Auf dieser Basis fragt sie ein Student (Alois), der im kommenden Semester die Prüfung in Statistik schreiben muss will: "Sag mal, wenn ich 100 Stunden lerne und so mittel motiviert bin (bestenfalls), welche Note kann ich dann erwarten?". Mit Hilfe Ihrer Analyse können Sie diese Frage (und andere) beantworten. Natürlich könnten Sie es sich leicht machen und antworten: "Mei, der Notendurchschnitt war beim letzten Mal 2.7. Also ist 2.7 kein ganz doofer Tipp für deine Note." Ja, das ist keine doofe Antwort, aber man genauere Prognose machen, wenn man es geschickt anstellt. Da hilft Ihnen die Statistik (doch, wirklich).

Kurz gesagt gehen Sie so vor: Importieren Sie die Daten in R, starten Sie die nötigen R-Pakete und schauen Sie sich die Daten unter verschiedenen Blickwinkeln an. Dann nehmen Sie die vielversprechendsten Prädiktoren in ein Regressionsmodell und schauen sich an, wie gut die Vorhersage ist. Wiederholen Sie das ein paar Mal, bis Sie ein Modell haben, das Sie brauchbar finden. Mit diesem Modell sagen Sie dann die Noten der neuen Studis (Alois und Co.) vorher. Je genauer Ihre Vorhersage, desto besser ist Ihr Prüfungsergebnis.

## 2.3 Vorhersage

Neben der erklärenden, rückwärtsgerichteten Modellierung spielt insbesondere in der Praxis die *vorhersageorientierte* Modellierung eine wichtige Rolle: Ziel ist es, bei gegebenen, neuen Beobachtungen die noch unbekannten Werte der Zielvariablen  $y$  vorherzusagen, z.B. für neue Kunden auf Basis von soziodemographischen Daten den *Kundenwert* – möglichst genau – zu prognostizieren. Dies geschieht auf Basis der vorhandenen Daten der Bestandskunden, d.h. inklusive des für diese Kunden bekannten Kundenwertes.

Ihnen werden *zwei Teildatenmengen* zur Verfügung gestellt: Zum einen gibt es die Trainingsdaten (auch *Lerndaten* genannt) und zum anderen gibt es Anwendungsdaten (auch *Testdaten* genannt), auf die man das Modell anwendet.

1. Bei den Trainingsdaten (Train-Sample) liegen sowohl die erklärenden Variablen  $\mathbf{x} = (x_1, x_2, \dots, x_n)$  als auch die Zielvariable  $y$  vor. Auf diesen Trainingsdaten wird das Modell  $y = f(\mathbf{x}) + \epsilon = f(x_1, x_2, \dots, x_n) + \epsilon$  gebildet und durch  $\hat{f}(\cdot)$  geschätzt. Es ist also die Variable  $y$  vorherzusagen.
2. Dieses geschätzte Modell  $\hat{f}(\cdot)$  wird auf die Anwendungsdaten  $\mathbf{x}_0$ , für die (Ihnen) die Werte der Zielvariable  $y$  unbekannt sind, angewendet, d.h., es wird  $\hat{y}_0 := \hat{f}(\mathbf{x}_0)$  berechnet. Der unbekannte Wert  $y_0$  der Zielvariable  $y$  wird durch  $\hat{y}_0$  prognostiziert.

Liegt zu einem noch späteren Zeitpunkt der eingetroffene Wert  $y_0$  der Zielvariable  $y$  vor, so kann die eigene Vorhersage  $\hat{y}_0$  evaluiert werden, d.h. z.B. kann der Fehler  $e = y_0 - \hat{y}_0$  zwischen prognostiziertem Wert  $\hat{y}_0$  und wahren Wert  $y_0$  analysiert werden.

In der praktischen Anwendung können zeitlich drei aufeinanderfolgende Schritte unterschieden werden (vergleiche oben):

1. die *Trainingsphase*, d.h., die Phase für die sowohl erklärende ( $\mathbf{x}$ ) als auch die erklärte Variable ( $y$ ) bekannt sind. Hier wird das Modell geschätzt (gelernt):  $\hat{f}(\mathbf{x})$ . Dafür wird der Trainingsdatensatz genutzt.
2. In der folgenden *Anwendungsphase* sind nur die erklärenden Variablen ( $\mathbf{x}_0$ ) bekannt, nicht  $y_0$ . Auf Basis der Ergebnisse aus dem 1. Schritt wird  $\hat{y}_0 := \hat{f}(\mathbf{x}_0)$  prognostiziert.
3. Evt. gibt es später noch die *Evaluierungsphase*, für die dann auch die Zielvariable ( $y_0$ ) bekannt ist, so dass die Vorhersagegüte des Modells überprüft werden kann.

Im Computer kann man dieses Anwendungsszenario *simulieren*: man teilt die Datenmenge *zufällig* in eine Lern- bzw. Trainingsstichprobe (Trainingsdaten;  $(\mathbf{x}, y)$ ) und eine Teststichprobe (Anwendungsdaten,  $(\mathbf{x}_0)$ ) auf: Die Modellierung erfolgt auf den Trainingsdaten. Das Modell wird angewendet auf die Testdaten (Anwendungsdaten). Da man hier aber auch die Zielvariable ( $y_0$ ) kennt, kann damit das Modell evaluiert werden.

## 2.4 Hauptziel: Genaue Prognose

Ihre Aufgabe ist: Spielen Sie den Data-Scientist! Konstruieren Sie ein Modell auf Basis der Trainingsdaten ( $\mathbf{x}, y$ ) und sagen Sie für die Anwendungsdaten ( $\mathbf{x}_0$ ) die Zielvariable möglichst genau voraus ( $\hat{y}_0$ ).

Ihr(e) Dozent\*in kennt den Wert der Zielvariable ( $y_0$ ). Sie nicht.

Von zwei Prognosemodellen zum gleichen Datensatz ist dasjenige Modell besser, das weniger Vorhersagefehler aufweist (im Test-Datensatz), also genauer vorhersagt. Kurz gesagt: Genauer ist besser.

## 2.5 Zum Aufbau Ihrer Prognosedatei im CSV-Format

1. Die CSV-Datei muss aus genau zwei Spalten mit (exakt) folgenden Spaltennamen bestehen:
  - a. `id` : Den ID-Wert jedes vorhergesagten Wertes
  - b. `pred` : Der vorhergesagte Wert.
3. Umlaute sind zu ersetzen (also `süß` wird `suess` etc.).
4. Die CSV-Datei muss als *Spaltentrennzeichen* ein *Komma* verwenden und als *Dezimaltrennzeichen* einen *Punkt* (d.h. also die *Standardformatierung* einer CSV-Datei; *nicht* die deutsche Formatierung).

5. Die CSV-Datei muss genau die Anzahl an Zeilen aufweisen, die der Zeilenlänge im Test-Datensatz entspricht.
6. Prüfen Sie, dass Ihre CSV-Datei sich problemlos lesen lässt. Falls keine (funktionstüchtige) CSV-Datei eingereicht (hochgeladen) wurde, ist die Prüfung nicht bestanden. Tipp: Öffnen Sie die CSV-Datei mit einem Texteditor und schauen Sie sich an, ob alles vernünftig aussieht. Achtung: Öffnen Sie die CSV-Datei besser nicht mit Excel, da Excel einen Bug hat, der CSV-Dateien verfälschen kann auch ohne dass man die Datei speichert.

## 2.6 Einzureichende Dateien

1. Folgende\* Dateiarten\* sind einzureichen:
  1. Prognose: Ihre *Prognose-Datei* (CSV-Datei)
  2. Analyse: Ihr *Analyseskript* (R-, Rmd- oder Rmd-Notebook-Datei)
2. Weitere Dateien sind nicht einzureichen.
3. Komprimieren Sie die Dateien *nicht* (z.B. via *zip*).
4. Der Name jeder eingereichte Datei muss wie folgt lauten: `Nachname_Vorname_Matrikelnummer_Dateiart.Endung`. Beispiel:  
`Sauer_Sebastian_0123456_Prognose.csv` bzw. `Sauer_Sebastian_0123456_Analyse.Rmd`.

## 2.7 Gliederung Ihrer Analyse

Ihr Analysedokument stellt alle Ihre Schritte vor, die Sie im Rahmen der Bearbeitung der Prüfungsaufgabe unternommen haben, zumindest was die Analyse der Daten betrifft.

Das Dokument mischt drei Textarten: R-Syntax, R-Ausgaben sowie Prosa (normale Sprache). Alle drei Aspekte sind gleichermaßen wichtig und unabdingbar für diese Analyse.

Wenn Sie das Dokument als R-Markdown-Datei (Rmd-Datei) anlegen, müssen Sie R-Code in einem "R-Chunk" auszeichnen. Prosa wird in Rmd-Datei als Standard gesehen, sie brauchen ihn nicht extra auszuzeichnen (für R-Notebook-Dateien gilt das Gleiche). In R-Skript-Dateien ist es umgekehrt: Sie müssen R-Code nicht extra auszeichnen, da in R-Skripten R als "Standard-Text" gesehen wird. Hingegen müssen Sie Prosa als Kommentar einfügen. Es bleibt Ihnen überlassen, für welche Variante (R-, Rmd- oder R-Notebook) Sie sich entscheiden. Keine Option wird als besser oder schlechter gewertet (vermutlich ist Rmd für Sie am einfachsten).

Sie können Ihr Analysedokument z.B. so gliedern:

1. Forschungsfrage und Hintergrund (Beschreiben Sie kurz, worum es geht)
2. Vorbereitung (Pakete laden, Daten importieren, etc.)
3. Explorative Datenanalyse (Untersuchen Sie den Datensatz nach Auffälligkeiten, die Sie dann beim Modellieren nutzen)
4. Modellieren (Mehrere Prognosemodelle, z.B. via `lm(av ~ uv)`)
5. Vorhersagen (Vorhersage der Test-Daten anhand des besten Vorhersagemodells und Einreichen)

Die Gliederung ist kein Muss; andere Gliederung sind auch möglich. Entscheidend ist die fachliche Angemessenheit und die Reproduzierbarkeit.

### 2.7.1 Abschnitt Forschungsfrage und Hintergrund

In diesem Abschnitt passiert noch keine Statistik bzw. keine Analyse. Stattdessen stellen Sie in "normaler Sprache", also ohne intensiven Gebrauch vom (statistischen) Fachvokabular dar, was Ziel und was Hintergrund der Analyse ist. Sie können als Ziel bzw. Hintergrund den formalen Aspekt der Prüfung anführen, wichtiger sind aber inhaltliche bzw. fachliche Überlegungen: Worum geht es in der Analyse? Warum ist die Frage wichtig? Was wird untersucht? Anhand welcher Methodik wird die Frage untersucht?

### 2.7.2 Vorbereitung

In diesem Abschnitt Ihres Analysedokuments führen Sie die technische Vorbereitung durch. Das betrifft vor allem das Importieren der Daten und das Starten aller R-Pakete, die in der Analyse verwendet werden.

### 2.7.3 Explorative Datenanalyse

Die explorative Datenanalyse (EDA) meint sowohl die deskriptive Statistik als auch die Datenvisualisierung. Typische Schritte sind: das Bearbeiten (oder Entfernen) von Extremwerten und fehlenden Werten, die Untersuchung von Verteilungsformen oder das Suchen nach Mustern (Korrelationen, Gruppenunterschieden). Ein nützliches Ergebnis ist z.B. zu erkennen, welche Variablen sich als Prädiktoren eignen (für den nächsten Abschnitt der Modellierung). Ziel ist, dass Sie den folgenden Schritt vorbereiten, also Schritte unternehmen, damit Sie die AV möglichst gut vorhersagen können.

### 2.7.4 Modellierung

In diesem Schritt berechnen Sie Prognosemodelle. Das sind oft lineare Modelle, also etwa `lm(av ~ uv)`. Es empfiehlt sich, mehrere Modelle zu berechnen und zu schauen, welches dieser Kandidaten am besten ist. Die Güte eines Prognosemodells bemisst sich letztlich *nur* an der Präzision der Vorhersage *neuer* Daten, also des Test-Datensatzes. Wie gut Ihre Vorhersagen also wirklich sind, erfahren Sie erst mit der

Notenbekanntgabe. Allerdings können Sie die Trainingsdaten nutzen, um die Güte Ihres Modells abzuschätzen.

## 2.7.5 Vorhersagen

Schließlich entscheiden Sie sich für einen Modellkandidaten. Diesen Modellkandidaten nehmen Sie mit, um die (ihnen unbekannten) Werte der AV (Zielvariablen) vorherzusagen. Diese Vorhersagen - zusammen mit der ID für jede Vorhersage - speichern Sie als (reguläre) CSV-Datei ab und reichen Sie als Ihre Prüfungsleistung ein, zusammen mit Ihrer Analysedatei.

## 2.8 Tipps

### 2.8.1 Tipps für eine gute Prognose

- Schauen Sie in die Literatur.
- Evtl. kann eine Datenvorverarbeitung (Variablentransformation, z.B. log() oder die Elimination von Ausreißern) helfen.
- Überlegen Sie sich Kriterien zur Modell- und/oder Variablenauswahl. Auch hierfür gibt es Algorithmen und R-Funktionen.
- Vermeiden Sie Über-Anpassung (Overfitting).
- Vermeiden Sie viele fehlende Werte bei Ihrer Prognose. Fehlende Werte werden bei der Benotung mit dem Mittelwert (der vorhandenen Prognosewerte Ihrer Einreichung) aufgefüllt.
- Arbeiten Sie die bereitgestellten Fallstudien durch. Wenn Sie mehr tun möchten, finden Sie im Internet eine Fülle von weiteren Fallstudien.

### 2.8.2 Tipps zur Datenverarbeitung

- Ein "deutsches" Excel kann Standard-CSV-Dateien nicht ohne Weiteres lesen. Online-Dienste wie Google Sheets können dies allerdings.

### 2.8.3 Tipps zum Aufbau des Analyseskripts

- Zu Beginn des Skripts sollten alle verwendeten R-Pakete mittels `library()` gestartet werden.
- Zu Beginn des Skripts sollten die Daten von der vom Dozenten bereitgestellten URL importiert werden (*nicht* von der eigenen Festplatte, da das Skript sonst bei Dritten, wie Ihrem Prüfer, nicht lauffähig ist).

### 2.8.4 Sonstiges

- Legen Sie regelmäßig Sicherheitskopien Ihrer Arbeit an (ggf. auf einem anderen Datenträger).
- Achten Sie darauf, dass Sie nicht durcheinander kommen, in welcher Datei der aktuelle Stand Ihrer Arbeit liegt.

## 2.9 Bewertung

### 2.9.1 Kriterien

- Es gibt drei Bewertungskriterien:
  - *Formalit:* u.a. Reproduzierbarkeit der Analyse, Lesbarkeit der Syntax, Übersichtlichkeit der Analyse.
  - *Methode:* u.a. methodischer Anspruch und Korrektheit in der Explorativen Datenanalyse, Datenvorverarbeitung, Variablenauswahl und Modellierungsmethode.
  - *Inhalt:* **Vorhersagegüte.**
- Das zentrale Bewertungskriterium ist *Inhalt*; die übrigen beiden Kriterien fließen nur bei besonders guter oder schlechter Leistung in die Gesamtnote ein.
- Die quantitative Datenanalyse in Durchführung und Interpretation ist der Schwerpunkt dieser Arbeit. Zufälliges identisches Vorgehen, z.B. im R Code, ist sehr unwahrscheinlich und kann als **Plagiat** bewertet werden.
- Die Gesamtnote muss sich nicht als arithmetischer Mittelwert der Teilnoten ergeben.
- Es werden keine Teilnoten vergeben, sondern nur eine Gesamtnote wird vergeben.
- Es werden keine Hinweise vergeben, stattdessen gibt es einen Überblick an typischen Fehlern.
- Es wird keine Musterlösung veröffentlicht, um nachfolgende Kohorten nicht zu bevorteilen bzw. die aktuelle Kohorte nicht zu benachteiligen.

### 2.9.2 Kennzahl der Modellgüte

Die Güte der Vorhersage wird anhand des *mittleren Absolutfehlers* (mae) bemessen:

$$\text{mae} = \frac{1}{n} \sum_{i=1}^n |(y_i - \hat{y}_i)|$$

## 2.9.3 Notenstufen

Zur Vorhersagegüte: Die Vorhersagegüte eines einfachen Minimalmodells entspricht einer 4,0, die eines Referenzmodells des Dozenten einer 2,0.

Ihre Bewertung erfolgt entsprechend Ihrer Vorhersagegüte, d.h., sind Sie besser als das Referenzmodell erhalten Sie hier in diesem Teilaспект eine bessere Note als 2,0!

## 2.9.4 Bewertungsprozess

Der Gutachter legt im Nachgang der Prüfung alle Teilnehmern ihre jeweilige Wert der Kennzahl der Modellgüte offen. Außerdem werden die vorherzusagenden Daten veröffentlicht sowie die Grenzwerte für jede Notenstufe. Auf dieser Basis ist es allen Teilnehmern möglich, die Korrektheit Ihrer Note zu überprüfen.

## 2.10 Hinweise

Sie haben freie Methodenwahl bei der Modellierung und Vorverarbeitung. Nutzen Sie den Stoff wie im Unterricht gelernt; Sie können aber auch auf weitere Inhalte, die nicht im Unterricht behandelt wurden, zugreifen.

Eine Einführung in verschiedene Methoden gibt es z.B. bei Sebastian Sauer (2019): *Moderne Datenanalyse mit R*<sup>1</sup> aber auch bei Max Kuhn und Julia Silge (2021): *Tidy Modeling with R*<sup>2</sup>. Die Bücher beinhalten jeweils Beispiele und Anwendung mit R.

Auch ist es Ihnen überlassen, welche Variablen Sie zur Modellierung heranziehen – und ob Sie diese eventuell vorverarbeiten, d.h., transformieren, zusammenfassen, Ausreißer bereinigen o.Ä.. Denken Sie nur daran, die Datentransformation, die Sie auf den Trainingsdaten durchführen, auch auf den Testdaten (Anwendungsdaten) durchzuführen.

Hinweise zur Modellwahl usw. gibt es auch in erwähnter Literatur, aber auch in vielen Büchern zum Thema Data-Science.

**Alles, was Sie tun, Datenvorverarbeitung, Modellierung und Anwenden, muss transparent und reproduzierbar sein.** Im Übrigen lautet die Aufgabe: Finden Sie ein Modell, von dem Sie glauben, dass es die Testdaten gut vorhersagt.  $\hat{y} = 42$  ist zwar eine schöne Antwort, trifft die Wirklichkeit aber leider nicht immer. Eine gute Modellierung auf den *Trainingsdaten* (z.B. hohes  $R^2$ ) bedeutet nicht zwangsläufig eine gute Vorhersage (*Test-Set*).

## 2.11 Formalia

1. Es sind nur Einzelarbeiten zulässig.
2. In der Analyse muss als Ausgangspunkt der vom/von der Dozenten/in bereitgestellten Datensatz genutzt werden.
3. Alle Analyseschritte bzw. alle Veränderungen an den Daten müssen im (eingereichten) *Analyseskript* nachvollziehbar (transparent und reproduzierbar) aufgeführt sein. Das *Analyseskript* ist als R-Skript, Rmd-Datei oder Rmd-Notebook-Datei abzugeben. Sie können die bereitgestellte Vorlage als *Analyseskript* nutzen (`Template-Dokumentation-Vorhersagemodellierung.Rmd`).
4. Das *Analyseskript* muss funktionstüchtig für den Prüfer sein: Alle Befehle müssen ohne Fehlermeldung durchlaufen (abgesehen von etwaiger Installation fehlender Pakete).
5. Es dürfen keine weiteren Informationen (Daten) als die vom Dozenten ausgegebenen verwendet werden. Sonstige Hilfe (z.B. von Dritten) ist ebenfalls unzulässig.
6. Nichtbeachtung der für dieses Modul formulierten Regeln kann zu Nichtbestehen oder Punkteabzug führen.
7. Der Schwerpunkt dieser Hausarbeit liegt auf der quantitativen Modellierung, der formale Anspruch liegt daher unter dem von anderen Hausarbeiten.
8. Es muss keine Literatur zitiert werden.
9. Ein ausgedrucktes Exemplar muss nicht abgegeben werden.
10. Während der Prüfungsphase werden keine inhaltlichen Fragen ("wie macht man nochmal eine Log-Transformation?") und keine technischen Fragen ("wie installiert man nochmal ein R-Paket?") beantwortet.

## 2.12 Ich brauche Hilfe!

### 2.12.1 Wo finde ich Beispiele und Vorlagen?

Im Rahmen des Unterrichts wurden mehrere Fallstudien erarbeitet bzw. bereitgestellt, diese dienen Ihnen als ideale Vorlage.

Eine Beispiel-Modellierung finden Sie in der Datei `Beispielanalyse-Prognose-Wettbewerb.Rmd`. Eine beispielhafte Vorlage (Template), die Sie als Richtschnur nutzen können, ist mit der Datei `Template-Vorhersagemodellierung.Rmd` hier (<https://github.com/sebastiansauer/Lehre/blob/main/Hinweise/Prognosewettbewerb/Template-Vorhersagemodellierung.Rmd>) bereitgestellt.

Im Internet finden sich viele Fallstudien, von denen Sie sich inspirieren lassen können.

## 2.12.2 Materialsammlung

In [diesem Ordner]((<https://github.com/sebastiansauer/Lehre/tree/main/Hinweise/Prognosewettbewerb>)) (<https://github.com/sebastiansauer/Lehre/tree/main/Hinweise/Prognosewettbewerb>)) finden Sie eine Materialsammlung zum Prognosewettbewerb.

## 2.12.3 Videos

Diese Playlist (<https://youtube.com/playlist?list=PLRR4REmBgplH6uG8LZWPTSMReX1OFxfUx>) beinhaltet Videos, die die Rahmenbedingungen der Prüfungsleistung vorstellt.

## 2.13 Plagiatskontrolle

Die eingereichten Arbeiten können automatisiert auf Plagiate überprüft werden. Gibt es substanzielle Überschneidungen zwischen zwei (oder mehr) Arbeiten, werden alle betreffenden Arbeiten mit *ungenügend* bewertet oder es folgt eine Abwertung der Note.

# I Themen

## 3 Statistisches Lernen

Benötigte R-Pakete für dieses Kapitel:

```
library(tidyverse)
```

### 3.1 Lernsteuerung

#### 3.1.1 Vorbereitung

- Lesen Sie die Hinweise zum Modul.
- Installieren (oder Updaten) Sie die für dieses Modul angegebene Software.
- Lesen Sie die Literatur.

#### 3.1.2 Lernziele

- Sie können erläutern, was man unter statistischem Lernen versteht.
- Sie wissen, war Overfitting ist, wie es entsteht, und wie es vermieden werden kann.
- Sie kennen verschiedenen Arten von statistischem Lernen und können Algorithmen zu diesen Arten zuordnen.

#### 3.1.3 Literatur

- Rhys, Kap. 1
- evtl. Sauer, Kap. 15

#### 3.1.4 Hinweise

- Bitte beachten Sie die Hinweise zum Präsenzunterricht und der Streamingoption.
- Bitte stellen Sie sicher, dass Sie einen einsatzbereiten Computer haben und dass die angegebene Software (in aktueller Version) läuft.

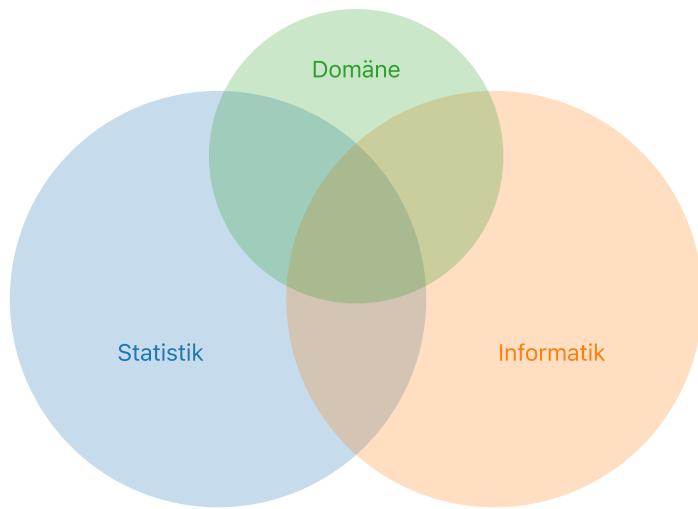
## 3.2 Was ist Data Science?

Es gibt mehrere Definitionen von *Data Science*, aber keinen kompletten Konsens. Baumer, Kaplan, and Horton (2017) definieren Data Science wie folgt (S. 4):

The science of extracting meaningful information from data

Auf der anderen Seite entgegen viele Statistiker: "Hey, das machen wir doch schon immer!".

Eine Antwort auf diesen Einwand ist, dass in Data Science nicht nur die Statistik eine Rolle spielt, sondern auch die Informatik sowie - zu einem geringen Teil - die Fachwissenschaft ("Domäne"), die sozusagen den Empfänger bzw. die Kunden oder den Rahmen stellt. Dieser "Dreiklang" ist in folgendem Venn-Diagramm dargestellt.



### 3.3 Was ist Machine Learning?

*Maschinelles Lernen* (ML), oft auch (synonym) als *statistisches Lernen* (statistical learning) bezeichnet, ist ein Teilgebiet der *künstlichen Intelligenz* (KI; artificial intelligence, AI) (Rhys 2020). ML wird auch als *data-based* bezeichnet in Abgrenzung von *rule-based*, was auch als "klassische KI" bezeichnet wird, vgl. Abb. 3.1.

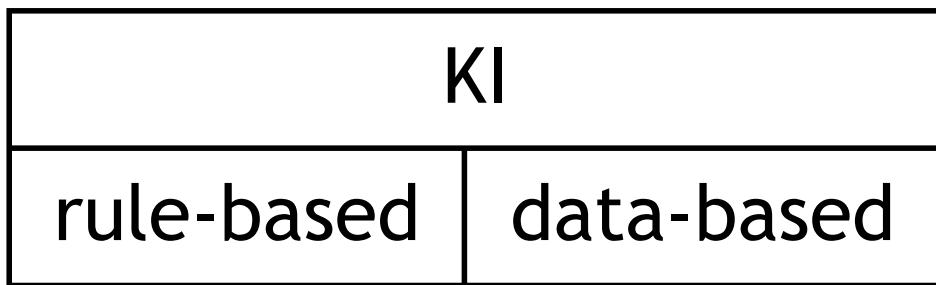


Abbildung 3.1: KI und Maschinelles Lernen

In beiden Fällen finden Algorithmen Verwendung. Algorithmen sind nichts anderes als genaue Schritt-für-Schritt-Anleitungen, um etwas zu erledigen. Ein Kochrezept ist ein klassisches Beispiel für einen Algorithmus.

Hier (<https://www.c-programming-simple-steps.com/images/xsum-two-numbers-h.png.pagespeed.ic.AM9WYFPgEo.webp>) findet sich ein Beispiel für einen einfachen Additionsalgorithmus.

Es gibt viele ML-Algorithmen, vgl. Abb. 3.2.

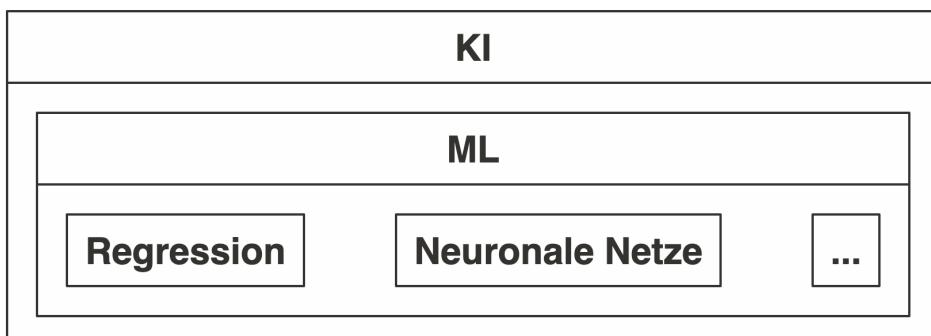


Abbildung 3.2: ML-Matroschka

### 3.3.1 Rule-based

Klassische (ältere) KI implementiert Regeln "hartverdrahtet" in ein Computersystem. Nutzer füttern Daten in dieses System. Das System leitet dann daraus Antworten ab.

Regeln kann man prototypisch mit *Wenn-Dann-Abfragen* darstellen:

```

lernzeit <- c(0, 10, 10, 20)
schlauer_nebensitzer <- c(FALSE, FALSE, TRUE, TRUE)

for (i in 1:4) {
  if (lernzeit[i] > 10) {
    print("bestanden!")
  } else {
    if (schlauer_nebensitzer[i] == TRUE) {
      print("bestanden!")
    } else print("Durchgefallen!")
  }
}

```

```

## [1] "Durchgefallen!"
## [1] "Durchgefallen!"
## [1] "bestanden!"
## [1] "bestanden!"

```

Sicherlich könnte man das schlauer programmieren, vielleicht so:

```

d <-
  tibble(
  lernzeit = c(0, 10, 10, 20),
  schlauer_nebensitzer = c(FALSE, FALSE, TRUE, TRUE)
)

d %>%
  mutate(bestanden = ifelse(lernzeit > 10 | schlauer_nebensitzer == TRUE, TRUE, FALSE))

## # A tibble: 4 × 3
##   lernzeit schlauer_nebensitzer bestanden
##     <dbl> <lgl>                <lgl>
## 1       0 FALSE                 FALSE
## 2      10 FALSE                 FALSE
## 3      10 TRUE                  TRUE
## 4      20 TRUE                  TRUE

```

### 3.3.2 Data-based

ML hat zum Ziel, Regeln aus den Daten zu lernen. Man füttert Daten und Antworten in das System, das System gibt Regeln zurück.

James et al. (2021) definieren ML so: Nehmen wir an, wir haben die abhängige Variable  $Y$  und  $p$  Prädiktoren,  $X_1, X_2, \dots, X_p$ . Weiter nehmen wir an, die Beziehung zwischen  $Y$  und  $X = (X_1, X_2, \dots, X_p)$  kann durch eine Funktion  $f$  beschrieben werden. Das kann man so darstellen:

$$Y = f(X) + \epsilon$$

ML kann man auffassen als eine Menge an Verfahren, um  $f$  zu schätzen.

Ein Beispiel ist in Abb. 3.3 gezeigt (James et al. 2021).

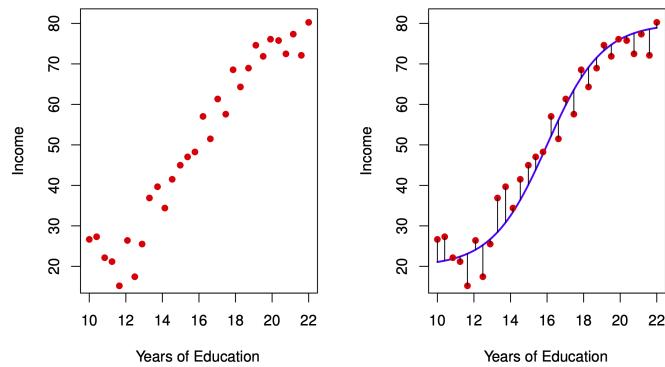


Abbildung 3.3: Vorhersage des Einkommens durch Ausbildungsjahre

Natürlich kann  $X$  mehr als eine Variable beinhalten, vgl. Abb. 3.4 (James et al. 2021).

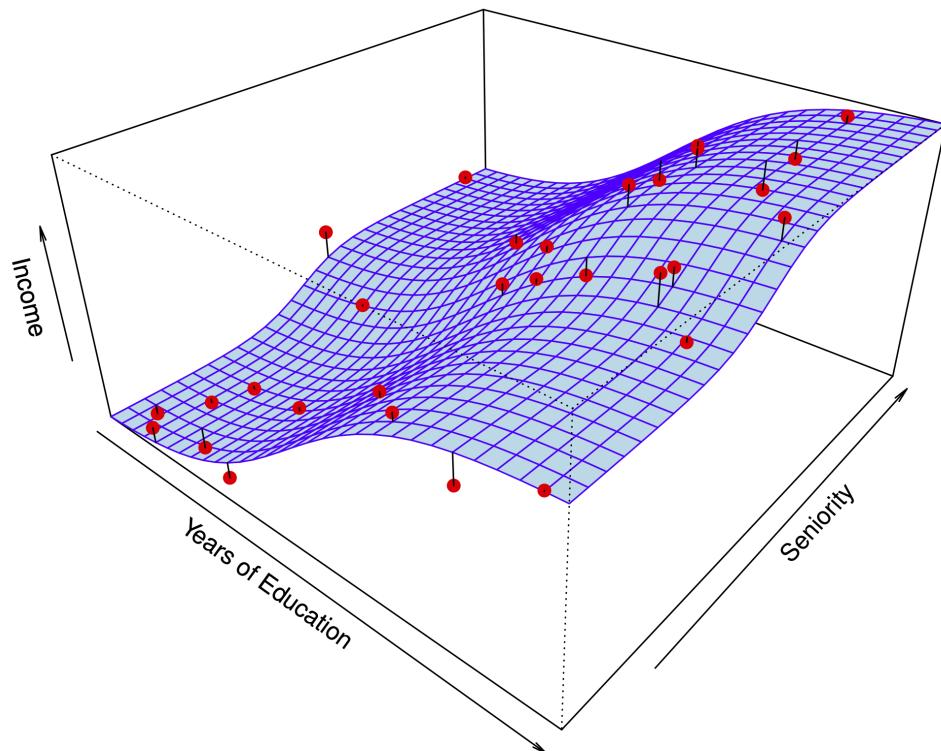


Abbildung 3.4: Vorhersage des Einkommens als Funktion von Ausbildungsjahren und Dienstjahren

Anders gesagt: traditionelle KI-Systeme werden mit Daten und Regeln gefüttert und liefern Antworten. ML-Systeme werden mit Daten und Antworten gefüttert und liefern Regeln zurück, vgl. Abb. 3.5.

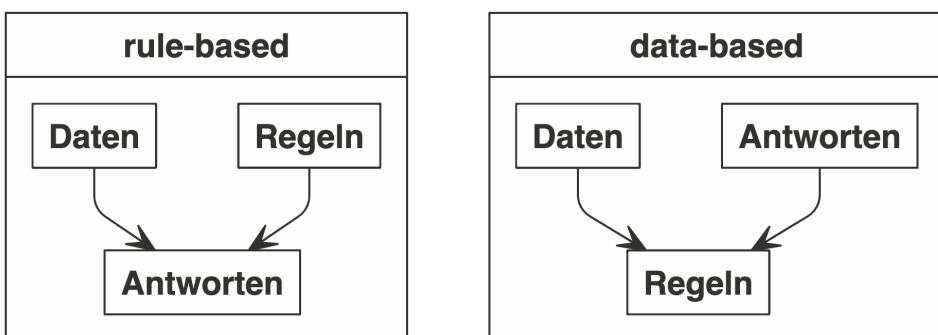


Abbildung 3.5: Vergleich von klassischer KI und ML

## 3.4 Modell vs. Algorithmus

### 3.4.1 Modell

Ein Modell, s. Abb. 3.6 (Spurzem 2017)!



Abbildung 3.6: Ein Modell-Auto

Wie man sieht, ist ein Modell eine vereinfachte Repräsentation eines Gegenstands.

Der Gegenstand definiert (gestaltet) das Modell. Das Modell ist eine Vereinfachung des Gegenstands, vgl. Abb. 3.7.

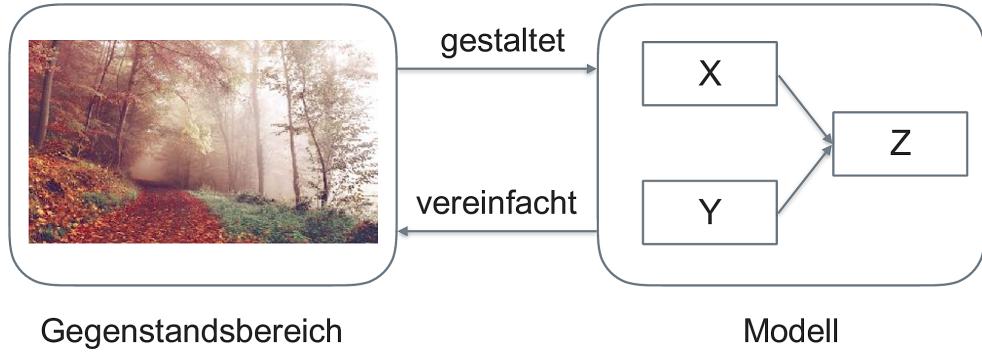


Abbildung 3.7: Gegenstand und Modell

Im maschinellen Lernen meint ein Modell, praktisch gesehen, die Regeln, die aus den Daten gelernt wurden.

### 3.4.2 Beispiel für einen ML-Algorithmus

Unter einem ML-Algorithmus versteht man das (mathematische oder statistische) Verfahren, anhand dessen die Beziehung zwischen  $X$  und  $Y$  „gelernt“ wird. Bei Rhys (2020) (S. 9) findet sich dazu ein Beispiel, das kurz zusammengefasst etwa so lautet:

*Beispiel eines Regressionsalgoritmus*

1. Setze Gerade in die Daten mit  $b_0 = \hat{y}, b_1 = 0$
2. Berechne  $MSS = \sum (y_i - \hat{y}_i)^2$
3. „Drehe“ die Gerade ein bisschen, d.h. erhöhe  $b_1^{neu} = b_1^{alt} + 0.1$
4. Wiederhole 2-3 solange, bis  $MSS < \text{Zielwert}$

Diesen Algorithmus kann man „von Hand“ z.B. mit dieser App (<https://shinyapps.org/showapp.php?app=https://shiny.psy.lmu.de/felix/lmfit&by=Felix%20Sch%C3%B6nbradt&title=Find-a-fit!&shorttitle=Find-a-fit!>) durchspielen.

## 3.5 Taxonomie

Methoden des maschinellen Lernens lassen sich verschiedentlich gliedern. Eine typische Gliederung unterscheidet in *supervidierte* (geleitete) und *nicht-supervidierte* (ungeleitete) Algorithmen, s. Abb. 3.8.

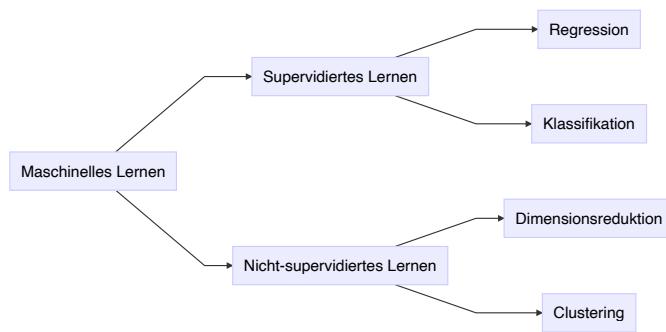


Abbildung 3.8: Taxonomie der Arten des maschinellen Lernens

### 3.5.1 Geleitetes Lernen

Die zwei Phasen des geleiteten Lernens sind in Abb. 3.9 dargestellt.

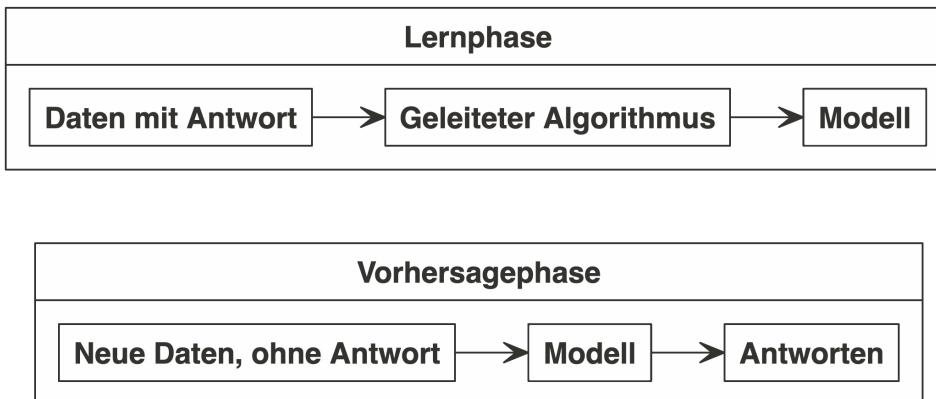
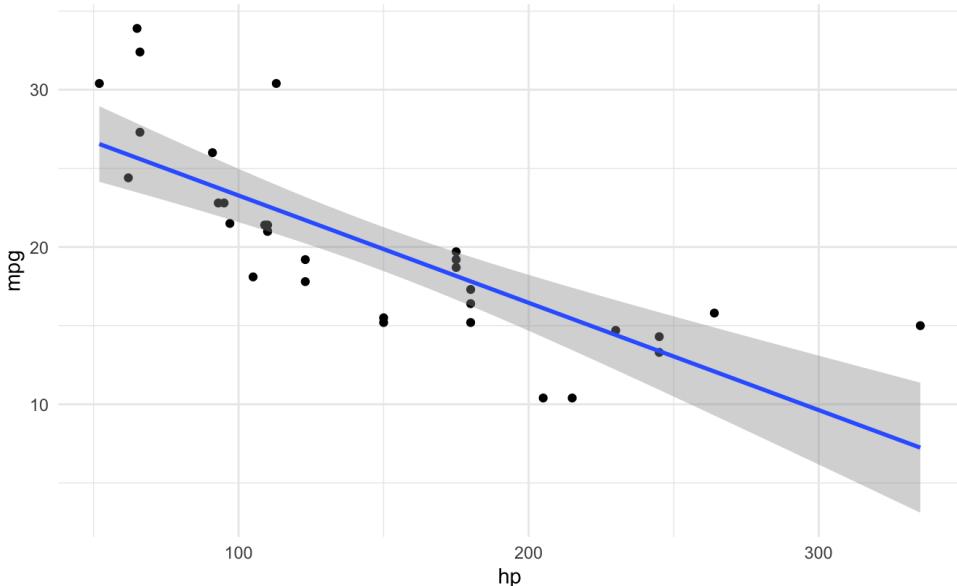


Abbildung 3.9: Geleitetes Lernen geschieht in zwei Phasen

#### 3.5.1.1 Regression: Numerische Vorhersage

```

ggplot(mtcars) +
  aes(x = hp, y = mpg) +
  geom_point() +
  geom_smooth(method = "lm") +
  theme_minimal()
  
```



Die Modellgüte eines numerischen Vorhersagemodells wird oft mit (einem der) folgenden *Gütekoeffizienten* gemessen:

- Mean Squared Error (Mittlerer Quadratfehler):

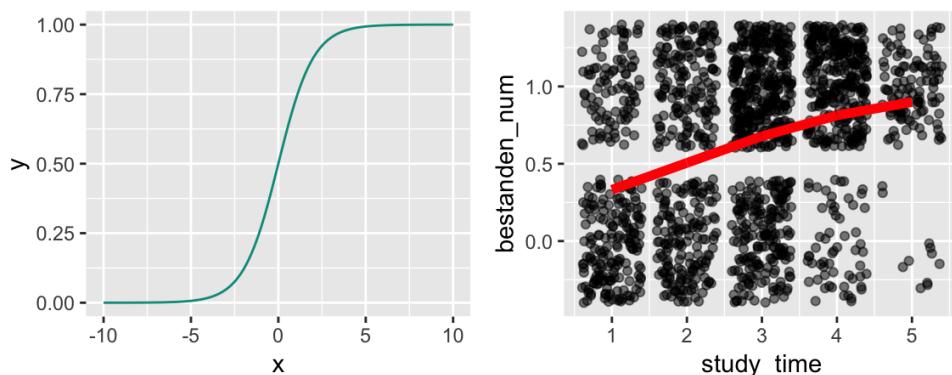
$$MSE := \frac{1}{n} \sum (y_i - \hat{y}_i)^2$$

- Mean Absolute Error (Mittlerer Absolutfehler):

$$MAE := \frac{1}{n} \sum |(y_i - \hat{y}_i)|$$

Wir sind nicht daran interessiert die Vorhersagegenauigkeit in den bekannten Daten einzuschätzen, sondern im Hinblick auf neue Daten, die in der Lernphase dem Modell nicht bekannt waren.

### 3.5.1.2 Klassifikation: Nominale Vorhersage



Die Modellgüte eines numerischen Vorhersagemodells wird oft mit folgendem *Gütekoeffizienten* gemessen:

- Mittlerer Klassifikationsfehler  $e$ :

$$e := \frac{1}{n} I(y_i \neq \hat{y}_i)$$

Dabei ist  $I$  eine Indikatorfunktion, die 1 zurückliefert, wenn tatsächlicher Wert und vorhergesagter Wert identisch sind.

## 3.5.2 Ungeleitetes Lernen

Die zwei Phasen des ungeleiteten Lernens sind in Abb. 3.10 dargestellt.



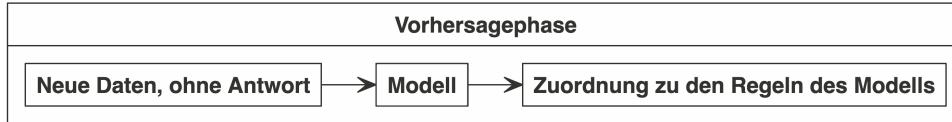


Abbildung 3.10: Die zwei Phasen des ungeleiteten Lernens

Ungeleitetes Lernen kann wiederum in zwei Arten unterteilen, vgl. Abb. 3.11:

1. Fallreduzierendes Modellieren (Clustering)
2. Dimensionsreduzierendes Modellieren (z.B. Faktorenanalyse)

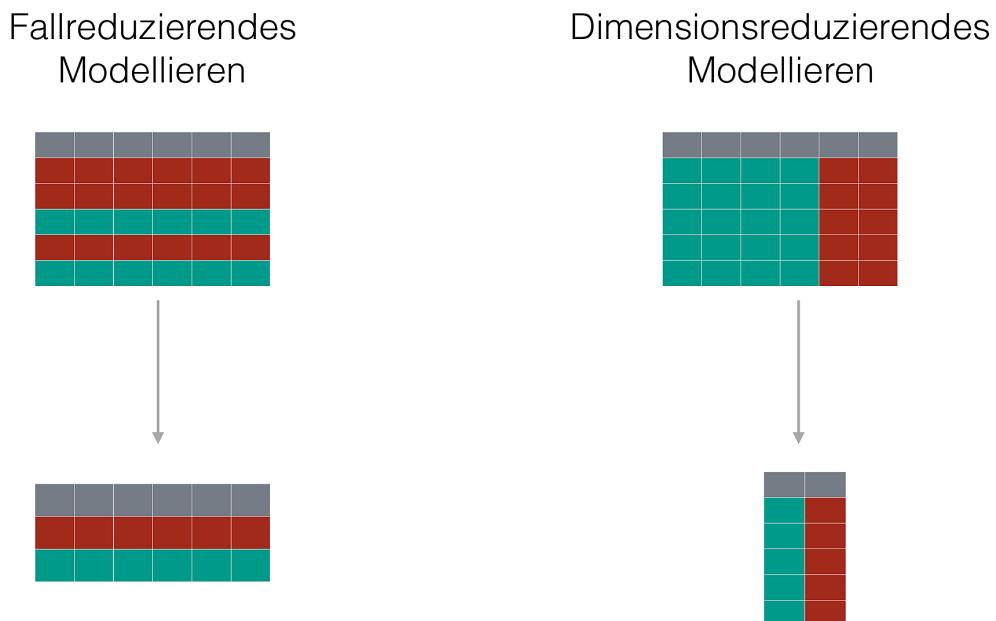


Abbildung 3.11: Zwei Arten von ungeleittem Modellieren

## 3.6 Ziele des ML

Man kann vier Ziele des ML unterscheiden, s. Abb. 3.12.

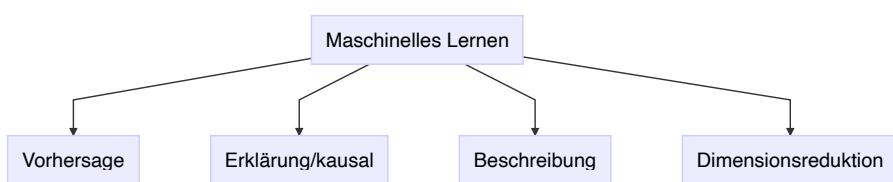


Abbildung 3.12: Ziele des maschinellen Lernens

*Vorhersage* bezieht sich auf die Schätzung der Werte von Zielvariablen (sowie die damit verbundene Unsicherheit). *Erklärung* meint die kausale Analyse von Zusammenhängen. *Beschreibung* ist praktisch gleichzusetzen mit der Verwendung von deskriptiven Statistiken. *Dimensionsreduktion* ist ein Oberbegriff für Verfahren, die die Anzahl der Variablen (Spalten) oder der Beobachtungen (Zeilen) verringert.

Wie "gut" ein Modell ist, quantifiziert man in verschiedenen Kennzahlen; man spricht von Modellgüte oder *model fit*. Je schlechter die Modellgüte, desto höher der *Modellfehler*, vgl. Abb. 3.13.

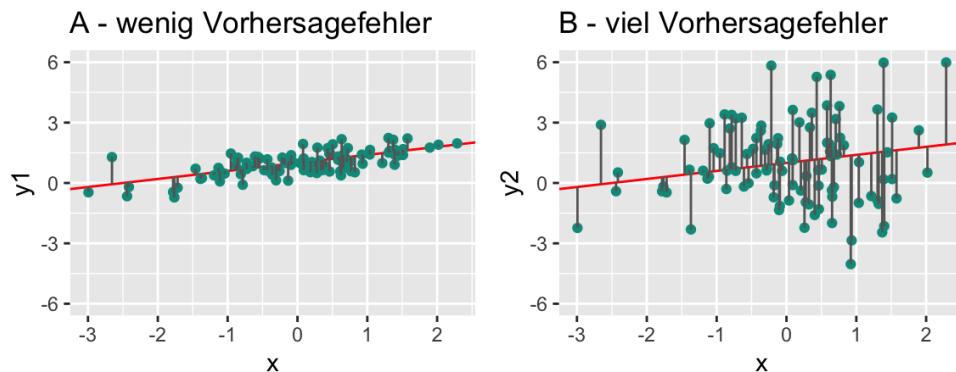


Abbildung 3.13: Wenig (links) vs. viel (rechts) Vorhersagefehler

Die Modellgüte eines Modells ist nur relevant für *neue Beobachtungen*, an denen das Modell nicht trainiert wurde.

## 3.7 Über- vs. Unteranpassung

*Overfitting*: Ein Modell sagt die Trainingsdaten zu genau vorher - es nimmt Rauschen als "bare Münze", also fälschlich als Signal. Solche Modelle haben zu viel *Varianz* in ihren Vorhersagen.

*Underfitting*: Ein Modell ist zu simpel (ungenau, grobkörnig) - es unterschlägt Nuancen des tatsächlichen Musters. Solche Modelle haben zu viel *Verzerrung* (Bias) in ihren Vorhersagen.

Welches der folgenden Modelle (B,C,D) passt am besten zu den Daten (A), s. Abb. 3.14, vgl. (Sauer 2019), Kap. 15.

```
knitr:::include_graphics("img/overfitting-4-plots-1.png")
```

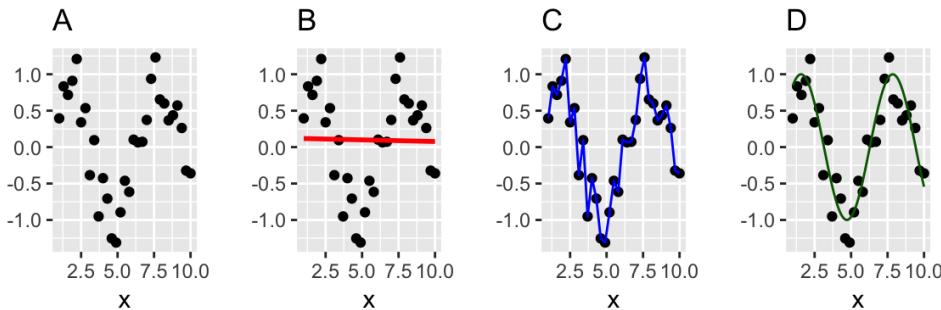


Abbildung 3.14: Over- vs. Underfitting

Welches Modell wird wohl neue Daten am besten vorhersagen? Was meinen Sie?

Modell D zeigt sehr gute Beschreibung ("Retrodiktion") der Werte, anhand derer das Modell trainiert wurde ("Trainingsstichprobe"). Wird es aber "ehrlich" getestet, d.h. anhand neuer Daten ("Test-Stichprobe"), wird es vermutlich *nicht* so gut abschneiden.

Es gilt, ein Modell mit "mittlerer" Komplexität zu finden, um Über- und Unteranpassung in Grenzen zu halten. Leider ist es nicht möglich, vorab zu sagen, was der richtige, "mittlere" Wert an Komplexität eines Modells ist, vgl. Abb. 3.15 aus (Sauer 2019).

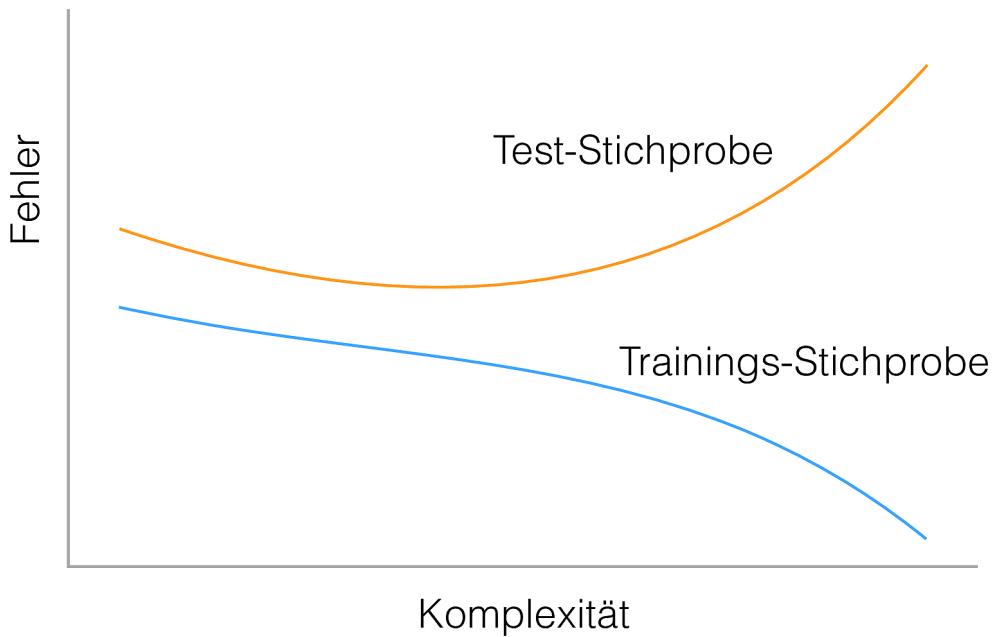
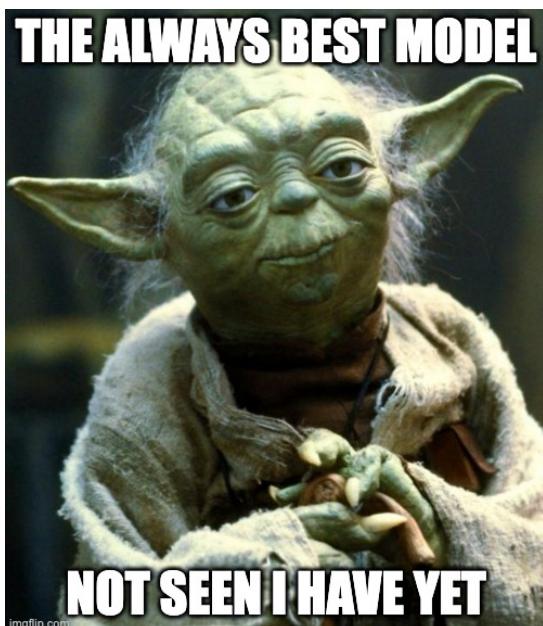


Abbildung 3.15: Mittlere Modellkomplexität führt zur besten Vorhersagegüte

### 3.8 No free lunch



(https://imgflip.com/i/687izk)

from Imgflip Meme Generator (<https://imgflip.com/memegenerator>)

Wenn  $f$  (die Beziehung zwischen  $Y$  und  $X$ , auch *datengenerierender Prozess* genannt) linear oder fast linear ist, dann wird ein lineare Modell gute Vorhersagen liefern, vgl. Abb. 3.16 aus James et al. (2021), dort zeigt die schwarze Linie den "wahren Zusammenhang", also  $f$  an. In orange sieht man ein lineares Modell, in grün ein hoch komplexes Modell, das sich in einer "wackligen" Funktion - also mit hoher Varianz - niederschlägt. Das grüne Modell könnte z.B. ein Polynom-Modell hohen Grades sein, z. B.  $y = b_0 + b_1x^{10} + b_2x^9 + \dots + b_{11}x^1 + \epsilon$ . Das lineare Modell hat hingegen wenig Varianz und in diesem Fall wenig Bias. Daher ist es für dieses  $f$  gut passend. Die grüne Funktion zeigt dagegen Überanpassung (overfitting), also viel Modellfehler (für eine Test-Stichprobe).

Die grüne Funktion in Abb. 3.16 wird neue, beim Modelltraining unbekannte Beobachtungen ( $y_0$ ) vergleichsweise schlecht vorhersagen. In Abb. 3.17 ist es umgekehrt.

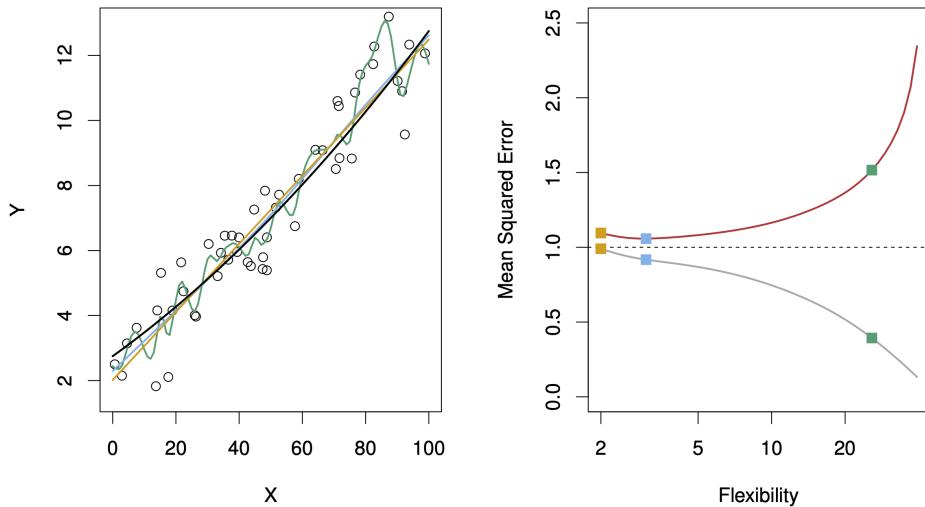


Abbildung 3.16: Ein lineare Funktion verlangt ein lineares Modell; ein nichtlineares Modell wird in einem höheren Vorhersagefehler (bei neuen Daten!) resultieren.

Betrachten wir im Gegensatz dazu Abb. 3.17 aus James et al. (2021), die (in schwarz) eine hochgradig *nichtlineare* Funktion  $f$  zeigt. Entsprechend wird das lineare Modell (orange) nur schlechte Vorhersagen erreichen - es hat zu viel Bias, da zu simpel. Ein lineares Modell wird der Komplexität von  $f$  nicht gerecht, Unteranpassung (underfitting) liegt vor.

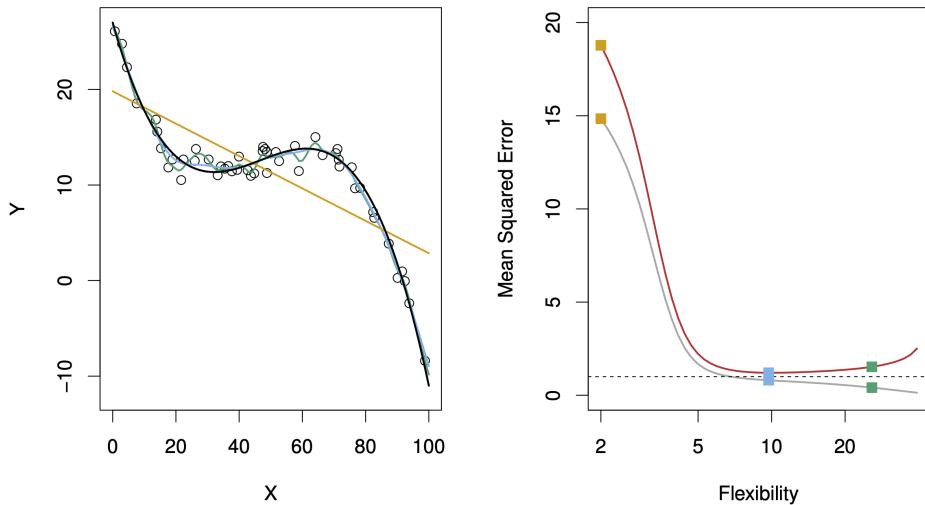


Abbildung 3.17: Eine nichtlineare Funktion (schwarz) verlangt eine nichtlineares Modell. Ein lineares Modell (orange) ist unterangepasst und hat eine schlechte Vorhersageleistung.

## 3.9 Bias-Varianz-Abwägung

Der Gesamtfehler  $E$  des Modells ist die Summe dreier Terme:

$$E = (y - \hat{y}) = \text{Bias} + \text{Varianz} + \epsilon$$

Dabei meint  $\epsilon$  den *nicht reduzierbaren Fehler*, z.B. weil dem Modell Informationen fehlen. So kann man etwa auf der Motivation von Studenten keine perfekte Vorhersage ihrer Noten erreichen (lehrt die Erfahrung).

Bias und Varianz sind Kontrahenten: Ein Modell, das wenig Bias hat, neigt tendenziell zu wenig Varianz und umgekehrt, vgl. Abb. 3.18 aus Sauer (2019).

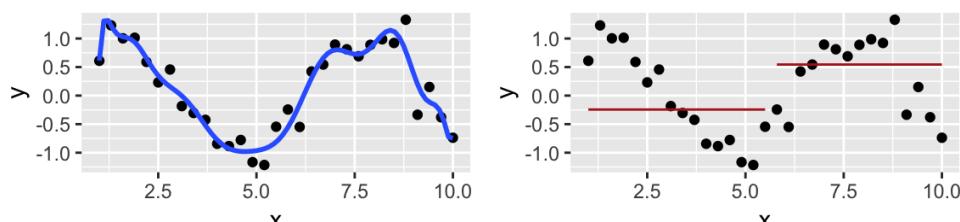


Abbildung 3.18: Abwägung von Bias vs. Varianz

## 3.10 Aufgaben

- Machen Sie sich mit ‘Kaggle’ vertraut (<https://www.kaggle.com/>)
- Bearbeiten Sie die Fallstudie ‘TitaRnic’ auf Kaggle (<https://www.kaggle.com/code/headsortails/tidy-titarnic/report>)
- Machen Sie sich mit dieser einfachen Fallstudie zur linearen Regression vertraut: The Movie Data Base Revenue (Kaggle) (<https://www.kaggle.com/code/ssauer/notebook9188bfa616>)

## 3.11 Vertiefung

- Verdienst einer deutschen Data Scientistin (<https://www.zeit.de/arbeit/2020-10/data-scientist-gehalt-geldanlage-programmieren-kontoauszug>)
- Weitere Fallstudie zum Thema Regression auf Kaggle (<https://www.kaggle.com/micahshull/r-bike-sharing-linear-regression>)
- Crashkurs Data Science (Coursera, Johns Hopkins University) mit ‘Star-Dozenten’ (<https://www.coursera.org/learn/data-science-course>)
- Arbeiten Sie diese Regressionsfallstudie (zum Thema Gehalt) auf Kaggle auf (<https://www.kaggle.com/pranjalpandey12/performing-simple-linear-regression-in-r>)
- Werfen Sie einen Blick in diese Fallstudie auf Kaggle zum Thema Hauspreise (<https://www.kaggle.com/lazaro97/data-preprocessing-and-linear-regression-with-r>)
- Wiederholen Sie unser Vorgehen in der Fallstudie zu den Flugverspätungen (<https://data-se.netlify.app/2021/03/10/fallstudie-modellierung-von-flugversp%C3%A4tungen/>)

## 4 R, zweiter Blick

Benötigte R-Pakete für dieses Kapitel:

```
library(tidyverse)
library(tidymodels)
```

## 4.1 Lernsteuerung

### 4.1.1 Vorbereitung

- Lesen Sie die Literatur.

### 4.1.2 Lernziele

- Sie können Funktionen, in R schreiben.
- Sie können Datensätze vom Lang- und Breit-Format wechseln.
- Sie können Wiederholungsstrukturen wie Mapping-Funktionen anwenden.
- Sie können eine dplyr-Funktion auf mehrere Spalten gleichzeitig anwenden.

### 4.1.3 Literatur

- Rhys, Kap. 2
- MODAR, Kap. 5

## 4.2 Objekttypen in R

Näheres zu Objekttypen findet sich in Sauer (2019), Kap. 5.2.

### 4.2.1 Überblick

In R ist praktisch alles ein Objekt. Ein Objekt meint ein im Computerspeicher repräsentiertes Ding, etwa eine Tabelle.

Vektoren und Dataframes (Tibbles) sind die vielleicht gängigsten Objektarten in R (vgl. Abb. 4.1, aus Sauer (2019)).

## Vektoren

3	Berta	2
1		
1		
2		
2		
2		

ID	Name	Note1
1	Anna	1
2	Anna	1
3	Berta	2
4	Carla	2
...	...	...

Abbildung 4.1: Zentrale Objektarten in R

Es gibt in R keine (Objekte für) Skalare (einzelne Zahlen). Stattdessen nutzt R Vektoren der Länge 1.

Ein nützliches Schema stammt aus Wickham and Grolemund (2016), s. Abb. 4.2.

## Vectors

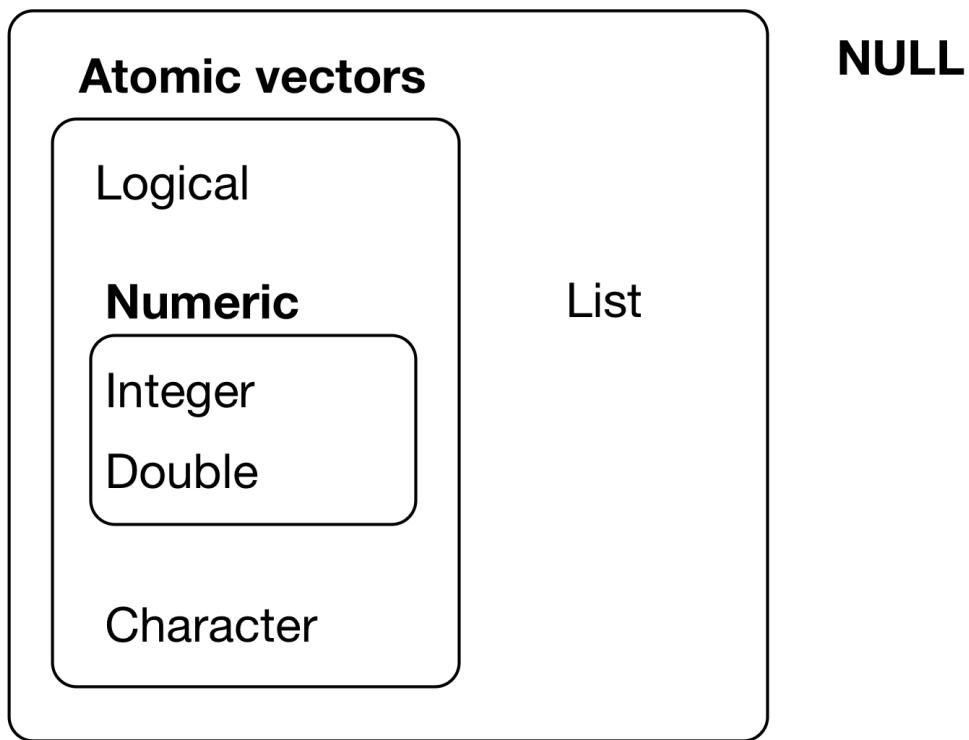


Abbildung 4.2: Objektarten hierarchisch gegliedert

### 4.2.2 Taxonomie

Unter *homogenen* Objektiven verstehen wir Datenstrukturen, die nur eine Art von Daten (wie Text oder Ganze Zahlen) fassen. Sonstige Objekte nennen wir *heterogen*.

- Homogene Objekte
  - Vektoren
  - Matrizen
- Heterogen
  - Liste
  - Dataframes (Tibbles)

#### 4.2.2.1 Vektoren

Vektoren sind insofern zentral in R, als dass die übrigen Datenstrukturen auf ihnen aufbauen, vgl. Abb. 4.3 aus Sauer (2019).

Reine (atomare) Vektoren in R sind eine geordnete Liste von Daten eines Typs.

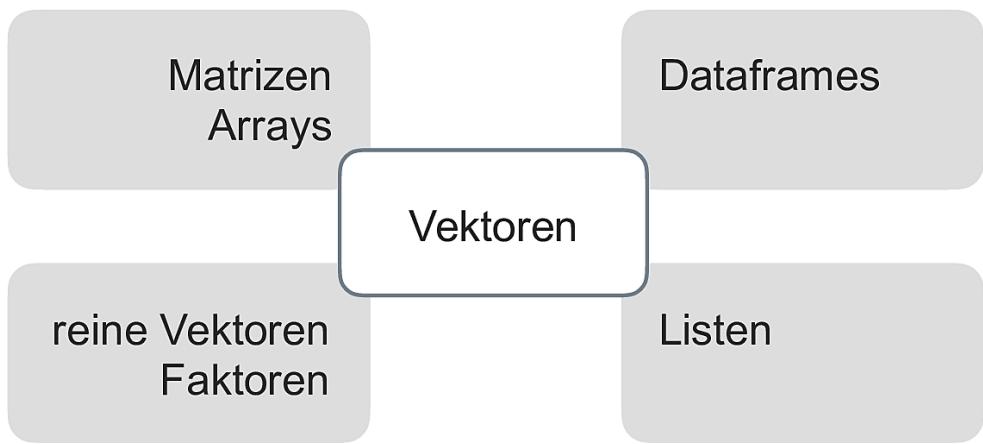


Abbildung 4.3: Vektoren stehen im Zentrum der Datenstrukturen in R

```

ein_vektor <- c(1, 2, 3)
noch_ein_vektor <- c("A", "B", "C")
logischer_vektor <- c(TRUE, FALSE, TRUE)
  
```

Mit `str()` kann man sich die Struktur eines Objekts ausgeben lassen:

```

str(ein_vektor)

## num [1:3] 1 2 3

str(noch_ein_vektor)

## chr [1:3] "A" "B" "C"

str(logischer_vektor)

## logi [1:3] TRUE FALSE TRUE
  
```

Vektoren können von folgenden Typen sein:

- Kommazahlen (`double`) genannt
- Ganzzahlig (`integer`, auch mit `L` für `Long` abgekürzt)
- Text (`character`, `String`)
- logische Ausdrücke (`logical` oder `lg1`) mit `TRUE` oder `FALSE`

Kommazahlen und Ganze Zahlen zusammen bilden den Typ `numeric` (numerisch) in R.

```
knitr:::opts_chunk$set(echo = TRUE)
```

Den Typ eines Vektors kann man mit `typeof()` ausgeben lassen:

```

typeof(ein_vektor)

## [1] "double"
  
```

#### 4.2.2.2 Faktoren

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

Interessant:

```

str(sex)

## Factor w/ 2 levels "Frau", "Mann": 2 1 1
  
```

Vertiefende Informationen findet sich in Wickham and Grolemund (2016).

#### 4.2.2.3 Listen

```
eine_liste <- list(titel = "Einführung",
                    woche = 1,
                    datum = c("2022-03-14", "2202-03-21"),
                    lernziele = c("dies", "jenes", "und noch mehr"),
                    lehre = c(TRUE, TRUE, TRUE)
)
str(eine_liste)
```

```
## List of 5
## $ titel      : chr "Einführung"
## $ woche      : num 1
## $ datum      : chr [1:2] "2022-03-14" "2202-03-21"
## $ lernziele: chr [1:3] "dies" "jenes" "und noch mehr"
## $ lehre      : logi [1:3] TRUE TRUE TRUE
```

#### 4.2.2.4 Tibbles

Für `tibble()` brauchen wir `tidyverse`:

```
library(tidyverse)
```

```
studentis <-  
  tibble(  
    name = c("Anna", "Berta"),  
    motivation = c(10, 20),  
    noten = c(1.3, 1.7)  
  )  
str(studentis)
```

```
## # tibble [2 x 3] (S3:tbl_df/tbl/data.frame)
## # $ name      : chr [1:2] "Anna" "Berta"
## # $ motivation: num [1:2] 10 20
## # $ noten     : num [1:2] 1.3 1.7
```

### 4.2.3 Indizieren

Einen Teil eines Objekts auszulesen, bezeichnen wir als *Indizieren*.

### 4.2.3.1 Reine Vektoren

Zur Erinnerung:

`str(ein_vektor)`

```
## num [1:3] 1 2 3
```

ein vektor[1]

## [1] 1

```
ein vektor[s(1-2)]
```

## [1] 1 2

ein Vektor [1 3]

## Error in air.vektor[1: 21]: incorrect number of dimensions

Man darf Vektoren auch wie Listen ansprechen, also eine doppelte Eckklammer zum Indizieren verwenden.

ein vektor[[2]]

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

Der Grund ist, dass Listen auch Vektoren sind, nur eben ein besonderer Fall eines Vektors:

```
is.vector(eine_liste)
```

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

Was passiert, wenn man bei einem Vektor der Länge 3 das 4. Element indiziert?

```
ein_vektor[4]
```

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

Ein schnödes `NA` ist die Antwort. Das ist interessant: Wir bekommen keine Fehlermeldung, sondern den Hinweis, das angesprochene Element sei leer bzw. nicht verfügbar.

In Sauer (2019), Kap. 5.3.1 findet man weitere Indizierungsmöglichkeiten für reine Vektoren.

#### 4.2.3.2 Listen

```
eine_liste %>% str()
```

```
## List of 5
## $ titel    : chr "Einführung"
## $ Woche   : num 1
## $ datum    : chr [1:2] "2022-03-14" "2202-03-21"
## $ lernziele: chr [1:3] "dies" "jenes" "und noch mehr"
## $ lehre    : logi [1:3] TRUE TRUE TRUE
```

Listen können wie Vektoren, also mit `[` ausgelesen werden. Dann wird eine Liste zurückgegeben.

```
eine_liste[1]
```

```
## $titel
## [1] "Einführung"
```

```
eine_liste[2]
```

```
## $woche
## [1] 1
```

Das hat den technischen Hintergrund, dass Listen als eine bestimmte Art von Vektoren implementiert sind.

Mann kann auch die “doppelte Eckklammer”, `[[` zum Auslesen verwenden; dann wird anstelle einer Liste die einfachere Struktur eines Vektors zurückgegeben:

```
eine_liste[[1]]
```

```
## [1] "Einführung"
```

Man könnte sagen, die “äußere Schicht” des Objekts, die Liste, wird abgeschält, und man bekommt die “innere” Schicht, den Vektor.

Mann die Elemente der Liste entweder mit ihrer Positionsnummer (1, 2, ...) oder, sofern vorhanden, ihren Namen ansprechen:

```
eine_liste[["titel"]]
```

```
## [1] "Einführung"
```

Dann gibt es noch den Dollar-Operator, mit dem Mann benannte Elemente von Listen ansprechen kann:

```
eine_liste$titel
```

```
## [1] "Einführung"
```

Man kann auch tiefer in eine Liste hinein indizieren. Sagen wir, uns interessiert das 4. Element der Liste `eine_liste` - und davon das erste Element.

Das geht dann so:

```
eine_liste[[4]][[1]]
```

```
## [1] "dies"
```

Eine einfachere Art des Indizierens von Listen bietet die Funktion `pluck()`, aus dem Paket `purrr`, das Hilfen für den Umgang mit Listen bietet.

```
pluck(eine_liste, 4)
```

```
## [1] "dies"           "jenes"          "und noch mehr"
```

Und jetzt aus dem 4. Element das 1. Element:

```
pluck(eine_liste, 4, 1)
```

```
## [1] "dies"
```

Probieren Sie mal, aus einer Liste der Länge 5 das 6. Element auszulesen:

```
eine_liste %>% length()
```

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

```
eine_liste[[6]]
```

```
## Error in eine_liste[[6]]: subscript out of bounds
```

Unser Versuch wird mit einer Fehlermeldung quittiert.

Sprechen wir die Liste wie einen (atomaren) Vektor an, bekommen wir hingegen ein `NA` bzw. ein `NULL`:

```
eine_liste[6]
```

```
## $<NA>
## NULL
```

### 4.2.3.3 Tibbles

Tibbles lassen sich sowohl wie ein Vektor als auch wie eine Liste indizieren.

```
studentis[1]
```

```
## # A tibble: 2 × 1
##   name
##   <chr>
## 1 Anna
## 2 Berta
```

Die Indizierung eines Tibbles mit der einfachen Eckklammer liefert einen Tibble zurück.

```
studentis["name"]
```

```
## # A tibble: 2 × 1
##   name
##   <chr>
## 1 Anna
## 2 Berta
```

Mit doppelter Eckklammer bekommt man, analog zur Liste, einen Vektor zurück:

```
studentis[[ "name" ]]
```

```
## [1] "Anna"  "Berta"
```

Beim Dollar-Operator kommt auch eine Liste zurück:

```
studentis$name
```

```
## [1] "Anna"  "Berta"
```

## 4.2.4 Weiterführende Hinweise

- Tutorial ([https://jennybc.github.io/purrr-tutorial/bk00\\_vectors-and-lists.html](https://jennybc.github.io/purrr-tutorial/bk00_vectors-and-lists.html)) zum Themen Indizieren von Listen von Jenny BC.

## 4.2.5 Indizieren mit dem Tidyverse

Natürlich kann man auch die Tidyverse-Verben zum Indizieren verwenden. Das bietet sich an, wenn zwei Bedingungen erfüllt sind:

1. Wenn man einen Tibble als Input und als Output hat
2. Wenn man nicht programmieren möchte

## 4.3 Datensätze von lang nach breit umformatieren

Manchmal findet man Datensätze im sog. *langen* Format vor, manchmal im *breiten*.

In der Regel müssen die Daten “tidy” sein, was meist dem langen Format entspricht, vgl. Abb. 4.4 aus Sauer (2019).

```
knitr:::include_graphics("img/gather_spread.png")
```

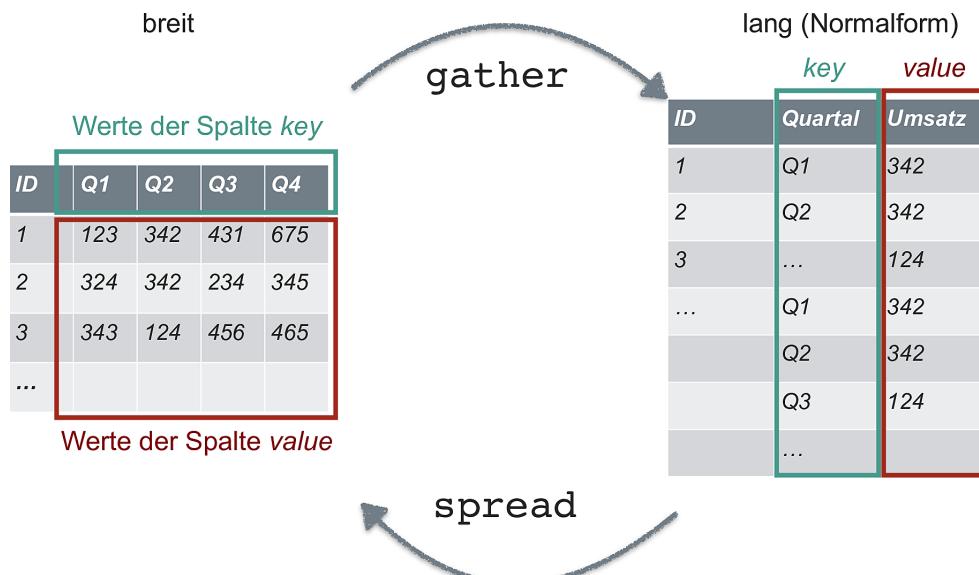


Abbildung 4.4: Von lang nach breit und zurück

In einer neueren Version des Tidyverse werden diese beiden Befehle umbenannt bzw. erweitert:

- *gather()* -> *pivot\_longer()*
- *spread()* -> *pivot\_wider()*

Weitere Informationen findet sich in Wickham and Grolemund (2016), in diesem Abschnitt, 12.3 ([https://r4ds.had.co.nz/tidy-data.html?q=pivot\\_%23pivoting](https://r4ds.had.co.nz/tidy-data.html?q=pivot_%23pivoting)).

## 4.4 Funktionen

Eine Funktion kann man sich als analog zu einer Variable vorstellen. Es ist ein Objekt, das nicht Daten, sondern Syntax beinhaltet, vgl. Abb. 4.5 aus Sauer (2019).

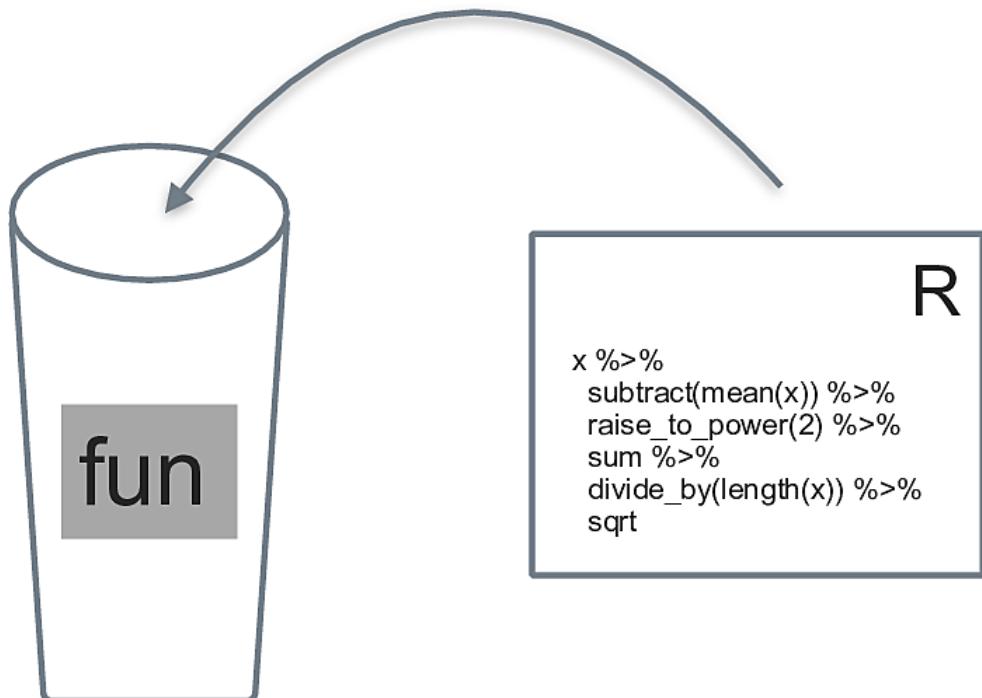


Abbildung 4.5: Sinnbild einer Funktion

```
mittelwert <- function(x){

  summe <- sum(x, na.rm = TRUE)
  mw <- summe/length(x)
  return(mw)

}
```

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

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

Weitere Informationen finden sich in Kapitel 19 (<https://r4ds.had.co.nz/functions.html>) in Wickham and Grolemund (2016). Alternativ findet sich ein Abschnitt dazu (28.1) in Sauer (2019).

## 4.5 Wiederholungen programmieren

Häufig möchte man eine Operation mehrfach ausführen. Ein Beispiel wäre die Anzahl der fehlenden Werte pro Spalte auslesen. Natürlich kann man die Abfrage einfach häufig tippen, nervt aber irgendwann. Daher braucht's Strukturen, die *Wiederholungen* beschreiben.

Dafür gibt es verschiedene Ansätze.

### 4.5.1 across()

Handelt es sich um Spalten von Tibbles, dann bietet sich die Funktion `across(.col, .fns)` an. `across` wendet eine oder mehrere Funktionen (mit `.fns` bezeichnet) auf die Spalten `.col` an.

Das erklärt sich am besten mit einem Beispiel:

Natürlich hätte man in diesem Fall auch anders vorgehen können:

```
mtcars %>%
  summarise(across(.cols = everything(),
    .fns = mean))
```

```
##      mpg     cyl     disp      hp      drat      wt      qsec      vs      am
## 1 20.09062 6.1875 230.7219 146.6875 3.596563 3.21725 17.84875 0.4375 0.40625
##   gear   carb
## 1 3.6875 2.8125
```

Möchte man der Funktion `.fns` Parameter übergeben, so nutzt man diese Syntax ("Purrr-Lambda"):

```
mtcars %>%
  summarise(across(.cols = everything(),
    .fns = ~ mean(., na.rm = TRUE)))
```

```
##      mpg cyl   disp   hp drat   wt  qsec   vs   am
## 1 20.09062 6.1875 230.7219 146.6875 3.596563 3.21725 17.84875 0.4375 0.40625
##   gear carb
## 1 3.6875 2.8125
```

Hier (<https://www.rebeccabarter.com/blog/2020-07-09-across/>) findet sich ein guter Überblick zu `across()`.

## 4.5.2 `map()`

`map()` ist eine Funktion aus dem R-Paket `purrr` und Teil des Tidyverse.

`map(x, f)` wenden die Funktion `f` auf jedes Element von `x` an. Ist `x` ein Tibble, so wird `f` demnach auf jede Spalte von `x` angewendet (“zugeordnet”, daher `map`), vgl. Abb. 4.6 aus Sauer (2019).

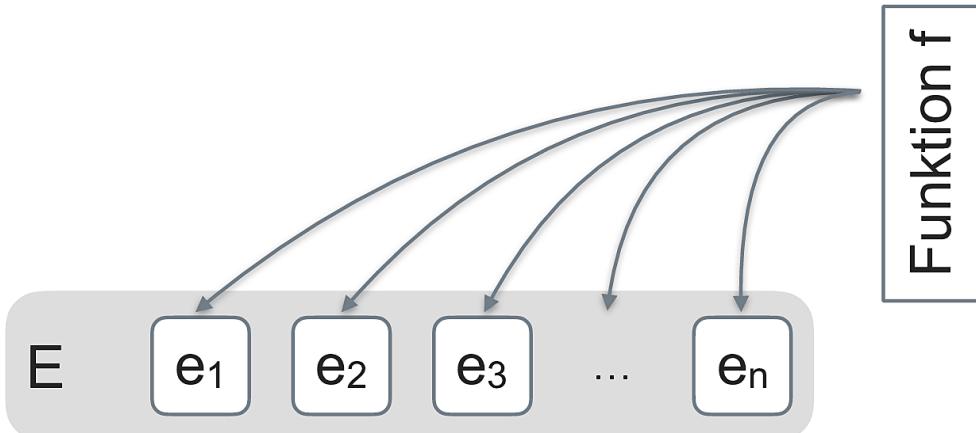


Abbildung 4.6: Sinnbild für `map`

Hier ein Beispiel-Code:

```
data(mtcars)

mtcars <- mtcars %>% select(1:3) # nur die ersten 3 Spalten

map(mtcars, mean)
```

```
## $mpg
## [1] 20.09062
##
## $cyl
## [1] 6.1875
##
## $disp
## [1] 230.7219
```

Möchte man der gemappten Funktion Parameter übergeben, nutzt man wieder die “Kringel-Schreibweise”:

```
map(mtcars, ~ mean(., na.rm = TRUE))
```

```
## $mpg
## [1] 20.09062
##
## $cyl
## [1] 6.1875
##
## $disp
## [1] 230.7219
```

## 4.5.3 Weiterführende Hinweise

Weiteres zu `map()` findet sich z.B. in Wickham and Grolemund (2016), Kapitel 21.5 (<https://r4ds.had.co.nz/iteration.html#the-map-functions>) oder in Sauer (2019), Kap. 28.2.

## 4.6 Listenspalten

### 4.6.1 Wozu Listenspalten?

Listenspalten sind immer dann sinnvoll, wenn eine einfache Tabelle nicht komplex genug für unsere Daten ist.

Zwei Fälle stechen dabei ins Auge:

1. Unsere Datenstruktur ist nicht rechteckig
2. In einer Zelle der Tabelle soll mehr als ein einzelner Wert stehen: vielleicht ein Vektor, eine Liste oder eine Tabelle

Der erstere Fall (nicht rechteckig) ließe sich noch einfach lösen, in dem man mit `NA` auffüllt.

Der zweite Fall verlangt schlichtweg nach komplexeren Datenstrukturen.

Kap. 25.3 (<https://r4ds.had.co.nz/many-models.html?q=list#creating-list-columns>) aus Wickham and Grolemund (2016) bietet einen guten Einstieg in das Konzept von Listenspalten (list-columns) in R.

### 4.6.2 Beispiele für Listenspalten

#### 4.6.2.1 tidymodel

Wenn wir mit `tidymodels` arbeiten, werden wir mit Listenspalten zu tun haben. Daher ist es praktisch, sich schon mal damit zu beschäftigen.

Hier ein Beispiel für eine  $v = 3$ -fache Kreuzvalidierung:

```
library(tidymodels)
mtcars_cv <-
  vfold_cv(mtcars, v = 3)

mtcars_cv
```

```
## # 3-fold cross-validation
## # A tibble: 3 × 2
##   splits      id
##   <list>      <chr>
## 1 <split [21/11]> Fold1
## 2 <split [21/11]> Fold2
## 3 <split [22/10]> Fold3
```

Betrachten wir das Objekt `mtcars_cv` näher. Die Musik spielt in der 1. Spalte.

Lesen wir den Inhalt der 1. Spalte, 1 Zeile aus (nennen wir das mal “Position 1,1”):

```
pos11 <- mtcars_cv[[1]][[1]]
pos11
```

```
## <Analysis/Assess/Total>
## <21/11/32>
```

In dieser Zelle findet sich eine Aufteilung des Komplettdatensatzes in den Analyseteil (Analysis sample) und den Assessmentteil (Assessment Sample).

Schauen wir jetzt in dieses Objekt näher an. Das können wir mit `str()` tun. `str()` zeigt uns die Struktur eines Objekts.

```
str(pos11)
```

```
## List of 4
## $ data : 'data.frame': 32 obs. of 3 variables:
##   ..$ mpg : num [1:32] 21 21 22.8 21.4 18.7 ...
##   ..$ cyl : num [1:32] 6 6 4 6 8 4 4 6 ...
##   ..$ disp: num [1:32] 160 160 108 258 360 ...
##   $ in_id : int [1:21] 1 3 4 5 7 10 12 13 15 16 ...
##   $ out_id: logi NA
##   $ id    : tibble [1 × 1] (S3: tbl_df/tbl/data.frame)
##     ..$ id: chr "Fold1"
##   - attr(*, "class")= chr [1:2] "vfold_split" "rsplit"
```

Oh! `pos11` ist eine Liste, und zwar eine durchaus komplexe. Wir müssen erkennen, dass in einer einzelnen Zelle dieses Dataframes viel mehr steht, als ein Skalar bzw. ein einzelnes, atomares Element.

Damit handelt es sich bei Spalte 1 dieses Dataframes (`mtcars_cv`) also um eine Listenspalte.

Üben wir uns noch etwas im Indizieren.

Sprechen wir in `pos11` das erste Element an (`data`) und davon das erste Element:

```
pos11[["data"]][[1]]
```

```
## [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2 10.4
## [16] 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7
## [31] 15.0 21.4
```

Wir haben hier die doppelten Eckklammern benutzt, um den “eigentlichen” oder “inneren” Vektor zu bekommen, nicht die “außen” herumgewickelte Liste. Zur Erinnerung: Ein Dataframe ist ein Spezialfall einer Liste, also auch eine Liste, nur eine mit bestimmten Eigenschaften.

Zum Vergleich indizieren wir mal mit einer einfachen Eckklammer:

```
pos11[["data"]][1] %>%
  head()
```

```
##          mpg
## Mazda RX4     21.0
## Mazda RX4 Wag 21.0
## Datsun 710    22.8
## Hornet 4 Drive 21.4
## Hornet Sportabout 18.7
## Valiant       18.1
```

Mit `pluck()` bekommen wir das gleiche Ergebnis, nur etwas komfortabler, da wir keine Eckklammern tippen müssen:

```
pluck(pos11, "data", 1, 1)
```

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

Wie man sieht, können wir beliebig tief in das Objekt hineinindizieren.

#### 4.6.2.2 Kurs DataScience1

Ein Kurs, wie dieser, kann anhand einer “Deskriptoren” wie Titel der Inhalte, Lernziele, Literatur und so weiter zusammengefasst werden. Diese Deskriptoren kann man wiederum jeder Kurswoche oder jedem Kursabschnitt zuordnen, so dass eine zweidimensionale Struktur resultiert. Eine Tabelle, einfach gesagt, etwa so:

```
tibble::tribble(
  ~Nr, ~Titel, ~Literatur, ~Aufgaben,
  1L, "Titel1", "Literatur1", "Aufgaben1",
  2L, "Titel2", "Literatur2", "Aufgaben2",
  3L, "Titel3", "Literatur3", "Aufgaben3"
) %>%
  gt::gt()
```

	Nr	Titel	Literatur	Aufgaben
1	Titel1	Literatur1	Aufgaben1	
2	Titel2	Literatur2	Aufgaben2	
3	Titel3	Literatur3	Aufgaben3	

Wie man sieht, entspricht jede Spalte einem Deskriptor des Kurses, und jede Zeile entspricht einem Thema (oder Woche oder Abschnitt) des Kurses.

Jetzt ist es nur so, dass einzelne Zellen dieser Tabelle nicht aus nur einem Element bestehen. So könnte etwa “Aufgaben1” aus mehreren Aufgaben bestehen, die jeweils wiederum aus mehreren (Text-)Elementen bestehen. Oder “Literatur2” besteht vielleicht aus zwei Literaturquellen.

Kurz gesagt, wir brauchen eine Tabelle, die erlaubt, in einer Zelle mehr als ein einzelnes Element zu packen. Listenspalten erlauben das.

Schauen wir uns die "Mastertabelle" dieses Kurses an zur Illustration.

Zunächst source n wir die nötigen Funktionen.

```
source("https://raw.githubusercontent.com/sebastiansauer/Lehre/main/R-Code/render-course-sections.R")
```

In Ihrem Environment sollten Sie jetzt die gesourceten Funktionen sehen. Mit Klick auf den Funktionsnamen können Sie diese Funktionen auch betrachten.

Die Deskriptoren des Kurses speisen sich aus zwei Textdateien, gespeichert im sog. YAML-Format, ein einfaches Textformat, und hier nicht weiter von Belang.

Zum einen eine Datei mit den Datumsangaben:

```
course_dates_file <- "https://raw.githubusercontent.com/sebastiansauer/datascience1/main/course-dates.yaml"
```

Zum anderen eine Datei mit den Deskriptoren, die unabhängig vom Datum sind:

```
content_file <- "https://raw.githubusercontent.com/sebastiansauer/datascience1/main/_modul-ueberblick.yaml"
```

Im Github-Repo (<https://github.com/sebastiansauer/datascience1>) dieses Kurses können Sie die Dateien komfortabel betrachten.

Die "Mastertabelle" kann man mit folgender Funktion erstellen:

```
mastertable <- build_master_course_table(  
  course_dates_file = course_dates_file,  
  content_file = content_file)
```

Betrachten Sie die Tabelle in Ruhe! Sie werden sehen, dass einige Spalten komplex sind, also mehr als nur einen einzelnen Wert enthalten:

```
mastertable[["Vorbereitung"]][[1]]
```

```
## [1] "Lesen Sie die Hinweise zum Modul."  
## [2] "Installieren (oder Updaten) Sie die für dieses Modul angegebene Software."  
## [3] "Lesen Sie die Literatur."
```

Gerade haben wir aus dem Objekt mastertable, ein Dataframe, die Spalte mit dem Namen *Vorbereitung* ausgelesen und aus dieser Spalte das erste Element. Dieses erste Element ist ein Textvektor der Länge 3.

Daraus könnten wir z.B. das zweite Element auslesen:

```
mastertable[["Vorbereitung"]][[1]][2]
```

```
## [1] "Installieren (oder Updaten) Sie die für dieses Modul angegebene Software."
```

Was würde passieren, wenn wir anstelle der doppelten Eckklammer einfache Eckklammern verwenden würden?

```
mastertable["Vorbereitung"] %>% class()
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

```
mastertable[["Vorbereitung"]] %>% class()
```

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

Das macht noch keinen großen Unterschied, aber schauen wir mal weiter.

Wenn wir das erste Element der Spalte "Vorbereitung" mit *doppelter* Eckklammer ansprechen, bekommen wir einen Text-Vektor (der Länge drei) zurück.

```
mastertable[["Vorbereitung"]][[1]]
```

```
## [1] "Lesen Sie die Hinweise zum Modul."  
## [2] "Installieren (oder Updaten) Sie die für dieses Modul angegebene Software."  
## [3] "Lesen Sie die Literatur."
```

Jetzt können wir, wie oben getan, diese einzelnen Elemente ansprechen.

Aber: Wenn wir das erste Element der Spalte "Vorbereitung" mit einfacher Eckklammer ansprechen, bekommen wir eine Liste *mit einem Element* zurück.

```
mastertable[["Vorbereitung"]][1]
```

```
## [[1]]
## [1] "Lesen Sie die Hinweise zum Modul."
## [2] "Installieren (oder Updaten) Sie die für dieses Modul angegeben Software."
## [3] "Lesen Sie die Literatur."
```

Wir können also nicht (ohne weiteres "Abschälen") z.B. das zweite Element des Text-Vektors ("Installieren Sie...") auslesen:

```
mastertable[["Vorbereitung"]][1] %>% pluck(2)
```

```
## NULL
```

Wenn Sie sich über `pluck()` wundern, Sie hätten synonym auch schreiben können:

```
mastertable[["Vorbereitung"]][1][2]
```

```
## [[1]]
## NULL
```

Da die Liste nur aus einem Element besteht, könnten wir z.B. nicht das zweite Element der Liste ansprechen:

```
mastertable[["Vorbereitung"]][1][[2]]
```

```
## Error in mastertable[["Vorbereitung"]][1][[2]]: subscript out of bounds
```

Haben wir aber die doppelte Eckklammer verwendet, so bekommen wir einen Vektor der Länge drei zurück (vom Typ Text), und daher können wir die Elemente 1 bis 3 ansprechen:

```
mastertable[["Vorbereitung"]][1][[2]]
```

```
## [1] "Installieren (oder Updaten) Sie die für dieses Modul angegeben Software."
```

Dabei ist es egal, ob Sie einfache oder doppelte Eckklammern benutzen, da Listen auch Vektoren sind.

## 4.6.3 Programmieren mit dem Tidyverse

Das Programmieren mit dem Tidyvers ist nicht ganz einfach und hier nicht näher ausgeführt. Eine Einführung findet sich z.B.

- Tidyeval in fünf Minuten (Video) (<https://www.youtube.com/watch?v=nERXS3ssntw>)
- In Kapiteln 17-21 in Advanced R, 2nd Ed (<https://adv-r.hadley.nz/>)
- Ein Überblicksdiagramm findet sich hier (<https://twitter.com/lapply/status/1493908215796535296/photo/1>) Quelle (<https://twitter.com/lapply/status/1493908215796535296?t=P0SbLJAd0Yd97hYPzNMxMg&s=09>).

## 4.7 R ist schwierig

Manche behaupten, R sei ein Inferno ([https://www.burns-stat.com/pages/Tutor/R\\_inferno.pdf](https://www.burns-stat.com/pages/Tutor/R_inferno.pdf)).

Zum Glück gibt es auch aufmunternde Stimmen:

```
praise::praise()
```

```
## [1] "You are epic!"
```

Hat jemand einen guten Rat für uns? Vielleicht ist der häufigste Rate, dass man die Dokumentation lesen solle (<https://en.wikipedia.org/wiki/RTFM>).

## 4.8 Aufgaben

### 1. Aufgabe

Schreiben Sie eine Funktion, mit folgendem Algorithmus, wobei ein beliebiger Vektor als Eingabe verlangt wird.

1. Zähle die Anzahl verschiedener Werte.
2. Wenn es nur zwei verschiedene Werte gibt, gebe `TRUE` zurück, ansonsten `FALSE`.

Hinweise:

- Wählen Sie einen treffenden Namen für Ihre Funktion (nutzen Sie am besten ein konsistentes Namensschema).
- Wichtigster Tipp: Googeln :-)
- Verschiedene Werte eines Vektors gibt die Funktion `unique()` zurück.
- Vermutlich gibt es schon viele Lösungen (Implementierungen) für diese Funktion. Ist nur als Übung gedacht :-)

## Lösung

```
has_two_levels <- function(vec){

  # input: vector of any type
  # value: number of unique values (double)

  tmp <- length(unique(vec))
  tmp == 2
}
```

`levels` ist ein Ausdruck, der nahelegt, dass es sich um *verschiedene* Werte ("Ausprägungen" oder "Stufen") handelt.

Alternativ könnte man die Funktion auch so schreiben:

```
has_two_values <- function(vec){

  # input: vector of any type
  # value: number of unique values (double)

  step1 <- unique(vec)
  step2 <- length(step1)
  step3 <- if(step2 == 2) {
    out <- TRUE
  } else {
    out <- FALSE
  }

  out <- step2
  return(out)
}
```

Das ist expliziter, aber länger.

Wenn man es genau nimmt, heißt *binär*, dass es nur die Werte `0` und `1` gibt. Das ist ein strengeres Kriterium, wie dass es zwei beliebigen verschiedenen Werte gibt (s. unten dazu ein Vorschlag).

Testen wir unsere Funktion, Test 1:

```
x <- c(1,2, 3, 3, 3, 1)
x2 <- c("A", "B")
```

```
has_two_levels(x2)
```

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

```
has_two_levels(x)
```

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

Test 2:

```
data(mtcars)
```

Wir wenden unsere neue Funktion mit Tidyverse-Methoden an:

```
mtcars %>%
  summarise(am_has_two_values = has_two_levels(am),
            mpg_has_two_values = has_two_levels(mpg))
```

```
##   am_has_two_values mpg_has_two_values
## 1             TRUE                 FALSE
```

Bonus!

Verwenden Sie `across()` (`{dplyr}`), um alle Spalten von `mtcars` mit `has_two_levels()` zu überprüfen.

```
mtcars %>%
  summarise(across(everything(),
                  has_two_levels))
```

```
##   mpg cyl disp hp drat wt
## 1 FALSE FALSE FALSE FALSE FALSE FALSE
##   qsec vs am gear carb
## 1 FALSE TRUE TRUE FALSE FALSE
```

Kann man auch so schreiben:

```
mtcars %>%
  summarise(across(everything(),
                  ~ has_two_levels(.)))
```

```
##   mpg cyl disp hp drat wt
## 1 FALSE FALSE FALSE FALSE FALSE FALSE
##   qsec vs am gear carb
## 1 FALSE TRUE TRUE FALSE FALSE
```

Bonus-Bonus:

So könnte man eine Funktion schreiben, die prüft, ob die Ausprägungen eines Vektors binär sind, d.h. nur `0` oder `1`:

```
is_binary <- function(var){
  return(all(var %in% c(0,1)))
}
```

## 2. Aufgabe

Schreiben Sie eine Funktion zur Berechnung der Anzahl der fehlenden Werte in einem (numerischen) Vektor!

Hinweise:

- Wählen Sie einen treffenden Namen für Ihre Funktion (nutzen Sie am besten ein konsistentes Namensschema).

### Lösung

```
na_n <- function(num_vec) {
  # input: num. vector (int or double)
  # value: count of missing values (double)

  if (!is.numeric(num_vec)) stop("Numeric input is needed!")
  out <- sum(is.na(num_vec))
  return(out)
}
```

Test:

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

```
x2 <- c("A", "B", NA)
```

```
na_n(x)
```

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

Bei `x2` sollte ein Fehler aufgeworfen werden:

```
na_n(x2)
```

```
## Error in na_n(x2): Numeric input is needed!
```

Gut!

Testen wir weiter, jetzt mit dem Datensatz `mtcars`:

```
mtcars[1, c(1,2,3,4)] <- NA # Zeilen/Spalte
```

```
mtcars %>%
  summarise(mpg_na_n = na_n(mpg))
```

```
##   mpg_na_n
## 1          1
```

BONUS!

Verwenden Sie `across()`, um die fehlenden Werte in allen Spalten von `mtcars` zu zählen.

Wer schnell ist, der nehme gerne `nycflights13::flights` (aus dem Paket `{nycflights13}` oder per CSV-Datei aus geeigneter Stelle aus dem Internet. Meistens geht es schneller, die Daten aus einem Paket zu laden mit `data(flights)` nachdem man `library(nycflights13)` geschrieben hat).

```
mtcars %>%
  summarise(across(everything(), na_n))
```

```
##   mpg cyl disp hp drat wt qsec vs am
## 1   1    1    1    1    0    0    0    0    0
##   gear carb
## 1     0    0
```

Mit `pivot_longer()` ist es häufig übersichtlicher bzw. besser für weitere Bearbeitungsschritte, wie das folgende Beispiel zeigt:

```
mtcars %>%
  summarise(across(everything(), na_n)) %>%
  pivot_longer(everything()) %>%
  filter(value > 0)
```

```
## # A tibble: 4 × 2
##   name   value
##   <chr> <int>
## 1 mpg      1
## 2 cyl      1
## 3 disp     1
## 4 hp       1
```

`flights`:

```
data(flights, package = "nycflights13")

flights %>%
  select(where(is.numeric)) %>%
  summarise(across(everything(),
                  na_n)) %>%
  pivot_longer(everything()) %>%
  arrange(-value) %>% # Sortiere absteigend nach Anzahl der fehlenden Werte
  filter(value > 0) # Zeige nur Variablen mit fehlenden Werten
```

```
## # A tibble: 5 × 2
##   name   value
##   <chr> <int>
## 1 arr_delay 9430
## 2 air_time   9430
## 3 arr_time   8713
## 4 dep_time   8255
## 5 dep_delay  8255
```

### 3. Aufgabe

Erstellen Sie für jede Variable aus `mtcars` ein Histogramm!

Hinweise:

- Nutzen Sie eine Wiederholungsstruktur. Schreiben Sie prägnante Syntax.
- Optional: Lassen Sie dichotome (zweiwertige) Variablen aus.
- Hier (<https://data-se.netlify.app/2021/02/06/plotting-multiple-plots-using-purrr-map-and-ggplot/>) (und an vielen anderen Stellen im Netz) finden Sie Tipps.

### Lösung

```
has_two_levels <- function(vec){

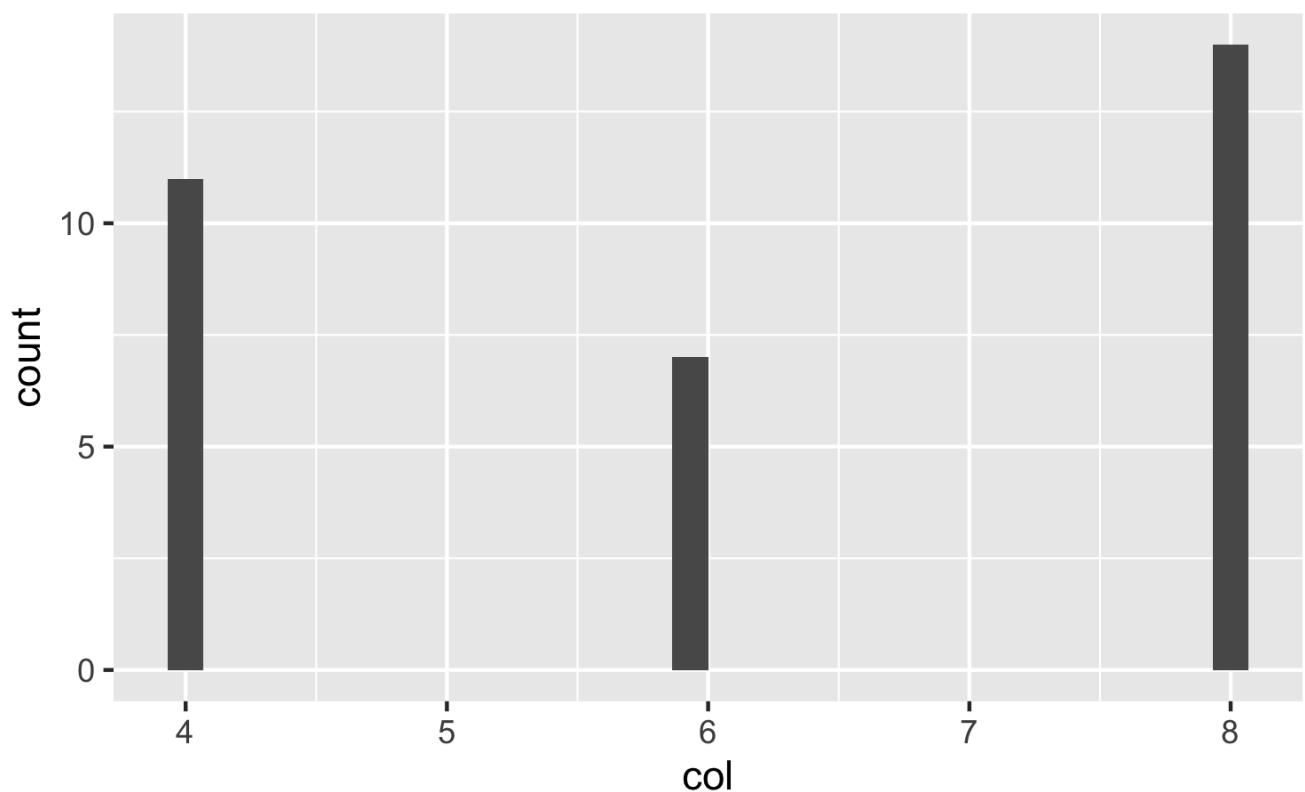
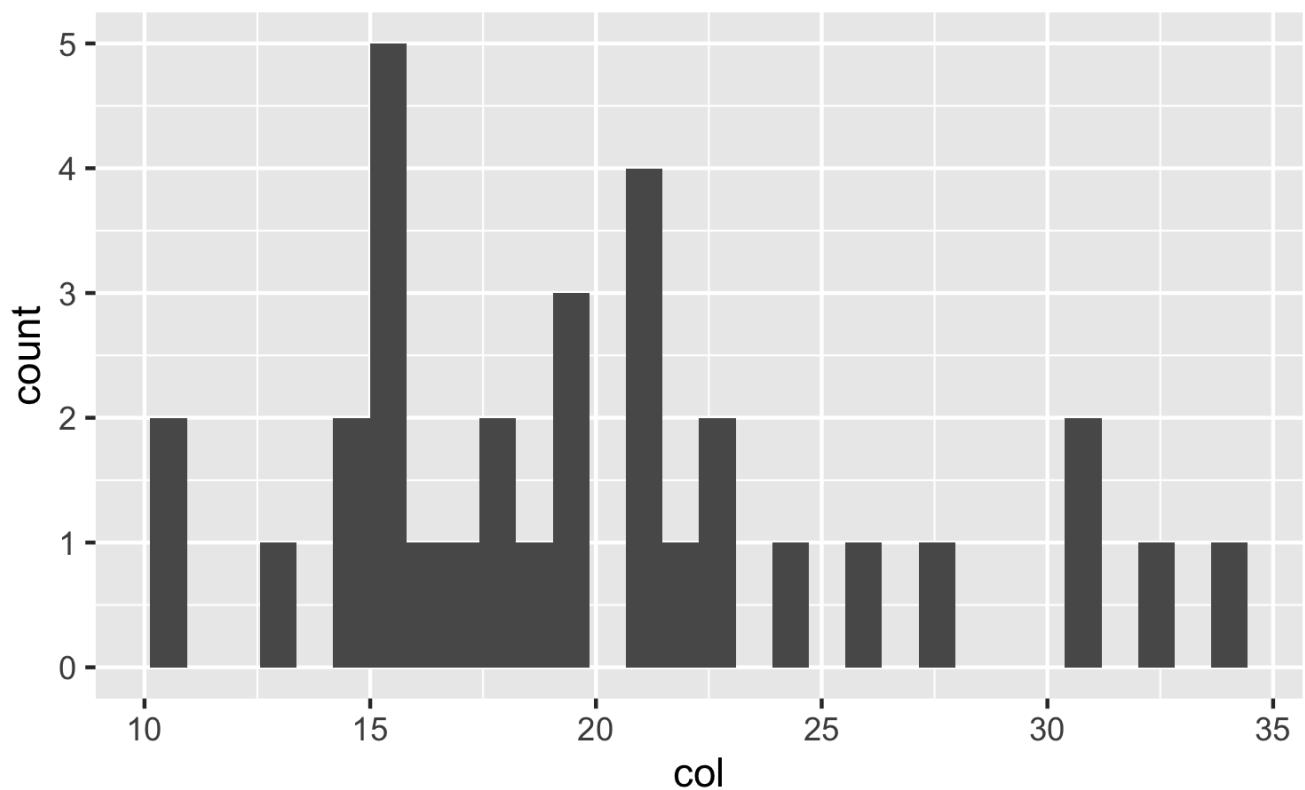
  # input: vector of any type
  # value: number of unique values (double)

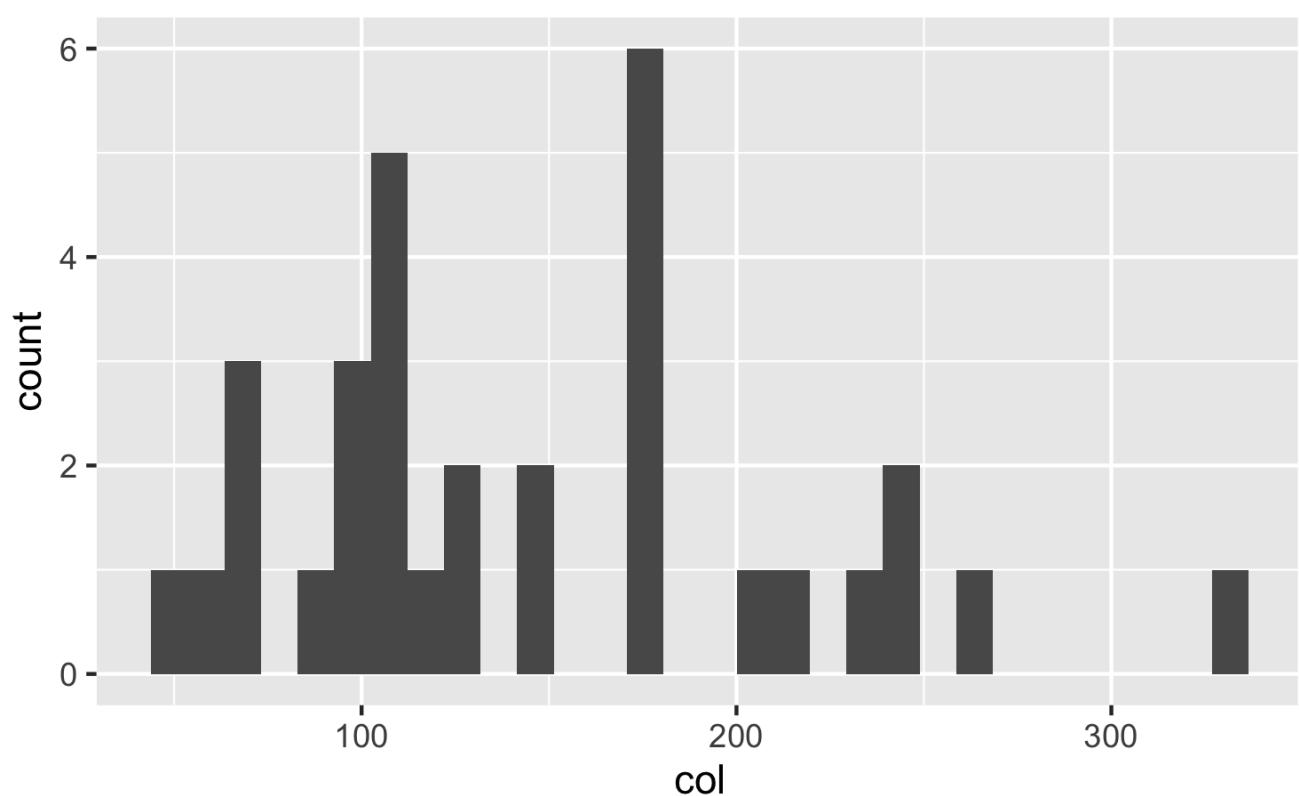
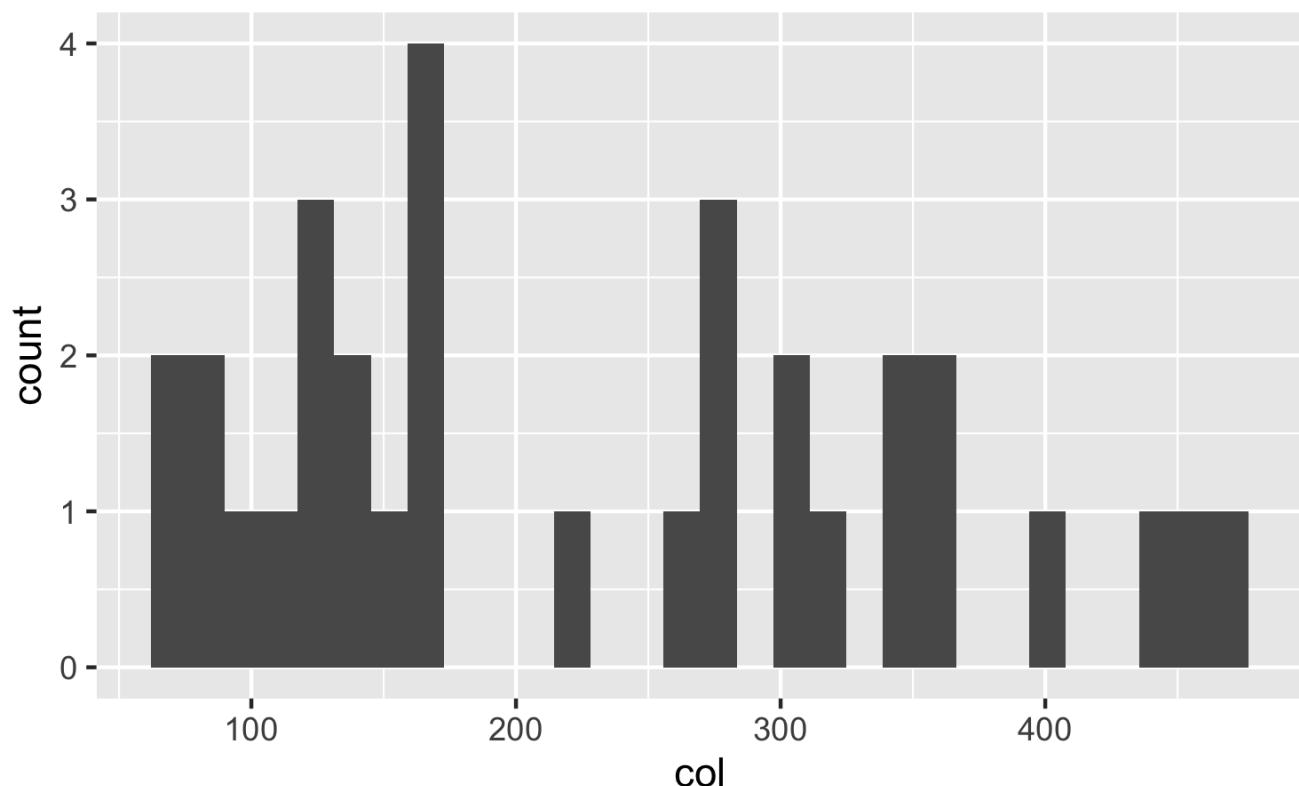
  tmp <- length(unique(vec))
  tmp == 2
}
```

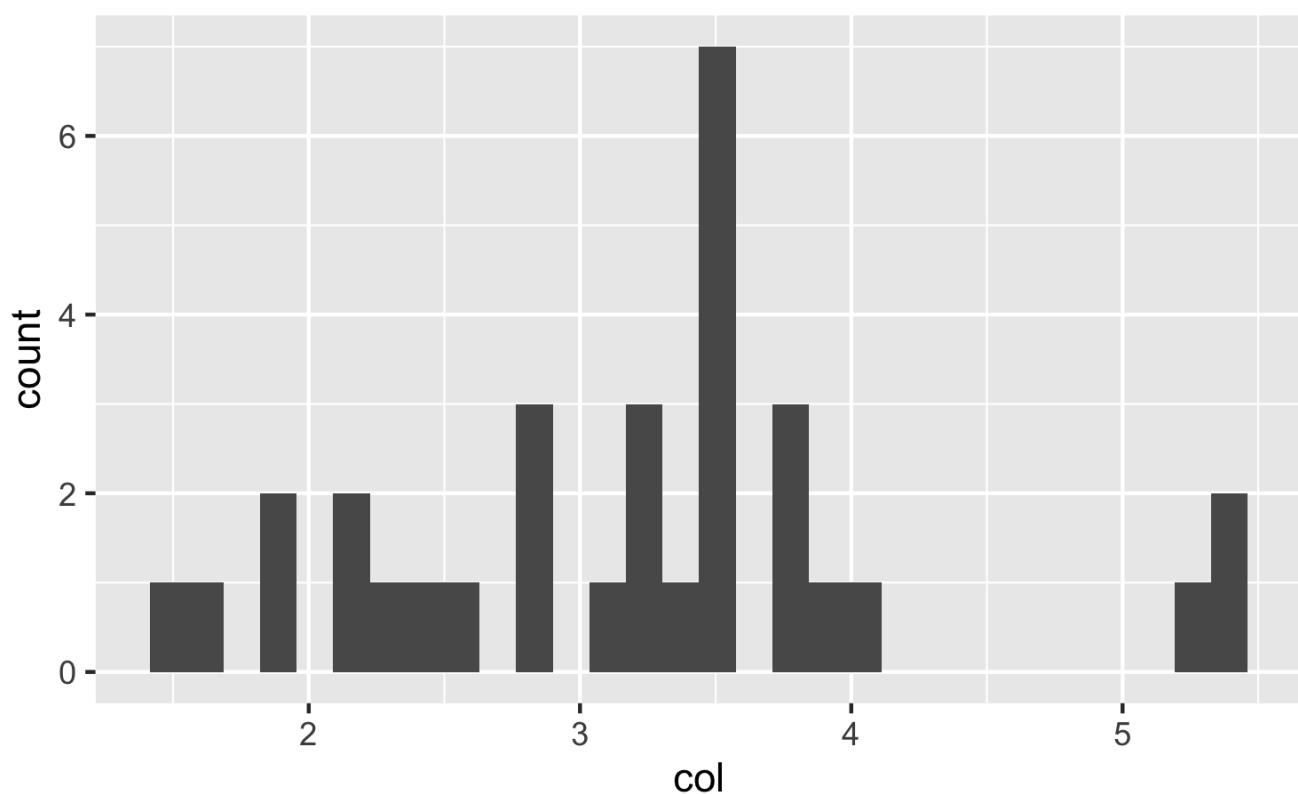
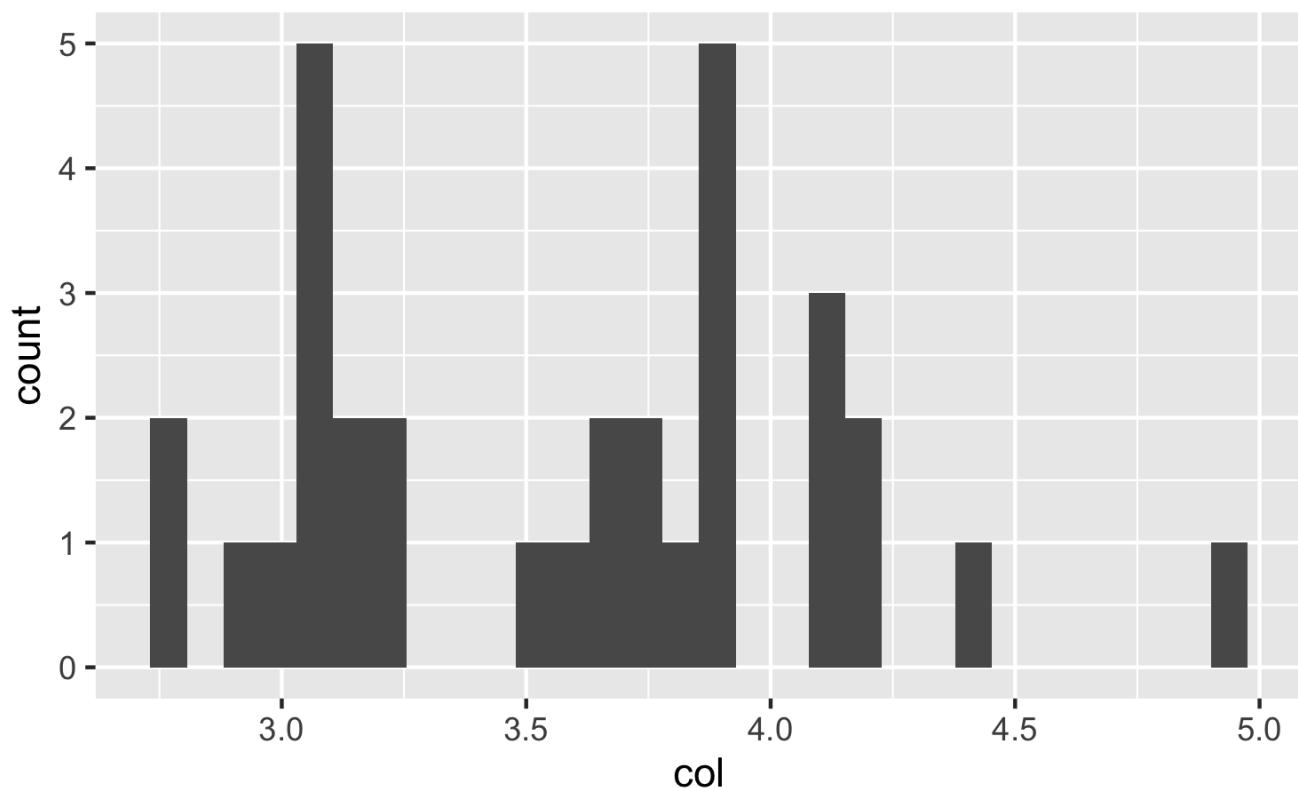
```
mtcars_hist <- function(col){
  mtcars %>%
    ggplot(aes(x = col)) +
    geom_histogram()
}
```

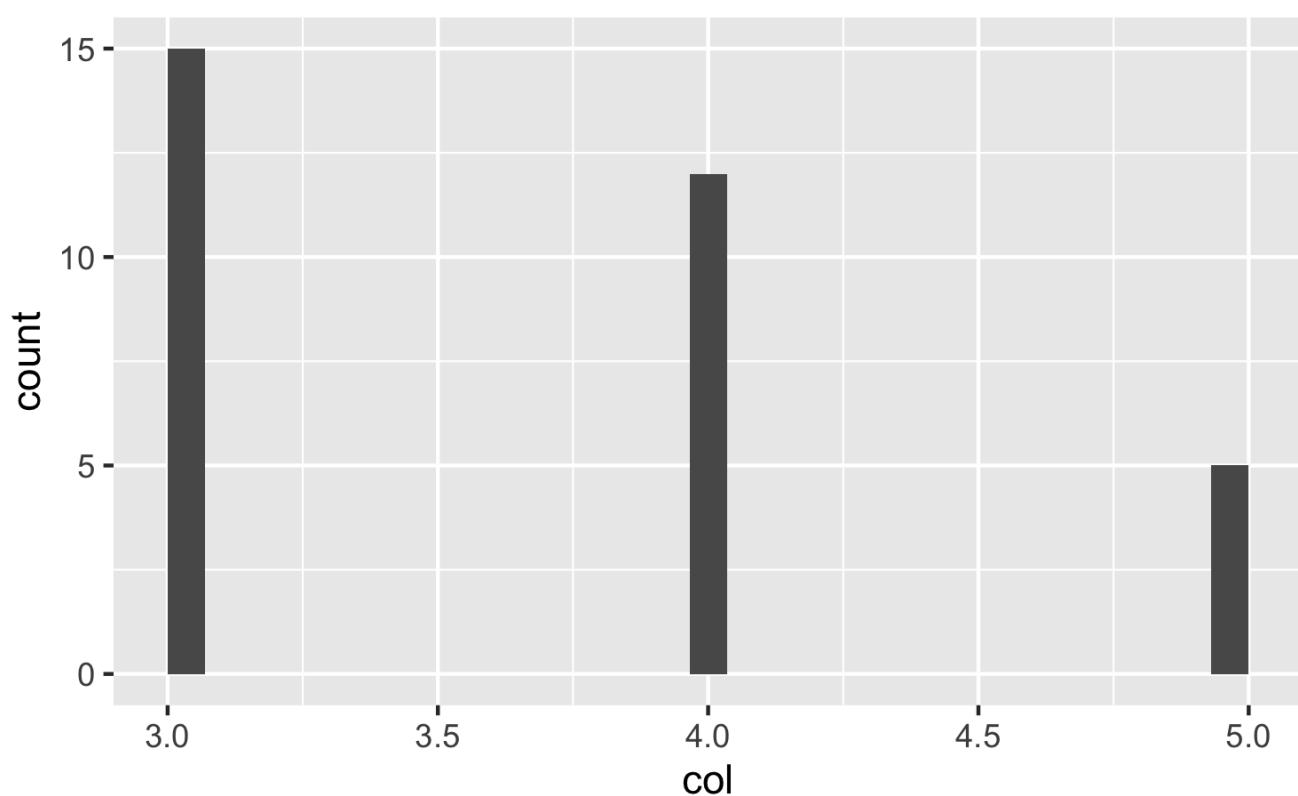
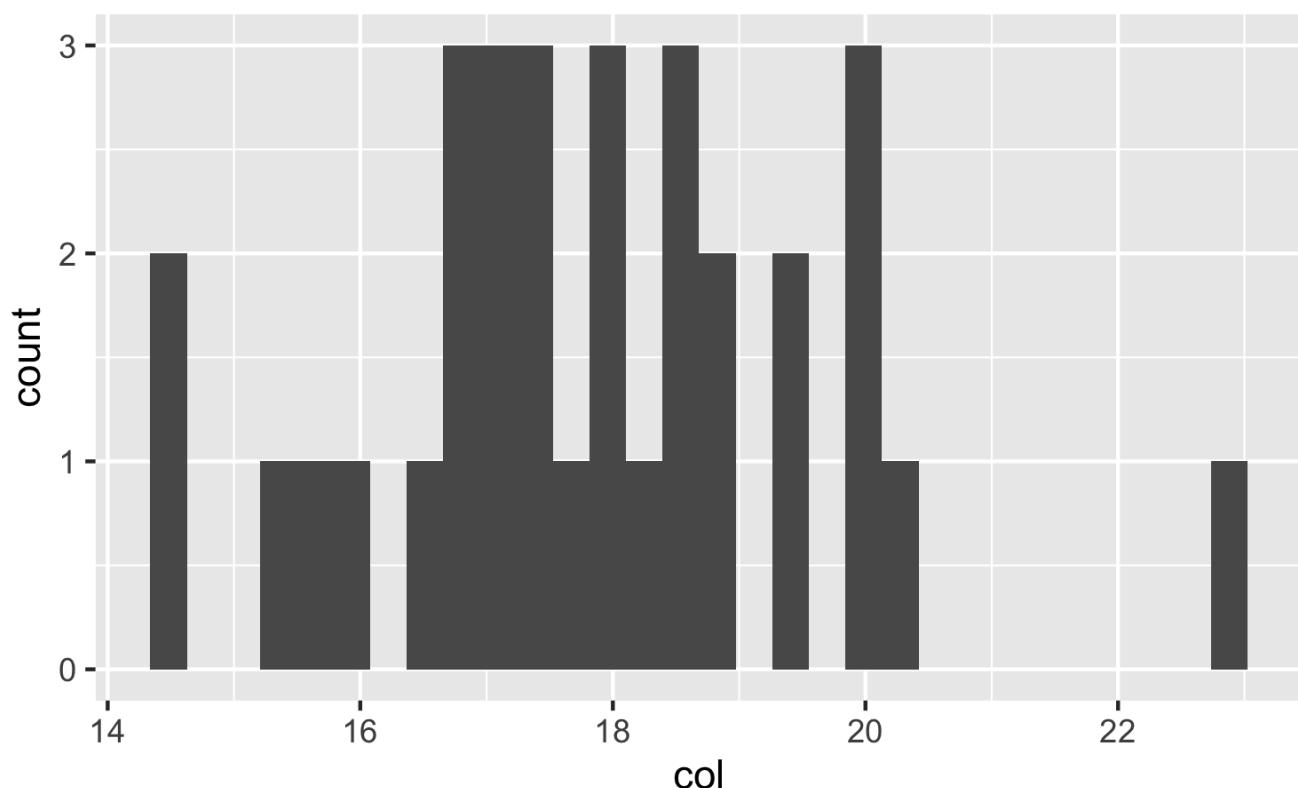
```
mtcars %>%
  select(where(is.numeric)) %>%
  select(where(negate(has_two_levels))) %>%  # `!` zum Negieren geht leider nicht
  map(mtcars_hist)
```

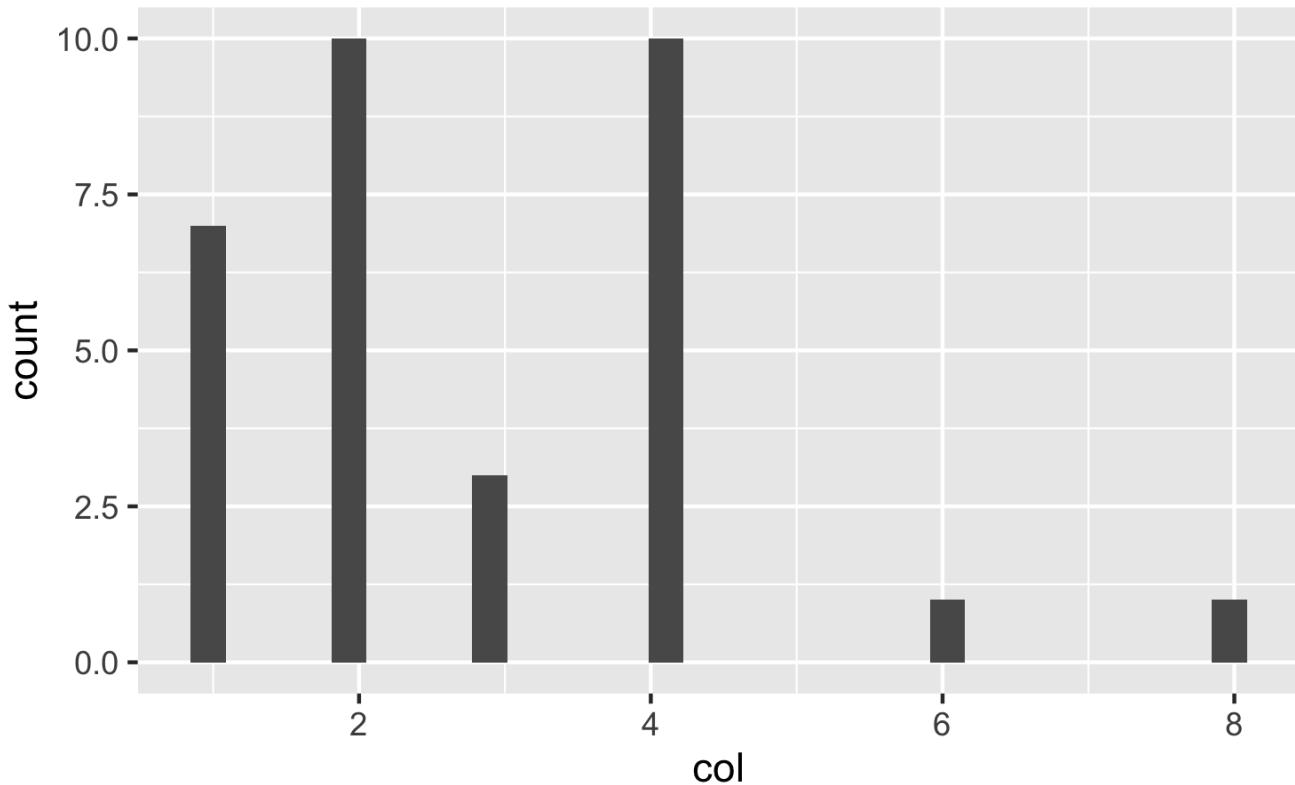
```
## $mpg
##
## $cyl
##
## $disp
##
## $hp
##
## $drat
##
## $wt
##
## $qsec
##
## $gear
##
## $carb
```











## 4.9 Vertiefung

- Funktionale Programmierung mit R (<https://albert-rapp.de/post/2021-09-16-similar-data-and-list-like-columns/>)
- Lernen Sie Wiederholungsstrukturen mit ggplot (<https://data-se.netlify.app/2021/02/06/plotting-multiple-plots-using-purrr-map-and-ggplot/>)
- Fallstudie Flugverspätungen (<https://data-se.netlify.app/2021/03/10/fallstudie-modellierung-von-flugversp%C3%A4tungen/>)
- Fallstudie Getreideernte (<https://juliasilge.com/blog/crop-yields/>)

## 5 tidymodels

Benötigte R-Pakete für dieses Kapitel:

```
library(tidyverse)
library(tidymodels)
```

{tidymodels} ist ein *Metapaket*: Ein (R)-Paket, das mehrere andere Paket startet und uns damit das Leben einfacher macht. Eine Liste der R-Pakete, die durch tidymodels gestartet werden, findet sich hier (<https://www.tidymodels.org/packages/>). Probieren Sie auch mal ?tidymodels .

Eine Liste aller Pakete, die in Tidymodels benutzt werden, die dependencies , kann man sich so ausgeben lassen:

```
pkg_deps(x = "tidymodels", recursive = FALSE)
```

## 5.1 Lernsteuerung

### 5.1.1 Vorbereitung

- Lesen Sie TMWR, Kapitel 1 (<https://www.tmwr.org/software-modeling.html>)
- Lesen Sie übrige Literatur zu diesem Thema

### 5.1.2 Lernziele

- Sie sind in der Lage, Regressionsmodelle mit dem tidymodels-Ansatz zu spezifizieren

### 5.1.3 Literatur

- TMWR, Kap. 1, 5, 6, 7, 8, 9

## 5.2 Daten

Dieser Abschnitt bezieht sich auf Kapitel 4 (<https://www.tmwr.org/ames.html>) in Silge and Kuhn (2022).

Wir benutzen den Datensatz zu Immobilienpreise aus dem Ames County ([https://en.wikipedia.org/wiki/Ames,\\_Iowa](https://en.wikipedia.org/wiki/Ames,_Iowa)) in Iowa, USA, gelegen im Zentrum des Landes.

```
data(ames) # Daten wurden über tidymodels mit geladen
ames <- ames %>% mutate(Sale_Price = log10(Sale_Price))
```

Hier wurde die AV log-transformiert. Das hat zwei (wichtige) Effekte:

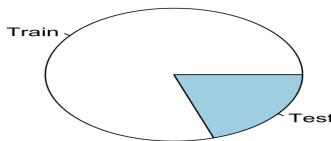
1. Die Verteilung ist symmetrischer, näher an der Normalverteilung. Damit gibt es mehr Daten im Hauptbereich des Ranges von `Sale_Price`, was die Vorhersagen stabiler machen dürfte.
2. Logarithmiert man die Y-Variable, so kommt dies einem multiplikativen Modell gleich (<https://sebastiansauer.github.io/2021-sose/QuantMeth1/Vertiefung/Log-Log-Regression.html#19>), s. auch hier (<https://data-se.netlify.app/2021/06/17/ein-beispiel-zum-nutzen-einer-log-transformation/>).

## 5.3 Train- vs Test-Datensatz aufteilen

Dieser Abschnitt bezieht sich auf Kapitel 5 (<https://www.tmwr.org/splitting.html>) in Silge and Kuhn (2022).

Das Aufteilen in Train- und Test-Datensatz ist einer der wesentlichen Grundsätze im maschinellen Lernen. Das Ziel ist, Overfitting abzuwenden.

Im Train-Datensatz werden alle Modelle berechnet. Der Test-Datensatz wird nur *einmal* verwendet, und zwar zur Überprüfung der Modellgüte.



Praktisch funktioniert das in Silge and Kuhn (2022) wie folgt.

Wir laden die Daten und erstellen einen Index, der jeder Beobachtung die Zuteilung zu Train- bzw. zum Test-Datensatz zuweist:

```
ames_split <- initial_split(ames, prop = 0.80, strata = Sale_Price)
```

`initial_split()` speichert für spätere komfortable Verwendung auch die Daten. Aber eben auch der Index, der bestimmt, welche Beobachtung im Train-Set landet:

```
ames_split$in_id %>% head(n = 10)
```

```
## [1] 2 27 28 30 31 35 83 84 89 120
```

```
length(ames_split$in_id)
```

```
## [1] 2342
```

Praktisch ist auch, dass die AV-Verteilung in beiden Datensätzen ähnlich gehalten wird (Stratifizierung), das besorgt das Argument `strata`.

Die eigentlich Aufteilung in die zwei Datensätze geht dann so:

```
ames_train <- training(ames_split)
ames_test <- testing(ames_split)
```

## 5.4 Grundlagen der Modellierung mit tidymodels

Dieser Abschnitt bezieht sich auf Kapitel 6 (<https://www.tmwr.org/models.html>) in Silge and Kuhn (2022).

`tidymodels` ist eine Sammlung mehrerer, zusammengehöriger Pakete, eben zum Thema statistische Modellieren.

Das kann man analog zur Sammlung `tidyverse` verstehen, zu der z.B. das R-Paket `dplyr` gehört.

Das R-Paket innerhalb von `tidymodels`, das zum "Fitten" von Modellen zuständig ist, heißt `parsnip` (<https://parsnip.tidymodels.org/>).

Eine Liste der verfügbaren Modelltypen, Modelimplementierungen und Modellparameter, die in Parsnip aktuell unterstützt werden, findet sich hier (<https://www.tidymodels.org/find/parsnip/>).

## 5.4.1 Modelle spezifizieren

Ein (statistisches) Modell wird in Tidymodels mit drei Elementen spezifiziert, vgl. Abb. 5.1.

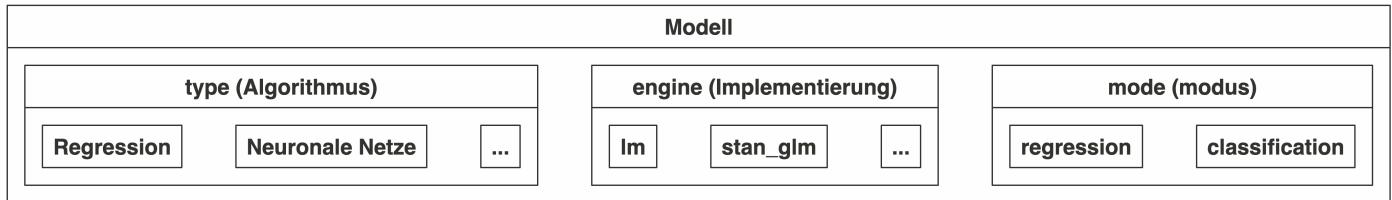


Abbildung 5.1: Definition eines Models in tidymodels

Die Definition eines Modells in tidymodels folgt diesen Ideen:

1. Das Modell sollte unabhängig von den Daten spezifiziert sein
2. Das Modell sollte unabhängig von den Variablen (AV, UVs) spezifiziert sein
3. Das Modell sollte unabhängig von etwaiger Vorverarbeitung (z.B. z-Transformation) spezifiziert sein

Da bei einer linearen Regression nur der Modus "Regression" möglich ist, muss der Modus in diesem Fall nicht angegeben werden. Tidymodels erkennt das automatisch.

```
lm_model <-
  linear_reg() %>%  # Algorithmus, Modelltyp
  set_engine("lm")  # Implementierung
  # Modus hier nicht nötig, da lineare Modelle immer numerisch klassifizieren
```

## 5.4.2 Modelle berechnen

Nach Rhys (2020) ist ein Modell sogar erst ein Modell, wenn die Koeffizienten berechnet sind. Tidymodels kennt diese Unterscheidung nicht. Stattdessen spricht man in Tidymodels von einem "gefitteten" Modell, sobald es berechnet ist. Ähnlich fancy könnte man von einem "instantiierten" Modell sprechen.

Für das Beispiel der einfachen linearen Regression heißt das, das Modell ist gefittet, sobald die Steigung und der Achsenabschnitt (sowie die Residualstreuung) berechnet sind.

```
lm_form_fit <-
  lm_model %>%
  fit(Sale_Price ~ Longitude + Latitude, data = ames_train)
```

## 5.4.3 Vorhersagen

Im maschinellen Lernen ist man primär an den Vorhersagen interessiert, häufig nur an Punktschätzungen. Schauen wir uns also zunächst diese an.

Vorhersagen bekommt man recht einfach mit der `predict()` Methode:

```
predict(lm_form_fit, new_data = ames_test) %>%
  head()
```

```
## # A tibble: 6 × 1
##   .pred
##   <dbl>
## 1 5.28
## 2 5.28
## 3 5.25
## 4 5.25
## 5 5.26
## 6 5.24
```

Die Syntax lautet `predict(modell, daten_zum_vorhersagen)`.

## 5.4.4 Vorhersagen im Train-Datensatz

Vorhersagen im Train-Datensatz machen keinen Sinn, da sie nicht gegen Overfitting geschützt sind und daher deutlich zu optimistisch sein können.

Bei einer linearen Regression ist diese Gefahr nicht so hoch, aber bei anderen, flexibleren Modellen, ist diese Gefahr absurd groß.

## 5.4.5 Modellkoeffizienten im Train-Datensatz

Gibt man den Namen des Modellobjekts ein, so wird ein Überblick an relevanten Modellergebnissen am Bildschirm gedruckt:

```
lm_form_fit
```

```
## parsnip model object
##
## Call:
## stats::lm(formula = Sale_Price ~ Longitude + Latitude, data = data)
##
## Coefficients:
## (Intercept)    Longitude     Latitude
## -301.402       -2.018        2.799
```

Innerhalb des Ergebnisobjekts findet sich eine Liste namens `fit`, in der die Koeffizienten (der "Fit") abgelegt sind:

```
lm_form_fit %>% pluck("fit")
```

```
## 
## Call:
## stats::lm(formula = Sale_Price ~ Longitude + Latitude, data = data)
##
## Coefficients:
## (Intercept)    Longitude     Latitude
## -301.402       -2.018        2.799
```

Zum Herausholen dieser Infos kann man auch die Funktion `extract_fit_engine()` verwenden:

```
lm_fit <-
  lm_form_fit %>%
  extract_fit_engine()

lm_fit
```

```
## 
## Call:
## stats::lm(formula = Sale_Price ~ Longitude + Latitude, data = data)
##
## Coefficients:
## (Intercept)    Longitude     Latitude
## -301.402       -2.018        2.799
```

Das extrahierte Objekt ist, in diesem Fall, das typische `lm()` Objekt. Entsprechend kann man darauaf `coef()` oder `summary()` anwenden.

```
coef(lm_fit)
```

```
## (Intercept) Longitude Latitude
## -301.402383 -2.017755 2.799489
```

```
summary(lm_fit)
```

```
##
## Call:
## stats::lm(formula = Sale_Price ~ Longitude + Latitude, data = data)
##
## Residuals:
##     Min      1Q  Median      3Q     Max 
## -1.02433 -0.09711 -0.01715  0.09649  0.57880 
## 
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)    
## (Intercept) -301.4024    14.4190  -20.90   <2e-16 ***
## Longitude     -2.0178     0.1285  -15.71   <2e-16 ***
## Latitude      2.7995     0.1790   15.64   <2e-16 ***
## ---      
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## 
## Residual standard error: 0.1597 on 2339 degrees of freedom
## Multiple R-squared:  0.1685, Adjusted R-squared:  0.1678 
## F-statistic: 237 on 2 and 2339 DF, p-value: < 2.2e-16
```

Schicker sind die Pendant-Befehle aus `broom`, die jeweils einen Tibble zuückliefern:

```
library(broom)
tidy(lm_fit) # Koeffizienten
```

```
## # A tibble: 3 × 5
##   term      estimate std.error statistic  p.value
##   <chr>     <dbl>     <dbl>     <dbl>     <dbl>
## 1 (Intercept) -301.      14.4      -20.9  4.27e-89
## 2 Longitude     -2.02     0.128     -15.7  6.33e-53
## 3 Latitude      2.80      0.179     15.6  1.61e-52
```

```
glance(lm_fit) # Modellgüte
```

```
## # A tibble: 1 × 12
##   r.squared adj.r.squared sigma statistic  p.value    df logLik     AIC     BIC
##   <dbl>        <dbl> <dbl>     <dbl>     <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 0.169       0.168 0.160     237.  1.82e-94     2  974. -1941. -1918.
## # ... with 3 more variables: deviance <dbl>, df.residual <int>, nobs <int>
```

## 5.4.6 Parsnip RStudio add-in

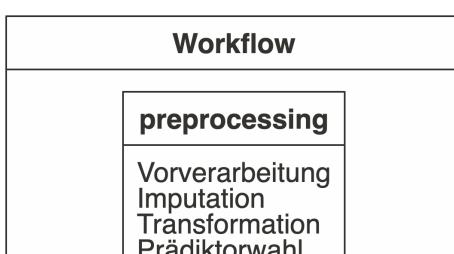
Mit dem Add-in von Parsnip kann man sich eine Modellspezifikation per Klick ausgeben lassen. Nett!

```
parsnip_addin()
```

## 5.5 Workflows

Dieser Abschnitt bezieht sich auf Kapitel 7 (<https://www.tmwr.org/workflows.html>) in Silge and Kuhn (2022).

### 5.5.1 Konzept des Workflows in Tidymodels



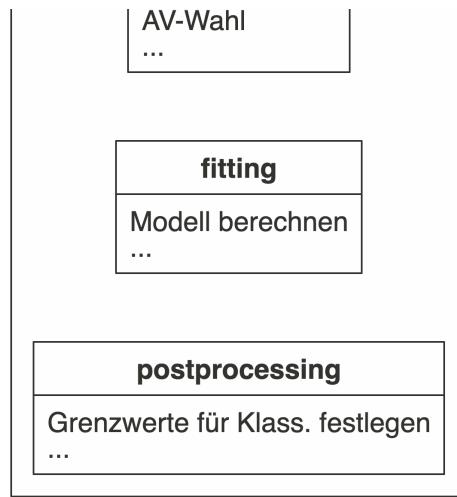


Abbildung 5.2: Definition eines Models in tidyverse

## 5.5.2 Einfaches Beispiel

Wir initialisieren einen Workflow, verzichten auf Vorverarbeitung und fügen ein Modell hinzu:

```
lm_workflow <-
  workflow() %>% # init
  add_model(lm_model) %>% # Modell hinzufügen
  add_formula(Sale_Price ~ Longitude + Latitude) # Modellformel hinzufügen
```

Werfen wir einen Blick in das Workflow-Objekt:

```
lm_workflow

## -- Workflow --
## Preprocessor: Formula
## Model: linear_reg()
##
## -- Preprocessor --
## Sale_Price ~ Longitude + Latitude
##
## -- Model --
## Linear Regression Model Specification (regression)
##
## Computational engine: lm
```

Wie man sieht, gehört die Modellformel ( $y \sim x$ ) zur Vorverarbeitung aus Sicht von Tidyverse.

Was war nochmal im Objekt `lm_model` enthalten?

```
lm_model

## Linear Regression Model Specification (regression)
##
## Computational engine: lm
```

Jetzt können wir das Modell berechnen (fitten):

```
lm_fit <-
  lm_workflow %>%
  fit(ames_train)
```

Natürlich kann man synonym auch schreiben:

```
lm_fit <- fit(lm_wflow, ames_train)
```

Schauen wir uns das Ergebnis an:

```
lm_fit
```

```
## == Workflow [trained] =====
## Preprocessor: Formula
## Model: linear_reg()
##
## — Preprocessor -----
## Sale_Price ~ Longitude + Latitude
##
## — Model -----
##
## Call:
## stats::lm(formula = ..y ~ ., data = data)
##
## Coefficients:
## (Intercept) Longitude Latitude
## -301.402     -2.018      2.799
```

### 5.5.3 Vorhersage mit einem Workflow

Die Vorhersage mit einem Tidymodels-Workflow ist einerseits komfortabel, da man einfach sagen kann:

“Nimm die richtigen Koeffizienten des Modells aus dem Train-Set und wende sie auf das Test-Sample an. Berechne mir die Vorhersagen und die Modellgüte.”

So sieht das aus:

```
final_lm_res <- last_fit(lm_workflow, ames_split)
final_lm_res
```

```
## # Resampling results
## # Manual resampling
## # A tibble: 1 × 6
##   splits          id       .metrics .notes   .predictions .workflow
##   <list>         <chr>    <list>  <list>   <list>      <list>
## 1 <split [2342/588]> train/test split <tibble> <tibble> <tibble>  <workflow>
```

Andererseits wird auch ein recht komplexes Objekt zurückgeliefert, das man erst mal durchschauen muss.

Wie man sieht, gibt es mehrere Listenpalten. Besonders interessant erscheinen natürlich die Listenpalten `.metrics` und `.predictions`.

Schauen wir uns die Vorhersagen an.

```
lm_preds <- final_lm_res %>% pluck(".predictions", 1)
```

Es gibt auch eine Funktion, die obige Zeile vereinfacht (also synonym ist):

```
lm_preds <- collect_predictions(final_lm_res)
lm_preds %>% slice_head(n = 5)
```

```
## # A tibble: 5 × 5
##   id       .pred .row Sale_Price .config
##   <chr>    <dbl> <int>      <dbl> <chr>
## 1 train/test split  5.28     9        5.37 Preprocessor1_Model1
## 2 train/test split  5.28    10        5.28 Preprocessor1_Model1
## 3 train/test split  5.25    17        5.21 Preprocessor1_Model1
## 4 train/test split  5.25    21        5.28 Preprocessor1_Model1
## 5 train/test split  5.26    23        5.33 Preprocessor1_Model1
```

### 5.5.4 Modellgüte

Dieser Abschnitt bezieht sich auf Kapitel 9 (<https://www.tmwr.org/performance.html>) in Silge and Kuhn (2022).

Die Vorhersagen bilden die Basis für die Modellgüte (“Metriken”), die schon fertig berechnet im Objekt `final_lm_res` liegen und mit `collect_metrics` herausgenommen werden können:

```
lm_metrics <- collect_metrics(final_lm_res)
```

```
.metric .estimator .estimate .config
```

.metric	.estimator	.estimate	.config
rmse	standard	$1.66 \times 10^{-1}$	Preprocessor1_Model1
rsq	standard	$1.88 \times 10^{-1}$	Preprocessor1_Model1

Man kann auch angeben, welche Metriken der Modellgüte man bekommen möchte:

```
ames_metrics <- metric_set(rmse, rsq, mae)
```

```
ames_metrics(data = lm_preds,
             truth = Sale_Price,
             estimate = .pred)
```

```
## # A tibble: 3 × 3
##   .metric .estimator .estimate
##   <chr>   <chr>        <dbl>
## 1 rmse    standard     0.166
## 2 rsq     standard     0.188
## 3 mae     standard     0.125
```

## 5.5.5 Vorhersage von Hand

Man kann sich die Metriken auch von Hand ausgeben lassen, wenn man direktere Kontrolle haben möchte als mit `last_fit` und `collect_metrics`.

```
ames_test_small <- ames_test %>% slice(1:5)
predict(lm_form_fit, new_data = ames_test_small)
```

```
## # A tibble: 5 × 1
##   .pred
##   <dbl>
## 1 5.28
## 2 5.28
## 3 5.25
## 4 5.25
## 5 5.26
```

Jetzt binden wir die Spalten zusammen, also die "Wahrheit" ( $y$ ) und die Vorhersagen:

```
ames_test_small2 <-
  ames_test_small %>%
  select(Sale_Price) %>%
  bind_cols(predict(lm_form_fit, ames_test_small)) %>%
  # Add 95% prediction intervals to the results:
  bind_cols(predict(lm_form_fit, ames_test_small, type = "pred_int"))
```

```
rsq(ames_test_small2,
    truth = Sale_Price,
    estimate = .pred
  )
```

```
## # A tibble: 1 × 3
##   .metric .estimator .estimate
##   <chr>   <chr>        <dbl>
## 1 rsq     standard     0.116
```

Andere Koeffizienten der Modellgüte können mit `rmse` oder `mae` abgerufen werden.

## 5.6 Rezepte zur Vorverarbeitung

Dieser Abschnitt bezieht sich auf Kapitel 8 (<https://www.tmwr.org/recipes.html>) in Silge and Kuhn (2022).

### 5.6.1 Was ist Rezept und wozu ist es gut?

So könnte ein typischer Aufruf von `lm()` aussehen:

```
lm(Sale_Price ~ Neighborhood + log10(Gr_Liv_Area) + Year_Built + Bldg_Type,
  data = ames)
```

Neben dem Fitten des Modells besorgt die Formel-Schreibweise noch einige zusätzliche nützliche Vorarbeitung:

1. Definition von AV und AV
2. Log-Transformation von Gr\_Liv\_Area
3. Transformation der nominalen Variablen in Dummy-Variablen

Das ist schön und nützlich, hat aber auch Nachteile:

1. Das Modell wird nicht nur spezifiziert, sondern auch gleich berechnet. Das ist unpraktisch, weil man die Modellformel vielleicht in anderen Modell wiederverwenden möchte. Außerdem kann das Berechnen lange dauern.
2. Die Schritte sind ineinander vermengt, so dass man nicht einfach und übersichtlich die einzelnen Schritte bearbeiten kann.

Praktischer wäre also, die Schritte der Vorverarbeitung zu ent-flechten. Das geht mit einem "Rezept" aus Tidmoodels:

```
simple_ames <-
  recipe(Sale_Price ~ Neighborhood + Gr_Liv_Area + Year_Built + Bldg_Type,
         data = ames_train) %>%
  step_log(Gr_Liv_Area, base = 10) %>%
  step_dummy(all_nominal_predictors())
simple_ames
```

```
## Recipe
##
## Inputs:
##
##   role #variables
##   outcome      1
##   predictor     4
##
## Operations:
##
## Log transformation on Gr_Liv_Area
## Dummy variables from all_nominal_predictors()
```

Ein Rezept berechnet kein Modell. Es macht nichts außer die Vorverarbeitung des Modells zu spezifizieren (inklusive der Modellformel).

## 5.6.2 Workflows mit Rezepten

Jetzt definieren wir den Workflow nicht nur mit einer Modellformel, sondern mit einem Rezept:

```
lm_workflow <-
  workflow() %>%
  add_model(lm_model) %>%
  add_recipe(simple_ames)
```

Sonst hat sich nichts geändert.

Wie vorher, können wir jetzt das Modell berechnen.

```
lm_fit <- fit(lm_workflow, ames_train)
```

```
final_lm_res <- last_fit(lm_workflow, ames_split)
final_lm_res
```

```
## # Resampling results
## # Manual resampling
## # A tibble: 1 × 6
##   splits           id       .metrics .notes   .predictions .workflow
##   <list>          <chr>    <list>   <list>   <list>      <list>
## 1 <split [2342/588]> train/test split <tibble> <tibble> <tibble>    <workflow>
##
## There were issues with some computations:
##
## - Warning(s) x1: prediction from a rank-deficient fit may be misleading
##
## Use `collect_notes(object)` for more information.
```

```
lm_metrics <- collect_metrics(final_lm_res)
lm_metrics

## # A tibble: 2 × 4
##   .metric .estimator .estimate .config
##   <chr>   <chr>        <dbl> <chr>
## 1 rmse    standard     0.0857 Preprocessor1_Model1
## 2 rsq     standard      0.785  Preprocessor1_Model1
```

## 5.6.3 Spaltenrollen

Eine praktische Funktion ist es, bestimmte Spalten nicht als Prädiktor, sondern als ID-Variable zu nutzen. Das kann man in Tidymodels komfortabel wie folgt angeben:

```
ames_recipe <-
  simple_ames %>%
  update_role(Neighborhood, new_role = "id")

ames_recipe
```

```
## Recipe
##
## Inputs:
##
##       role #variables
##       id          1
##       outcome      1
##       predictor    3
##
## Operations:
##
## Log transformation on Gr_Liv_Area
## Dummy variables from all_nominal_predictors()
```

## 5.6.4 Fazit

Mehr zu Rezepten findet sich hier (<https://recipes.tidymodels.org/>). Ein Überblick zu allen Schritten der Vorverarbeitung findet sich hier (<https://recipes.tidymodels.org/articles/recipes.html>).

## 5.7 Aufgaben

- Fallstudie Seegurken (<https://www.tidymodels.org/start/models/>)
- Sehr einfache Fallstudie zur Modellierung einer Regression mit tidymodels (<https://juliasilge.com/blog/student-debt/>)
- Fallstudie zur linearen Regression mit Tidymodels (<https://www.gmudatamining.com/lesson-10-r-tutorial.html>)

# 6 kNN

## 6.1 Lernsteuerung

### 6.1.1 Lernziele

- Sie sind in der Lage, einfache Klassifikationsmodelle zu spezifizieren mit tidymodels
- Sie können den knn-Algorithmus erläutern
- Sie können den knn-Algorithmus in tidymodels anwenden
- Sie können die Gütemetriken von Klassifikationsmodellen einschätzen

### 6.1.2 Literatur

- Rhys, Kap. 3
- Timbers et al., Kap. 5 (<https://datasciencebook.ca/classification.html#classification>)

## 6.2 Überblick

In diesem Kapitel geht es um das Verfahren *KNN, K-Nächste-Nachbarn*.

```
knitr::opts_chunk$set(echo = TRUE)
```

Benötigte R-Pakete für dieses Kapitel:

```
library(tidymodels)
```

## 6.3 Intuitive Erklärung

*K-Nächste-Nachbarn* (*k nearest neighbors*, kNN) ist ein einfacher Algorithmus des maschinellen Lernens, der sowohl für Klassifikation als auch für numerische Vorhersage (Regression) genutzt werden kann. Wir werden kNN als Beispiel für eine Klassifikation betrachten.

Betrachten wir ein einführendes Beispiel von Rhys (2020), für das es eine Online-Quelle (<https://livebook.manning.com/book/machine-learning-for-mortals-mere-and-otherwise/chapter-3/22>) gibt. Stellen Sie sich vor, wir laufen durch englische Landschaft, vielleicht die Grafschaft Kent, und sehen ein kleines Tier durch das Gras huschen. Eine Schlange?! In England gibt es (laut Rhys (2020)) nur eine giftige Schlange, die Otter (Adder). Eine andere Schlange, die Grass Snake ist nicht giftig, und dann kommt noch der Slow Worm in Frage, der gar nicht zur Familie der Schlangen gehört. Primär interessiert uns die Frage, haben wir jetzt eine Otter gesehen? Oder was für ein Tier war es?

Zum Glück wissen wir einiges über Schlangen bzw. schlangenähnliche Tiere Englands. Nämlich können wir die betreffenden Tierarten in Größe und Aggressivität einschätzen, das ist in Abbildung 6.1 dargestellt.

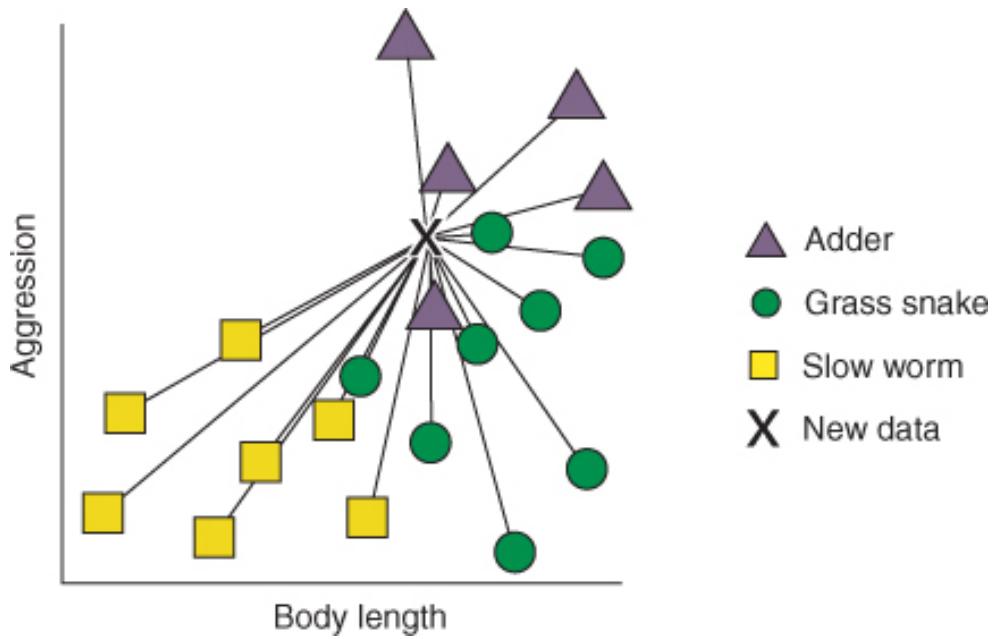


Abbildung 6.1: Haben wir gerade eine Otter gesehen?

Der Algorithmus von kNN sieht einfach gesagt vor, dass wir schauen, welcher Tierarten Tiere mit ähnlicher Aggressivität und Größe angehören. Die Tierart die bei diesen "Nachbarn" hinsichtlich Ähnlichkeit relevanter Merkmale am häufigsten vertreten ist, ordnen wir die bisher unklassifizierte Beobachtung zu.

Etwas zugespitzt:

Wenn es quakt wie eine Ente, läuft wie eine Ente und aussieht wie eine Ente, dann ist es eine Ente.

Die Anzahl  $k$  der nächsten Nachbarn können wir frei wählen; der Wert wird *nicht* vom Algorithmus bestimmt. Solche vom Nutzer zu bestimmenden Größen nennt man auch *Tuningparameter*.

## 6.4 Krebsdiagnostik

Betrachten wir ein Beispiel von Timbers, Campbell, and Lee (2022), das hier (<https://datasciencebook.ca/classification.html#classification-with-k-nearest-neighbors>) frei eingesehen werden kann.

Die Daten sind so zu beziehen:

```
data_url <- "https://raw.githubusercontent.com/UBC-DSCI/introduction-to-datascience/master/data/wdbc.csv"
cancer <- read_csv(data_url)
```

In diesem Beispiel versuchen wir Tumore der Brust zu klassifizieren, ob sie einen schweren Verlauf (maligne, engl. malignant) oder einen weniger schweren Verlauf (benigne, engl. benign) erwarten lassen. Der Datensatz ist hier (<https://datasciencebook.ca/classification.html#describing-the-variables-in-the-cancer-data-set>) näher erläutert.

Wie in Abb. 6.2 ersichtlich, steht eine Tumordiagnose (malignant vs. benign) in Abhängigkeit von Umfang (engl. perimeter) und Konkavität ([https://de.wikipedia.org/wiki/Konvexe\\_und\\_konkave\\_Funktionen](https://de.wikipedia.org/wiki/Konvexe_und_konkave_Funktionen)), die “Gekrümmtheit nach innen”.

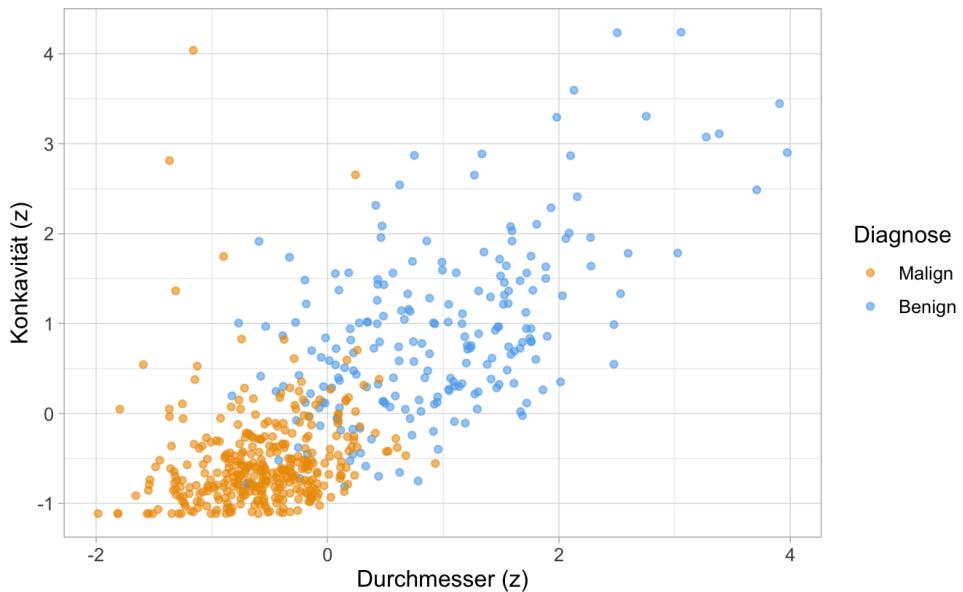


Abbildung 6.2: Streudiagramm zur Einschätzung von Tumordiagnosen

In diesem Code-Beispiel wird die seit R 4.1.0 verfügbare R-native Pfeife verwendet. Wichtig ist vielleicht vor allem, dass diese Funktion nicht läuft auf R-Versionen vor 4.1.0. Einige Unterschiede zur seit längerem bekannten Magrittr-Pfeife sind hier (<https://stackoverflow.com/questions/67633022/what-are-the-differences-between-rs-new-native-pipe-and-the-magrittr-pipe>) erläutert.

Wichtig ist, dass die Merkmale standardisiert sind, also eine identische Skalierung aufweisen, da sonst das Merkmal mit kleinerer Skala weniger in die Berechnung der Nähe (bzw. Abstand) eingeht.

Für einen neuen, bisher unklassifizierten Fall suchen nur nun nach einer Diagnose, also nach der am besten passenden Diagnose (maligne oder benigne), s. Abb. 6.3, wieder aus Timbers, Campbell, and Lee (2022). Ihr Quellcode für dieses Diagramm (und das ganze Kapitel) findet sich hier (<https://github.com/UBC-DSCI/introduction-to-datascience/blob/master/classification1.Rmd>).

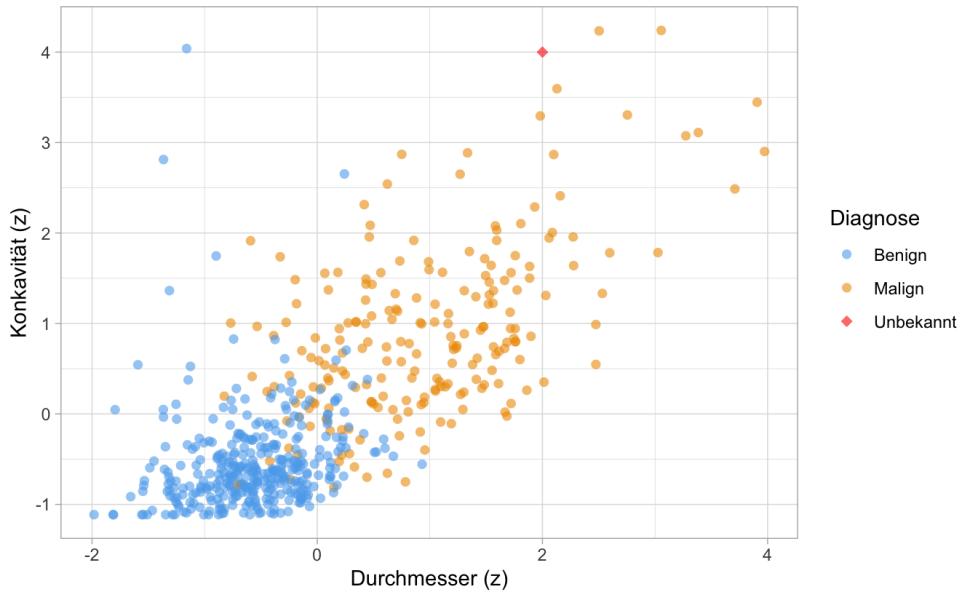


Abbildung 6.3: Ein neuer Fall, bisher unklassifiziert

Wir können zunächst den (im euklidischen Koordinatensystem) nächst gelegenen Fall (der “nächste Nachbar”) betrachten, und vereinbaren, dass wir dessen Klasse als Schätzwert für den unklassifizierten Fall übernehmen, s. Abb. 6.4.

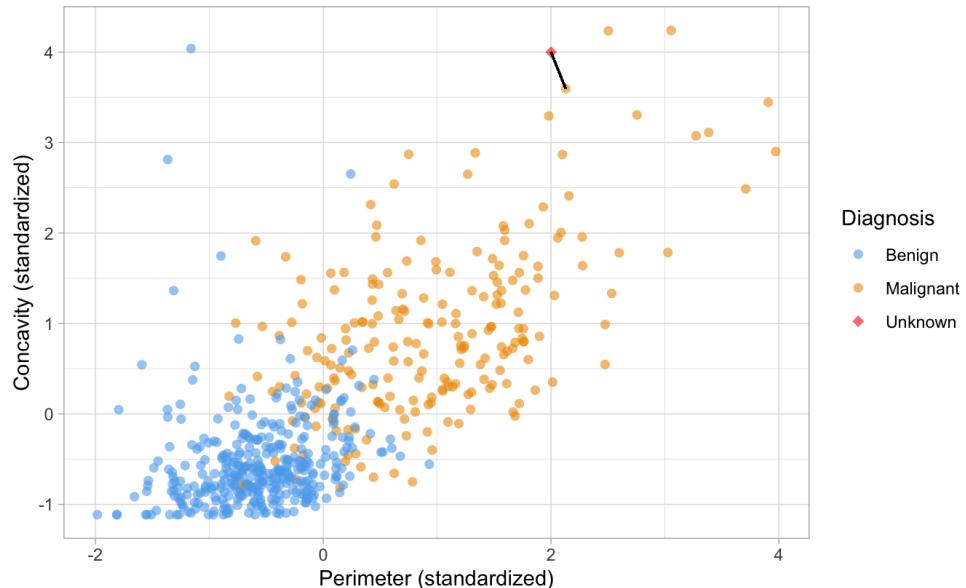


Abbildung 6.4: Ein nächster Nachbar

Betrachten wir einen anderen zu klassifizierenden Fall, s. Abb 6.5. Ob hier die Klassifikation von "benign" korrekt ist? Womöglich nicht, denn viele andere Nachbarn, die etwas weiter weg gelegen sind, gehören zur anderen Diagnose, malign.

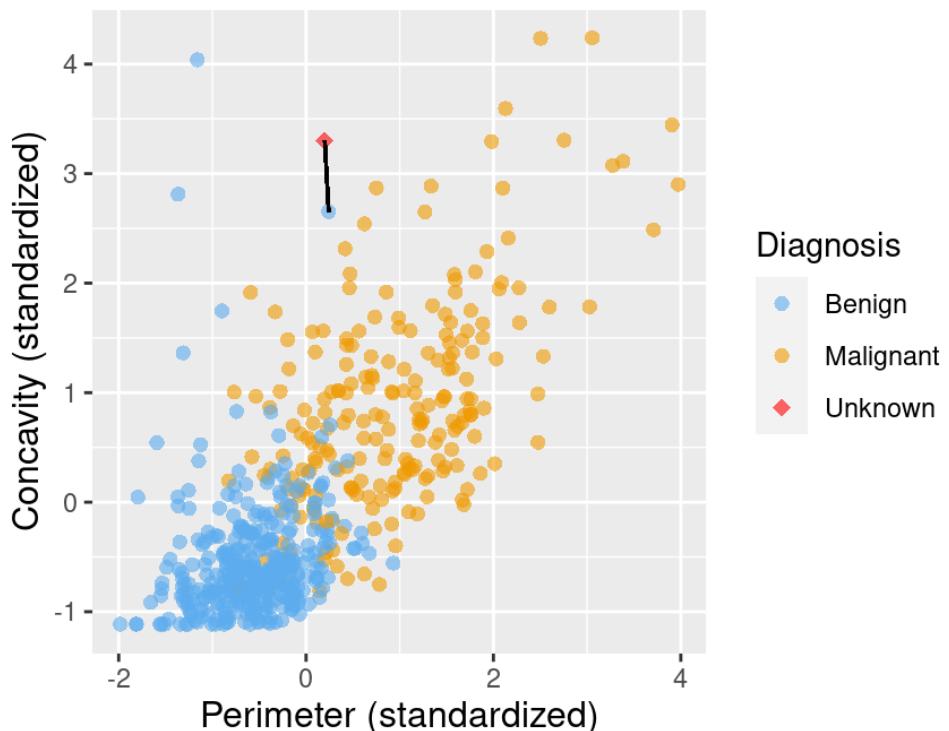
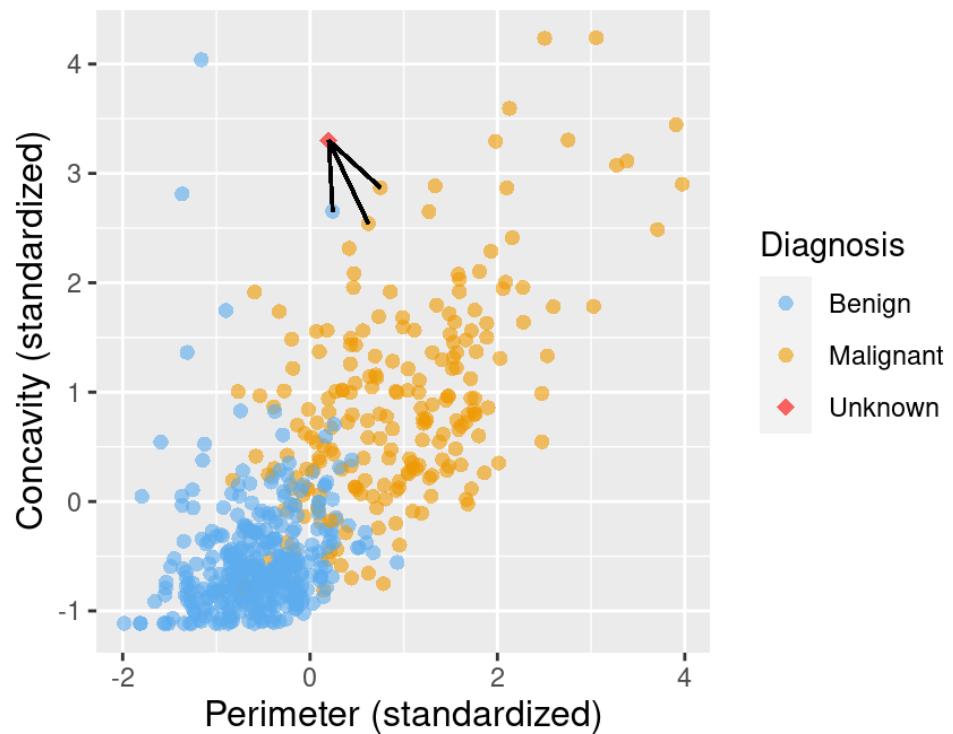


Abbildung 6.5: Trügt der nächste Nachbar?

Um die Vorhersage zu verbessern, können wir nicht nur den nächst gelegenen Nachbarn betrachten, sondern die  $k$  nächst gelegenen, z.B.  $k = 3$ , s. Abb 6.6.

Abbildung 6.6: kNN mit  $k=3$ 

Die Entscheidungsregel ist dann einfach eine Mehrheitsentscheidung: Wir klassifizieren den neuen Fall entsprechend der Mehrheit in den  $k$  nächst gelegenen Nachbarn.

## 6.5 Berechnung der Nähe

Es gibt verschiedenen Algorithmen, um die Nähe bzw. Distanz der Nachbarn zum zu klassifizieren Fall zu berechnen.

Eine gebräuchliche Methode ist der *euklidische* Abstand, der mit Pythagoras berechnet werden kann, s. Abb. 6.7 aus Sauer (2019).

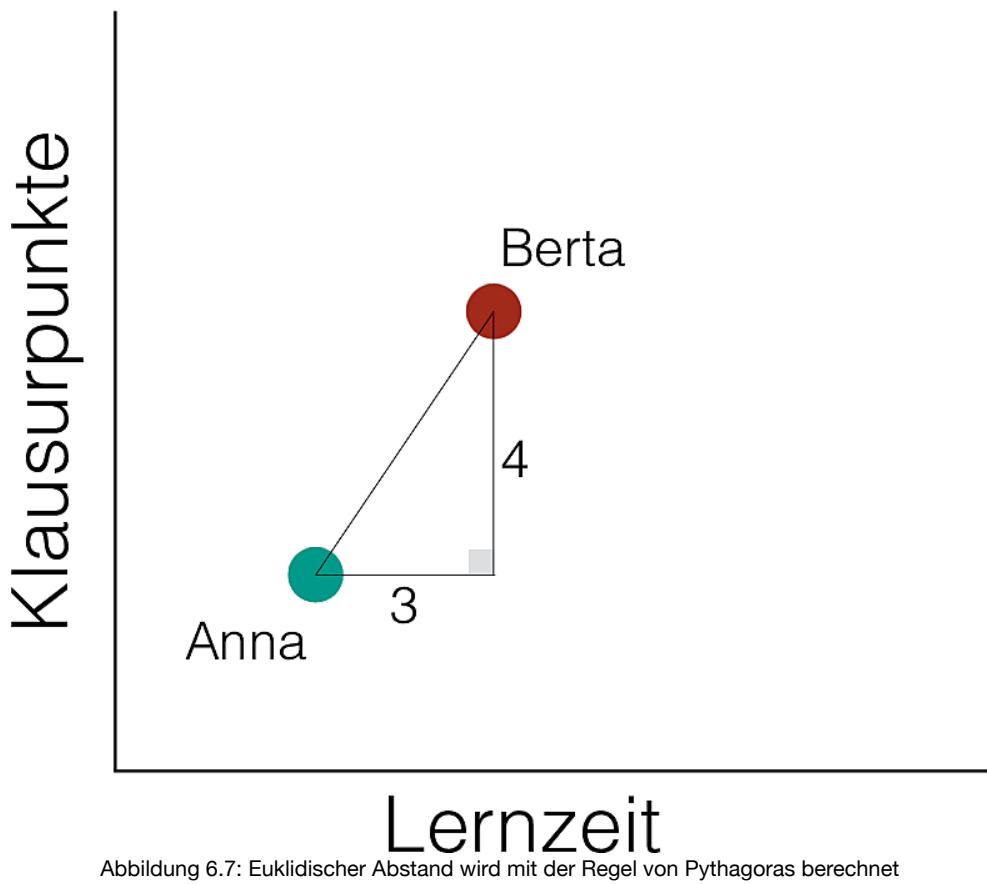


Abbildung 6.7: Euklidischer Abstand wird mit der Regel von Pythagoras berechnet

Wie war das noch mal?

$$c^2 = a^2 + b^2$$

Im Beispiel oben also:

$$c^2 = 3^2 + 4^2 = 5^2$$

Damit gilt:  $c = \sqrt{c^2} = \sqrt{5^2} = 5$ .

Im 2D-Raum ist das so einfach, dass man das (fast) mit bloßem Augenschein entscheiden kann. In mehr als 2 Dimensionen wird es aber schwierig für das Auge, wie ein Beispiel (<https://datasciencebook.ca/classification.html#more-than-two-explanatory-variables>) aus Timbers, Campbell, and Lee (2022) zeigt.

Allerdings kann man den guten alten Pythagoras auch auf Dreiecke mit mehr als zwei Dimensionen anwenden, s. Abb. 6.8 aus Sauer (2019), Kap. 21.1.2.

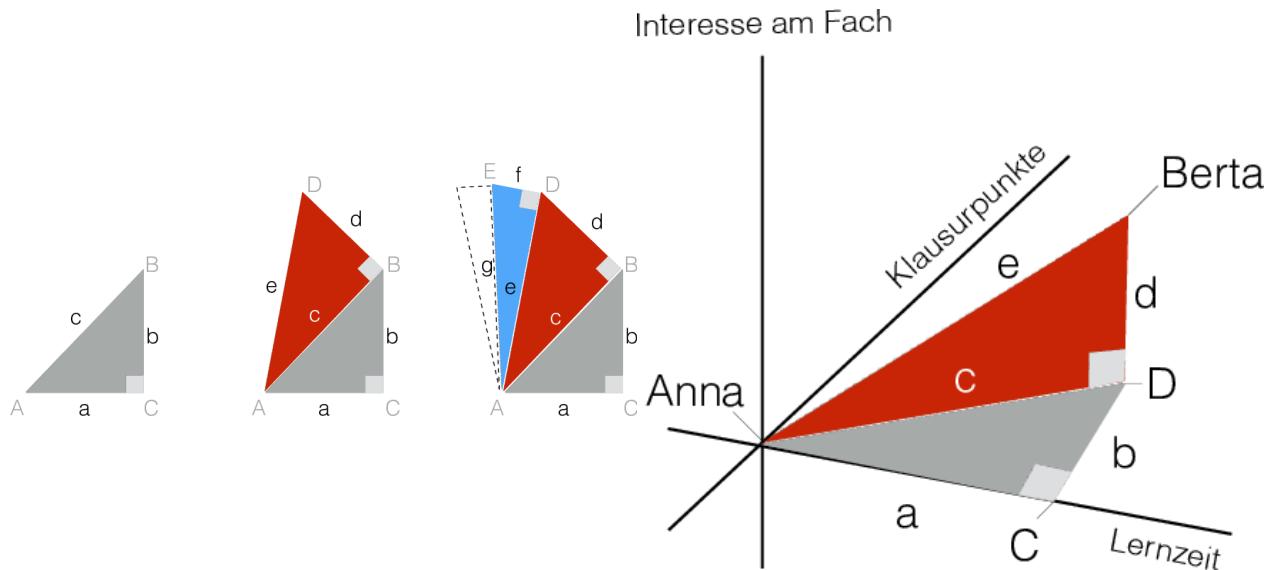


Abbildung 6.8: Pythagoras in der Ebene (links) und in 3D (rechts)

Bleiben wir beim Beispiel von Anna und Berta und nehmen wir eine dritte Variable hinzu (Statistikliebe). Sagen wir, der Unterschied in dieser dritten Variable zwischen Anna und Berta betrage 2.

Es gilt:

$$\begin{aligned} e^2 &= c^2 + d^2 \\ e^2 &= 5^2 + 2^2 \\ e^2 &= 25 + 4 \\ e &= \sqrt{29} \approx 5.4 \end{aligned}$$

## 6.6 kNN mit Tidymodels

### 6.6.1 Analog zu Timbers et al.

Eine Anwendung von kNN mit Tidymodels ist in Timbers, Campbell, and Lee (2022), Kap. 5.6, hier (<https://datasciencebook.ca/classification.html#k-nearest-neighbors-with-tidymodels>) beschrieben.

Die Daten aus Timbers, Campbell, and Lee (2022) finden sich in diesem Github-Repo (<https://github.com/UBC-DSCI/introduction-to-datasience/tree/master/data>)-

Die (z-transformierten) Daten zur Tumorklassifikation können hier ([https://raw.githubusercontent.com/UBC-DSCI/data-science-a-first-intro-worksheets/main/worksheet\\_classification1/data/clean-wdbc-data.csv](https://raw.githubusercontent.com/UBC-DSCI/data-science-a-first-intro-worksheets/main/worksheet_classification1/data/clean-wdbc-data.csv)) bezogen werden.

```
data_url <- "https://raw.githubusercontent.com/UBC-DSCI/introduction-to-datasience/master/data/wdbc.csv"
cancer <- read_csv(data_url)
```

Timbers, Campbell, and Lee (2022) verwenden in Kap. 5 auch noch nicht standardisierte Daten, `unscales_wdbc.csv`, die hier ([https://raw.githubusercontent.com/UBC-DSCI/introduction-to-datasience/master/data/unscaled\\_wdbc.csv](https://raw.githubusercontent.com/UBC-DSCI/introduction-to-datasience/master/data/unscaled_wdbc.csv)) als CSV-Datei heruntergeladen werden können.

```
cancer_unscales_path <- "https://raw.githubusercontent.com/UBC-DSCI/introduction-to-datasience/master/data/unscaled_wdbc.csv"

unscaled_cancer <- read_csv(cancer_unscales_path) |>
  mutate(Class = as_factor(Class)) |>
  select(Class, Area, Smoothness)
unscaled_cancer
```

```
## # A tibble: 569 × 3
##   Class Area Smoothness
##   <fct> <dbl>      <dbl>
## 1 M     1001      0.118
## 2 M     1326      0.0847
## 3 M     1203      0.110
## 4 M     386.      0.142
## 5 M     1297      0.100
## 6 M     477.      0.128
## 7 M     1040      0.0946
## 8 M     578.      0.119
## 9 M     520.      0.127
## 10 M    476.      0.119
## # ... with 559 more rows
```

## 6.6.2 Rezept definieren

```
uc_recipe <- recipe(Class ~ ., data = unscaled_cancer)
print(uc_recipe)
```

```
## Recipe
##
## Inputs:
##
##       role #variables
##   outcome      1
## predictor      2
```

Und jetzt die z-Transformation:

```
uc_recipe <-
  uc_recipe |>
  step_scale(all_predictors()) |>
  step_center(all_predictors())
```

Die Schritte `prep()` und `bake()` sparen wir uns, da `fit()` und `predict()` das für uns besorgen.

## 6.6.3 Modell definieren

```
knn_spec <- nearest_neighbor(weight_func = "rectangular", neighbors = 5) |>
  set_engine("kknn") |>
  set_mode("classification")
knn_spec
```

```
## K-Nearest Neighbor Model Specification (classification)
##
## Main Arguments:
##   neighbors = 5
##   weight_func = rectangular
##
## Computational engine: kknn
```

## 6.6.4 Workflow definieren

```
knn_fit <- workflow() |>
  add_recipe(uc_recipe) |>
  add_model(knn_spec) |>
  fit(data = unscaled_cancer)

knn_fit
```

```

## == Workflow [trained] =====
## Preprocessor: Recipe
## Model: nearest_neighbor()
##
## — Preprocessor ——————
## 2 Recipe Steps
##
## • step_scale()
## • step_center()
##
## — Model ——————
##
## Call:
## kknn::train.kknn(formula = ..y ~ ., data = data, ks = min_rows(5,      data, 5), kernel = ~"rectangular")
##
## Type of response variable: nominal
## Minimal misclassification: 0.1107206
## Best kernel: rectangular
## Best k: 5

```

## 6.6.5 Vorhersagen

```

new_observation <- tibble(Area = c(500, 1500), Smoothness = c(0.075, 0.1))
prediction <- predict(knn_fit, new_observation)

prediction

```

```

## # A tibble: 2 × 1
##   .pred_class
##   <fct>
## 1 B
## 2 M

```

## 6.7 Mit Train-Test-Aufteilung

Im Kapitel 5 greifen Timbers, Campbell, and Lee (2022) die Aufteilung in Train- vs. Test-Sample noch nicht auf (aber in Kapitel 6).

Da in diesem Kurs diese Aufteilung aber schon besprochen wurde, soll dies hier auch dargestellt werden.

```

cancer_split <- initial_split(cancer, prop = 0.75, strata = Class)
cancer_train <- training(cancer_split)
cancer_test <- testing(cancer_split)

```

### 6.7.1 Rezept definieren

```

cancer_recipe <- recipe(Class ~ Smoothness + Concavity, data = cancer_train) |>
  step_scale(all_predictors()) |>
  step_center(all_predictors())

```

### 6.7.2 Modell definieren

```

knn_spec <- nearest_neighbor(weight_func = "rectangular", neighbors = 3) |>
  set_engine("kknn") |>
  set_mode("classification")

```

### 6.7.3 Workflow definieren

```

knn_fit <- workflow() |>
  add_recipe(cancer_recipe) |>
  add_model(knn_spec) |>
  fit(data = cancer_train)

knn_fit

```

```

## == Workflow [trained] =====
## Preprocessor: Recipe
## Model: nearest_neighbor()
##
## — Preprocessor ——————
## 2 Recipe Steps
##
## • step_scale()
## • step_center()
##
## — Model ——————
##
## Call:
## kknn::train.kknn(formula = ..y ~ ., data = data, ks = min_rows(3,      data, 5), kernel = ~"rectangular")
##
## Type of response variable: nominal
## Minimal misclassification: 0.1126761
## Best kernel: rectangular
## Best k: 3

```

## 6.7.4 Vorhersagen

Im Gegensatz zu Timbers, Campbell, and Lee (2022) verwenden wir hier `last_fit()` und `collect_metrics()`, da wir dies bereits eingeführt haben und künftig darauf aufbauen werden.

```

cancer_test_fit <- last_fit(knn_fit, cancer_split)

cancer_test_fit

## # Resampling results
## # Manual resampling
## # A tibble: 1 × 6
##   splits           id       .metrics .notes   .predictions .workflow
##   <list>         <chr>    <list>  <list>   <list>      <list>
## 1 <split [426/143]> train/test split <tibble> <tibble> <tibble>  <workflow>

```

## 6.7.5 Modellgüte

```

cancer_test_fit %>% collect_metrics()

## # A tibble: 2 × 4
##   .metric .estimator .estimate .config
##   <chr>   <chr>     <dbl> <chr>
## 1 accuracy binary     0.853 Preprocessor1_Model
## 2 roc_auc   binary     0.892 Preprocessor1_Model

```

Die eigentlichen Predictions stecken in der Listenpalte `.predictions` im Fit-Objekt:

```

names(cancer_test_fit)

## [1] "splits"        "id"          ".metrics"      ".notes"       ".predictions"
## [6] ".workflow"

```

Genau genommen ist `.predictions` eine Spalte, in der in jeder Zeile (und damit Zelle) eine Tabelle (Tibble) steht. Wir haben nur eine Zeile und wollen das erste Element dieser Spalte herausziehen. Da hilft `pluck()`:

```

cancer_test_predictions <-
cancer_test_fit %>%
  pluck(".predictions", 1)

confusion <- cancer_test_predictions |>
  conf_mat(truth = Class, estimate = .pred_class)

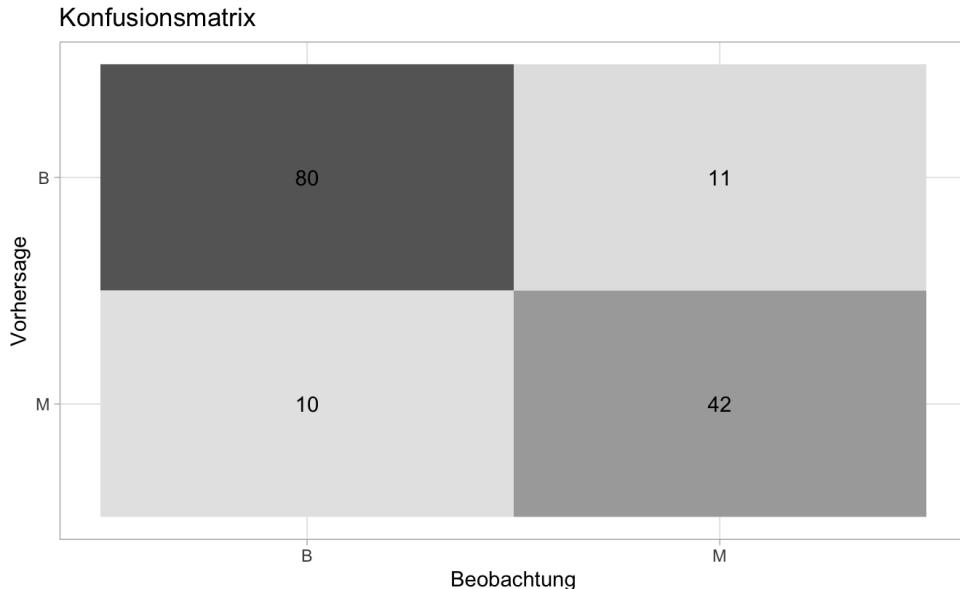
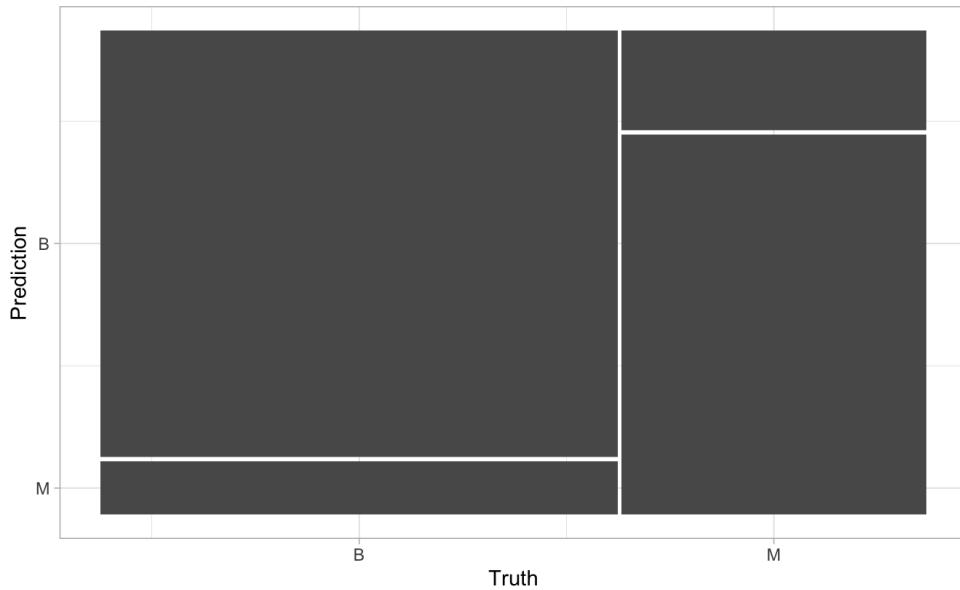
confusion

```

```
##          Truth
## Prediction B M
##          B 80 11
##          M 10 42
```

## 6.7.6 Visualisierung

```
autoplot(confusion, type = "mosaic")
autoplot(confusion, type = "heatmap") +
  labs(x = "Beobachtung",
       y = "Vorhersage",
       title = "Konfusionsmatrix")
```



## 6.8 Kennzahlen der Klassifikation

In Sauer (2019), Kap. 19.6, findet sich einige Erklärung zu Kennzahlen der Klassifikationsgüte.

Ein Test kann vier verschiedenen Ergebnisse haben:

Tabelle 6.1: Vier Arten von Ergebnissen von Klassifikationen

Wahrheit	Als negativ (-) vorhergesagt	Als positiv (+) vorhergesagt	Summe
In Wahrheit negativ (-)	Richtig negativ (RN)	Falsch positiv (FP)	N
In Wahrheit positiv (+)	Falsch negativ (FN)	Richtig positiv (RN)	P

Wahrheit	Als negativ (-) vorhergesagt	Als positiv (+) vorhergesagt	Summe
Summe	N*	P*	N+P

Es gibt eine verwirrende Vielfalt von Kennzahlen, um die Güte einer Klassifikation einzuschätzen. Hier sind einige davon:

### Geläufige Kennwerte der Klassifikation.

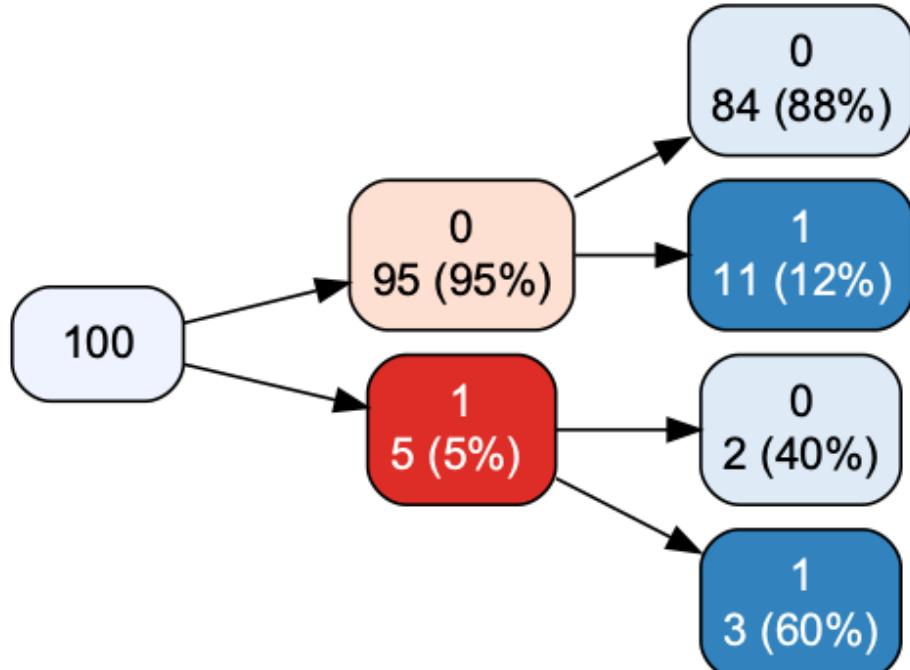
F: Falsch. R: Richtig. P: Positiv. N: Negativ

Name	Definition	Synonyme
FP-Rate	FP/N	Alphafehler, Typ-1-Fehler, 1-Spezifität, Fehlalarm
RP-Rate	RP/N	Power, Sensitivität, 1-Betafehler, Recall
FN-Rate	FN/N	Fehlender Alarm, Befafehler
RN-Rate	RN/N	Spezifität, 1-Alphafehler
Pos. Vorhersagewert	RP/P*	Präzision, Relevanz
Neg. Vorhersagewert	RN/N*	Segreganz
Richtigkeit	(RP+RN)/(N+P)	Korrektklassifikationsrate, Gesamtgenauigkeit

## 6.9 Krebstest-Beispiel

Betrachten wir Daten eines fiktiven Krebstest, aber realistischen Daten.

```
## # A tibble: 1 × 7
##   format width height colorspace matte filesize density
##   <chr>   <int>  <int> <chr>     <lgl>    <int> <chr>
## 1 PNG      500    429 sRGB      TRUE     40643 72x72
```



Krebs Test

Wie gut ist dieser Test? Berechnen wir einige Kennzahlen.

Da die Funktionen zur Klassifikation stets einen Faktor wollen, wandeln wir die relevanten Spalten zuerst in einen Faktor um (aktuell sind es numerische Spalten).

```
krebstest <-
  krebstest %>%
  mutate(Krebs = factor(Krebs),
        Test = factor(Test))
```

Gesamtgenauigkeit:

```
accuracy(krebstest, truth = Krebs, estimate = Test)
```

```
## # A tibble: 1 × 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy  binary      0.87
```

Sensitivität:

```
sens(krebstest, truth = Krebs, estimate = Test)
```

```
## # A tibble: 1 × 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 sens     binary      0.884
```

Spezifität:

```
spec(krebstest, truth = Krebs, estimate = Test)
```

```
## # A tibble: 1 × 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 spec     binary      0.6
```

Kappa:

```
kap(krebstest, truth = Krebs, estimate = Test)
```

```
## # A tibble: 1 × 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 kap      binary      0.261
```

Positiver Vorhersagewert:

```
ppv(krebstest, truth = Krebs, estimate = Test)
```

```
## # A tibble: 1 × 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 ppv     binary      0.977
```

Negativer Vorhersagewert:

```
npv(krebstest, truth = Krebs, estimate = Test)
```

```
## # A tibble: 1 × 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 npv     binary      0.214
```

Während Sensitivität und Spezifität sehr hoch sind, ist die der negative Vorhersagewert sehr gering:

Wenn man einen positiven Test erhält, ist die Wahrscheinlichkeit, in Wahrheit krank zu sein gering, zum Glück!

## 6.10 Aufgaben

- Arbeiten Sie sich so gut als möglich durch diese Analyse zum Verlauf von Covid-Fällen (<https://github.com/sebastiansauer/covid-icu>)
- Fallstudie zur Modellierung einer logististischen Regression mit tidymodels (<https://onezero.blog/modelling-binary-logistic-regression-using-tidymodels-library-in-r-part-1/>)
- Fallstudie zu Vulkanausbrüchen (<https://juliasilge.com/blog/multinomial-volcano-eruptions/>)
- Fallstudie Himalaya (<https://juliasilge.com/blog/himalayan-climbing/>)

# 7 Resampling und Tuning

Benötigte R-Pakete für dieses Kapitel:

```
library(tidyverse)
library(tidymodels)
library(tune) # wird nicht automatisch mit tidymodels gestartet
```

## 7.1 Lernsteuerung

### 7.1.1 Vorbereitung

- Lesen Sie die Literatur.

### 7.1.2 Lernziele

- Sie verstehen den Nutzen von Resampling und Tuning im maschinellen Nutzen.
- Sie können Methoden des Resampling und Tunings mit Hilfe von Tidymodels anwenden.

### 7.1.3 Literatur

- Rhys, Kap. 3
- TMWR, Kap. 10, 12

## 7.2 Überblick

Der Standardablauf des maschinellen Lernens ist in Abb. 7.1 dargestellt. Eine alternative, hilfreich Abbildung findet sich hier (<https://www.tmwr.org/resampling.html>) in Kap. 10.2 in Silge and Kuhn (2022).

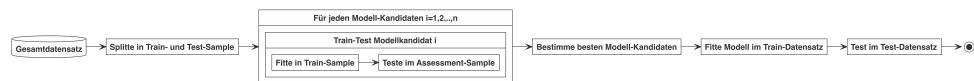


Abbildung 7.1: Standardablauf des maschinellen Lernens mit Tuning und Resampling

## 7.3 tidymodels

### 7.3.1 Datensatz aufteilen

```
data(ames)

set.seed(4595)
data_split <- initial_split(ames, strata = "Sale_Price")

ames_train <- training(data_split)
ames_test <- testing(data_split)
```

### 7.3.2 Rezept, Modell und Workflow definieren

In gewohnter Weise definieren wir den Workflow mit einem kNN-Modell.

```

ames_rec <-
  recipe(Sale_Price ~ ., data = ames_train) %>%
  step_log(Sale_Price, base = 10) %>%
  step_other(Neighborhood, threshold = .1) %>%
  step_dummy(all_nominal()) %>%
  step_zv(all_predictors())

knn_model <-
  nearest_neighbor(
    mode = "regression",
  ) %>%
  set_engine("kknn")

ames_wflow <-
  workflow() %>%
  add_recipe(ames_rec) %>%
  add_model(knn_model)

```

Das kNN-Modell ist noch *nicht berechnet*, es ist nur ein “Rezept” erstellt:

```
knn_model
```

```

## K-Nearest Neighbor Model Specification (regression)
##
## Computational engine: kknn

```

```
ames_wflow
```

```

## == Workflow ==
## Preprocessor: Recipe
## Model: nearest_neighbor()
##
## — Preprocessor —
## 4 Recipe Steps
##
## • step_log()
## • step_other()
## • step_dummy()
## • step_zv()
##
## — Model —
## K-Nearest Neighbor Model Specification (regression)
##
## Computational engine: kknn

```

## 7.4 Resampling

Vergleichen Sie die drei Fälle, die sich in der Nutzung von Train- und Test-Sample unterscheiden:

1. Wir fitten ein Klassifikationsmodell in einer Stichprobe, sagen die Y-Werte dieser Stichprobe “vorher”. Wir finden eine Gesamtgenauigkeit von 80%.
2. Wir fitten ein Klassifikationsmodell in einem Teil der ursprünglichen Stichprobe (Train-Sample) und sagen Y-die Werte im verbleibenden Teil der ursprünglichen Stichprobe vorher (Test-Sample). Wir finden eine Gesamtgenauigkeit von 70%.
3. Wir wiederholen Fall 2 noch drei Mal mit jeweils anderer Zuweisung der Fälle zum Train- bzw. zum Test-Sample. Wir finden insgesamt folgende Werte an Gesamtgenauigkeit: 70%, 70%, 65%, 75%.

Welchen der drei Fälle finden Sie am sinnvollsten? Warum?

## 7.5 Illustration des Resampling

*Resampling* stellt einen Oberbegriff dar; *Kreuzvalidierung* ist ein Unterbegriff dazu. Es gibt noch andere Arten des Resampling, etwa *Bootstrapping* oder *Leave-One-Out-Cross-Validation (LOOCV)*.

Im Folgenden ist nur die Kreuzvalidierung dargestellt, da es eines der wichtigsten und vielleicht das Wichtigste ist. In vielen Quellen finden sich Erläuterungen anderer Verfahren dargestellt, etwa in Silge and Kuhn (2022), James et al. (2021) oder Rhys (2020).

### 7.5.1 Einfache v-fache Kreuzvalidierung

Abb. 7.2 illustriert die zufällige Aufteilung von  $n = 10$  Fällen der Originalstichprobe auf eine Train- bzw. Test-Stichprobe. Man spricht von Kreuzvalidierung (cross validation, CV).

In diesem Fall wurden 70% der ( $n = 10$ ) Fälle der Train-Stichprobe zugewiesen (der Rest der Test-Stichprobe); ein willkürlicher, aber nicht unüblicher Anteil. Diese Aufteilung wurde  $v = 3$  Mal vorgenommen, es resultieren drei "Resampling-Stichproben", die manchmal auch als "Faltungen" bezeichnet werden.

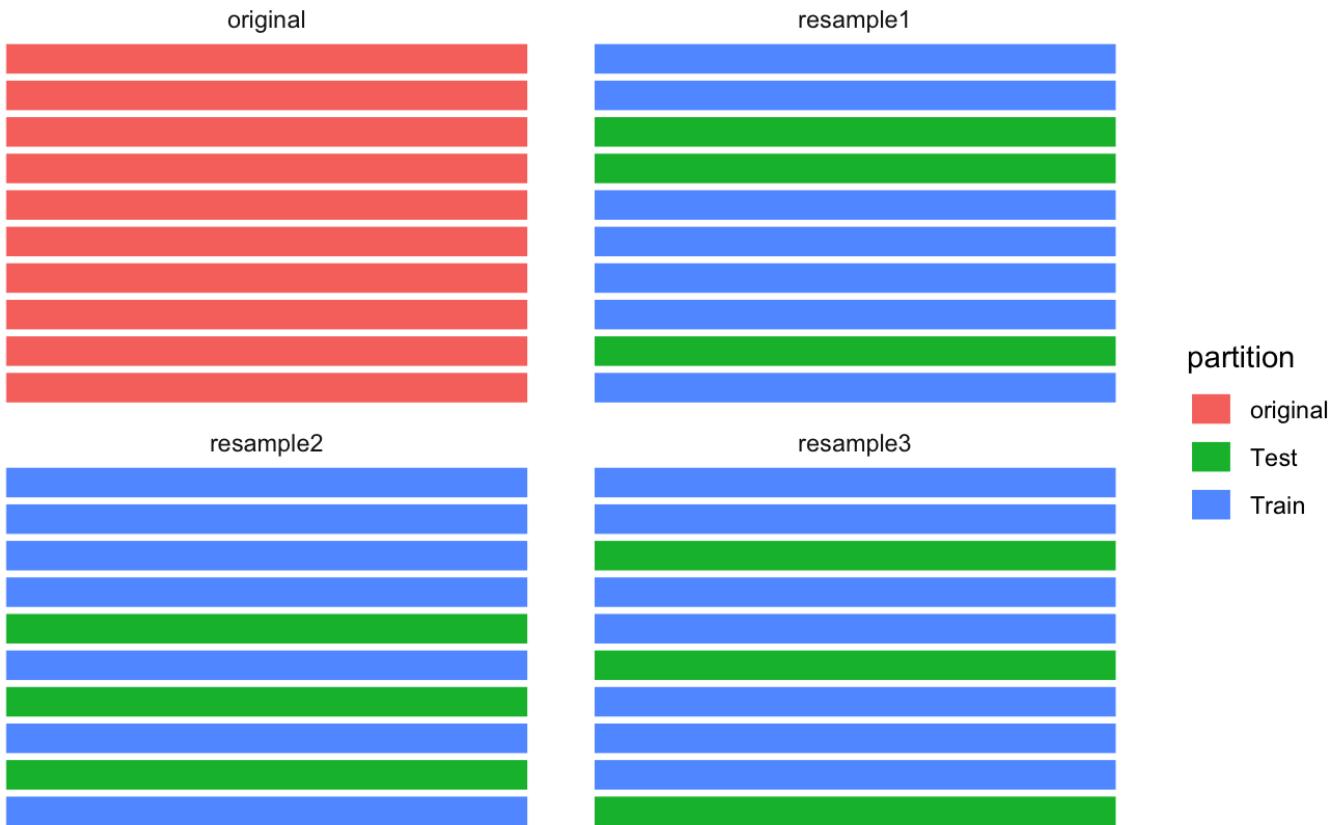


Abbildung 7.2: Resampling: Eine Stichprobe wird mehrfach (hier 3 Mal) zu 70% in ein Train- und zu 30% in die Test-Stichprobe aufgeteilt

Sauer (2019) stellt das Resampling so dar (S. 259), s. Abb. 7.3.

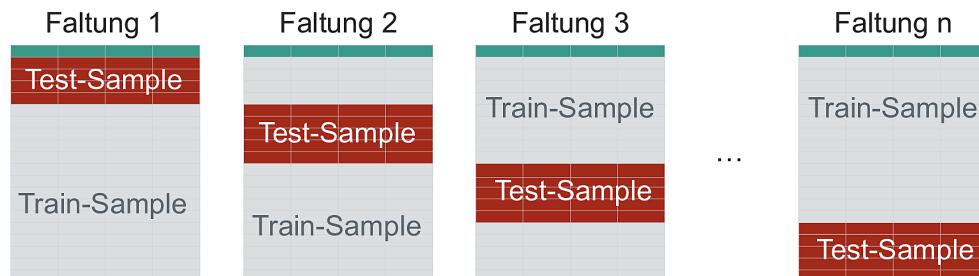


Abbildung 7.3: Kreuzvalidierung, Aufteilung in Train- vs. Testsample

Der Gesamtfehler der Vorhersage wird als Mittelwerte der Vorhersagefehler in den einzelnen Faltungen berechnet.

Warum ist die Vorhersage besser, wenn man mehrere Faltungen, mehrere Schätzungen für  $y$  also, vornimmt?

Der Grund ist das Gesetz der großen Zahl, nachdem sich eine Schätzung in Mittelwert und Variabilität stabilisiert mit steigendem Stichprobenumfang, dem wahren Mittelwert also präziser schätzt. Bei Normalverteilungen klappt das gut, bei randlastigen Verteilungen leider nicht mehr (Taleb 2019).

Häufig werden  $v = 10$  Faltungen verwendet, was sich empirisch als guter Kompromiss von Rechenaufwand und Fehlerreduktion herausgestellt hat.

## 7.5.2 Wiederholte Kreuzvalidierung

Die  $r$ -fach wiederholte Kreuzvalidierung wiederholte die einfache Kreuzvalidierung mehrfach (nämlich  $r = 4$  mal), Sauer (2019) stellt das Resampling so dar (S. 259), s. Abb. 7.4.

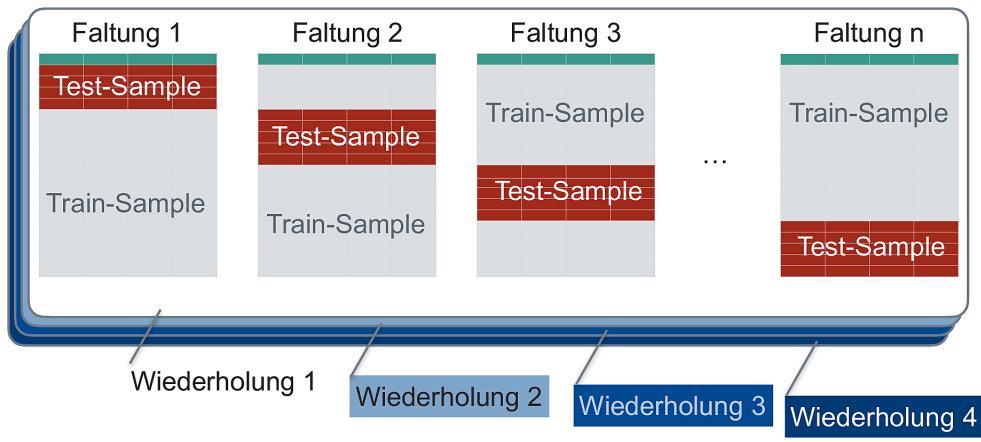
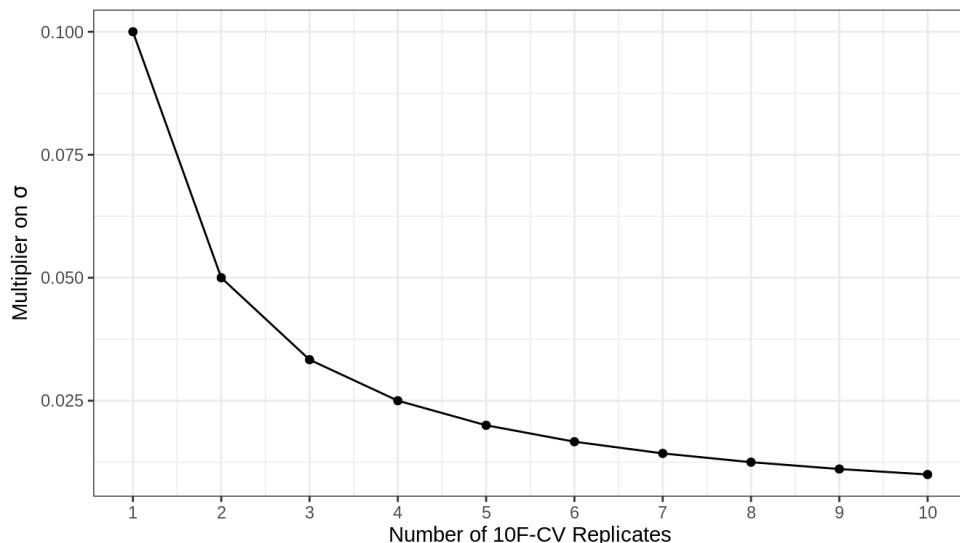


Abbildung 7.4: Wiederholte Kreuzvalidierung

Die wiederholte Kreuzvalidierung reduziert den Standardfehler der Vorhersagen.

Silge and Kuhn (2022) zeigen die Verringerung des Schätzfehlers als Funktion der  $r$  Wiederholungen dar, s. Abb. 7.5.

Abbildung 7.5: Reduktion des Schätzfehlers als Funktion der  $r$  Wiederholungen der Kreuzvalidierung

Warum ist die Wiederholung der Kreuzvalidierung nützlich?

Die Kreuzvalidierung liefert einen Schätzwert der Modellparameter, die wahren Modellparameter werden also anhand einer Stichprobe von  $n = 1$  geschätzt. Mit höherem Stichprobenumfang kann diese Schätzung natürlich präzisiert werden.

Da jede Stichprobenverteilung bei  $n \rightarrow \infty$  normalverteilt ist - ein zentrales Theorem der Statistik, der *Zentrale Grenzwertsatz* (Central Limit Theorem) - kann man hoffen, dass sich eine bestimmte Stichprobenverteilung bei kleinerem  $n$  ebenfalls annähernd normalverteilt<sup>3</sup>. Dann sind die Quantile bekannt und man kann die Streuung des Schätzers,  $\sigma_{\hat{x}}$ , z.B. für den Mittelwert, einfach schätzen:

$$\sigma_{\hat{x}} = \frac{\sigma}{\sqrt{n}}$$

### 7.5.3 Resampling passiert im Train-Sample

Wichtig zu beachten ist, dass die Resampling nur im Train-Sample stattfindet. Das Test-Sample bleibt unangerührt. Dieser Sachverhalt ist in Abb. 7.6, aus Silge and Kuhn (2022), illustriert.

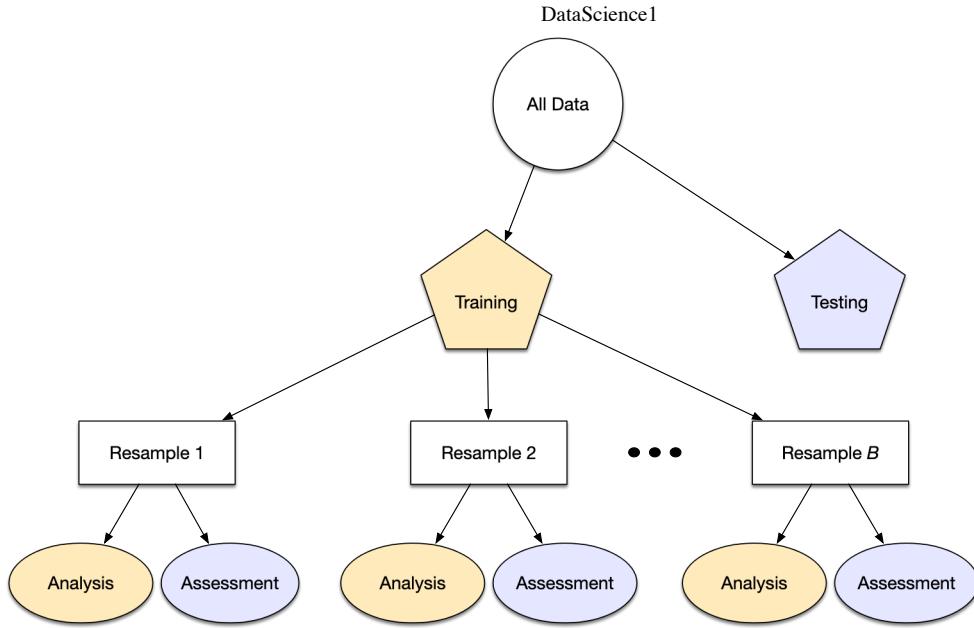


Abbildung 7.6: Resampling im Train-, nicht im Test-Sample

Wie in Abb. 7.6 dargestellt, wird das Modell im *Analysis-Sample* berechnet (gefittet), und im *Assessment-Sample* auf Modellgüte hin überprüft.

Die letzliche Modellgüte ist dann die Zusammenfassung (Mittelwert) der einzelnen Resamples.

## 7.5.4 Andere Illustrationen

Es gibt eine Reihe vergleichbarer Illustrationen in anderen Büchern:

- Timbers, Campbell & Lee, 2022, Kap. 6 (<https://datasciencebook.ca/img/cv.png>)
- Silge & Kuhn, 2022, Abb. 10.1 (<https://datasciencebook.ca/img/cv.png>)
- Silge & Kuhn, 2022, Abb. 10.2 (<https://www.tmwr.org/premade/three-CV.svg>)
- Silge & Kuhn, 2022, Abb. 10.3 (<https://www.tmwr.org/premade/three-CV-iter.svg>)
- James, Witten, hastie & Tishirani, 2021, Abb. 5.3

## 7.6 Gesetz der großen Zahl

Nach dem *Gesetz der großen Zahl* (Law of Large Numbers) sollte sich der Mittelwert einer großen Stichprobe dem theoretischen Mittelwert der zugrundeliegenden Verteilung (Population, datengenerierender Prozess) sehr nahe kommen.

$$\lim_{n \rightarrow \infty} \sum_{i=1}^n \frac{X_i}{n} = \bar{X}$$

David Salazar visualisiert das folgendermaßen in diesem Post (<https://david-salazar.github.io/2020/04/17/fat-vs-thin-does-lln-work/>) seines lezenswerten Blogs (<https://david-salazar.github.io/>), s. Abb. 7.7.

```
# source: https://david-salazar.github.io/2020/04/17/fat-vs-thin-does-lnn-work/
samples <- 1000

thin <- rnorm(samples, sd = 20)

cumulative_mean <- function(numbers) {
  x <- seq(1, length(numbers))
  cum_mean <- cumsum(numbers)/x
  cum_mean
}

thin_cum_mean <- cumulative_mean(thin)

thin_cum_mean %>%
  tibble(running_mean = .) %>%
  add_rownames(var = 'number_samples') %>%
  mutate(number_samples = as.double(number_samples)) %>%
  arrange(number_samples) %>%
  ggplot(aes(x = number_samples, y = running_mean)) +
  geom_line(color = 'dodgerblue4') +
  geom_hline(yintercept = 0, linetype = 2, color = 'red') +
  hrbrthemes::theme_ipsum_rc(grid = 'Y') +
  scale_x_continuous(labels = scales::comma) +
  labs(x = "Stichprobengröße",
       title = "Gesetz der großen Zahl",
       subtitle = "Kumulierter Mittelwert aus einer Normalverteilung mit sd=20")
```

## Gesetz der großen Zahl

Kumulierter Mittelwert aus einer Normalverteilung mit sd=20

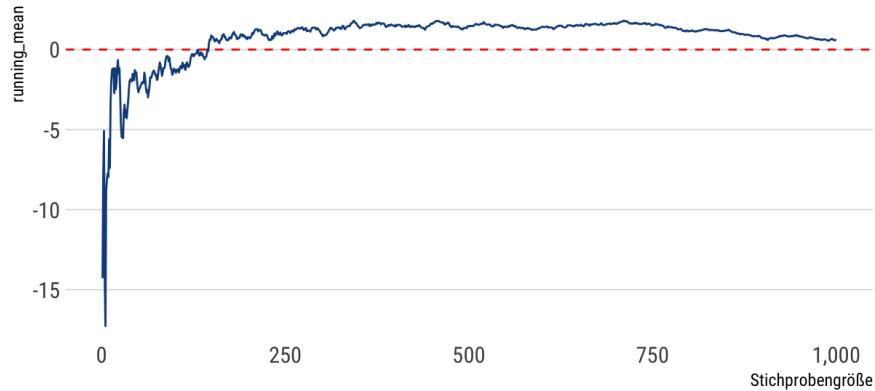


Abbildung 7.7: Gesetz der großen Zahl

Wie man sieht, nähert sich der empirische Mittelwert (also in der Stichprobe) immer mehr dem theoretischen Mittelwert, 0, an.

Achtung: Bei randlastigen Verteilungen darf man dieses schöne, wohlerzogene Verhalten nicht erwarten (Taleb 2019).

## 7.7 Über- und Unteranpassung an einem Beispiel

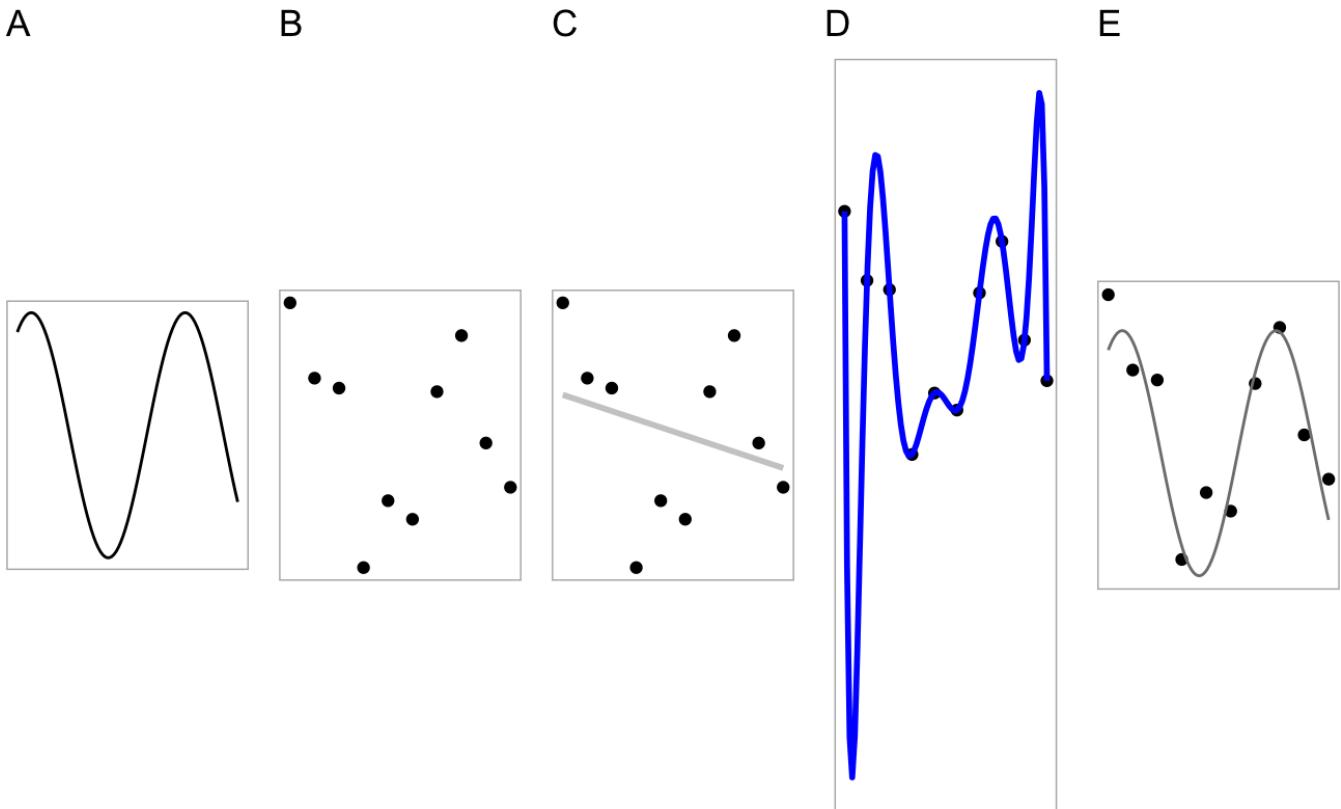


Abbildung 7.8: Welches Modell (Teile C-E) passt am besten zu den Daten (Teil B)? Die ‘wahre Funktion’, der datengenerierende Prozess ist im Teil A dargestellt

Abb. 7.8 zeigt:

- Teil A: Die ‘wahre Funktion’,  $f$ , die die Daten erzeugt. Man spricht auch von der “datengenerierenden Funktion”. Wir gehen gemeinhin davon aus, dass es eine wahre Funktion gibt. Das heißt nicht, dass die wahre Funktion die Daten perfekt erklärt, schließlich kann die Funktion zwar wahr, aber unvollständig sein oder unsere Messinstrumente sind nicht perfekt präzise.
- Teil B: Die Daten, erzeugt aus A plus etwas zufälliges Fehler (Rauschen).
- Teil C: Ein zu einfaches Modell: Unteranpassung. Vorhersagen in einer neuen Stichprobe (basierend auf dem datengenerierenden Prozess aus A) werden nicht so gut sein.
- Teil D: Ein zu komplexes Modell: Überanpassung. Vorhersagen in einer neuen Stichprobe (basierend auf dem datengenerierenden Prozess aus A) werden nicht so gut sein.
- Teil E: Ein Modell mittlerer Komplexität. Keine Überanpassung, keine Unteranpassung. Vorhersagen in einer neuen Stichprobe (basierend auf dem datengenerierenden Prozess aus A) werden gut sein.

## 7.8 CV in tidymodels

### 7.8.1 CV definieren

So kann man eine *einfache v-fache Kreuzvalidierung* in Tidymodels auszeichnen:

```
set.seed(2453)
ames_folds <- vfold_cv(ames_train, strata = "Sale_Price")
ames_folds
```

```
## # 10-fold cross-validation using stratification
## # A tibble: 10 × 2
##   splits      id
##   <list>      <chr>
## 1 1 <split [1976/221]> Fold01
## 2 2 <split [1976/221]> Fold02
## 3 3 <split [1976/221]> Fold03
## 4 4 <split [1976/221]> Fold04
## 5 5 <split [1977/220]> Fold05
## 6 6 <split [1977/220]> Fold06
## 7 7 <split [1978/219]> Fold07
## 8 8 <split [1978/219]> Fold08
## 9 9 <split [1979/218]> Fold09
## 10 10 <split [1980/217]> Fold10
```

Werfen wir einen Blick in die Spalte `splits`, erste Zeile:

```
ames_folds %>% pluck(1, 1)
```

```
## <Analysis/Assess/Total>
## <1976/221/2197>
```

Möchte man die Defaults von `vfold_cv` wissen, schaut man in der Hilfe nach: `?vfold_cv`:

```
vfold_cv(data, v = 10, repeats = 1, strata = NULL, breaks = 4, pool = 0.1, ...)
```

Probieren wir  $v = 5$  und  $r = 2$ :

```
ames_folds_rep <- vfold_cv(ames_train,
                           strata = "Sale_Price",
                           v = 5,
                           repeats = 2)
ames_folds_rep
```

```
## # 5-fold cross-validation repeated 2 times using stratification
## # A tibble: 10 × 3
##   splits           id    id2
##   <list>          <chr> <chr>
## 1 1 <split [1756/441]> Repeat1 Fold1
## 2 2 <split [1757/440]> Repeat1 Fold2
## 3 3 <split [1757/440]> Repeat1 Fold3
## 4 4 <split [1758/439]> Repeat1 Fold4
## 5 5 <split [1760/437]> Repeat1 Fold5
## 6 6 <split [1756/441]> Repeat2 Fold1
## 7 7 <split [1757/440]> Repeat2 Fold2
## 8 8 <split [1757/440]> Repeat2 Fold3
## 9 9 <split [1758/439]> Repeat2 Fold4
## 10 10 <split [1760/437]> Repeat2 Fold5
```

## 7.8.2 Resamples fitten

Hat unser Computer mehrere Rechenkerne, dann können wir diese nutzen und die Berechnungen beschleunigen. Im Standard wird sonst nur ein Kern verwendet.

```
mycores <- parallel::detectCores(logical = FALSE)
mycores
```

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

Auf Unix/MacOC-Systemen kann man dann die Anzahl der parallelen Kerne so einstellen:

```
library(doMC)
registerDoMC(cores = mycores)
```

So, und jetzt fitten wir die Resamples und trachten die Modellgüte in den Resamples:

```
ames_resamples_fit <-
  ames_wflow %>%
  fit_resamples(ames_folds)

ames_resamples_fit %>%
  collect_metrics()
```

```
## # A tibble: 2 × 6
##   .metric .estimator   mean   n std_err .config
##   <chr>   <chr>     <dbl> <int>  <dbl> <chr>
## 1 rmse    standard    0.0928    10  0.00187 Preprocessor1_Model1
## 2 rsq     standard    0.722     10  0.00864 Preprocessor1_Model1
```

Natürlich interessiert uns primär die Modellgüte im Test-Sample:

```

final_ames <-
last_fit(ames_wflow, data_split)

final_ames %>%
collect_metrics()

## # A tibble: 2 × 4
##   .metric .estimator .estimate .config
##   <chr>   <chr>        <dbl> <chr>
## 1 rmse    standard     0.103 Preprocessor1_Model
## 2 rsq     standard     0.678 Preprocessor1_Model

```

## 7.9 Tuning

### 7.9.1 Tuning auszeichnen

In der Modellspezifikation des Modells können wir mit `tune()` auszeichnen, welche Parameter wir tunen möchten. Wir können

```

knn_model <-
nearest_neighbor(
  mode = "regression",
  neighbors = tune()
) %>%
set_engine("kknn")

```

Wir können dem Tuningparameter auch einen Namen (ID/Laben) geben, z.B. "K":

```

knn_model <-
nearest_neighbor(
  mode = "regression",
  neighbors = tune("K")
) %>%
set_engine("kknn")

```

### 7.9.2 Grid Search vs. Iterative Search

Im K-Nächste-Nachbarn-Modell ist der vorhergesagte Wert,  $\hat{y}$  für eine neue Beobachtung  $x_0$  wie folgt definiert:

$$\hat{y} = \frac{1}{K} \sum_{\ell=1}^K x_{\ell}^{*},$$

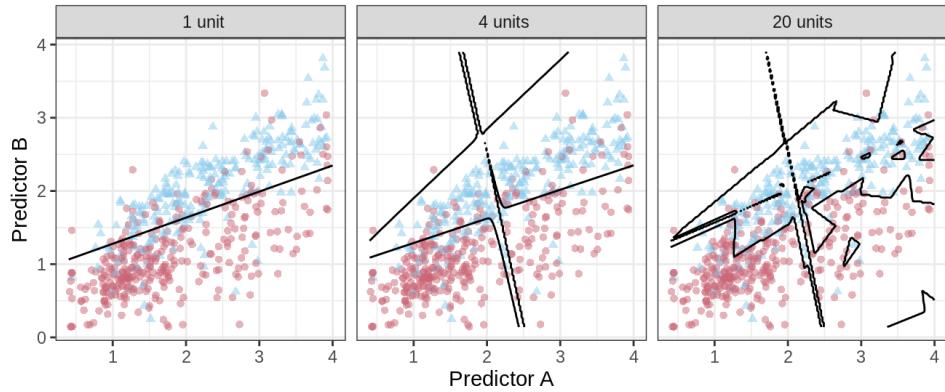
wobei  $K$  die Anzahl der zu berücksichtigen nächsten Nachbarn darstellt und  $x_{\ell}^{*}$  die Werte dieser berücksichtigten Nachbarn.

Die Wahl von  $K$  hat einen gewaltigen Einfluss auf die Vorhersagen und damit auf die Vorhersagegüte. Allerdings wird  $K$  nicht vom Modell geschätzt. Es liegt an den Nutzi, diesen Wert zu wählen.

Parameter dieser Art (die von den Nutzi zu bestimmen sind, nicht vom Algorithmus), nennt man *Tuningparameter*.

Abbildung 7.9 aus Silge and Kuhn (2022) stellt exemplarisch dar, welchen großen Einfluss die Wahl des Werts eines Tuningparameters auf die Vorhersagen eines Modells haben.

## Training Set



## Test Set

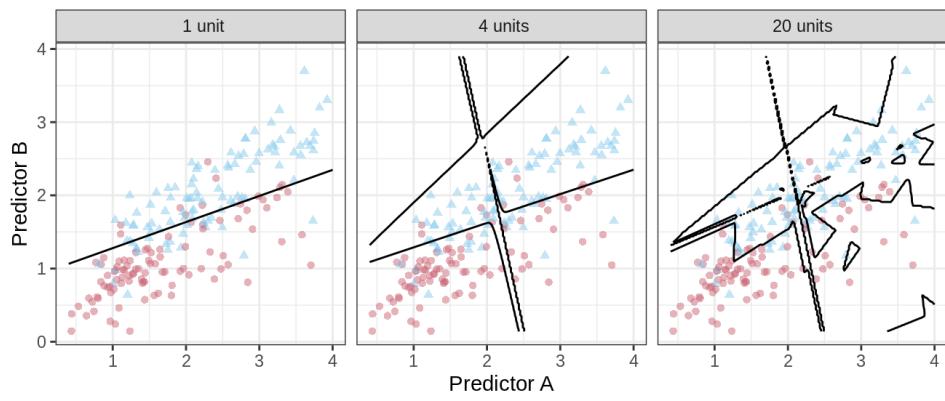


Abbildung 7.9: Overfitting als Funktion der Modellparameter und insofern als Problem de Wahl der Tuningparameter

Aber wie wählt man "gute" Werte der Tuningparater? Zwei Ansätze, grob gesprochen, bieten sich an.

1. *Grid Search*: Probiere viele Werte aus und schaue, welcher der beste ist. Dabei musst du hoffen, dass du die Werte erwischst, die nicht nur im Train-, sondern auch im Test-Sample gut funktionieren werden.
2. *Iterative Search*: Wenn du einen Wert eines Tuningparameters hast, nutze diesen, um intelligenter einen neuen Wert eines Tuningparameters zu finden.

Der Unterschied beider Ansätze ist in Silge and Kuhn (2022) wie in Abb. 7.10 dargestellt.

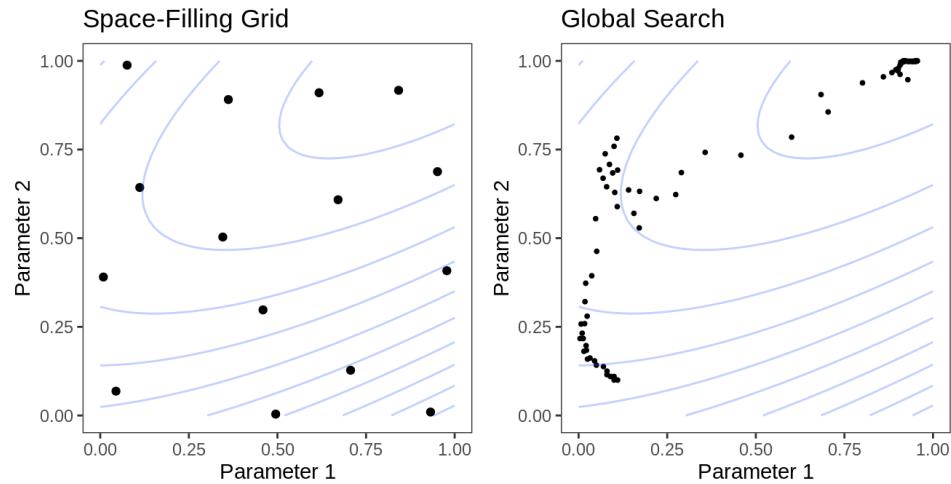


Abbildung 7.10: Links: Grid Search. Rechts: Iterative Search2

In `tidymodels` kann man mit `tune()` angeben, dass man einen bestimmten Parameter tunen möchte. `tidymodels` führt das dann ohne weiteres Federlesens für uns durch.

## 7.10 Tuning mit Tidymodels

### 7.10.0.1 Tuningparameter betrachten

Möchte man wissen, welche und wie viele Tuningparameter `tidymodels` in einem Modell berücksichtigt, kann man `extract_parameter_set_dials()` aufrufen:

```
extract_parameter_set_dials(knn_model)
```

```
## Collection of 1 parameters for tuning
##
##  identifier      type    object
##        K neighbors nparam[+]
```

Die Ausgabe informiert uns, dass es nur einen Tuningparameter gibt in diesem Modell und dass der Name (Label, ID) des Tuningparameters "K" ist. Außerdem sollen die Anzahl der Nachbarn getunnt werden. Der Tuningparameter ist numerisch; das sieht man an `nparam[+]`.

Schauen wir uns mal an, auf welchen Wertebereich `tidymodels` den Parameter  $K$  begrenzt hat:

```
knn_model %>%
  extract_parameter_dials("K")
```

```
## # Nearest Neighbors (quantitative)
## Range: [1, 15]
```

Aktualisieren wir mal unseren Workflow entsprechend:

```
ames_wflow <-
  ames_wflow %>%
  update_model(knn_model)
```

Wir können auch Einfluss nehmen und angeben, dass die Grenzen des Wertebereichs zwischen 1 und 50 liegen soll (für den Tuningparameter `neighbors`):

```
ames_set <-
  extract_parameter_set_dials(ames_wflow) %>%
  update(K = neighbors(c(1, 50)))

ames_set
```

```
## Collection of 1 parameters for tuning
##
## identifier      type    object
##                 K neighbors nparam[+]
```

## 7.10.1 Datenabhängige Tuningparameter

Manche Tuningparameter kann man nur bestimmen, wenn man den Datensatz kennt. So ist die Anzahl der Prädiktoren, `mtry` in einem Random-Forest-Modell sinnvollerweise als Funktion der Prädiktorenanzahl zu wählen. Der Workflow kennt aber den Datensatz nicht. Daher muss der Workflow noch "finalisiert" oder "aktualisiert" werden, um den Wertebereich (Unter- und Obergrenze) eines Tuningparameters zu bestimmen.

Wenn wir im Rezept aber z.B. die Anzahl der Prädiktoren verändert haben, möchten wir die Grenzen des Wertebereichs für `mtry` (oder andere Tuningparameter) vielleicht nicht händisch, "hartverdrahtet" selber bestimmen, sondern lieber den Computer anweisen, und sinngemäß sagen: "Warte mal mit der Bestimmung der Werte der Tuningparameter, bis du den Datensatz bzw. dessen Dimensionen kennst. Merk dir, dass du, wenn du den Datensatz kennst, die Werte des Tuningparameter noch ändern musst. Und tu das dann auch." Dazu später mehr.

```
ames_set <-
  workflow() %>%
  add_model(knn_model) %>%
  add_recipe(ames_rec) %>%
  extract_parameter_set_dials() %>%
  finalize(ames_train)
```

## 7.10.2 Modelle mit Tuning berechnen

Nachdem wir die Tuningwerte bestimmt haben, können wir jetzt das Modell berechnen: Für jeden Wert des Tuningparameters wird ein Modell berechnet:

```
ames_grid_search <-
  tune_grid(
    ames_wf,
    resamples = ames_folds
  )
ames_grid_search

## # Tuning results
## # 10-fold cross-validation using stratification
## # A tibble: 10 × 4
##   splits          id    .metrics      .notes
##   <list>         <chr> <list>        <list>
## 1 <split [1976/221]> Fold01 <tibble [16 × 5]> <tibble [0 × 3]>
## 2 <split [1976/221]> Fold02 <tibble [16 × 5]> <tibble [0 × 3]>
## 3 <split [1976/221]> Fold03 <tibble [16 × 5]> <tibble [0 × 3]>
## 4 <split [1976/221]> Fold04 <tibble [16 × 5]> <tibble [0 × 3]>
## 5 <split [1977/220]> Fold05 <tibble [16 × 5]> <tibble [0 × 3]>
## 6 <split [1977/220]> Fold06 <tibble [16 × 5]> <tibble [0 × 3]>
## 7 <split [1978/219]> Fold07 <tibble [16 × 5]> <tibble [0 × 3]>
## 8 <split [1978/219]> Fold08 <tibble [16 × 5]> <tibble [0 × 3]>
## 9 <split [1979/218]> Fold09 <tibble [16 × 5]> <tibble [0 × 3]>
## 10 <split [1980/217]> Fold10 <tibble [16 × 5]> <tibble [0 × 3]>
```

Im Default berechnet `tiymodels` 10 Kandidatenmodelle.

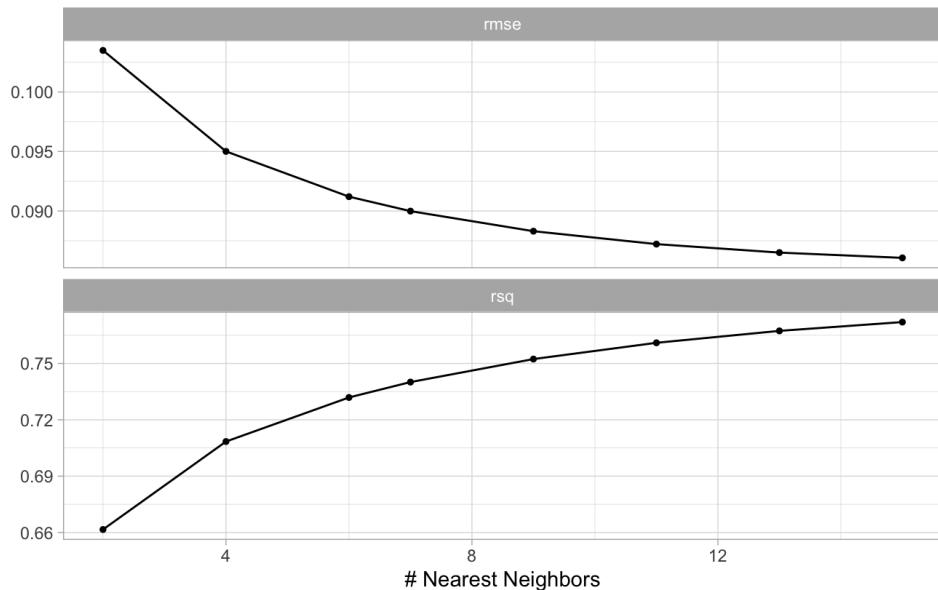
Die Spalte `.metrics` beinhaltet die Modellgüte für jedes Kandidatenmodell.

```
ames_grid_search %>%
  collect_metrics()
```

```
## # A tibble: 16 × 7
##   K .metric .estimator  mean    n std_err .config
##   <int> <chr>   <chr>     <dbl> <int>   <dbl> <chr>
## 1 2 rmse    standard  0.103    10 0.00213 Preprocessor1_Model1
## 2 2 rsq     standard  0.662    10 0.0112  Preprocessor1_Model1
## 3 4 rmse    standard  0.0950   10 0.00188 Preprocessor1_Model2
## 4 4 rsq     standard  0.708    10 0.00916 Preprocessor1_Model2
## 5 6 rmse    standard  0.0912   10 0.00189 Preprocessor1_Model3
## 6 6 rsq     standard  0.732    10 0.00842 Preprocessor1_Model3
## 7 7 rmse    standard  0.0900   10 0.00192 Preprocessor1_Model4
## 8 7 rsq     standard  0.740    10 0.00829 Preprocessor1_Model4
## 9 9 rmse    standard  0.0883   10 0.00201 Preprocessor1_Model5
## 10 9 rsq    standard  0.752    10 0.00827 Preprocessor1_Model5
## 11 11 rmse   standard  0.0872   10 0.00211 Preprocessor1_Model6
## 12 11 rsq    standard  0.761    10 0.00845 Preprocessor1_Model6
## 13 13 rmse   standard  0.0865   10 0.00217 Preprocessor1_Model7
## 14 13 rsq    standard  0.767    10 0.00848 Preprocessor1_Model7
## 15 15 rmse   standard  0.0861   10 0.00221 Preprocessor1_Model8
## 16 15 rsq    standard  0.772    10 0.00850 Preprocessor1_Model8
```

Das können wir uns einfach visualisieren lassen:

```
autoplot(ames_grid_search)
```



Auf Basis dieser Ergebnisse könnte es Sinn machen, noch größere Werte für  $K$  zu überprüfen.

### 7.10.3 Vorhersage im Test-Sample

Welches Modellkandidat war jetzt am besten?

```
show_best(ames_grid_search)
```

```
## # A tibble: 5 × 7
##   K .metric .estimator  mean    n std_err .config
##   <int> <chr>   <chr>     <dbl> <int>   <dbl> <chr>
## 1 15 rmse    standard  0.0861   10 0.00221 Preprocessor1_Model8
## 2 13 rmse    standard  0.0865   10 0.00217 Preprocessor1_Model7
## 3 11 rmse    standard  0.0872   10 0.00211 Preprocessor1_Model6
## 4 9 rmse    standard  0.0883   10 0.00201 Preprocessor1_Model5
## 5 7 rmse    standard  0.0900   10 0.00192 Preprocessor1_Model4
```

Wählen wir jetzt mal das beste Modell aus (im Sinne des Optimierungskriteriums):

```
select_best(ames_grid_search)
```

```
## # A tibble: 1 × 2
##       K .config
##   <int> <chr>
## 1     15 Preprocessor1_Model8
```

Ok, notieren wir uns die Kombination der Tuningparameterwerte im besten Kandidatenmodell. In diesem Fall hat das Modell nur einen Tuningparameter:

```
ames_knn_best_params <-
  tibble(K = 15)
```

Unser Workflow weiß noch nicht, welche Tuningparameterwerte am besten sind:

```
ames_wflow
```

```
## == Workflow ==
## Preprocessor: Recipe
## Model: nearest_neighbor()
##
## — Preprocessor ——————
## 4 Recipe Steps
##
## • step_log()
## • step_other()
## • step_dummy()
## • step_zv()
##
## — Model ——————
## K-Nearest Neighbor Model Specification (regression)
##
## Main Arguments:
##   neighbors = tune("K")
##
## Computational engine: kknn
```

`neighbors = tune("K")` sagt uns, dass er diesen Parameter tunen will. Das haben wir jetzt ja erledigt. Wir wollen für das Test-Sample nur noch einen Wert, eben aus dem besten Kandidatenmodell, verwenden:

```
ames_final_wflow <-
  ames_wflow %>%
  finalize_workflow(ames_knn_best_params)

ames_final_wflow
```

```
## == Workflow ==
## Preprocessor: Recipe
## Model: nearest_neighbor()
##
## — Preprocessor ——————
## 4 Recipe Steps
##
## • step_log()
## • step_other()
## • step_dummy()
## • step_zv()
##
## — Model ——————
## K-Nearest Neighbor Model Specification (regression)
##
## Main Arguments:
##   neighbors = 15
##
## Computational engine: kknn
```

Wie man sieht, steht im Workflow nichts mehr von Tuningparameter.

Wir können jetzt das *ganze Train-Sample* fitten, also das Modell auf das ganze Train-Sample anwenden - nicht nur auf ein Analysis-Sample. Und mit den dann resultierenden Modellkoeffizienten sagen wir das TestSample vorher:

```

final_ames_knn_fit <-
  last_fit(ames_final_wflow, data_split)

final_ames_knn_fit

## # Resampling results
## # Manual resampling
## # A tibble: 1 × 6
##   splits           id       .metrics .notes   .predictions .workflow
##   <list>         <chr>     <list>   <list>   <list>      <list>
## 1 <split [2197/733]> train/test split <tibble> <tibble> <tibble>    <workflow>

```

Holen wir uns die Modellgüte:

```
collect_metrics(final_ames_knn_fit)
```

```

## # A tibble: 2 × 4
##   .metric .estimator .estimate .config
##   <chr>   <chr>        <dbl> <chr>
## 1 rmse    standard     0.0951 Preprocessor1_Model
## 2 rsq     standard     0.736  Preprocessor1_Model

```

## 7.11 Aufgaben

- Arbeiten Sie sich so gut als möglich durch diese Analyse zum Verlauf von Covid-Fällen (<https://github.com/sebastiansauer/covid-icu>)
- Fallstudie zur Modellierung einer logistischen Regression mit tidymodels (<https://onezero.blog/modelling-binary-logistic-regression-using-tidymodels-library-in-r-part-1/>)
- Fallstudie zu Vulkanausbrüchen (<https://juliasilge.com/blog/multinomial-volcano-eruptions/>)
- Fallstudie Himalaya (<https://juliasilge.com/blog/himalayan-climbing/>)

## 7.12 Vertiefung

- Fields arranged by purity, xkcd 435 (<https://xkcd.com/435/>)

# 8 Logistische Regression

Benötigte R-Pakete für dieses Kapitel:

```

library(tidyverse)
library(tidymodels)
library(easystats)

```

{datawizard} ist, wie Tidymodels und Tidyverse, ein Metapaket, ein R-Paket also, das mehrere Pakete verwaltet und startet. Hier (<https://easystats.github.io/easystats/>) findet sich mehr Info zu Easystats.

Achtung Easystats ist (noch) nicht auf CRAN. Sie können Easystats so installieren:

```
install.packages("easystats", repos = "https://easystats.r-universe.dev")
```

Einen flotten Spruch bekommen wir von Easystats gratis dazu:

```
easystats_zen()
```

```
## [1] "Patience you must have my young padawan."
```

## 8.1 Lernsteuerung

### 8.1.1 Vorbereitung

- Frischen Sie Ihr Wissen zur logistischen Regression auf bzw. machen Sie sich mit den Grundlagen des Verfahrens vertraut.

### 8.1.2 Lernziele

- Sie verstehen den Zusammenhang von linearen und logistischen Modellen

- Sie können die logistische Regression mit Methoden von tidymodels anwenden

### 8.1.3 Literatur

- Rhys, Kap. 4

## 8.2 Intuitive Erklärung

Die *logistische Regression* ist ein Spezialfall des linearen Modells (lineare Regression), der für *binäre* (dichotom) AV eingesetzt wird (es gibt auch eine Variante für multinominale AV). Es können eine oder mehrere UV in eine logistische Regression einfließen, mit beliebigem Skalenniveau.

Beispiele für Forschungsfragen, die mit der logistischen Regression modelliert werden sind:

- Welche Faktoren sind prädiktiv, um vorherzusagen, ob jemand einen Kredit zurückzahlt kann oder nicht?
- Haben weibliche Passagiere aus der 1. Klasse eine höhere Überlebenschance als andere Personen auf der Titanic?
- Welche Faktoren hängen damit zusammen, ob ein Kunde eine Webseite verlässt, bevor er einen Kauf abschließt?

Der Name stammt von der logistischen Funktion ([https://en.wikipedia.org/wiki/Logistic\\_function](https://en.wikipedia.org/wiki/Logistic_function)), die man in der einfachsten Form so darstellen kann:

$$f(x) = \frac{x}{1 + e^{-x}}$$

Da die AV als dichotom modelliert wird, spricht man von einer *Klassifikation*.

Allerdings ist das Modell reichhaltiger als eine bloße Klassifikation, die (im binären Fall) nur 1 Bit Information liefert: "ja" vs. "nein" bzw. 0 vs. 1.

Das Modell liefert nämlich nicht nur eine Klassifikation zurück, sondern auch eine *Indikation der Stärke* (epistemologisch) der Klassenzugehörigkeit.

Einfach gesagt heißt das, dass die logistische Regression eine Wahrscheinlichkeit der Klassenzugehörigkeit zurückliefert.



Abbildung 8.1: Definition eines Models in tidymodels

## 8.3 Profil

Das Profil des Modells kann man wie folgt charakterisieren, vgl. Tab. 8.1.

Tabelle 8.1: Profil der logistischen Regression

Merkmal	Logistische Regression
Klassifikation	ja
Regression	nein
Lerntyp	überwacht
parametrisch	ja

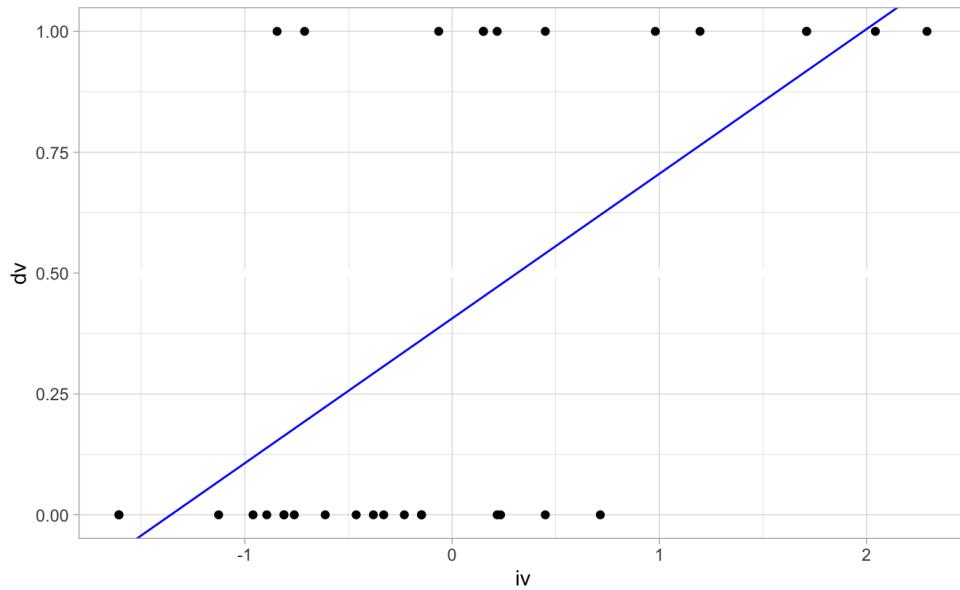
## 8.4 Warum nicht die lineare Regression verwenden?

Forschungsfrage: Kann man anhand des Spritverbrauchs vorhersagen, ob ein Auto eine Automatik- bzw. ein manuelle Schaltung hat? Anders gesagt: Hängt Spritverbrauch und Getriebeart? (Datensatz `mtcars`)

```
data(mtcars)
d <- 
  mtcars %>%
  mutate(mpg_z = standardize(mpg),
        iv = mpg_z,
        dv = am)

m81 <- lm(dv ~ iv, data = d)
coef(m81)
```

```
## (Intercept)           iv
##   0.4062500   0.2993109
```



$Pr(am = 1 | m91, mpg\_z = 0) = 0.46$ : Die Wahrscheinlichkeit einer manuelle Schaltung, gegeben einem durchschnittlichen Verbrauch (und dem Modell `m81`) liegt bei knapp 50%.

### 8.4.1 Lineare Modelle running wild

Wie groß ist die Wahrscheinlichkeit für eine manuelle Schaltung ...

- ... bei `mpg_z = -2` ?

```
predict(m81, newdata = data.frame(iv = -2))
```

```
##           1
## -0.1923719
```

$Pr(\hat{y}) < 0$  macht keinen Sinn. ⚡

- ... bei `mpg_z = +2` ?

```
predict(m81, newdata = data.frame(iv = +2))
```

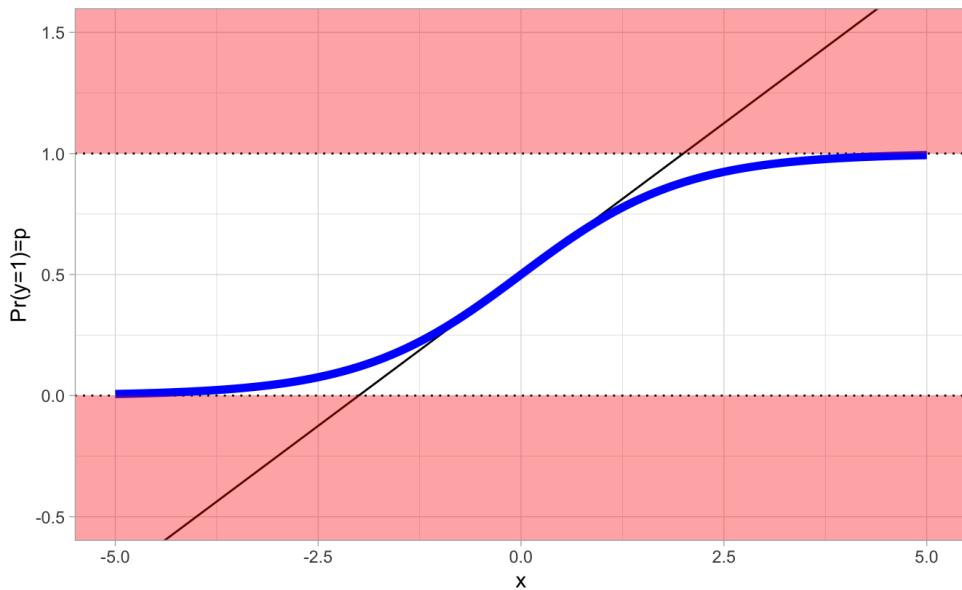
```
##           1
## 1.004872
```

$Pr(\hat{y}) > 1$  macht keinen Sinn. ⚡

Schauen Sie sich mal die Vorhersage an für `mpg_z=5` 🤓

### 8.4.2 Wir müssen die Regressionsgerade umbiegen

... wenn der vorhergesagte Wert eine Wahrscheinlichkeit,  $p_i$ , ist.



Die schwarze Gerade verlässt den Wertebereich der Wahrscheinlichkeit. Die blaue Kurve,  $f$ , bleibt im erlaubten Bereich,  $Pr(y) \in [0, 1]$ . Wir müssen also die linke oder die rechte Seite des linearen Modells transformieren:  $p_i = f(\alpha + \beta \cdot x)$  bzw.:

$$f(p) = \alpha + \beta \cdot x$$

$f$  nennt man eine *Link-Funktion*.

### 8.4.3 Verallgemeinerte lineare Modelle zur Rettung

Für metrische AV mit theoretisch unendlichen Grenzen des Wertebereichs haben wir bisher eine Normalverteilung verwendet:

$$y_i \sim N(\mu_p, \sigma)$$

Dann ist die Normalverteilung eine voraussetzungsarme Wahl (maximiert die Entropie).

Aber wenn die AV *binär* ist bzw. *Häufigkeiten* modelliert, braucht man eine Variable die nur positive Werte zulässt.

Diese Verallgemeinerung des linearen Modells bezeichnet man als *verallgemeinertes lineares Modell* (generalized linear model, GLM).

Im Falle einer binären (bzw. dichotomen) AV liegt eine bestimmte Form des GLM vor, die man als *logistische Regression* bezeichnet.

## 8.5 Der Logit-Link

Der *Logit-Link* wird auch L, logit, Log-Odds oder Logit-Funktion genannt.

Er "biegt" die lineare Funktion in die richtige Form.

Der Logit-Link ordnet einen Parameter, der als Wahrscheinlichkeitsmasse definiert ist (und daher im Bereich von 0 bis 1 liegt), einem linearen Modell zu (das jeden beliebigen reellen Wert annehmen kann):

$$\text{logit}(p_i) = \alpha + \beta x_i$$

- Die Logit-Funktion L ist definiert als der (natürliche) Logarithmus des Verhältnisses der Wahrscheinlichkeit zu Gegenwahrscheinlichkeit:

$$L = \log \frac{p_i}{1 - p_i}$$

- Das Verhältnis der Wahrscheinlichkeit zu Gegenwahrscheinlichkeit nennt man auch *Odds*.
- Also:

$$L = \log \frac{p_i}{1 - p_i} = \alpha + \beta x_i$$

## 8.6 Aber warum?

*Forschungsfrage:* Hängt das Überleben (statistisch) auf der Titanic vom Geschlecht ab?

Wie war eigentlich *insgesamt*, also ohne auf einen (oder mehrere) Prädiktoren zu bedingen, die Überlebenswahrscheinlichkeit?

```
data(titanic_train, package = "titanic")
m82 <- lm(Survived ~ 1, data = titanic_train)
coef(m82)
```

```
## (Intercept)
##  0.3838384
```

Die Wahrscheinlichkeit zu Überleben  $Pr(y = 1)$  lag bei einem guten Drittel (0.38).

Das hätte man auch so ausrechnen:

```
titanic_train %>%
  count(Survived) %>%
  mutate(prop = n/sum(n))
```

```
##   Survived   n     prop
## 1         0 549 0.6161616
## 2         1 342 0.3838384
```

Anders gesagt:  $p(y = 1) = \frac{549}{549+342} \approx 0.38$

## 8.6.1 tidymodels, m83

Berechnen wir jetzt ein lineares Modell für die AV `Survived` mit dem Geschlecht als Pädiktor:

```
d <-
  titanic_train %>%
  filter(Fare > 0) %>%
  mutate(iv = log(Fare),
        dv = factor(Survived))
```

Die Faktorstufen, genannt `levels` von `Survived` sind:

```
levels(d$dv)
```

```
## [1] "0" "1"
```

Und zwar genau in dieser Reihenfolge.

## 8.7 lm83, glm

Die klassischen Methoden in R, ein logistisches Modell zu berechnen, ist mit der Funktion `glm()`. Tidymodels greift intern auf diese Funktion zurück. Daher sind die Ergebnisse numerisch identisch.

```
lm83 <- glm(dv ~ iv, data = d, family = "binomial")
coef(lm83)
```

```
## (Intercept)      iv
## -2.6827432   0.7479317
```

- AV: Überleben (binär/Faktor)
- UV: Ticketpreis

Mit `{easystats}` kann man sich `model_parameters()` einfach ausgeben lassen:

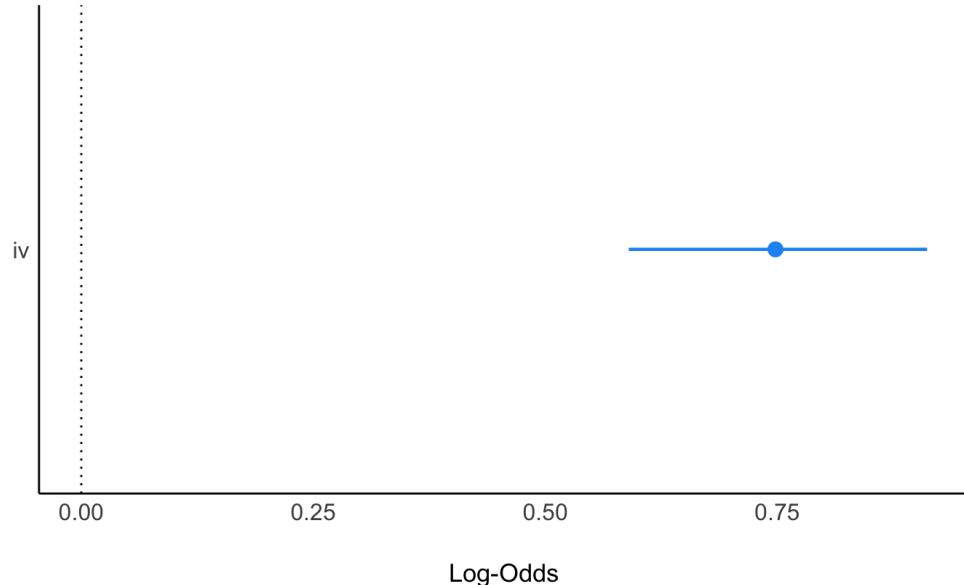
```
library(easystats)
```

```
model_parameters(lm83)
```

## Parameter	Log-Odds	SE	95% CI	z	p
## (Intercept)	-2.68	0.26	[-3.19, -2.19]	-10.46	< .001
## iv	0.75	0.08	[ 0.59,  0.91]	9.13	< .001

Und auch visualisieren lassen:

```
plot(model_parameters(lm83))
```



## 8.8 m83, tidymodels

Achtung! Bei tidymodels muss bei einer Klassifikation die AV vom Type `factor` sein. Außerdem wird bei `tidymodels`, im Gegensatz zu (`g`)`lm` nicht die zweite, sondern die erste als Ereignis modelliert wird.

Daher wechseln wir die referenzkategorie, wir “re-leveln”, mit `relevel()`:

```
d2 <-  
d %>%  
mutate(dv = relevel(dv, ref = "1"))
```

Check:

```
levels(d2$dv)
```

```
## [1] "1" "0"
```

Passt.

Die erste Stufe ist jetzt `1`, also Überleben.

Jetzt berechnen wir das Modell in gewohnter Weise mit `tidymodels`.

```
m83_mod <-  
logistic_reg()  
  
m83_rec <-  
recipe(dv ~ iv, data = d2)  
  
m83_wf <-  
workflow() %>%  
add_model(m83_mod) %>%  
add_recipe(m83_rec)  
  
m83_fit <-  
fit(m83_wf, data = d2)
```

Hier sind die Koeffizienten, die kann man sich aus `m83_fit` herausziehen:

term	estimate	std.error	statistic	p.value
(Intercept)	2.68	0.26	10.46	0.00
iv	-0.75	0.08	-9.13	0.00

```
## [1] 2.6827432 -0.7479317
```

```
## # A tibble: 2 × 5
##   term      estimate std.error statistic p.value
##   <chr>     <dbl>    <dbl>     <dbl>    <dbl>
## 1 (Intercept) 2.68     0.256    10.5  1.26e-25
## 2 iv          -0.748    0.0819   -9.13  6.87e-20
```

Die Koeffizienten werden in Logits angegeben.

In Abb. 8.2 ist das Modell und die Daten visualisiert.

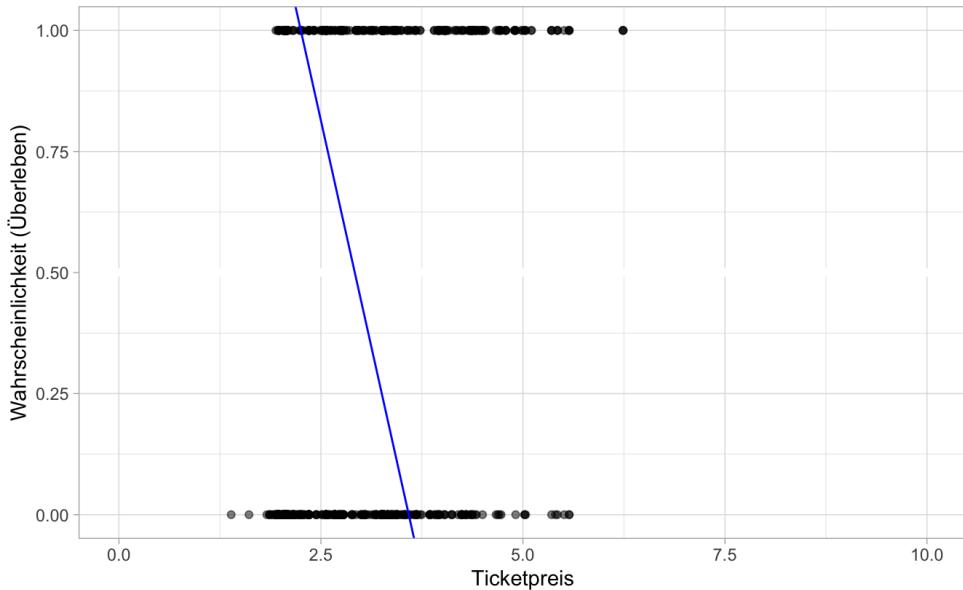


Abbildung 8.2: m83 und die Titanic-Daten

Definieren wir als  $y = 1$  das zu modellierende Ereignis, hier "Überleben auf der Titanic" (hat also überlebt).

Wie wir oben schon gesehen haben, funktioniert die lineare Regression nicht einwandfrei bei binären (oder dichotomen) AV.

## 8.8.1 Wahrscheinlichkeit in Odds

Probieren wir Folgendes: Rechnen wir die Wahrscheinlichkeit zu Überlegen für  $y$ , kurz  $p$ , in *Odds* (Chancen) um.

$$\text{odds} = \frac{p}{1-p}$$

In R:

```
odds <- 0.38 / 0.62
odds
```

```
## [1] 0.6129032
```

Bildlich gesprochen sagen die Odds: für 38 Menschen, die überlebt haben, kommen (ca.) 62 Menschen, die nicht überlebt haben, s. Abb. 8.3.

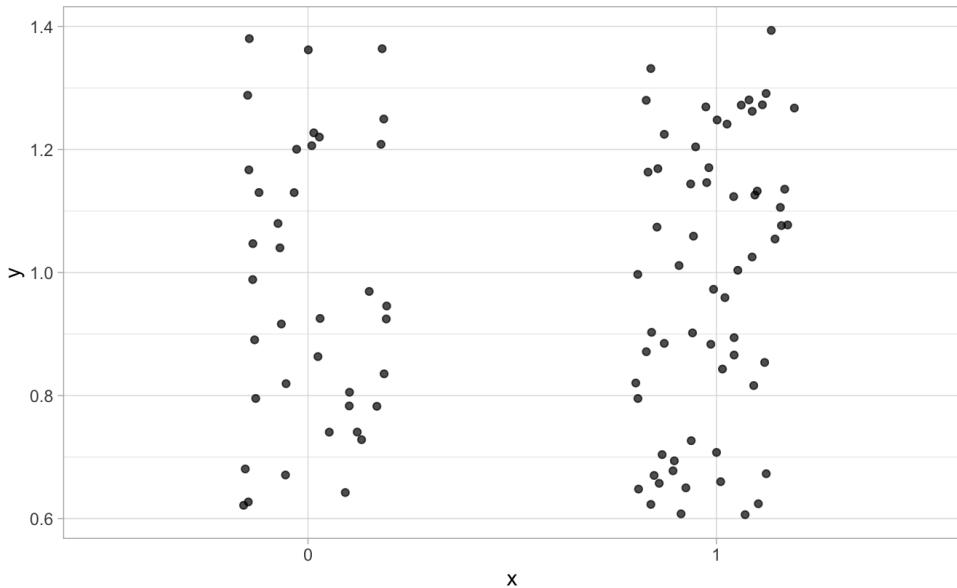


Abbildung 8.3: Odds: 38 zu 62

Plotten wir die Odds als Funktion der UV, s. Abb. 8.4.

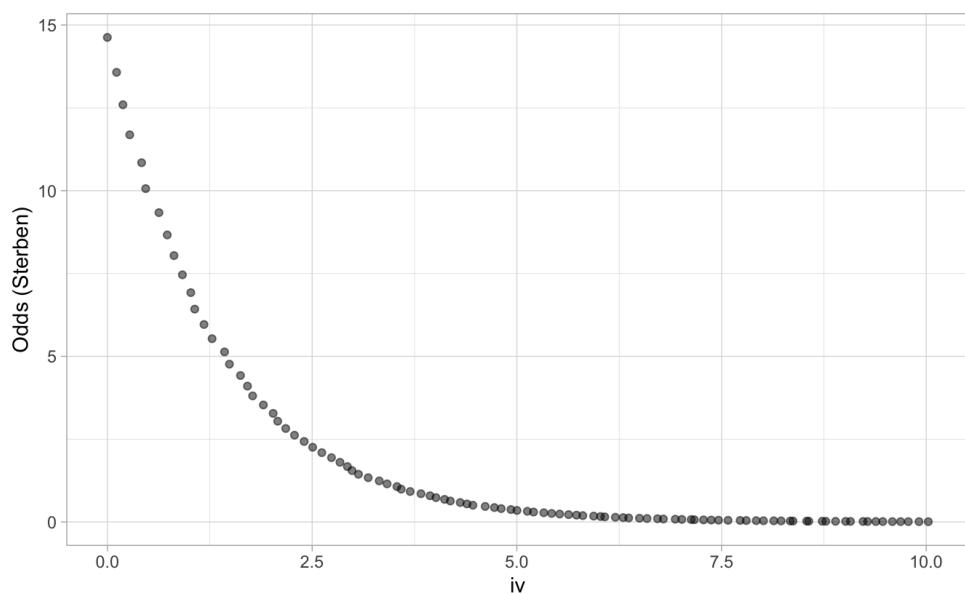


Abbildung 8.4: Odds als Funktion der UV

Wir sind noch nicht am Ziel; die Variable ist noch nicht "richtig gebogen".

## 8.8.2 Von Odds zu Log-Odds

Wenn wir jetzt den Logarithmus berechnen der Log-Odds bekommen wir eine "brav gebogenen" Funktion, die Log-Odds, L, als Funktion der UV, s. Abb. 8.5.

$$L = \log(\text{odds}) = \log\left(\frac{p}{1-p}\right)$$

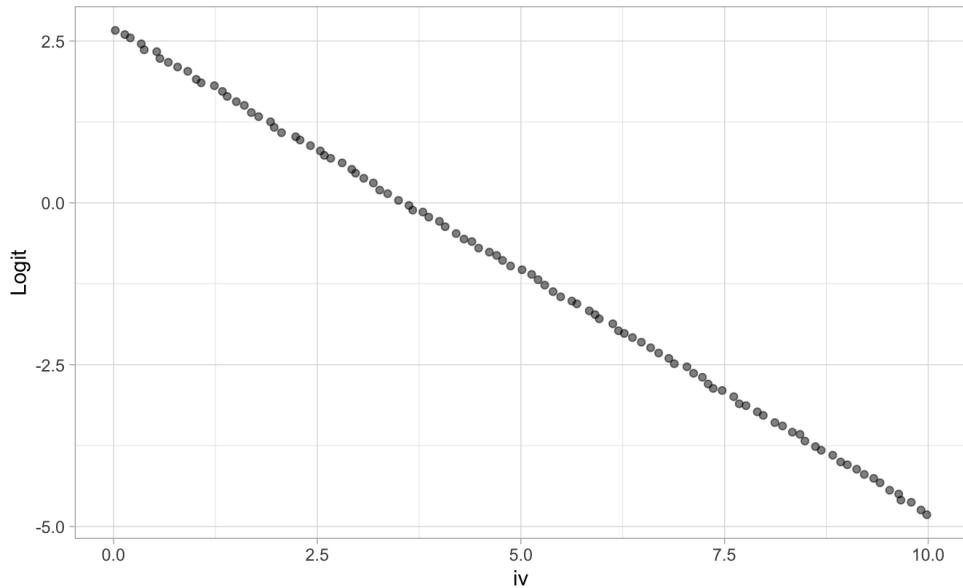


Abbildung 8.5: Logit als Funktion der UV

Linear!

Es gilt also:

$$\text{log-odds} = b_0 + b_1 x$$

Log-Odds (Log-Odds) bezeichnet man auch als *Logits*.

## 8.9 Inverser Logit

Um nach  $p$  aufzulösen, müssen wir einige Algebra bemühen:

$$\begin{aligned} \log \frac{p}{1-p} &= \alpha + \beta x && \text{Exponentieren} \\ \frac{p}{1-p} &= e^{\alpha + \beta x} \\ p_i &= e^{\alpha + \beta x_i} (1-p) && \text{Zur Vereinfachung: } x := e^{\alpha + \beta x_i} \\ p_i &= x(1-p) \\ &= x - xp \\ p + px &= x \\ p(1+x) &= x \\ p &= \frac{x}{1+x} && \text{Lösen wir } x \text{ wieder auf.} \\ p &= \frac{e^{\alpha + \beta x_i}}{1 + e^{\alpha + \beta x_i}} = L^{-1} \end{aligned}$$

Diese Funktion nennt man auch *inverser Logit*,  $\text{logit}^{-1}$ ,  $L^{-1}$ .

Zum Glück macht das alles die Rechenmaschine für uns 😊.

## 8.10 Vom Logit zur Klasse

Praktisch können wir uns die Logits und ihre zugehörige Wahrscheinlichkeit einfach ausgeben lassen mit R. Und die vorhergesagte Klasse (`.pred_class`) auch:

```
d3 <-  
d2 %>%  
bind_cols(predict(m83_fit, new_data = d2, type = "prob")) %>%  
bind_cols(predict(m83_fit, new_data = d2)) %>% # Klasse  
bind_cols(logits = predict(m83_fit, new_data = d2, type = "raw")) # Logits  
  
d3 %>%  
slice_head(n = 3) %>%  
select(-Name, -last_col())
```

```
##                                     Name    logits
## 1 Braund, Mr. Owen Harris 1.2010894
## 2 Cumings, Mrs. John Bradley (Florence Briggs Thayer) -0.5084287
## 3 Heikkinen, Miss. Laina 1.1345079
```

## 8.10.1 Grenzwert wechseln

Im Standard wird 50% als Grenzwert für die vorhergesagte Klasse  $c$  genommen:

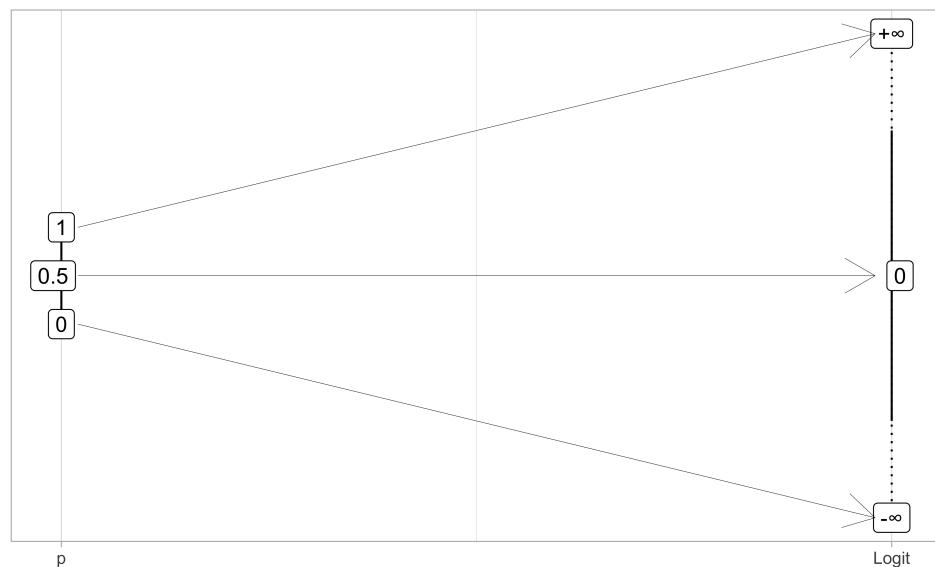
- wenn  $p \leq .5 \rightarrow c = 0$
- wenn  $p > .5 \rightarrow c = 1$

Man kann aber den Grenzwert beliebig wählen, um Kosten-Nutzen-Abwägungen zu optimieren; mehr dazu findet sich z.B. hier (<https://probably.tidymodels.org/articles/where-to-use.html>).

## 8.11 Logit und Inverser Logit

### 8.11.1 Logit

$(0, 1) \rightarrow (-\infty, +\infty)$

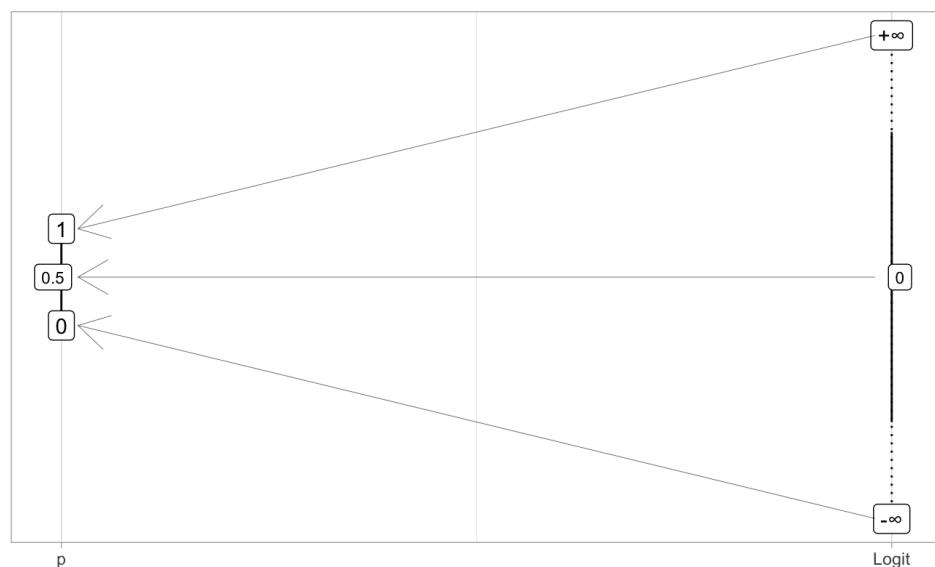


Praktisch, um Wahrscheinlichkeit zu modellieren.

$$p \rightarrow \text{logit} \rightarrow \alpha + \beta x$$

### 8.11.2 Inv-Logit

$(-\infty, +\infty) \rightarrow (0, 1)$



Praktisch, um in Wahrscheinlichkeiten umzurechnen.

$$p \leftarrow \text{inv-logit} \leftarrow \alpha + \beta x$$

## 8.12 Logistische Regression im Überblick

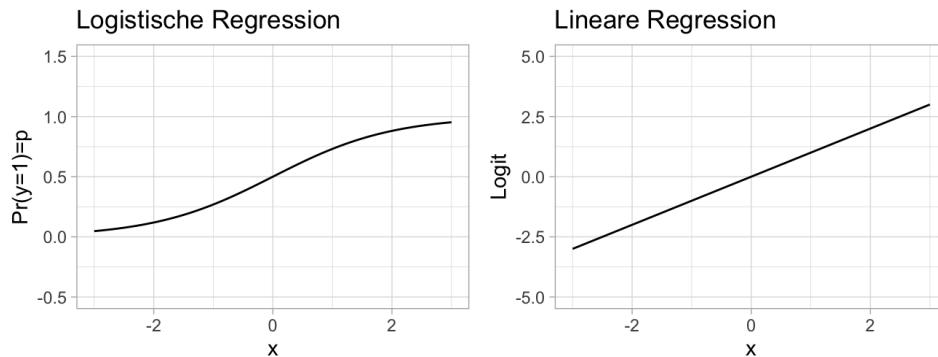
- Eine Regression mit binomial verteilter AV und Logit-Link nennt man *logistische Regression*.
- Man verwendet die logistische Regression um binomial verteilte AV zu modellieren, z.B.
  - Wie hoch ist die Wahrscheinlichkeit, dass ein Kunde das Produkt kauft?
  - Wie hoch ist die Wahrscheinlichkeit, dass ein Mitarbeiter kündigt?
  - Wie hoch ist die Wahrscheinlichkeit, die Klausur zu bestehen?
- Die logistische Regression ist eine normale, lineare Regression für den Logit von  $Pr(y = 1)$ , wobei  $y$  (AV) binomialverteilt mit  $n = 1$  angenommen wird:

$$y_i \sim B(1, p_i)$$

$$\text{logit}(p_i) = \alpha + \beta x_i$$

- Da es sich um eine normale, lineare Regression handelt, sind alle bekannten Methoden und Techniken der linearen Regression zulässig.
- Da Logits nicht einfach zu interpretieren sind, rechnet man nach der Berechnung des Modells den Logit häufig in Wahrscheinlichkeiten um.

### 8.12.1 Die Koeffizienten sind schwer zu interpretieren



- In der logistischen Regression gilt *nicht* mehr, dass eine konstante Veränderung in der UV mit einer konstanten Veränderung in der AV einhergeht.
- Stattdessen geht eine konstante Veränderung in der UV mit einer konstanten Veränderung im *Logit* der AV einher.
- Beim logistischen Modell hier gilt, dass in der Nähe von  $x = 0$  die größte Veränderung in  $p$  von stattfindet; je weiter weg von  $x = 0$ , desto geringer ist die Veränderung in  $p$ .

### 8.12.2 Logits vs. Wahrscheinlichkeiten

```
konvert_logits <-  
  tibble(  
    logit = c( -10, -3,  
              -2, -1, -0.5, -.25,  
              0,  
              .25, .5, 1, 2,  
              3, 10),  
    p = rstanarm:::invlogit(logit)  
  ) %>%  
  gt() %>%  
  fmt_number(everything(), decimals = 2)
```

## 8.13 Aufgaben

- Fallstudien zu Studiengebühren (<https://juliasilge.com/blog/tuition-resampling/>)
- 1. Modell der Fallstudie Hotel Bookings (<https://www.tidymodels.org/start/case-study/>)
- Aufgaben zur logistischen Regression, PDF (<https://github.com/sebastiansauer/datascience1/blob/main/Aufgaben/Thema8-Loesungen1.pdf>)

## 8.14 Vertiefung

- Fallstudie Diabetes mit logistischer Regression (<https://medium.com/the-researchers-guide/modelling-binary-logistic-regression-using-tidymodels-library-in-r-part-1-c1bdce0ac055>)

# 9 Entscheidungsbäume

## 9.1 Vorbereitung

In diesem Kapitel werden folgende R-Pakete benötigt:

```
library(titanic) # Datensatz Titanic
library(rpart) # Berechnung von Entscheidungsbäumen
library(tidymodels)
library(tictoc) # Zeitmessung
```

## 9.2 Lernsteuerung

## 9.3 Anatomie eines Baumes

Ein Baum 🌳 hat (u.a.):

- Wurzel
- Blätter
- Äste

In einem *Entscheidungsbaum* ist die Terminologie ähnlich, s. Abb. 9.1. Allgemein gesagt, kann ein Entscheidungsbaum in einem baumähnlichen Graphen visualisiert werden. Dort gibt es Knoten, die durch Kanten verbunden sind, wobei zu einem Knoten genau ein Kanten führt.

Ein Beispiel für einen einfachen Baum sowie die zugehörige *rekursive Partitionierung* ist in Abb. 9.1 dargestellt; man erkennt  $R = 3$  Regionen bzw. Blätter (James et al. 2021).

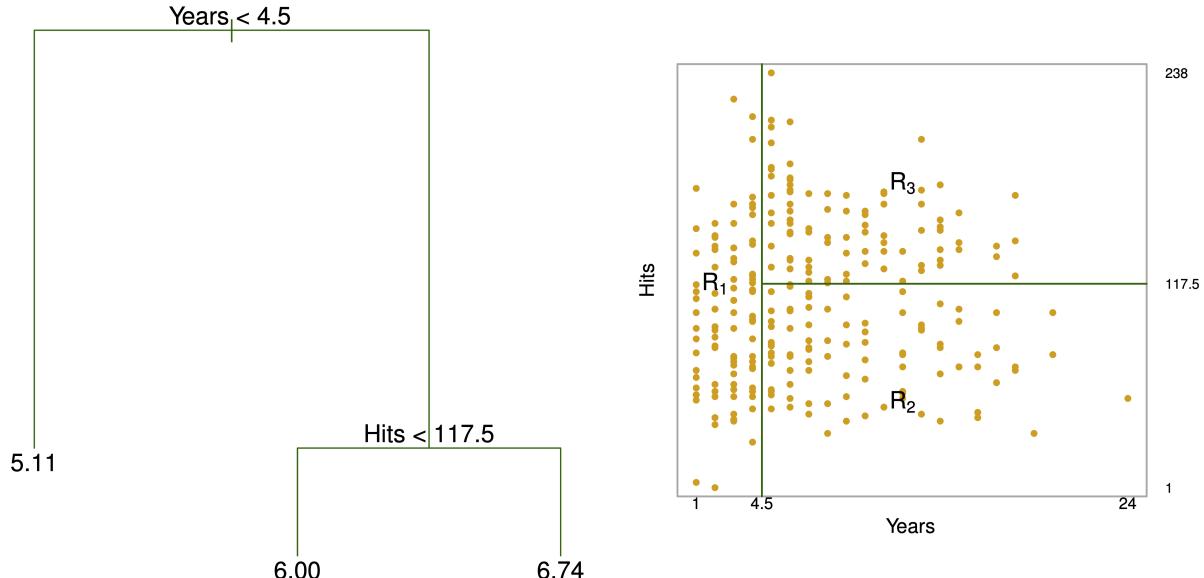


Abbildung 9.1: Einfaches Beispiel für einen Baum sowie der zugehörigen rekursiven Partitionierung

In Abb. 9.1 wird der Knoten an der Spitze auch als *Wurzel(knoten)* bezeichnet. Von diesem Knoten entspringen alle Pfade. Ein Pfad ist die geordnete Menge der Pfade mit ihren Knoten ausgehend von der Wurzel bis zu einem Blatt. Knoten, aus denen kein Kanten mehr wegführt ("Endknoten") werden als *Blätter* bezeichnet. Von einem Knoten gehen zwei Kanten aus (oder gar keine). Knoten, von denen zwei Kanten ausgehen, spiegeln eine *Bedingung* (Prüfung) wider, im Sinne einer Aussage, die mit ja oder nein beantwortet werden kann. Die Anzahl der Knoten eines Pfads entsprechen den *Ebenen* bzw. der Tiefe des Baumes. Von der obersten Ebene (Wurzelknoten) kann man die  $e$  Ebenen aufsteigend durchnummerieren, beginnend bei 1: 1, 2, ...,  $e$ .

## 9.4 Bäume als Regelmaschinen rekursiver Partitionierung

Ein Baum kann man als eine Menge von *Regeln*, im Sinne von *Wenn-dann-sonst-Aussagen*, sehen:

```

Wenn Prädiktor A = 1 ist dann
| Wenn Prädiktor B = 0 ist dann p = 10%
| sonst p = 30%
sonst p = 50%

```

In diesem Fall, zwei Prädiktoren, ist der Prädiktorenraum in *drei Regionen* unterteilt: Der Baum hat drei Blätter.

Für Abb. 9.2 ergibt sich eine komplexere Aufteilung, s. auch Abb. 9.3.

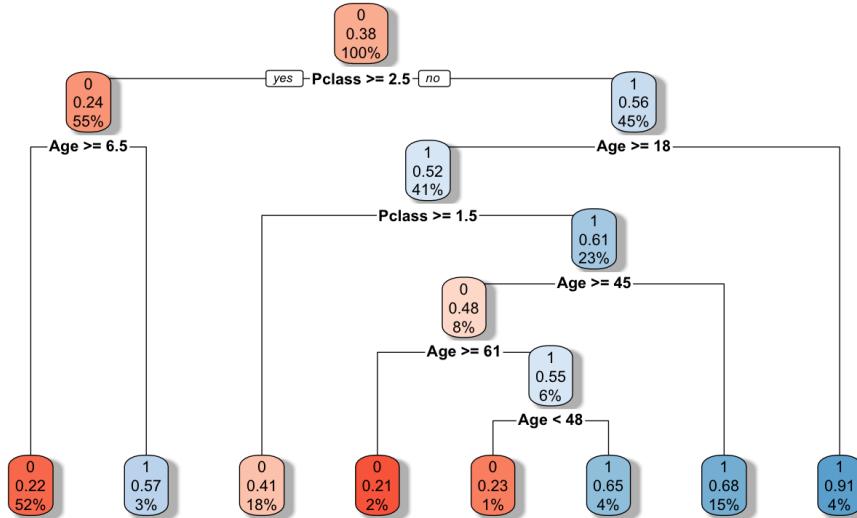


Abbildung 9.2: Beispiel für einen Entscheidungsbaum

Kleine Lesehilft für Abb. 9.2:

- Für jeden Knoten steht in der ersten Zeile der vorhergesagte Wert, z.B. 0 im Wurzelknoten
- darunter steht der Anteil (die Wahrscheinlichkeit) für die in diesem Knoten vorhergesagte Kategorie (0 oder 1)
- darunter (3. Zeile) steht der Anteil der Fälle (am Gesamt-Datensatz) in diesem Knoten, z.B. 100%

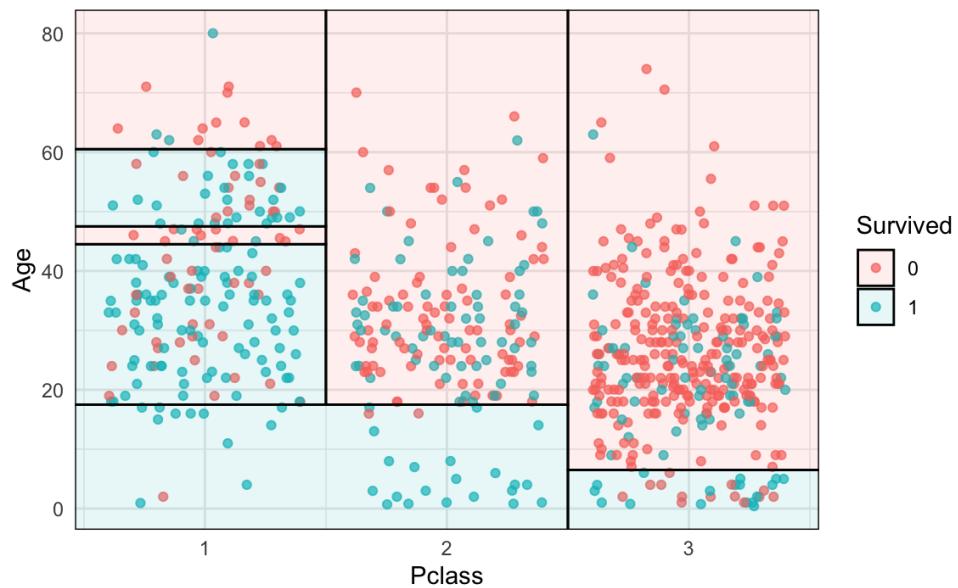


Abbildung 9.3: Partitionierung in Rechtecke durch Entscheidungsbäume

Wie der Algorithmus oben zeigt, wird der Prädiktoraum wiederholt (rekursiv) aufgeteilt, und zwar in Rechtecke, s. Abb. 9.3. Man nennt (eine Implementierung) dieses Algorithmus auch *rpart*.

Das Regelwerk zum Baum aus Abb. 9.2 sieht so aus:

```

## parsnip model object
##
## n= 891
##
## node), split, n, loss, yval, (yprob)
##   * denotes terminal node
##
## 1) root 891 342 0 (0.61616162 0.38383838)
##    2) Pclass>=2.5 491 119 0 (0.75763747 0.24236253)
##      4) Age>=6.5 461 102 0 (0.77874187 0.22125813) *
##      5) Age< 6.5 30 13 1 (0.43333333 0.56666667) *
##    3) Pclass< 2.5 400 177 1 (0.44250000 0.55750000)
##      6) Age>=17.5 365 174 1 (0.47671233 0.52328767)
##        12) Pclass>=1.5 161 66 0 (0.59006211 0.40993789) *
##        13) Pclass< 1.5 204 79 1 (0.38725490 0.61274510)
##          26) Age>=44.5 67 32 0 (0.52238806 0.47761194)
##            52) Age>=60.5 14 3 0 (0.78571429 0.21428571) *
##            53) Age< 60.5 53 24 1 (0.45283019 0.54716981)
##          106) Age< 47.5 13 3 0 (0.76923077 0.23076923) *
##          107) Age>=47.5 40 14 1 (0.35000000 0.65000000) *
##        27) Age< 44.5 137 44 1 (0.32116788 0.67883212) *
##    7) Age< 17.5 35 3 1 (0.08571429 0.91428571) *

```

Kleine Lesehilfe: Ander Wurzel root des Baumes, Knoten 1) haben wir 891 Fälle, von denen 342 nicht unserer Vorhersage yval entsprechen, also loss sind, das ist ein Anteil, (yprob) von 0.38. Unsere Vorhersage ist 0, da das die Mehrheit in diesem Knoten ist, dieser Anteil beträgt ca. 61%. In der Klammer stehen also die Wahrscheinlichkeiten für alle Ausprägungen von Y:, 0 und 1, in diesem Fall. Entsprechendes gilt für jeden weiteren Knoten.

Ein kurzer Check der Häufigkeit am Wurzelknoten:

```
count(titanic_train, Survived)
```

```

##   Survived   n
## 1       0 549
## 2       1 342

```

Solche Entscheidungsbäume zu erstellen, ist nichts neues. Man kann sie mit einer einfachen Checkliste oder Entscheidungssystem vergleichen. Der Unterschied zu Entscheidungsbäumen im maschinellen Lernen ist nur, dass die Regeln aus den Daten gelernt werden, man muss sie nicht vorab kennen.

Noch ein Beispiel ist in Abb. 9.4 gezeigt (James et al. 2021): Oben links zeigt eine *unmögliche* Partitionierung (für einen Entscheidungsbaum). Oben rechts zeigt die Regionen, die sich durch den Entscheidungsbaum unten links ergeben. Untenrechts ist der Baum in 3D dargestellt.

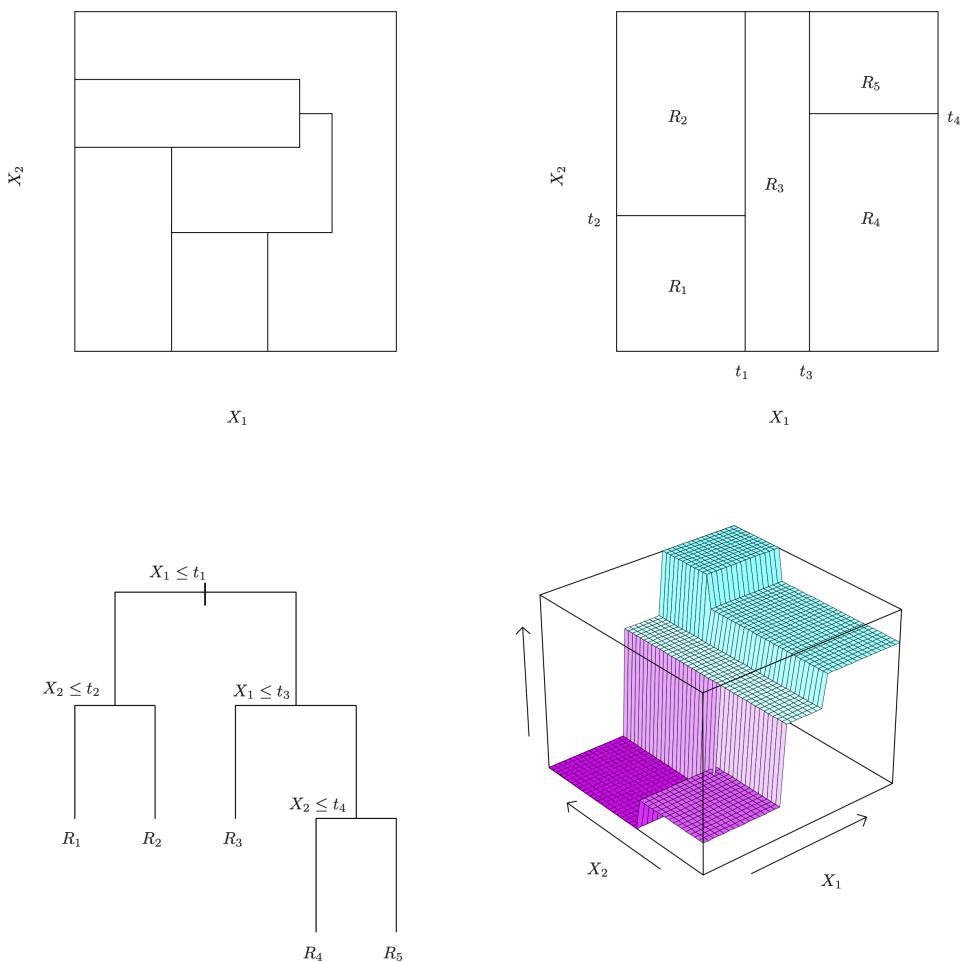


Abbildung 9.4: Ein weiteres Beispiel zur Darstellung von Entscheidungsbäumen

## 9.5 Klassifikation

Bäume können für Zwecke der Klassifikation (nominal skalierte AV) oder Regression (numerische AV) verwendet werden. Betrachten wir zunächst die binäre Klassifikation, also für eine zweistufige (nominalskalierte) AV. Das Ziel des Entscheidungsmodell-Algorithmus ist es, zu Blättern zu kommen, die möglichst "sortenrein" sind, sich also möglichst klar für eine (der beiden) Klassen  $A$  oder  $B$  aussprechen. Nach dem Motto: "Wenn Prädiktor 1 kleiner  $x$  und wenn Prädiktor 2 gleich  $y$ , dann handelt es sich beim vorliegenden Fall ziemlich sicher um Klasse  $A$ ."

Je homogener die Verteilung der AV pro Blatt, desto genauer die Vorhersagen.

Unsere Vorhersage in einem Blatt entspricht der Merheit bzw. der häufigsten Kategorie in diesem Blatt.

## 9.6 Gini als Optimierungskriterium

Es gibt mehrere Kennzahlen, die zur Optimierung bzw. zur Entscheidung zum Aufbau des Entscheidungsbaums herangezogen werden. Zwei übliche sind der *Gini-Koeffizient* und die *Entropie*. Bei Kennzahlen sind Maß für die Homogenität oder "Sortenreinheit" (vs. Heterogenität, engl. auch *impurity*).

Den Algorithmus zur Erzeugung des Baumes kann man so darstellen:

```

Wiederhole für jede Ebene
| prüfe für alle Prädiktoren alle möglichen Bedingungen
| wähle denjenigen Prädiktor mit derjenigen Bedingung, der die Homogenität maximiert
solange bis Abbruchkriterium erreicht ist.
  
```

Ein Bedingung könnte sein `Age >= 18` oder `years < 4.5`.

Es kommen mehrere Abbruchkriterium in Frage:

- Eine Mindestanzahl von Beobachtungen pro Knoten wird unterschritten (`minsplit`)
- Die maximale Anzahl an Ebenen ist erreicht (`maxdepth`)
- Die minimale Zahl an Beobachtungen eines Blatts wird unterschritten (`minbucket`)

Der Gini-Koeffizient ist im Fall einer UV mit zwei Stufen,  $c_A$  und  $c_B$ , so definiert:

$$G = 1 - \left( p(c_A)^2 + (1 - p(c_A))^2 \right)$$

Der Algorithmus ist "gierig" (greedy): Optimiert werden lokal optimale Aufteilungen, auch wenn das bei späteren Aufteilungen im Baum dann insgesamt zu geringerer Homogenität führt.

Die Entropie ist definiert als

$$D = - \sum_{k=1}^K p_k \cdot \log(p_k),$$

wobei  $K$  die Anzahl der Kategorien indiziert.

Gini-Koeffizient und Entropie kommen oft zu ähnlichen numerischen Ergebnissen, so dass wir uns im Folgenden auf den Gini-Koeffizienten konzentrieren werden.

### Beispiel

Vergleichen wir drei Bedingungen mit jeweils  $n = 20$  Fällen, die zu unterschiedlich homogenen Knoten führen:

- 10/10
- 15/5
- 19/1

Was ist jeweils der Wert des Gini-Koeffizienten?

```
G1 <- 1 - ((10/20)^2 + (10/20)^2)
G1
```

```
## [1] 0.5
```

```
G2 <- 1 - ((15/20)^2 + (5/20)^2)
G2
```

```
## [1] 0.375
```

```
G3 <- 1 - ((19/20)^2 + (1/20)^2)
G3
```

```
## [1] 0.095
```

Wie man sieht, sinkt der Wert des Gini-Koeffizienten ("G-Wert"), je homogener die Verteilung ist. *Maximal heterogen* ("gemischt") ist die Verteilung, wenn alle Werte gleich oft vorkommen, in diesem Fall also 50%/50%.

Neben dem G-Wert für einzelne Knoten kann man den G-Wert für eine Aufteilung ("Split") berechnen, also die Frage beantworten, ob die Aufteilung eines Knoten in zwei zu mehr Homogenität führt. Der G-Wert einer Aufteilung ist die gewichtete Summe der G-Werte der beiden Knoten (links,  $l$  und rechts,  $r$ ):

$$G_{split} = p(l)G_l + p(r)G_r$$

Der *Gewinn* (gain) an Homogenität ist dann die Differenz des G-Werts der kleineren Ebene und der Aufteilung:

$$G_{gain} = G - G_{split}$$

Der Algorithmus kann auch bei UV mit mehr als zwei, also  $K$  Stufen,  $c_1, c_2, \dots, c_K$  verwendet werden:

$$G = 1 - \sum_{k=1}^K p(c_k)^2$$

## 9.7 Metrische Prädiktoren

Außerdem ist es möglich, Bedingung bei *metrischen* UV auf ihre Homogenität hin zu bewerten, also Aufteilungen der Art `years < 4.5` zu tätigen. Dazu muss man einen Wert identifizieren, bei dem man auftrennt.

Das geht in etwa so:

Sortiere die Werte eines Prädiktors (aufsteigend)  
 Für jedes Paar an aufeinanderfolgenden Werten berechne den G-Wert  
 Finde das Paar mit dem höchsten G-Wert aus allen Paaren  
 Nimm den Mittelwert der beiden Werte dieses Paares: Das ist der Aufteilungswert

Abbildung 9.5 stellt dieses Vorgehen schematisch dar (Rhys 2020).

```
knitr::include_graphics("img/fig7-5_alt.jpeg")
```

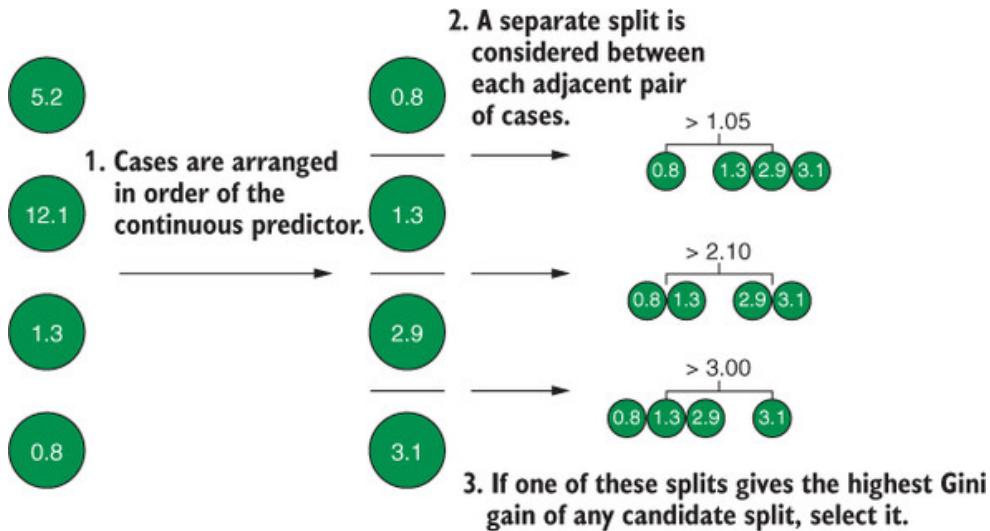


Abbildung 9.5: Aufteilungswert bei metrischen Prädiktoren

## 9.8 Regressionsbäume

Bei Regressionsbäumen wird nicht ein Homogenitätsmaß wie der Gini-Koeffizient als Optimierungskriterium herangezogen, sondern die *RSS* (Residual Sum of Squares) bietet sich an.

Die  $J$  Regionen (Partitionierungen) des Prädiktorraums  $R_1, R_2, \dots, R_J$  müssen so gewählt werden, dass RSS minimal ist:

$$RSS = \sum_{j=1}^J \sum_{i \in R_j} (u_i - \hat{y}_{R_j})^2,$$

wobei  $\hat{y}$  der (vom Baum) vorhergesagte Wert ist für die  $j$ -te Region.

## 9.9 Baum beschneiden

Ein Problem mit Entscheidungsbäumen ist, dass ein zu komplexer Baum, „zu verästelt“ sozusagen, in hohem Maße Overfitting ausgesetzt ist: Bei höheren Ebenen im Baum ist die Anzahl der Beobachtungen zwangsläufig klein, was bedeutet, dass viel Rauschen gefittet wird.

Um das Overfitting zu vermeiden, gibt es zwei auf der Hand liegende Maßnahmen:

1. Den Baum nicht so groß werden lassen
2. Den Baum „zurückschneiden“

Die 1. Maßnahme beruht auf dem Festlegen einer maximalen Zahl an Ebenen (`maxdepth`) oder einer minimalen Zahl an Fällen pro Knoten (`minsplit`) oder im Blatt (`minbucket`).

Die 2. Maßnahme, das Zurückschneiden (pruning) des Baumes hat als Idee, einen „Teilbaum“  $T$  zu finden, der so klein wie möglich ist, aber so gut wie möglich präzise Vorhersagen erlaubt. Dazu belegen wir die RSS eines Teilbaums (subtree) mit einem Strafterm  $s = \alpha |T|$ , wobei  $|T|$  die Anzahl der Blätter des Baums entspricht.  $\alpha$  ist ein Tuningparameter, also ein Wert, der nicht vom Modell berechnet wird, sondern von uns gesetzt werden muss - zumeist durch schlichtes Ausprobieren.  $\alpha$  wägt ab zwischen Komplexität und Fit (geringe RSS). Wenn  $\alpha = 0$  haben wir eine normalen, unbeschnittenen Baum  $T_0$ . Je größer  $\alpha$  wird, desto höher wird der „Preis“ für viele Blätter, also für Komplexität und der Baum wird kleiner. Dieses Vorgehen nennt man auch *cost complexity pruning*. Daher nennt man den zugehörigen Tuningparameter auch *Cost Complexity*  $C_p$ .

## 9.10 Das Rechteck schlägt zurück

Entscheidungsbäume zeichnen sich durch rechtecke (rekursive) Partitionierungen des Prädiktorenraums aus. Lineare Modelle durch eine einfache lineare Partitionierung (wenn man Klassifizieren möchte), Abb. 9.6 verdeutlicht diesen Unterschied (James et al. 2021).

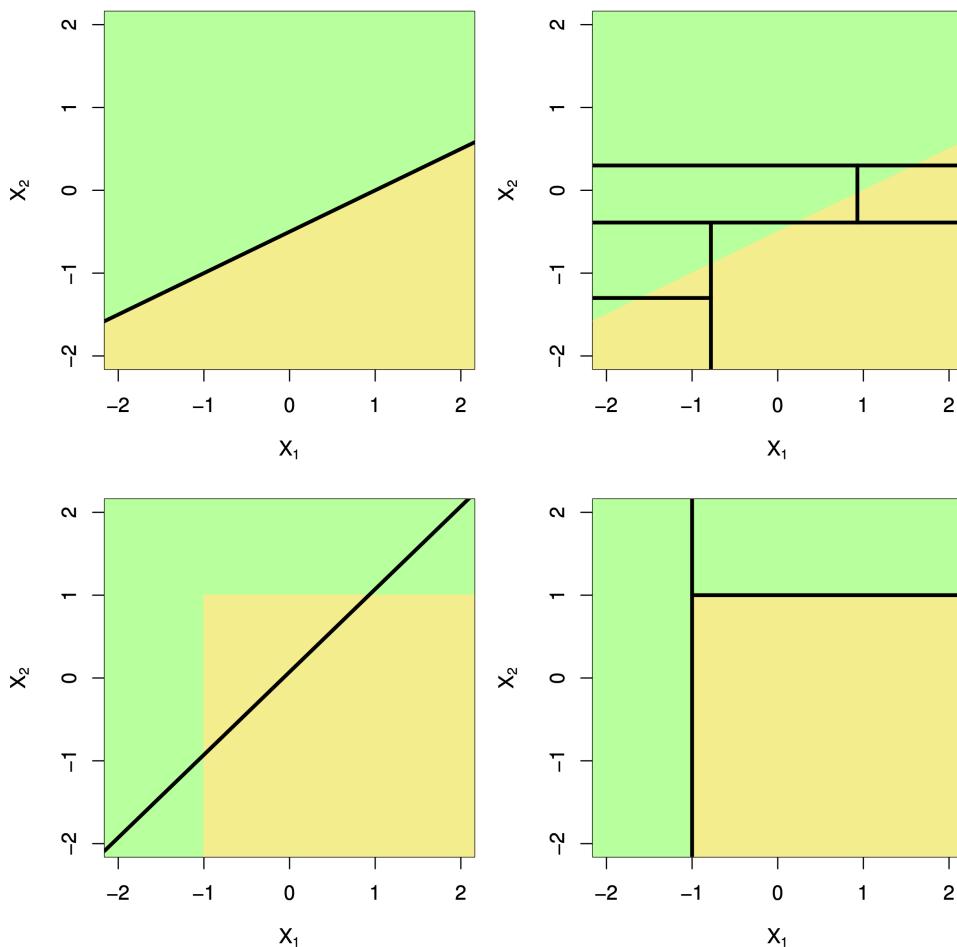


Abbildung 9.6: Rechteckige vs. lineare Partitionierung

Jetzt kann sich fragen: Welches Vorgehen ist besser - das rechteckige oder das lineare Partitionierungen. Da gibt es eine klare Antwort: Es kommt drauf an. Wie Abb. 9.6 gibt es Datenlagen, in denen das eine Vorgehen zu homogenerer Klassifikation führt und Situationen, in denen das andere Vorgehen besser ist, vgl. Abb. 9.7.



Abbildung 9.7: Free Lunch?

## 9.11 Tidymodels

Probieren wir den Algorithmus Entscheidungsbäume an einem einfachen Beispiel in R mit Tidymodels aus.

Die Aufgabe sei, Spritverbrauch (möglichst exakt) vorherzusagen.

Ein ähnliches Beispiel, mit analogem Vorgehen, findet sich in dieser Fallstudie (<https://juliasilge.com/blog/wind-turbine/>).

### 9.11.1 Initiale Datenaufteilung

```
library(tidymodels)

data("mtcars")

set.seed(42) # Reproduzierbarkeit
d_split <- initial_split(mtcars, strata = mpg)

## Warning: The number of observations in each quantile is below the recommended threshold of 20.
## • Stratification will use 1 breaks instead.

## Warning: Too little data to stratify.
## • Resampling will be unstratified.

d_train <- training(d_split)
d_test <- testing(d_split)
```

Die Warnung zeigt uns, dass der Datensatz sehr klein ist; stimmt. Ignorieren wir hier einfach.

Wie man auf der Hilfeseite der Funktion ([https://rsample.tidymodels.org/reference/initial\\_split.html](https://rsample.tidymodels.org/reference/initial_split.html)) sieht, wird per Voreinstellung 3/1 aufgeteilt, also 75% in das Train-Sample, 25% der Daten ins Test-Sample.

Bei  $n = 32$  finden also 8 Autos ihren Weg ins Test-Sample und die übrigen 24 ins Train-Sample. Bei der kleinen Zahl könnte man sich (berechtigterweise) fragen, ob es Sinn macht, die spärlichen Daten noch mit einem Test-Sample weiter zu dezimieren. Der Einwand ist nicht unberechtigt, allerdings zieht der Verzicht auf ein Test-Sample andere Probleme, Overfitting namentlich, nach sich.

## 9.11.2 Kreuzvalidierung definieren

```
d_cv <- vfold_cv(d_train, strata = mpg, repeats = 5, v = 5)
d_cv

## # 5-fold cross-validation repeated 5 times using stratification
## # A tibble: 25 × 3
##   splits      id     id2
##   <list>      <chr>  <chr>
## 1 1 <split [19/5]> Repeat1 Fold1
## 2 2 <split [19/5]> Repeat1 Fold2
## 3 3 <split [19/5]> Repeat1 Fold3
## 4 4 <split [19/5]> Repeat1 Fold4
## 5 5 <split [20/4]> Repeat1 Fold5
## 6 6 <split [19/5]> Repeat2 Fold1
## 7 7 <split [19/5]> Repeat2 Fold2
## 8 8 <split [19/5]> Repeat2 Fold3
## 9 9 <split [19/5]> Repeat2 Fold4
## 10 10 <split [20/4]> Repeat2 Fold5
## # ... with 15 more rows
```

Die Defaults (Voreinstellungen) der Funktion `vfold_cv()` können, wie immer, auf der Hilfeseite der Funktion ([https://rsample.tidymodels.org/reference/vfold\\_cv.html](https://rsample.tidymodels.org/reference/vfold_cv.html)) nachgelesen werden.

Da die Stichprobe sehr klein ist, bietet es sich an, eine kleine Zahl an Faltungen (`folds`) zu wählen. Bei 10 Faltungen beinhaltete eine Stichprobe gerade 10% der Fälle in Train-Sample, also etwa ... 2!

Zur Erinnerung: Je größer die Anzahl der Repeats, desto genauer schätzen wir die Modellgüte.

## 9.11.3 Rezept definieren

Hier ein einfaches Rezept:

```
recipe1 <-
  recipe(mpg ~ ., data = d_train) %>%
  step_impute_knn() %>%
  step_normalize() %>%
  step_dummy() %>%
  step_other(threshold = .1)
```

## 9.11.4 Modell definieren

```
tree_model <-
  decision_tree(
    cost_complexity = tune(),
    tree_depth = tune(),
    min_n = tune()
  ) %>%
  set_engine("rpart") %>%
  set_mode("regression")
```

Wenn Sie sich fragen, woher Sie die Optionen für die Tuningparameter wissen sollen: Schauen Sie mal in die Hilfeseite des Pakets {{dials}} (<https://dials.tidymodels.org/articles/Basics.html>); das Paket ist Teil von Tidymodels.

Die Berechnung des Modells läuft über das Paket `{}{rpart}`, was wir durch `set_engine()` festgelegt haben.

Der Parameter *Cost Complexity*,  $C_p$  oder manchmal auch mit  $\alpha$  bezeichnet, hat einen typischen Wertebereich von  $10^{-10}$  bis  $10^{-1}$ :

```
cost_complexity()
```

```
## Cost-Complexity Parameter (quantitative)
## Transformer: log-10 [1e-100, Inf]
## Range (transformed scale): [-10, -1]
```

Hier ist der Wert in Log-Einheiten angegeben. Wenn Sie sich fragen, woher Sie das bitteschön wissen sollen: Naja, es steht auf der Hilfeseite (<https://dials.tidymodels.org/articles/Basics.html>) 😊 .

Unser Modell ist also so definiert:

```
tree_model

## Decision Tree Model Specification (regression)
##
## Main Arguments:
##   cost_complexity = tune()
##   tree_depth = tune()
##   min_n = tune()
##
## Computational engine: rpart
```

Mit `tune()` weist man den betreffenden Parameter als "zu tunen" aus - gute Werte sollen durch Ausprobieren während des Berechnens bestimmt werden. Genauer gesagt soll das Modell für jeden Wert (oder jede Kombination an Werten von Tuningparametern) berechnet werden.

Eine Kombination an Tuningparameter-Werten, die ein Modell spezifizieren, sozusagen erst "fertig definieren", nennen wir einen *Modellkandidaten*.

Definieren wir also eine Tabelle (`grid`) mit Werten, die ausprobiert, "getuned" werden sollen. Wir haben oben drei Tuningparameter bestimmt. Sagen wir, wir hätten gerne jeweils 5 Werte pro Parameter.

```
tree_grid <-
  grid_regular(
    cost_complexity(),
    tree_depth(),
    min_n(),
    levels = 4
  )
```

Für jeden Parameter sind Wertebereiche definiert; dieser Wertebereich wird gleichmäßig (daher `grid regular`) aufgeteilt; die Anzahl der verschiedenen Werte pro Parameter wird durch `levels` gegeben.

Mehr dazu findet sich auf der Hilfeseite ([https://dials.tidymodels.org/reference/grid\\_regular.html](https://dials.tidymodels.org/reference/grid_regular.html)) zu `grid_regular()` .

Wenn man die alle miteinander durchprobiert, entstehen  $4^3$  Kombinationen, also Modellkandidaten.

Allgemeiner gesagt sind das bei  $n$  Tuningparametern mit jeweils  $m$  verschiedenen Werten  $m^n$  Möglichkeiten, spricht Modellkandidaten. Um diesen Faktor erhöht sich die Rechenzeit im Vergleich zu einem Modell ohne Tuning. Man sieht gleich, dass die Rechenzeit schnell unangenehm lang werden kann.

Entsprechend hat unsere Tabelle diese Zahl an Zeilen. Jede Zeile definiert einen Modellkandidaten, also eine Berechnung des Modells.

```
dim(tree_grid)

## [1] 64 3

head(tree_grid)

## # A tibble: 6 × 3
##   cost_complexity tree_depth min_n
##   <dbl>          <int> <int>
## 1 0.0000000001      1      2
## 2 0.0000001        1      2
## 3 0.0001           1      2
## 4 0.1              1      2
## 5 0.0000000001      5      2
## 6 0.0000001        5      2
```

Man beachte, dass außer *Definitionen* bisher nichts passiert ist – vor allem haben wir noch nichts berechnet. Sie scharren mit den Hufen? Wollen endlich loslegen? Also gut.

## 9.11.5 Workflow definieren

Fast vergessen: Wir brauchen noch einen Workflow.

```
tree_wf <-
  workflow() %>%
  add_model(tree_model) %>%
  add_recipe(recipe1)
```

## 9.11.6 Modell tunen und berechnen

Achtung: Das Modell zu berechnen kann etwas dauern. Es kann daher Sinn machen, das Modell abzuspeichern, so dass Sie beim erneuten Durchlaufen nicht nochmal berechnen müssen, sondern einfach von der Festplatte laden können; das setzt natürlich voraus, dass sich am Modell nichts geändert hat.

```
doParallel::registerDoParallel() # mehrere Kerne parallel nutzen

set.seed(42)
tic() # Stoppuhr an
trees_tuned <-
  tune_grid(
    object = tree_wf,
    grid = tree_grid,
    resamples = d_cv
  )
toc() # Stoppuhr aus
```

Es bietet sich in dem Fall an, das Ergebnis-Objekt als *R Data serialized* (rds) abzuspeichern:

```
write_rds(trees_tuned, "objects/trees1.rds")
```

Bzw. so wieder aus der RDS-Datei zu importieren:

```
trees_tuned <- read_rds("objects/trees1.rds")
```

Hier (<https://stackoverflow.com/questions/21370132/what-are-the-main-differences-between-r-data-files>) oder hier (<https://en.wikipedia.org/wiki/Serialization>) kann man einiges zum Unterschied einer RDS-Datei vs. einer "normalen" R-Data-Datei nachlesen. Wenn man möchte 😊.

```
trees_tuned
```

```
## # Tuning results
## # 5-fold cross-validation repeated 5 times using stratification
## # A tibble: 25 × 5
##   splits      id     id2 .metrics      .notes
##   <list>     <chr> <chr> <list>      <list>
## 1 <split [19/5]> Repeat1 Fold1 <tibble [128 × 7]> <tibble [32 × 3]>
## 2 <split [19/5]> Repeat1 Fold2 <tibble [128 × 7]> <tibble [32 × 3]>
## 3 <split [19/5]> Repeat1 Fold3 <tibble [128 × 7]> <tibble [32 × 3]>
## 4 <split [19/5]> Repeat1 Fold4 <tibble [128 × 7]> <tibble [32 × 3]>
## 5 <split [20/4]> Repeat1 Fold5 <tibble [128 × 7]> <tibble [32 × 3]>
## 6 <split [19/5]> Repeat2 Fold1 <tibble [128 × 7]> <tibble [32 × 3]>
## 7 <split [19/5]> Repeat2 Fold2 <tibble [128 × 7]> <tibble [32 × 3]>
## 8 <split [19/5]> Repeat2 Fold3 <tibble [128 × 7]> <tibble [32 × 3]>
## 9 <split [19/5]> Repeat2 Fold4 <tibble [128 × 7]> <tibble [32 × 3]>
## 10 <split [20/4]> Repeat2 Fold5 <tibble [128 × 7]> <tibble [32 × 3]>
## # ... with 15 more rows
##
## There were issues with some computations:
##
## - Warning(s) x320: 27 samples were requested but there were 19 rows in the data. 19 ... - Warning(s) x80:
## 27 samples were requested but there were 19 rows in the data. 19 ... - Warning(s) x320: 27 samples were requested but there were 19 rows in the data. 19 ... - Warning(s) x80: 27 samples were requested but there were 19 rows in the data. 19 ... - Warning(s) x4: 27 samples were requested but there were 19 rows in the data. 19 ...
##
## Use `collect_notes(object)` for more information.
```

Die Warnhinweise kann man sich so ausgeben lassen:

```
collect_notes(trees_tuned)
```

```
## # A tibble: 804 × 5
##   id     id2    location      type     note
##   <chr>  <chr>  <chr>       <chr>    <chr>
## 1 Repeat1 Fold1 preprocess 1/1, model 33/64 warning 27 samples were requeste...
## 2 Repeat1 Fold1 preprocess 1/1, model 34/64 warning 27 samples were requeste...
## 3 Repeat1 Fold1 preprocess 1/1, model 35/64 warning 27 samples were requeste...
## 4 Repeat1 Fold1 preprocess 1/1, model 36/64 warning 27 samples were requeste...
## 5 Repeat1 Fold1 preprocess 1/1, model 37/64 warning 27 samples were requeste...
## 6 Repeat1 Fold1 preprocess 1/1, model 38/64 warning 27 samples were requeste...
## 7 Repeat1 Fold1 preprocess 1/1, model 39/64 warning 27 samples were requeste...
## 8 Repeat1 Fold1 preprocess 1/1, model 40/64 warning 27 samples were requeste...
## 9 Repeat1 Fold1 preprocess 1/1, model 41/64 warning 27 samples were requeste...
## 10 Repeat1 Fold1 preprocess 1/1, model 42/64 warning 27 samples were requeste...
## # ... with 794 more rows
```

Wie gesagt, in diesem Fall war die Stichprobengröße sehr klein.

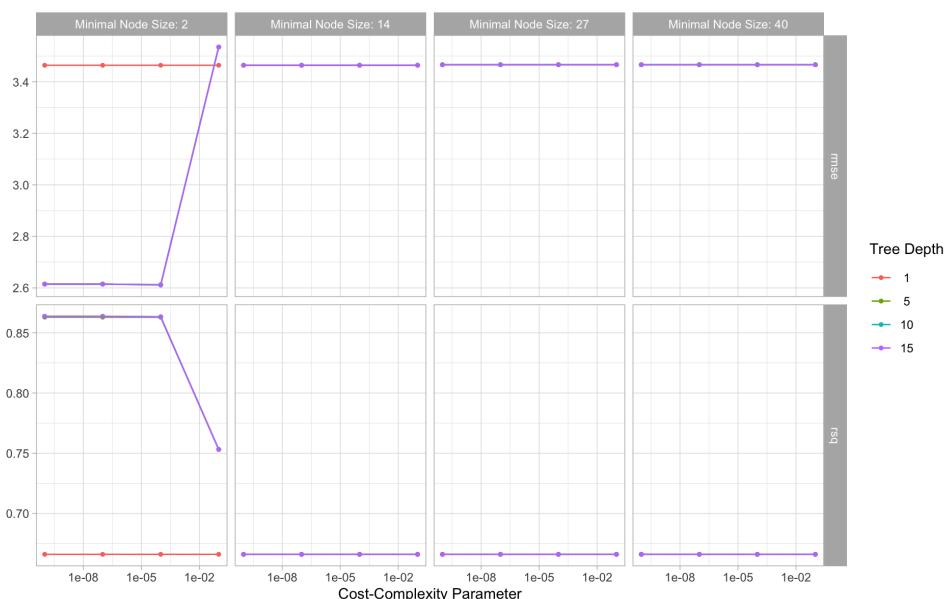
## 9.11.7 Modellgüte evaluieren

```
collect_metrics(trees_tuned)
```

```
## # A tibble: 128 × 9
##   cost_complexity tree_depth min_n .metric .estimator  mean     n std_err
##   <dbl>          <int>     <int> <chr>    <chr>     <dbl> <int>   <dbl>
## 1 0.0000000001     1        2 rmse    standard  3.46    25  0.223
## 2 0.0000000001     1        2 rsq     standard  0.666   21  0.0385
## 3 0.0000001        1        2 rmse    standard  3.46    25  0.223
## 4 0.0000001        1        2 rsq     standard  0.666   21  0.0385
## 5 0.0001           1        2 rmse    standard  3.46    25  0.223
## 6 0.0001           1        2 rsq     standard  0.666   21  0.0385
## 7 0.1              1        2 rmse    standard  3.46    25  0.223
## 8 0.1              1        2 rsq     standard  0.666   21  0.0385
## 9 0.0000000001     5        2 rmse    standard  2.62    25  0.265
## 10 0.0000000001    5        2 rsq     standard  0.863   25  0.0279
## # ... with 118 more rows, and 1 more variable: .config <chr>
```

Praktischerweise gibt es eine Autoplot-Funktion, um die besten Modellparameter auszulesen:

```
autoplot(trees_tuned)
```



## 9.11.8 Bestes Modell auswählen

Aus allen Modellkandidaten wählen wir jetzt das beste Modell aus:

```
select_best(trees_tuned)

## # A tibble: 1 × 4
##   cost_complexity tree_depth min_n .config
##       <dbl>        <int>    <int> <chr>
## 1      0.0001         5        2 Preprocessor1_Model07
```

Mit diesem besten Kandidaten definieren wir jetzt das “finale” Modell, wir “finalisieren” das Modell mit den besten Modellparametern:

```
tree_final <-
  finalize_model(tree_model, parameters = select_best(trees_tuned))

tree_final

## Decision Tree Model Specification (regression)
##
## Main Arguments:
##   cost_complexity = 1e-04
##   tree_depth = 5
##   min_n = 2
##
## Computational engine: rpart
```

Hier ist, unser finaler Baum 🌳.

Schließlich updaten wir mit dem finalen Baum noch den Workflow:

```
final_wf <-
  tree_wf %>%
  update_model(tree_final)
```

## 9.11.9 Final Fit

Jetzt fitten wir dieses Modell auf das ganze Train-Sample und predicten auf das Test-Sample:

```
tree_fit_final <-
  final_wf %>%
  last_fit(d_split)

tree_fit_final

## # Resampling results
## # Manual resampling
## # A tibble: 1 × 6
##   splits          id       .metrics .notes   .predictions   .workflow
##   <list>        <chr>     <list>  <list>   <list>        <list>
## 1 <split [24/8]> train/test split <tibble> <tibble> <tibble [8 × 4]> <workflow>
```

```
collect_metrics(tree_fit_final)
```

```
## # A tibble: 2 × 4
##   .metric .estimator .estimate .config
##   <chr>   <chr>      <dbl> <chr>
## 1 rmse    standard     3.93 Preprocessor1_Model1
## 2 rsq     standard     0.683 Preprocessor1_Model1
```

Voilà: Die Modellgüte für das Test-Sample: Im Schnitt liegen wir ca. 4 Meilen daneben mit unseren Vorhersagen, wenn wir RMSE mal so locker interpretieren wollen.

In der Regel ist übrigens RMSE interessanter als R-Quadrat, da R-Quadrat die Güte eines Korrelationsmusters vorhersagt, aber RMSE die Präzision der Vorhersage, also sozusagen die Kürze der Fehlerbalken.

## 9.11.10 Nur zum Spaß: Vergleich mit linearem Modell

Ein einfaches lineares Modell, was hätte das jetzt wohl für eine Modellgüte?

```
lm_model <-
  linear_reg()
```

```
lm_wf <-
  workflow() %>%
  add_model(lm_model) %>%
  add_recipe(recipe1)
```

```
tic()
lm_fit <-
  fit_resamples(
    lm_wf,
    resamples = d_cv
  )
toc()
```

```
## 7.572 sec elapsed
```

```
collect_metrics(lm_fit)
```

```
## # A tibble: 2 × 6
##   .metric .estimator  mean     n std_err .config
##   <chr>   <chr>     <dbl> <int>  <dbl> <chr>
## 1 rmse    standard    4.16    25  0.362 Preprocessor1_Model1
## 2 rsq     standard    0.624    25  0.0587 Preprocessor1_Model1
```

```
lm_fit_final <-
  last_fit(lm_wf, d_split)
```

Wie präzise ist die Vorhersage im Test-Sample?

```
collect_metrics(lm_fit_final)
```

```
## # A tibble: 2 × 4
##   .metric .estimator .estimate .config
##   <chr>   <chr>      <dbl> <chr>
## 1 rmse    standard     5.24 Preprocessor1_Model1
## 2 rsq     standard     0.434 Preprocessor1_Model1
```

Das lineare Modell schneidet etwas (deutlich?) schlechter ab als das einfache Baummodell.

Man beachte, dass die Modellgüte im Train-Sample höher ist als im Test-Sample (Overfitting).

## 10 Ensemble Lerner

### 10.1 Lernsteuerung

#### 10.1.1 Lernziele

- Sie können Algorithmen für Ensemble-Lernen erklären, d.i. Bagging, AdaBoost, XGBoost, Random Forest
- Sie wissen, anhand welche Tuningparameter man Overfitting bei diesen Algorithmen begrenzen kann
- Sie können diese Verfahren in R berechnen

#### 10.1.2 Literatur

- Rhys, Kap. 8

#### 10.1.3 Hinweise

- Nutzen Sie StackOverflow als Forum für Ihre Fragen - Hier ein Beispiel zu einer Fehlermeldung, die mir Kopfzerbrechen bereitete (<https://stackoverflow.com/questions/72333419/error-on-running-predict-in-tidymodels-error-in-dplyrselect-cant-su/72341769#72341769>)

## 10.2 Vorbereitung

In diesem Kapitel werden folgende R-Pakete benötigt:

```
library(tidymodels)
library(tictoc) # Zeitmessung
library(vip) # Variable importance plot
```

## 10.3 Hinweise zur Literatur

Die folgenden Ausführungen basieren primär auf Rhys (2020), aber auch auf James et al. (2021) und (weniger) Kuhn and Johnson (2013).

## 10.4 Wir brauchen einen Wald

Ein Pluspunkt von Entscheidungsbäumen ist ihre gute Interpretierbarkeit. Man könnte behaupten, dass Bäume eine typische Art des menschlichen Entscheidungsverhalten nachahmen: "Wenn A, dann tue B, ansonsten tue C" (etc.). Allerdings: Einzelne Entscheidungsbäume haben oft keine so gute Prognosegenauigkeit. Der oder zumindest ein Grund ist, dass sie (zwar wenig Bias aber) viel Varianz aufweisen. Das sieht man z.B. daran, dass die Vorhersagegenauigkeit stark schwankt, wählt man eine andere Aufteilung von Train- vs. Test-Sample. Anders gesagt: Bäume overfitten ziemlich schnell. Und obwohl das No-Free-Lunch-Theorem zu den Grundfesten des maschinellen Lernens (oder zu allem wissenschaftlichen Wissen) gehört, kann man festhalten, dass sog. *Ensemble-Lernen* fast immer besser sind als einzelne Baummodelle. Kurz gesagt: Wir brauchen einen Wald: 🌳🌳🌳<sup>4</sup>

## 10.5 Was ist ein Ensemble-Lerner?

Ensemble-Lerner kombinieren mehrere schwache Lerner zu einem starken Lerner. Das Paradebeispiel sind baumbasierte Modelle; darauf wird sich die folgende Ausführung auch begrenzen. Aber theoretisch kann man jede Art von Lerner kombinieren. Bei numerischer Prädiktion wird bei Ensemble-Lerner zumeist der Mittelwert als Optmierungskriterium herangezogen; bei Klassifikation (nominaler Prädiktion) hingegen die modale Klasse (also die häufigste). Warum hilft es, mehrere Modelle (Lerner) zu einem zu aggregieren? Die Antwort lautet, dass die Streuung der Mittelwerte sinkt, wenn die Stichprobengröße steigt. Zieht man Stichproben der Größe 1, werden die Mittelwerte stark variieren, aber bei größeren Stichproben (z.B. Größe 100) deutlich weniger<sup>5</sup>. Die Streuung der Mittelwerte in den Stichproben nennt man bekanntlich *Standardefehler* (*se*). Den *se* des Mittelwerts ( $se_M$ ) für eine normalverteilte Variable  $X \sim N(\mu, \sigma)$  gilt:  $se_M = \sigma / \sqrt{n}$ , wobei  $\sigma$  die SD der Verteilung und  $\mu$  den Erwartungswert ("Mittelwert") meint, und  $n$  ist die Stichprobengröße.

Je größer die Stichprobe, desto kleiner die Varianz des Schätzers (*ceteris paribus*). Anders gesagt: Größere Stichproben schätzen genauer als kleine Stichproben.

Aus diesem Grund bietet es sich an, schwache Lerner mit viel Varianz zu kombinieren, da die Varianz so verringert wird.

## 10.6 Bagging

### 10.6.1 Bootstrapping

Das erste baumbasierte Modell, was vorgestellt werden soll, basiert auf sog. *Bootstrapping*, ein Standardverfahren in der Statistik (James et al. 2021).

Bootstrapping ist eine Nachahmung für folgende Idee: Hätte man viele Stichproben aus der relevanten Verteilung, so könnte man z.B. die Genauigkeit eines Modells  $\hat{f}_{\bar{X}}$  zur Schätzung des Erwartungswertes  $\mu$  einfach dadurch bestimmen, indem man *se* berechnet, also die Streuung der Mittelwerte  $\bar{X}$  berechnet. Außerdem gilt, dass die Präzision der Schätzung des Erwartungswerts steigt mit steigendem Stichprobenumfang  $n$ . Wir könnten also für jede der  $B$  Stichproben,  $b = 1, \dots, B$ , ein (Baum-)Modell berechnen,  $\hat{f}^b$ , und dann deren Vorhersagen aggregieren (zum Mittelwert oder Modalwert). Das kann man formal so darstellen (James et al. 2021):

$$\hat{f}_{\bar{X}} = \frac{1}{B} \sum_{b=1}^B \hat{f}^b$$

Mit diesem Vorgehen kann die Varianz des Modells  $\hat{f}_{\bar{X}}$  verringert werden; die Vorhersagegenauigkeit steigt.

Leider haben wir in der Regel nicht viele ( $B$ ) Datensätze.

Daher "bauen" wir uns aus dem einzelnen Datensatz, der uns zur Verfügung steht, viele Datensätze. Das hört sich nach "too good to be true" an<sup>6</sup>. Weil es sich unglaublich anhört, nennt man das entsprechende Verfahren (gleich kommt es!) auch "Münchhausen-Methode", nach dem berühmten Lüggenbaron. Die Amerikaner ziehen sich übrigens nicht am Schopf aus dem Sumpf, sondern mit den Stiefelschläufen (die Cowboys wieder), daher spricht man im Amerikanischen auch von der "Bostrapping-Methode".

Diese "Pseudo-Stichproben" oder "Bostrapping-Stichproben" sind aber recht einfach zu gewinnen.. Gegeben sei Stichprobe der Größe  $n$ :

1. Ziehe mit Zurücklegen (ZmZ) aus der Stichprobe  $n$  Beobachtungen
2. Fertig ist die Bostrapping-Stichprobe.

Abb. 10.1 verdeutlicht das Prinzip des ZMZ, d.h. des Bootstrappings. Wie man sieht, sind die Bootstrap-Stichproben (rechts) vom gleichen Umfang  $n$  wie die Originalstichprobe (links). Allerdings kommen nicht alle Fälle (in der Regel) in den "Bootstrap-Beutel" (in bag), sondern einige Fälle werden oft mehrfach gezogen, so dass einige Fälle nicht gezogen werden (out of bag).

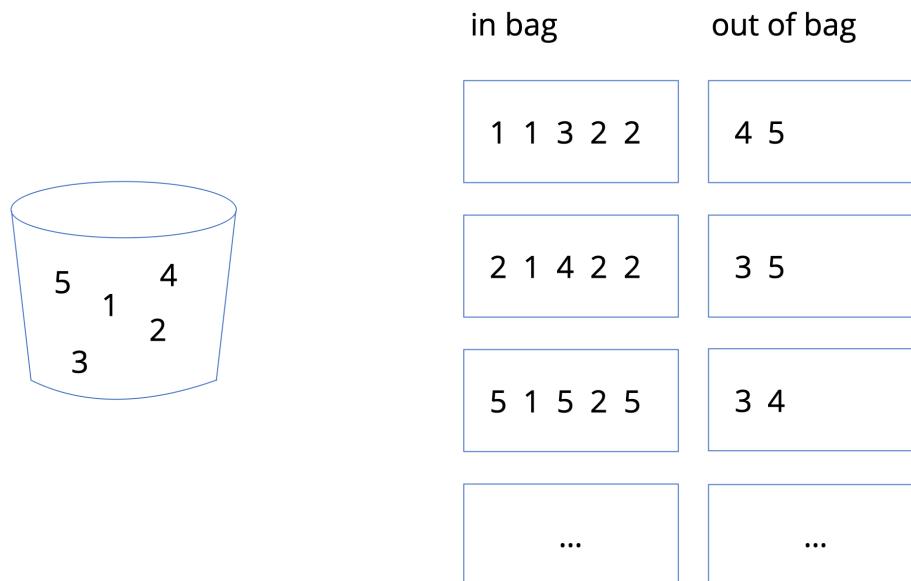


Abbildung 10.1: Bootstrapping: Der Topf links symbolisiert die Original-Stichprobe, aus der wir hier mehrere ZMZ-Stichproben ziehen (Rechts), dargestellt mit 'in bag'

Man kann zeigen, dass ca. 2/3 der Fälle gezogen werden, bzw. ca. 1/3 nicht gezogen werden. Die nicht gezogenen Fälle nennt man auch *out of bag* (OOB).

Für die Entwicklung des Bootstrapping wurde der Autor, Bradley Efron, im Jahr 2018 mit dem internationalen Preis für Statistik ausgezeichnet (<https://www.amstat.org/news-listing/2021/10/08/international-prize-in-statistics-awarded-to-bradley-efron>);

"While statistics offers no magic pill for quantitative scientific investigations, the bootstrap is the best statistical pain reliever ever produced," says Xiao-Li Meng, Whipple V. N. Jones Professor of Statistics at Harvard University."

## 10.7 Bagging-Algorithmus

Bagging, die Kurzform für Bootstrap-Aggregation ist wenig mehr als die Umsetzung des Bootstrappings.

Der Algorithmus von Bagging kann so beschrieben werden:

1. Wähle  $B$ , die Anzahl der Bootstrap-Stichproben und damit auch Anzahl der Submodelle (Lerner)
2. Ziehe  $B$  Bootstrap-Stichproben
3. Berechne das Modell  $\hat{f}^{*b}$  für jede der  $B$  Stichproben (typischerweise ein einfacher Baum)
4. Schicke die Test-Daten durch jedes Sub-Modell
5. Aggregiere ihre Vorhersage zu einem Wert (Modus bzw. Mittelwert) pro Fall aus dem Test-Sample, zu  $\hat{f}_{\text{bag}}$

Anders gesagt:

$$\hat{f}_{\text{bag}} = \frac{1}{B} \sum_{b=1}^B \hat{f}^{*b}$$

Der Bagging-Algorithmus ist in Abbildung 10.2 dargestellt.

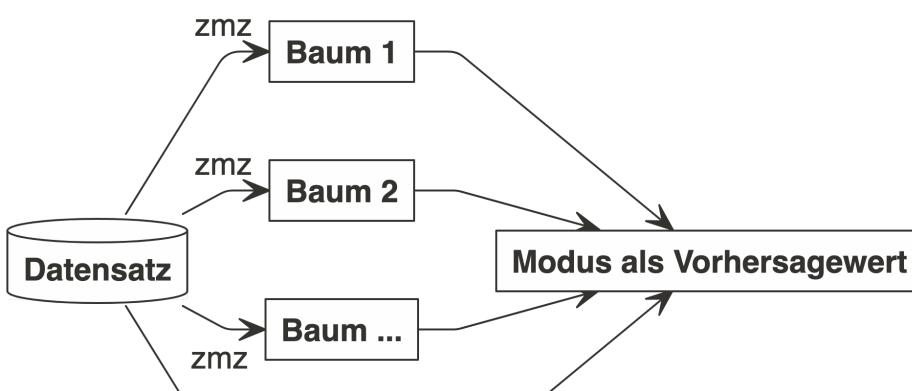




Abbildung 10.2: Bagging schematisch illustriert

Die Anzahl der Bäume (allgemeiner: Submodelle)  $B$  ist häufig im oberen drei- oder niedrigem vierstelligen Bereich, z.B.  $B = 1000$ . Eine gute Nachricht ist, dass Bagging nicht überanpasst, wenn  $B$  groß wird.

## 10.7.1 Variablenrelevanz

Man kann die Relevanz der Prädiktoren in einem Bagging-Modell auf mehrere Arten schätzen. Ein Weg (bei numerischer Prädiktion) ist, dass man die RSS-Verringerung, die durch Aufteilung anhand eines Prädiktors erzeugt wird, mittelt über alle beteiligten Bäume (Modelle). Bei Klassifikation kann man die analog die Reduktion des Gini-Wertes über alle Bäume mitteln und als Schätzwert für die Relevanz des Prädiktors heranziehen.

## 10.7.2 Out of Bag Vorhersagen

Da nicht alle Fälle der Stichprobe in das Modell einfließen (sondern nur ca. 2/3), kann der Rest der Fälle zur Vorhersage genutzt werden. Bagging erzeugt sozusagen innerhalb der Stichprobe selbstständig ein Train- und ein Test-Sample. Man spricht von *Out-of-Bag-Schätzung* (OOB-Schätzung). Der OOB-Fehler (z.B. MSE bei numerischen Modellen und Genauigkeit bei nominalen) ist eine valide Schätzung des typischen Test-Sample-Fehlers.

Hat man aber Tuningparameter, so wird man dennoch auf die typische Train-Test-Aufteilung zurückgreifen, um Overfitting durch das Ausprobieren der Tuning-Kandidaten zu vermeiden (was sonst zu Zufallstrefern führen würde bei genügend vielen Modellkandidaten).

## 10.8 Random Forests

Random Forests (“Zufallswälder”) sind eine Weiterentwicklung von Bagging-Modellen. Sie sind Bagging-Modelle, aber haben noch ein Ass im Ärmel: Und zwar wird an jedem Slit (Asteigabel, Aufteilung) *nur eine Zufallsauswahl an  $m$  Prädiktoren berücksichtigt*. Das hört sich verrückt an: “Wie, mit weniger Prädiktoren soll eine bessere Vorhersage erreicht werden!?” Ja, genau so ist es! Nehmen Sie an, es gibt im Datensatz einen sehr starken und ein paar mittelstarke Prädiktoren; der Rest der Prädiktoren ist wenig relevant. Wenn Sie jetzt viele “gebootstrapte”<sup>7</sup> ziehen, werden diese Bäume sehr ähnlich sein: Der stärkste Prädiktor steht vermutlich immer ob an der Wurzel, dann kommen die mittelstarken Prädiktoren. Jeder zusätzliche Baum trägt dann wenig neue Information bei. Anders gesagt: Die Vorhersagen der Bäume sind dann sehr ähnlich bzw. hoch korreliert. Bildet man den Mittelwert von hoch korrelierten Variablen, verringert sich leider die Varianz nur *wenig* im Vergleich zu nicht oder gering korrelierten Variablen (James et al. 2021). Dadurch dass Random Forests nur  $m$  der  $p$  Prädiktoren pro Split zulassen, werden die Bäume unterschiedlicher. Wir “dekorrelieren” die Bäume. Bildet man den Mittelwert von gering(er) korrelierten Variablen, so ist die Varianzreduktion höher – und die Vorhersage genauer. Lässt man pro Split  $m = p$  Prädiktoren zu, so gleicht Bagging dem Random Forest. Die Anzahl  $m$  der erlaubten Prädiktoren werden als Zufallstichprobe aus den  $p$  Prädiktoren des Datensatzes gezogen (ohne Zurücklegen).  $m$  ist ein Tuningparameter;  $m = \sqrt{p}$  ist ein beliebter Startwert. In den meisten Implementierungen wird  $m$  mit `mtry` bezeichnet (so auch in Tidymodels).

Der Random-Forest-Algorithmus ist in Abb. 10.3 illustriert.

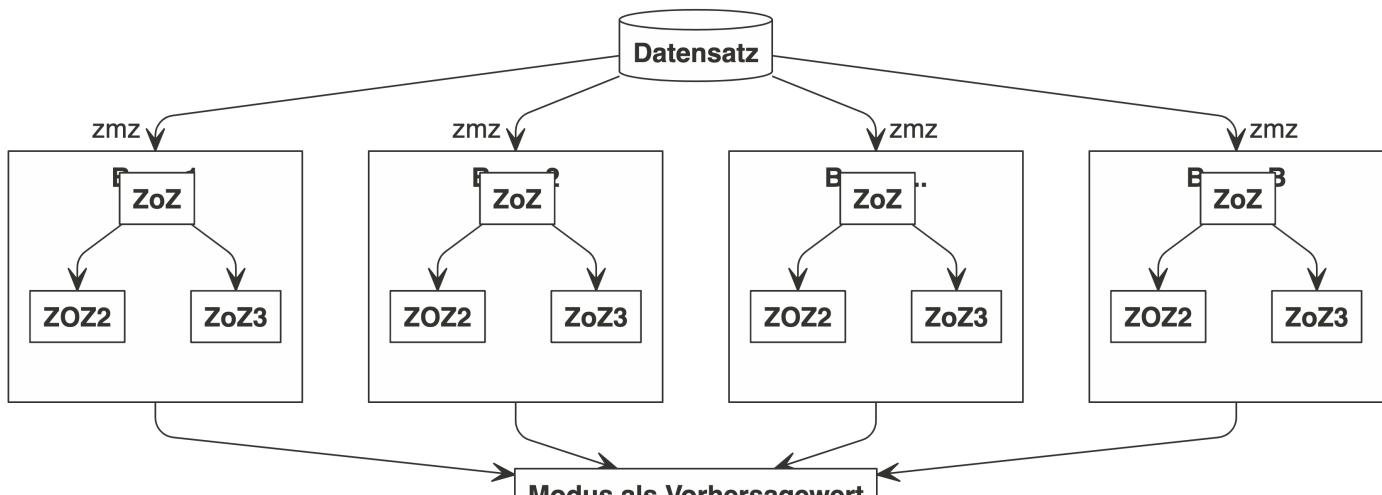


Abbildung 10.3: Zufallswälder durch Ziehen mit Zurücklegen (zmz) und Ziehen ohne Zurücklegen (ZoZ)

Abb. 10.4 vergleicht die Test-Sample-Vorhersagegüte von Bagging- und Random-Forest-Algorithmen aus James et al. (2021). In diesem Fall ist die Vorhersagegüte deutlich unter der OOB-Güte; laut James et al. (2021) ist dies hier "Zufall".

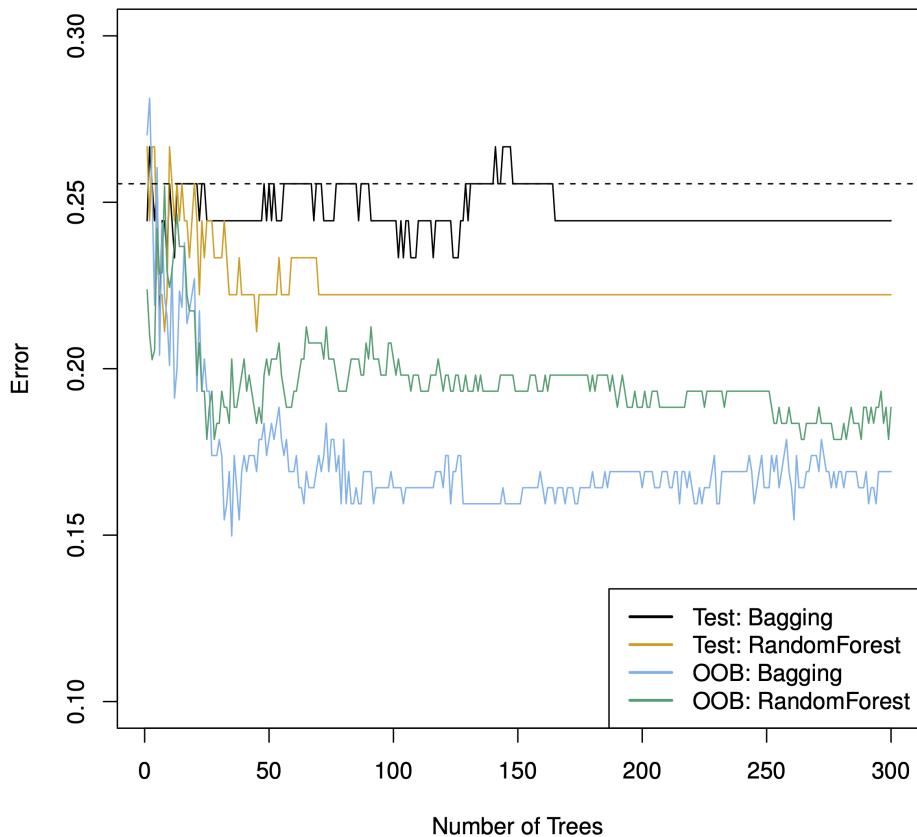


Abbildung 10.4: Test-Sample-Vorhersagegüte von Bagging- und Random-Forest-Algorithmen

Den Effekt von  $m$  (Anzahl der Prädiktoren pro Split) ist in Abb. 10.5 dargestellt (James et al. 2021). Man erkennt, dass der Zusatznutzen an zusätzlichen Bäumen,  $B$ , sich abschwächt.  $m = \sqrt{p}$  schneidet wie erwartet am besten ab.

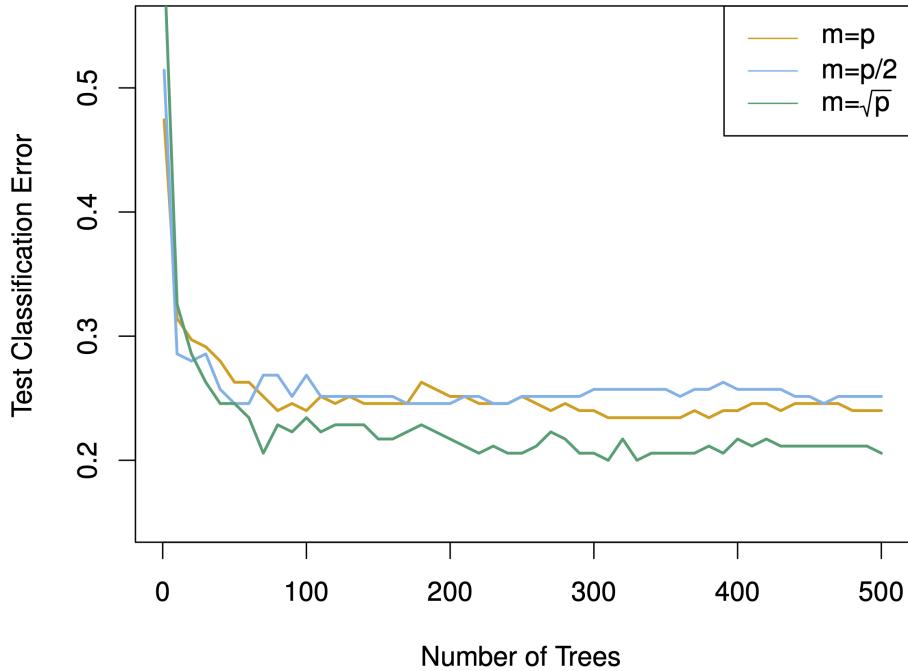


Abbildung 10.5: Test-Sample-Vorhersagegüte von Bagging- und Random-Forest-Algorithmen

## 10.9 Boosting

Im Unterschied zu Bagging und Random-Forest-Modellen wird beim Boosting der "Wald" *sequenziell* entwickelt, nicht gleichzeitig wie bei den anderen vorgestellten "Wald-Modellen". Die zwei bekanntesten Implementierungen bzw. Algorithmus-Varianten sind *AdaBoost* und *XGBoost*. Gerade XGBoost hat den Ruf, hervorragende Vorhersagen zu leisten. Auf Kaggle (<https://en.wikipedia.org/wiki/Kaggle>) gewinnt nach einigen Berichten oft XGBoost (<https://www.kaggle.com/code/msjgriffiths/r-what-algorithms-are-most-successful-on-kaggle/report>). Nur neuronale Netze schneiden besser ab. Random-Forest-Modelle kommen nach diesem Bereich auf Platz 3. Allerdings benötigen neuronale Netzen oft riesige Stichprobengrößen und bei spielen ihre Nuanciertheit vor allem bei komplexen Daten wie Bildern oder Sprache aus. Für "rechteckige" Daten (also aus einfachen, normalen Tabellen) wird ein baumbasiertes Modell oft besser abschneiden.

Die Idee des Boosting ist es, anschaulich gesprochen, aus Fehlern zu lernen: Fitte einen Baum, schau welche Fälle er schlecht vorhergesagt hat, konzentriere dich beim nächsten Baum auf diese Fälle und so weiter.

Wie andere Ensemble-Methoden auch kann Boosting theoretisch für beliebige Algorithmen eingesetzt werden. Es macht aber Sinn, Boosting bei "schwachen Lernern" einzusetzen. Typisches Beispiel ist ein einfacher Baum; "einfach" soll heißen, der Baum hat nur wenig Gabeln oder vielleicht sogar nur eine einzige. Dann spricht man von einem *Stumpf*, was intuitiv gut passt.

### 10.9.1 AdaBoost

Der AdaBoost-Algorithmus funktioniert, einfach dargestellt, wie folgt. Zuerst hat jeder Fall  $i$  im Datensatz das gleiche Gewicht. Die erste (und alle weiteren) Stichprobe werden per Bootstrapping aus dem Datensatz gezogen. Dabei ist die Wahrscheinlichkeit, gezogen zu werden, proportional zum Gewicht des Falles,  $w_i$ . Da im ersten Durchgang die Gewichte identisch sind, haben zunächst alle Fälle die gleiche Wahrscheinlichkeit, in das Bootstrap-Sample gezogen zu werden. Die Bäume bei AdaBoost sind eigentlich nur "Stümpfe": Sie bestehen aus einem einzelnen Split, s. Abb. 10.6.

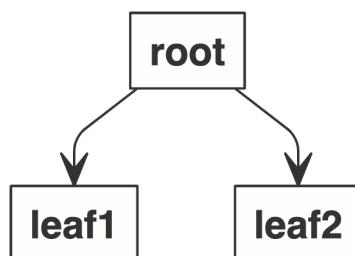


Abbildung 10.6: Ein Baumstumpf bei AdaBoost

Nach Berechnung des Baumes und der Vorhersagen werden die *richtig* klassifizierten Fälle heruntergewichtet und die falsch klassifizierten Fälle hoch gewichtet, also stärker gewichtet (bleiben wir aus Gründen der Einfachheit zunächst bei der Klassifikation). Dieses Vorgehen folgt dem Gedanken, dass man sich seine Fehler genauer anschauen muss, die falsch klassifizierten Fälle sozusagen mehr Aufmerksamkeit bedürfen. Das nächste (zweite) Modell zieht ein weiteres Bootstrap-Sample. Jetzt sind allerdings die Gewichte schon angepasst, so dass mehr Fälle, die im vorherigen Modell falsch klassifiziert wurden, in den neuen (zweiten) Baum gezogen werden. Das neue Modell hat also bessere Chancen, die Aspekte, die das Vorgänger-Modell übersah zu korrigieren bzw. zu lernen. Jetzt haben wir zwei Modelle. Die können wir aggregieren, genau wie beim Bagging: Der Modus der Vorhersage über alle (beide) Bäume hinweg ist dann die Vorhersage für einen bestimmten Fall ("Fall" und "Beobachtung" sind stets synonym für  $y_i$  zu verstehen). So wiederholt sich das Vorgehen für  $B$  Bäume: Die Gewichte werden angepasst, das neue Modell wird berechnet, alle Modelle machen ihre Vorhersagen, per Mehrheitsbeschluss - mit gewichteten Modellen - wird die Vorhersage bestimmt pro Fall. Irgendwann erreichen wir die vorab definierte Maximalzahl an Bäumen,  $B$ , und das Modell kommt zu einem Ende.

Da das Modell die Fehler seiner Vorgänger reduziert, wird der Bias im Gesamtmodell verringert. Da wir gleichzeitig auch Bagging vornehmen, wird aber die Varianz auch verringert. Klingt schon wieder (fast) nach Too-Good-to-be-True!

Das Gewicht  $w_i^b$  des  $i$ ten Falls im  $b$ ten Modell von  $B$  berechnet sich wie folgt (Rhys 2020):

$$w_i^b = \begin{cases} w_i^{b-1} \cdot e^{-\text{model weight}} & \text{wenn korrekt klassifiziert} \\ w_i^{b-1} \cdot e^{\text{model weight}} & \text{wenn inkorrekt klassifiziert} \end{cases}$$

Das *Modellgewicht*  $mw$  berechnet sich dabei so (Rhys 2020):

$$mw_b = 0.5 \cdot \log\left(\frac{1 - p(\text{inkorrekt})}{p(\text{korrekt})}\right) \propto L(p)$$

$p(\cdot)$  ist der Anteil (Wahrscheinlichkeit) einer Vorhersage.

Das Modellgewicht ist ein Faktor, der schlechtere Modelle bestraft. Das folgt dem Gedanken, dass schlechteren Modellen weniger Gehört geschenkt werden soll, aber schlecht klassifizierten Fällen mehr Gehör.

Das Vorgehen von AdaBoost ist in Abb. 10.7 illustriert.

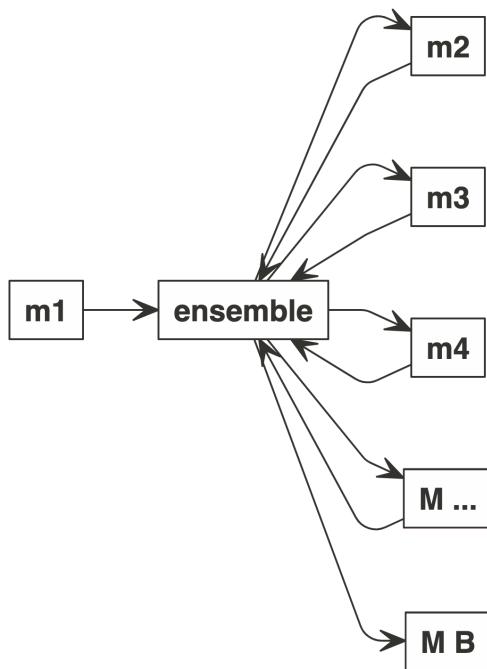


Abbildung 10.7: AdaBoost illustriert

## 10.9.2 XGBoost

XGBoost ist ein Gradientenverfahren, eine Methode also, die die Richtung des parziellen Ableitungskoeffizienten als Optimierungskriterium heranzieht. XGBoost ist ähnlich zu AdaBoost, nur dass *Residuen* modelliert werden, nicht  $y$ . Die Vorhersagefehler von  $\hat{y}^b$  werden die Zielvariable von  $\hat{y}^{b+1}$ . Ein Residuum ist der Vorhersagefehler, bei metrischen Modellen etwa RMSE, oder schlicht  $r_i = y_i - \hat{y}_i$ . Details finden sich z.B. hier (<https://arxiv.org/pdf/1603.02754.pdf>), dem Original XGBoost-Paper (Chen and Guestrin 2016).

Die hohe Vorhersagegenauigkeit von Boosting-Modellen ist exemplarisch in Abb. 10.8 dargestellt (James et al. 2021, S. 358ff). Allerdings verwenden die Autoren Friedmans (2001) *Gradient Boosting Machine*, eine weitere Variante des Boosting.

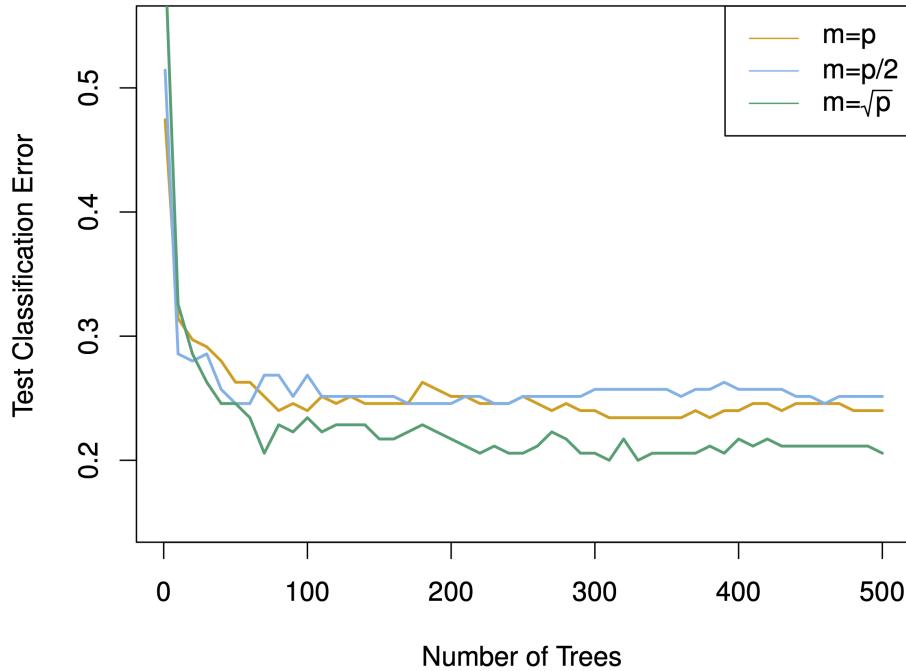


Abbildung 10.8: Vorhersagegüte von Boosting und Random Forest

## 10.10 Tidymodels

### 10.10.1 Datensatz Churn

Wir betrachten einen Datensatz zur Kundenabwanderung (Churn) aus dieser Quelle (<https://www.gmudatamining.com/lesson-13-r-tutorial.html>).

```
knitr::opts_chunk$set(echo = TRUE)
```

```
churn_df <- read_rds('https://gmudatamining.com/data/churn_data.rds')
```

Ein Blick in die Daten:

```
churn_df %>%
  head() %>%
  gt:::gt()
```

churn_df						
canceled_service	enrollment_discount	spouse_partner	dependents	phone_service	internet_service	online_security
yes	no	no	no	multiple_lines	fiber_optic	yes
yes	no	yes	yes	multiple_lines	fiber_optic	no
yes	yes	no	no	single_line	fiber_optic	no
yes	no	yes	yes	single_line	fiber_optic	yes
yes	yes	no	no	single_line	digital	no
yes	no	yes	no	single_line	fiber_optic	yes

### 10.10.2 Data Splitting und CV

Das Kreuzvalidieren (CV) fassen wir auch unter diesen Punkt.

```

churn_split <- initial_split(churn_df, prop = 0.75,
                             strata = canceled_service)

churn_training <- churn_split %>% training()

churn_test <- churn_split %>% testing()

churn_folds <- vfold_cv(churn_training, v = 5)

```

### 10.10.3 Feature Engineering

Hier definieren wir zwei Rezepte. Gleichzeitig verändern wir die Prädiktoren (normalisieren, dummysieren, ...). Das nennt man auch *Feature Engineering*.

```

churn_recipe1 <- recipe(canceled_service ~ ., data = churn_training) %>%
  step_normalize(all_numeric(), -all_outcomes()) %>%
  step_dummy(all_nominal(), -all_outcomes())

churn_recipe2 <- recipe(canceled_service ~ ., data = churn_training) %>%
  step_YeoJohnson(all_numeric(), -all_outcomes()) %>%
  step_normalize(all_numeric(), -all_outcomes()) %>%
  step_dummy(all_nominal(), -all_outcomes())

```

`step_YeoJohnson()` reduziert Schiefe in der Verteilung.

### 10.10.4 Modelle

```

tree_model <- decision_tree(cost_complexity = tune(),
                             tree_depth = tune(),
                             min_n = tune()) %>%
  set_engine('rpart') %>%
  set_mode('classification')

rf_model <- rand_forest(mtry = tune(),
                        trees = tune(),
                        min_n = tune()) %>%
  set_engine('ranger') %>%
  set_mode('classification')

boost_model <- boost_tree(mtry = tune(),
                           min_n = tune(),
                           trees = tune()) %>%
  set_engine("xgboost", nthreads = parallel::detectCores()) %>%
  set_mode("classification")

glm_model <- logistic_reg()

```

### 10.10.5 Workflows

Wir definieren ein Workflow-Set:

```

preproc <- list(rec1 = churn_recipe1, rec2 = churn_recipe2)
models <- list(tree1 = tree_model, rf1 = rf_model, boost1 = boost_model, glm1 = glm_model)

all_workflows <- workflow_set(preproc, models)

```

Infos zu `workflow_set` bekommt man wie gewohnt mit `?workflow_set`.

Im Standard werden alle Rezepte und Modelle miteinander kombiniert (`cross = TRUE`), also `preproc * models` Modelle gefittet.

### 10.10.6 Modelle berechnen mit Tuning, einzeln

Wir könnten jetzt jedes Modell einzeln tunen, wenn wir wollen.

#### 10.10.6.1 Baum

```
tree_wf <-
  workflow() %>%
  add_model(tree_model) %>%
  add_recipe(churn_recipe1)

tic()
tree_fit <-
  tree_wf %>%
  tune_grid(
    resamples = churn_folds,
    metrics = metric_set(roc_auc, sens, yardstick::spec)
  )
toc()
```

```
## 16.453 sec elapsed
```

Im Standard werden 10 Modellkandidaten getuned.

```
tree_fit
```

```
## # Tuning results
## # 5-fold cross-validation
## # A tibble: 5 × 4
##   splits           id     .metrics      .notes
##   <list>          <chr> <list>        <list>
## 1 <split [2393/599]> Fold1 <tibble [30 × 7]> <tibble [0 × 3]>
## 2 <split [2393/599]> Fold2 <tibble [30 × 7]> <tibble [0 × 3]>
## 3 <split [2394/598]> Fold3 <tibble [30 × 7]> <tibble [0 × 3]>
## 4 <split [2394/598]> Fold4 <tibble [30 × 7]> <tibble [0 × 3]>
## 5 <split [2394/598]> Fold5 <tibble [30 × 7]> <tibble [0 × 3]>
```

Schauen wir uns das Objekt etwas näher an:

```
tree_fit$.metrics[[1]]
```

```
## # A tibble: 30 × 7
##   cost_complexity tree_depth min_n .metric .estimator .estimate .config
##   <dbl>          <int> <int> <chr>   <chr>       <dbl> <chr>
## 1 4.47e- 9        14     18 sens    binary    0.848 Preprocessor1...
## 2 4.47e- 9        14     18 spec    binary    0.857 Preprocessor1...
## 3 4.47e- 9        14     18 roc_auc binary   0.917 Preprocessor1...
## 4 3.04e- 3         2      32 sens    binary   0.774 Preprocessor1...
## 5 3.04e- 3         2      32 spec    binary   0.843 Preprocessor1...
## 6 3.04e- 3         2      32 roc_auc binary  0.822 Preprocessor1...
## 7 1.12e-10        5      36 sens    binary   0.819 Preprocessor1...
## 8 1.12e-10        5      36 spec    binary   0.857 Preprocessor1...
## 9 1.12e-10        5      36 roc_auc binary  0.903 Preprocessor1...
## 10 9.36e- 9        4      39 sens    binary  0.856 Preprocessor1...
## # ... with 20 more rows
```

30 Zeilen: 3 Gütemetriken (Sens, Spec, ROC AUC) mit je 10 Werten (Submodellen), gibt 30 Koeffizienten.

Für jeden der 5 Faltungen haben wir also 10 Submodelle.

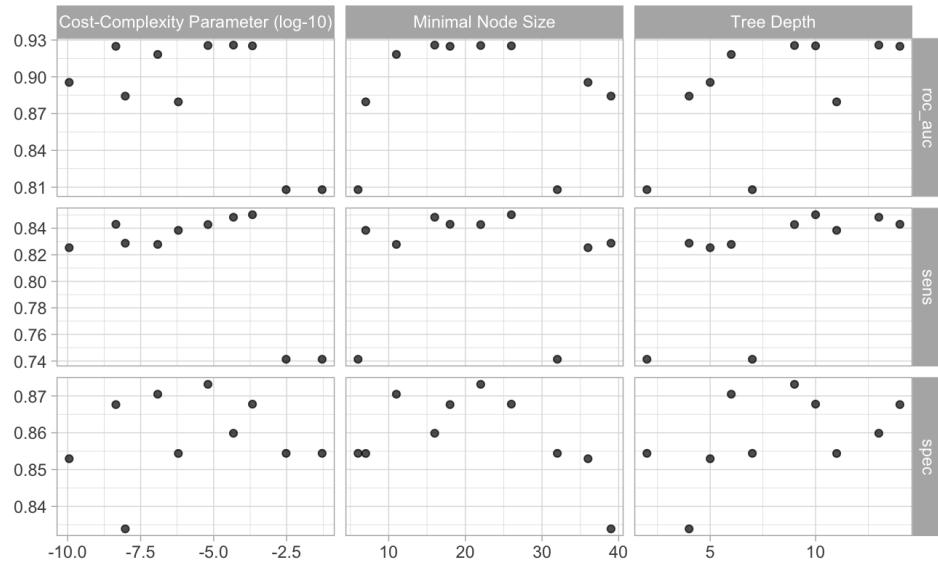
Welches Modell ist das beste?

```
show_best(tree_fit)
```

```
## # A tibble: 5 × 9
##   cost_complexity tree_depth min_n .metric .estimator  mean     n std_err
##   <dbl>          <int> <int> <chr>   <chr>       <dbl> <int> <dbl>
## 1 0.0000476        13     16 roc_auc binary    0.926    5 0.00171
## 2 0.00000638       9     22 roc_auc binary    0.926    5 0.00299
## 3 0.000213         10     26 roc_auc binary    0.925    5 0.00333
## 4 0.00000000447    14     18 roc_auc binary    0.925    5 0.00271
## 5 0.000000121      6     11 roc_auc binary    0.918    5 0.00443
## # ... with 1 more variable: .config <chr>
```

Aha, das sind die fünf besten Modelle, bzw. ihre Tuningparameter, ihre mittlere Güte zusammen mit dem Standardfehler.

```
autoplot(tree_fit)
```



## 10.10.6.2 RF

Was für Tuningparameter hat den der Algorithmus bzw. seine Implementierung?

```
show_model_info("rand_forest")
```

```
## Information for `rand_forest`  
## modes: unknown, classification, regression, censored regression  
##  
## engines:  
##   classification: randomForest, ranger, spark  
##   regression:     randomForest, ranger, spark  
##  
## arguments:  
##   ranger:  
##     mtry --> mtry  
##     trees --> num.trees  
##     min_n --> min.node.size  
##   randomForest:  
##     mtry --> mtry  
##     trees --> ntree  
##     min_n --> nodesize  
##   spark:  
##     mtry --> feature_subset_strategy  
##     trees --> num_trees  
##     min_n --> min_instances_per_node  
##  
## fit modules:  
##   engine mode  
##   ranger classification  
##   ranger regression  
##   randomForest classification  
##   randomForest regression  
##   spark classification  
##   spark regression  
##  
## prediction modules:  
##   mode engine methods  
##   classification randomForest class, prob, raw  
##   classification ranger class, conf_int, prob, raw  
##   classification spark class, prob  
##   regression randomForest numeric, raw  
##   regression ranger conf_int, numeric, raw  
##   regression spark numeric
```

Da die Berechnung einiges an Zeit braucht, kann man das (schon früher einmal berechnete) Ergebnisobjekt von der Festplatte lesen (sofern es existiert). Ansonsten berechnet man neu:

```
if (file.exists("objects/rf_fit1.rds")){
  rf_fit1 <- read_rds("objects/rf_fit1.rds")
} else {
  rf_wf1 <-
    workflow() %>%
    add_model(rf_model) %>%
    add_recipe(churn_recipe1)

  tic()
  rf_fit1 <-
    rf_wf1 %>%
    tune_grid(
      resamples = churn_folds,
      metrics = metric_set(roc_auc, sens, spec)
    )
  toc()
}
```

So kann man das berechnete Objekt abspeichern auf Festplatte, um künftig Zeit zu sparen:

```
write_rds(rf_fit1, file = "objects/rf_fit1.rds")
```

```
rf_fit1
```

```
## # Tuning results
## # 5-fold cross-validation
## # A tibble: 5 × 4
##   splits           id     .metrics       .notes
##   <list>          <chr> <list>        <list>
## 1 <split [2393/599]> Fold1 <tibble [30 × 7]> <tibble [0 × 3]>
## 2 <split [2393/599]> Fold2 <tibble [30 × 7]> <tibble [0 × 3]>
## 3 <split [2394/598]> Fold3 <tibble [30 × 7]> <tibble [0 × 3]>
## 4 <split [2394/598]> Fold4 <tibble [30 × 7]> <tibble [0 × 3]>
## 5 <split [2394/598]> Fold5 <tibble [30 × 7]> <tibble [0 × 3]>
```

```
show_best(rf_fit1)
```

```
## # A tibble: 5 × 9
##   mtry trees min_n .metric .estimator  mean    n std_err .config
##   <int> <int> <int> <chr>    <chr>    <dbl> <int> <dbl> <chr>
## 1     6   1686     18 roc_auc binary    0.958     5  0.00330 Preprocessor1_Model03
## 2     5     747     34 roc_auc binary    0.958     5  0.00324 Preprocessor1_Model10
## 3    10    818     22 roc_auc binary    0.956     5  0.00378 Preprocessor1_Model01
## 4     8     342      2 roc_auc binary    0.955     5  0.00361 Preprocessor1_Model09
## 5    13   1184     25 roc_auc binary    0.954     5  0.00423 Preprocessor1_Model08
```

### 10.10.6.3 XGBoost

```
boost_wf1 <-
  workflow() %>%
  add_model(boost_model) %>%
  add_recipe(churn_recipe1)

tic()
boost_fit1 <-
  boost_wf1 %>%
  tune_grid(
    resamples = churn_folds,
    metrics = metric_set(roc_auc, sens, spec)
  )
toc()
```

Wieder auf Festplatte speichern:

```
write_rds(boost_fit1, file = "objects/boost_fit1.rds")
```

Und so weiter.

## 10.10.7 Workflow-Set tunen

```
if (file.exists("objects/churn_model_set.rds")) {
  churn_model_set <- read_rds("objects/churn_model_set.rds")
} else {
  tic()
  churn_model_set <-
    all_workflows %>%
    workflow_map(
      resamples = churn_folds,
      grid = 20,
      metrics = metric_set(roc_auc),
      seed = 42, # reproducibility
      verbose = TRUE)
  toc()
}
```

Da die Berechnung schon etwas Zeit braucht, macht es Sinn, das Modell (bzw. das Ergebnisobjekt) auf Festplatte zu speichern:

```
write_rds(churn_model_set, file = "objects/churn_model_set.rds")
```

**Achtung** Dieser Schritt ist *gefährlich*: Wenn Sie Ihr Rezept und Fit-Objekt ändern, kriegt das Ihre Festplatte nicht unbedingt mit. Sie könnten also unbemerkt mit dem alten Objekt von Ihrer Festplatte weiterarbeiten, ohne durch eine Fehlermeldung gewarnt zu werden.

Entsprechend kann man das Modellobjekt wieder importieren, wenn einmal abgespeichert:

```
churn_model_set <- read_rds(file = "objects/churn_model_set.rds")
```

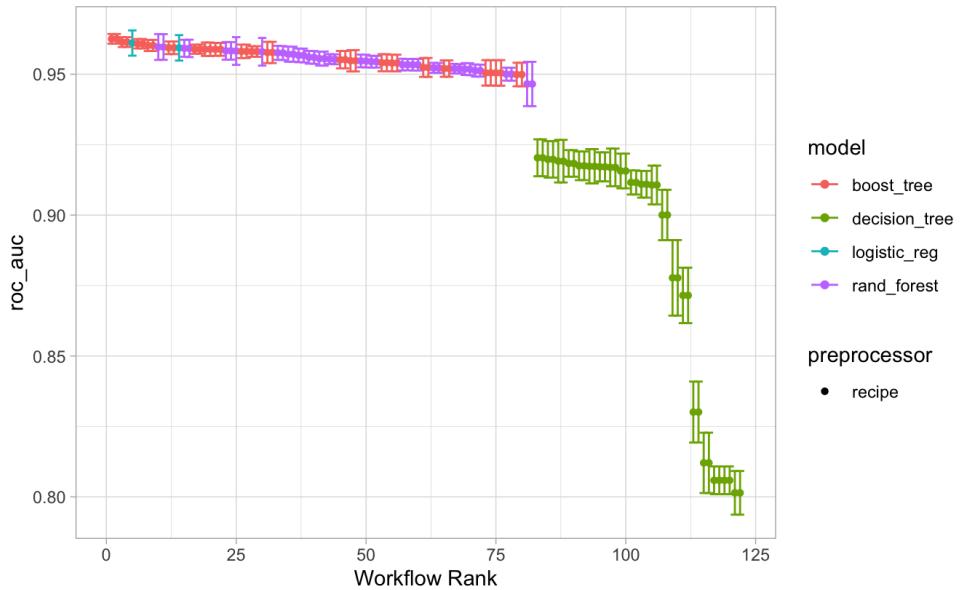
## 10.10.8 Ergebnisse im Train-Sest

Hier ist die Rangfolge der Modelle, geordnet nach mittlerem ROC AUC:

```
rank_results(churn_model_set, rank_metric = "roc_auc")
```

```
## # A tibble: 122 x 9
##   wflow_id .config   .metric  mean std_err  n preprocessor model  rank
##   <chr>     <chr>     <chr>   <dbl> <dbl> <int> <chr>       <chr> <int>
## 1 rec2_boost1 Preprocesso... roc_auc 0.963 0.00104  5 recipe      boos...     1
## 2 rec1_boost1 Preprocesso... roc_auc 0.963 0.00104  5 recipe      boos...     2
## 3 rec2_boost1 Preprocesso... roc_auc 0.961 0.00106  5 recipe      boos...     3
## 4 rec1_boost1 Preprocesso... roc_auc 0.961 0.00106  5 recipe      boos...     4
## 5 rec2_glm1   Preprocesso... roc_auc 0.961 0.00272  5 recipe      logi...     5
## 6 rec1_boost1 Preprocesso... roc_auc 0.961 0.00102  5 recipe      boos...     6
## 7 rec2_boost1 Preprocesso... roc_auc 0.961 0.00102  5 recipe      boos...     7
## 8 rec2_boost1 Preprocesso... roc_auc 0.960 0.00120  5 recipe      boos...     8
## 9 rec1_boost1 Preprocesso... roc_auc 0.960 0.00120  5 recipe      boos...     9
## 10 rec1_rf1    Preprocesso... roc_auc 0.960 0.00278  5 recipe      rand...    10
## # ... with 112 more rows
```

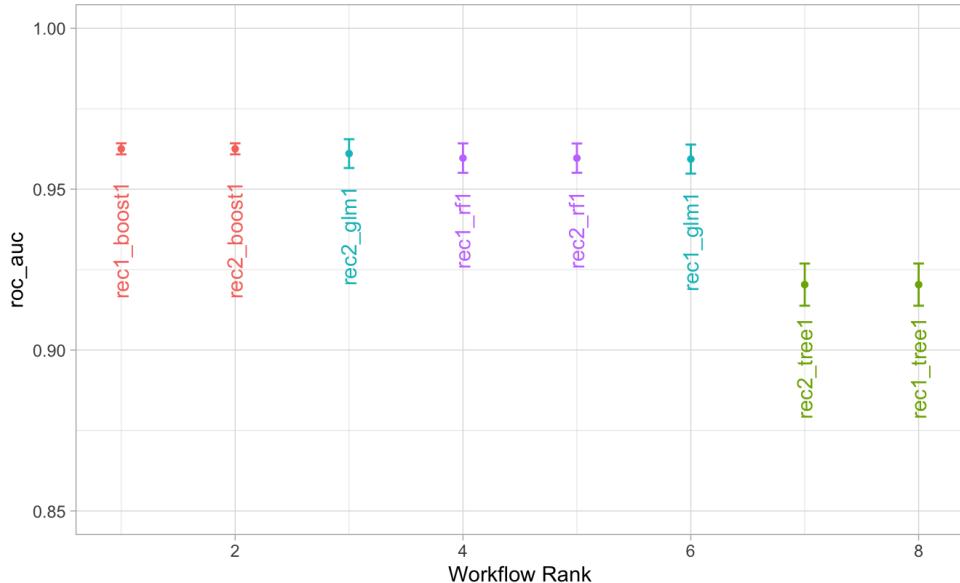
```
autoplot(churn_model_set, metric = "roc_auc")
```



## 10.10.9 Bestes Modell

Und hier nur der beste Kandidat pro Algorithmus:

```
autoplot(churn_model_set, metric = "roc_auc", select_best = "TRUE") +
  geom_text(aes(y = mean - .01, label = wflow_id), angle = 90, hjust = 1) +
  theme(legend.position = "none") +
  lims(y = c(0.85, 1))
```



Boosting hat - knapp - am besten abgeschnitten. Allerdings sind Random Forest und die schlichte, einfache logistische Regression auch fast genau so gut. Das wäre ein Grund für das einfachste Modell, das GLM, zu votieren. Zumal die Interpretierbarkeit am besten ist. Alternativ könnte man sich für das Boosting-Modell aussprechen.

Man kann sich das beste Submodell auch von Tidymodels bestimmen lassen. Das scheint aber (noch) nicht für ein Workflow-Set zu funktionieren, sondern nur für das Ergebnisobjekt von `tune_grid`.

```
select_best(churn_model_set, metric = "roc_auc")
```

```
## Error in `is_metric_maximize()`:
## ! Please check the value of `metric`.
```

`rf_fit1` haben wir mit `tune_grid()` berechnet; mit diesem Modell kann `select_best()` arbeiten:

```
select_best(rf_fit1)
```

```
## # A tibble: 1 × 4
##   mtry trees min_n .config
##   <int> <int> <int> <chr>
## 1     6    1686     18 Preprocessor1_Model03
```

Aber wir können uns händisch behelfen.

Schauen wir uns mal die Metriken (Vorhersagegüte) an:

```
churn_model_set %>%
  collect_metrics() %>%
  arrange(-mean)
```

```
## # A tibble: 122 × 9
##   wflow_id .config      preproc model .metric .estimator  mean     n std_err
##   <chr>     <chr>       <chr>   <chr>   <chr>      <dbl> <int>   <dbl>
## 1 rec1_boost1 Preprocesso... recipe  boos... roc_auc binary  0.963    5 0.00104
## 2 rec2_boost1 Preprocesso... recipe  boos... roc_auc binary  0.963    5 0.00104
## 3 rec1_boost1 Preprocesso... recipe  boos... roc_auc binary  0.961    5 0.00106
## 4 rec2_boost1 Preprocesso... recipe  boos... roc_auc binary  0.961    5 0.00106
## 5 rec2_glm1   Preprocesso... recipe  logi... roc_auc binary  0.961    5 0.00272
## 6 rec1_boost1 Preprocesso... recipe  boos... roc_auc binary  0.961    5 0.00102
## 7 rec2_boost1 Preprocesso... recipe  boos... roc_auc binary  0.961    5 0.00102
## 8 rec1_boost1 Preprocesso... recipe  boos... roc_auc binary  0.960    5 0.00120
## 9 rec2_boost1 Preprocesso... recipe  boos... roc_auc binary  0.960    5 0.00120
## 10 rec1_rf1   Preprocesso... recipe  rand... roc_auc binary 0.960    5 0.00278
## # ... with 112 more rows
```

`rec1_boost1` scheint das beste Modell zu sein.

```
best_model_params <-
extract_workflow_set_result(churn_model_set, "rec1_boost1") %>%
  select_best()

best_model_params
```

```
## # A tibble: 1 × 4
##   mtry trees min_n .config
##   <int> <int> <int> <chr>
## 1     6    80     21 Preprocessor1_Model05
```

## 10.10.10 Finalisieren

Wir entscheiden uns mal für das Boosting-Modell, `rec1_boost1`. Diesen Workflow, in finalisierter Form, brauchen wir für den “final Fit”. Finalisierte Form heißt:

- Schritt 1: Nimm den passenden Workflow, hier `rec1` und `boost1`; das hatte uns oben `rank_results()` verraten.
- Schritt 2: Update (Finalisiere) ihn mit den besten Tuningparameter-Werten

```
# Schritt 1:
best_wf <-
all_workflows %>%
  extract_workflow("rec1_boost1")

best_wf
```

```
## == Workflow ==
## Preprocessor: Recipe
## Model: boost_tree()
##
## — Preprocessor —
## 2 Recipe Steps
##
## • step_normalize()
## • step_dummy()
##
## — Model —
## Boosted Tree Model Specification (classification)
##
## Main Arguments:
##   mtry = tune()
##   trees = tune()
##   min_n = tune()
##
## Engine-Specific Arguments:
##   nthreads = parallel::detectCores()
##
## Computational engine: xgboost
```

Jetzt finalisieren wir den Workflow, d.h. wir setzen die Parameterwerte des besten Submodells ein:

```
# Schritt 2:
best_wf_finalized <-
  best_wf %>%
    finalize_workflow(best_model_params)

best_wf_finalized
```

```
## == Workflow ==
## Preprocessor: Recipe
## Model: boost_tree()
##
## — Preprocessor —
## 2 Recipe Steps
##
## • step_normalize()
## • step_dummy()
##
## — Model —
## Boosted Tree Model Specification (classification)
##
## Main Arguments:
##   mtry = 6
##   trees = 80
##   min_n = 21
##
## Engine-Specific Arguments:
##   nthreads = parallel::detectCores()
##
## Computational engine: xgboost
```

## 10.10.11 Last Fit

```
fit_final <-
  best_wf_finalized %>%
    last_fit(churn_split)

fit_final
```

```
## # Resampling results
## # Manual resampling
## # A tibble: 1 × 6
##   splits           id       .metrics .notes   .predictions .workflow
##   <list>        <chr>     <list>   <list>    <list>      <list>
## 1 <split [2992/998]> train/test split <tibble> <tibble> <tibble>    <workflow>
```

```
collect_metrics(fit_final)
```

```
## # A tibble: 2 × 4
##   .metric  .estimator .estimate .config
##   <chr>    <chr>      <dbl> <chr>
## 1 accuracy binary      0.890 Preprocessor1_Model1
## 2 roc_auc  binary      0.954 Preprocessor1_Model1
```

## 10.10.12 Variablenrelevanz

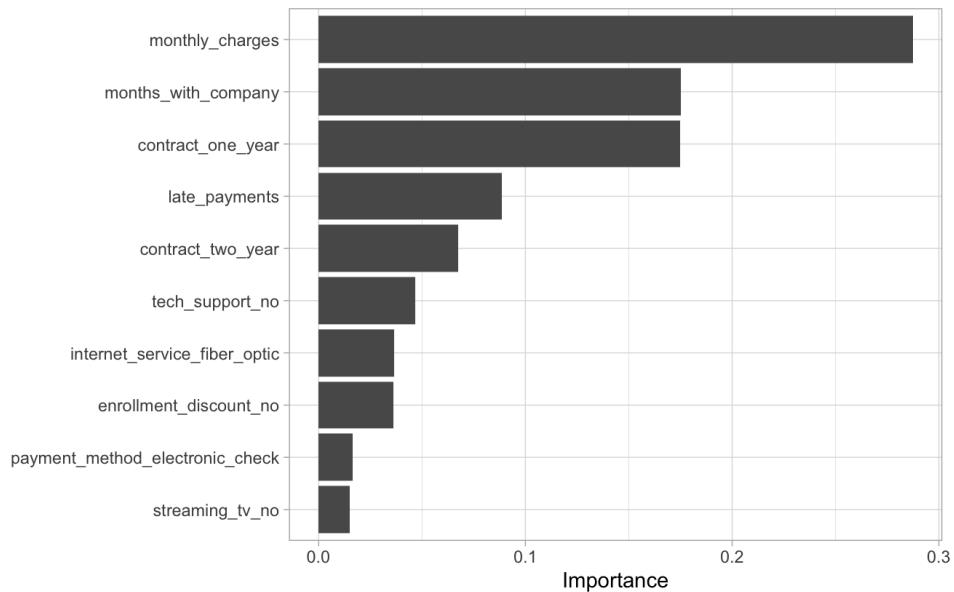
Um die Variablenrelevanz zu plotten, müssen wir aus dem Tidymodels-Ergebnisobjekt das eigentliche Ergebnisobjekt herausziehen, von der R-Funktion, die die eigentliche Berechnung durchführt, das wäre `glm()` bei einer logistischen Regression oder `xgboost::xgb.train()` bei XGBoost:

```
fit_final %>%
  extract_fit_parsnip()
```

```
## parsnip model object
##
## ##### xgb.Booster
## raw: 100.3 Kb
## call:
##
##   xgboost::xgb.train(params = list(eta = 0.3, max_depth = 6, gamma = 0,
##   colsample_bytree = 1, colsample_bynode = 0.285714285714286,
##   min_child_weight = 21L, subsample = 1, objective = "binary:logistic"),
##   data = x$data, nrounds = 80L, watchlist = x$watchlist, verbose = 0,
##   nthreads = 8L, nthread = 1)
## params (as set within xgb.train):
##
##   eta = "0.3", max_depth = "6", gamma = "0", colsample_bytree = "1",
##   colsample_bynode = "0.285714285714286", min_child_weight = "21",
##   subsample = "1", objective = "binary:logistic", nthreads = "8",
##   nthread = "1", validate_parameters = "TRUE"
## xgb.attributes:
##   niter
## callbacks:
##   cb.evaluation.log()
## # of features: 21
## niter: 80
## nfeatures : 21
## evaluation_log:
##   iter training_logloss
##     1      0.5694834
##     2      0.4810064
##   ---
##     79      0.1854236
##     80      0.1851494
```

Dieses Objekt übergeben wir dann an `{vip}`:

```
fit_final %>%
  extract_fit_parsnip() %>%
  vip()
```



## 10.10.13 ROC-Curve

Eine ROC-Kurve berechnet Sensitivität und Spezifität aus den Vorhersagen, bzw. aus dem Vergleich von Vorhersagen und wahren Wert (d.h. der beobachtete Wert).

Ziehen wir also zuerst die Vorhersagen heraus:

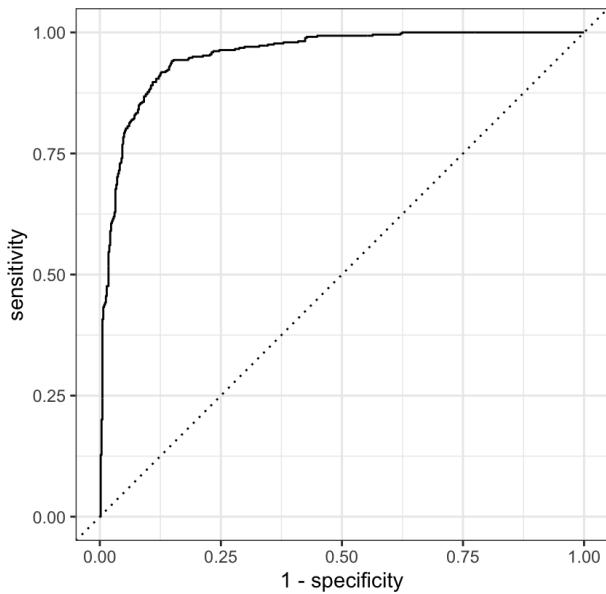
```
fit_final %>%
  collect_predictions()
```

```
## # A tibble: 998 × 7
##   id      .pred_yes .pred_no .row .pred_class canceled_service .config
##   <chr>     <dbl>    <dbl> <int> <fct>       <fct>      <chr>
## 1 train/test spl... 0.792    0.208     2 yes        yes       Prepro...
## 2 train/test spl... 0.773    0.227     6 yes        yes       Prepro...
## 3 train/test spl... 0.496    0.504    13 no         yes      Prepro...
## 4 train/test spl... 0.919    0.0809   15 yes        yes      Prepro...
## 5 train/test spl... 0.989    0.0111   18 yes        yes      Prepro...
## 6 train/test spl... 0.973    0.0267   21 yes        yes      Prepro...
## 7 train/test spl... 0.989    0.0111   23 yes        yes      Prepro...
## 8 train/test spl... 0.998    0.00154  26 yes        yes      Prepro...
## 9 train/test spl... 0.996    0.00388  27 yes        yes      Prepro...
## 10 train/test spl... 0.994   0.00642   41 yes        yes      Prepro...
## # ... with 988 more rows
```

Praktischerweise werden die "wahren Werte" (also die beobachteten Werte), `canceled_service`, ausch angegeben.

Dann berechnen wir die `roc_curve` und `autoplot` ten sie.

```
fit_final %>%
  collect_predictions() %>%
  roc_curve(canceled_service, .pred_yes) %>%
  autoplot()
```



## 10.11 Aufgaben

## 10.12 Aufgaben

- Aufgaben zu Tidymodels, PDF (<https://github.com/sebastiansauer/datascience1/blob/main/Aufgaben/Thema11-Loesungen1.pdf>)
- Aufgaben zu Tidymodels, HTML (<https://github.com/sebastiansauer/datascience1/blob/main/Aufgaben/Thema11-Loesungen1.html>)

## 10.13 Vertiefung

- Einfache Durchführung eines Modellierungs mit XGBoost (<https://data-se.netlify.app/2020/12/14/titanic-tidymodels-boost/>)
- Fallstudie Oregon Schools (<https://bcullen.rbind.io/post/2020-06-02-tidymodels-decision-tree-learning-in-r/>)
- Fallstudie Churn (<https://www.gmudatamining.com/lesson-13-r-tutorial.html>)
- Fallstudie Ikea (<https://juliasilge.com/blog/ikea-prices/>)
- Fallstudie Wasserquellen in Sierra Leone (<https://juliasilge.com/blog/water-sources/>)
- Fallstudie Bäume in San Francisco (<https://dev.to/juliasilge/tuning-random-forest-hyperparameters-in-r-with-tidyTuesday-trees-data-4ilh>)
- Fallstudie Vulkanausbrüche (<https://juliasilge.com/blog/multinomial-volcano-eruptions/>)
- Fallstudie Brettspiele mit XGBoost (<https://juliasilge.com/blog/board-games/>)

# 11 Regularisierte Modelle

## 11.1 Lernsteuerung

### 11.1.1 Lernziele

- Sie können Algorithmen für regularisierte lineare Modelle erklären, d.h. Lasso- und Ridge-Regression
- Sie wissen, anhand welche Tuningparameter man Overfitting bei diesen Algorithmen begrenzen kann
- Sie können diese Verfahren in R berechnen

### 11.1.2 Literatur

- Rhys, Kap. 11

### 11.1.3 Hinweise

- Rhys und ISLR sind eine gute Quelle zum Einstieg in das Thema

## 11.2 Vorbereitung

In diesem Kapitel werden folgende R-Pakete benötigt:

```
library(tidymodels)
library(tictoc) # Zeitmessung
```

## 11.3 Regularisierung

### 11.3.1 Was ist Regularisierung?

Regularisieren verweist auf "regulär"; laut Duden () bedeutet das Wort so viel wie "den Regeln, Bestimmungen, Vorschriften entsprechend; vorschriftsmäßig, ordnungsgemäß, richtig" oder "üblich".

Im Englischen spricht man auch von "penalized models", "bestrafte Modell" und von "shrinkage", von "Schrumpfung" im Zusammenhang mit dieser Art von Modellen.

Regularisierung ist ein Metralgorithmus, also ein Verfahren, was als zweiter Schritt "auf" verschiedene Modelle angewendet werden kann - zumeist aber auf lineare Modelle, worauf wir uns im Folgenden konzentrieren.

Das Ziel von Regularisierung ist es, Overfitting zu vermeiden, in dem die Komplexität eines Modells reduziert wird. Der Effekt von Regularisierung ist, dass die Varianz der Modelle verringert wird und damit das Overfitting. Der Preis ist, dass der Bias erhöht wird, aber oft geht die Rechnung auf, dass der Gewinn größer ist als der Verlust.

Im Kontext von linearen Modellen bedeutet das, dass die Koeffizienten ( $\beta$ s) im Betrag verringert werden durch Regularisierung, also in Richtung Null "geschrumpft" werden.

Dem liegt die Idee zugrunde, dass extreme Werte in den Koeffizienten vermutlich nicht "echt", sondern durch Rauschen fälschlich vorgegaukelt werden.

Die bekanntesten Vertreter dieser Modellart sind *Ridge Regression*, *L2*, das *Lasso*, *L1*, sowie *Elastic Net*.

### 11.3.2 Ähnliche Verfahren

Ein ähnliches Ziel wie der Regularisierung liegt dem Pruning zugrunde, dem nachträglichen Beschneiden von Entscheidungsbäumen. In beiden Fällen wird die Komplexität des Modells verringert, und damit die Varianz auf Kosten eines möglichen Anstiegs der Verzerrung (Bias) des Modells. Unterm Strich hofft man, dass der Gewinn die Kosten übersteigt und somit der Fit im Test-Sample besser wird.

Eine Andere Art der Regularisierung wird durch die Verwendung von Bayes-Modellen erreicht: Setzt man einen konservativen Prior, etwa mit Mittelwert Null und kleiner Streuung, so werden die Posteriori-Koeffizienten gegen Null hin geschrumpft werden.

Mit Mehrebenen-Modellen (Multi Level Models) lässt sich ein ähnlicher Effekt erreichen.

### 11.3.3 Normale Regression (OLS)

Man kann sich fragen, warum sollte man an der normalen Least-Square-Regression (OLS: Ordinary Least Square) weiter herumbasteln wollen, schließlich garantiert das Gauss-Markov-Theorem ([https://en.wikipedia.org/wiki/Gauss%20Markov\\_theorem](https://en.wikipedia.org/wiki/Gauss%20Markov_theorem)), dass eine lineare Regression den besten linearen unverzerrten Schätzwert (BLUE, best linear unbiased estimator) stellt, vorausgesetzt die Voraussetzungen der Regression sind erfüllt.

Ja, die Schätzwerte (Vorhersagen) der Regression sind BLUE, schätzen also den wahren Wert korrekt und maximal präzise. Das gilt (natürlich) nur, wenn die Voraussetzungen der Regression erfüllt sind, also vor allem, dass die Beziehung auch linear-additiv ist.

Zur Erinnerung, mit OLS minimiert man den quadrierten Fehler, *RSS*, Residual Sum of Square:

$$RSS = \sum_{i=1}^n \left( y_i - \beta_0 - \sum_{j=1}^p \beta_j x_{ij} \right)^2$$

Man sucht also diejenigen Koeffizientenwerte  $\beta$  (Argumente der Loss-Funktion RSS), die RSS minimieren:

$$\beta = \arg \min(RSS)\beta$$

Es handelt sich hier um Schätzwerte, die meist mit dem Hütchen  $\hat{\beta}$  ausgedrückt werden, hier aber zur einfacheren Notation weggelassen sind.

Abb. 11.1 visualisiert die Optimierung mit OLS Quelle (<https://www.crumplab.com/rstatsforpsych/regression.html>). An gleicher Stelle (<https://www.crumplab.com/rstatsforpsych/regression.html>) findet sich eine gute Darstellung zu den (mathematischen) Grundlagen der OLS-Regression.

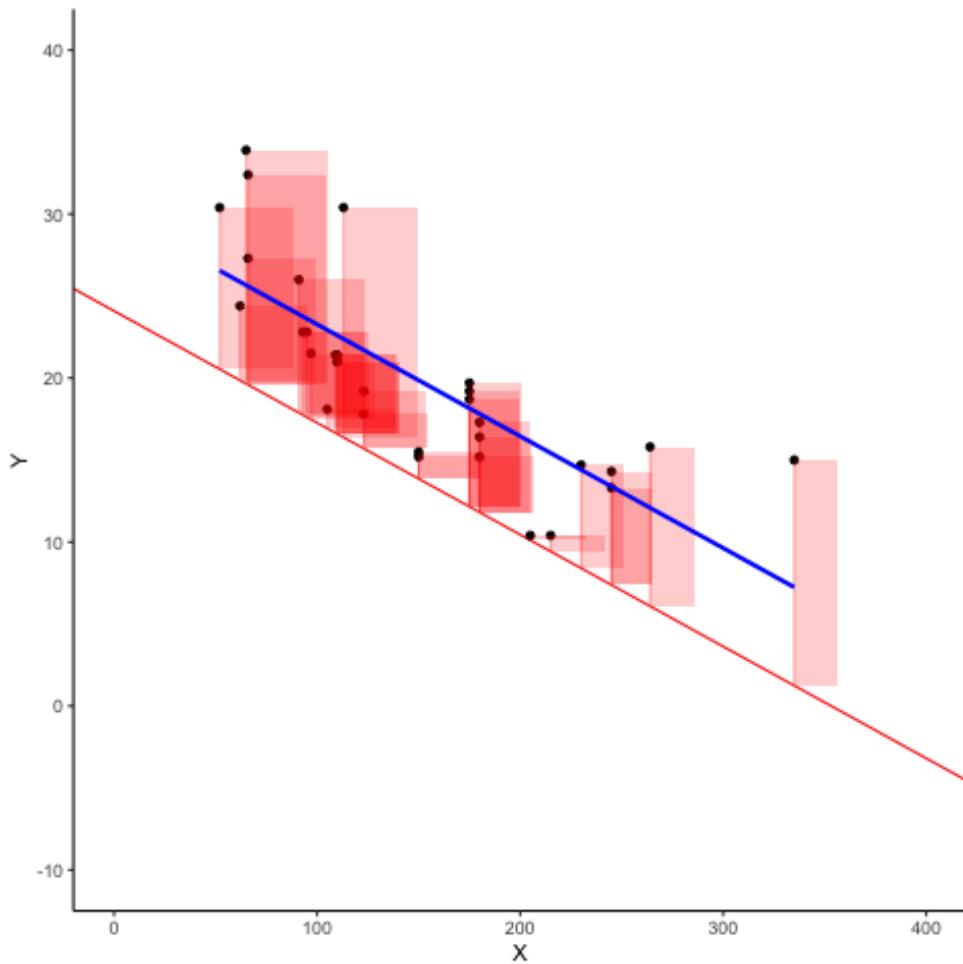


Abbildung 11.1: Visualisierung der Minimierung der RSS durch OLS

Übrigens nennt man Funktionen, die man minimiert mit Hilfe von Methoden des maschinellen Lernens mit dem Ziel die optimalen Koeffizienten (wie  $\beta$ s) zu finden, auch *Loss Functions* (Kostenfunktion).

Das Problem der Regression ist, dass die schöne Eigenschaft BLUE nur im *Train-Sample*, *nicht* (notwendig) im Test-Sample gilt.

## 11.4 Ridge Regression, L2

### 11.4.1 Strafterm

Ridge Regression ist sehr ähnlich zum OLS-Algorithmus, nur das ein "Strafterm aufgebrummt" wird, der *RSS* erhöht.

Der Gesamtterm, der optimiert wird,  $L_{L2}$  (Loss Level 2) ist also die Summe aus RSS und dem Strafterm:

$$L_{L2} = \text{RSS} + \text{Strafterm}$$

Der Strafterm ist so aufgebaut, dass (im Absolutbetrag) größere Koeffizienten mehr zum Fehler beitragen, also eine Funktion der (quadrierten) Summe der Absolutwerte der Koeffizienten:

$$\text{Strafterm} = \lambda \sum_{j=1}^p \beta_j^2$$

Man nennt den L2-Strafterm auch L2-Norm<sup>8</sup>.

Dabei ist  $\lambda$  (lambda) ein Tuningparameter, der bestimmt, wie stark die Bestrafung ausfällt. Den Wert von  $\lambda$  lassen wir durch Tuning bestimmen, wobei  $\lambda \in \mathbb{R}^+ \setminus \{0\}$ . Es gilt: Je größer lambda, desto stärker die Schrumpfung der Koeffizienten gegen Null, da der gesamte zu minimierende Term,  $L_{L2}$  entsprechend durch lambda vergrößert wird.

Der Begriff "L2" beschreibt dass es sich um eine quadrierte Normierung handelt.

Der Begriff "Norm" stammt aus der Vektoralgebra. Die L2-Norm eines Vektors  $\|v\|$  mit  $k$  Elementen ist so definiert Quelle (<https://towardsdatascience.com/intuitions-on-l1-and-l2-regularisation-235f2db4c261>):

$$\|v\| = \left( |v_1|^2 + |v_2|^2 + |v_i|^2 + \dots + |v_k|^2 \right)^{1/2}$$

wobei  $|v_i|$  den Absolutwert (Betrag) meint de Elements  $v_i$  meint. Im Falle von reellen Zahlen und Quadrierung braucht es hier die Absolutfunktion nicht.

Im Falle von zwei Elementen vereinfacht sich obiger Ausdruck zu:

$$\|v\| = \sqrt{v_1^2 + v_2^2}$$

Das ist nichts anderes als Pythagoras' Gesetz im euklidischen Raum.

Der Effekt von  $\lambda \sum_{j=1}^p \beta_j^2$  ist wie gesagt, dass die Koeffizienten in Richtung Null geschrumpft werden. Wenn  $\lambda = 0$ , resultiert OLS. Wenn  $\lambda \rightarrow \infty$ , werden alle Koeffizienten auf Null geschätzt werden, Abb. 11.2 verdeutlicht dies (James et al. 2021).

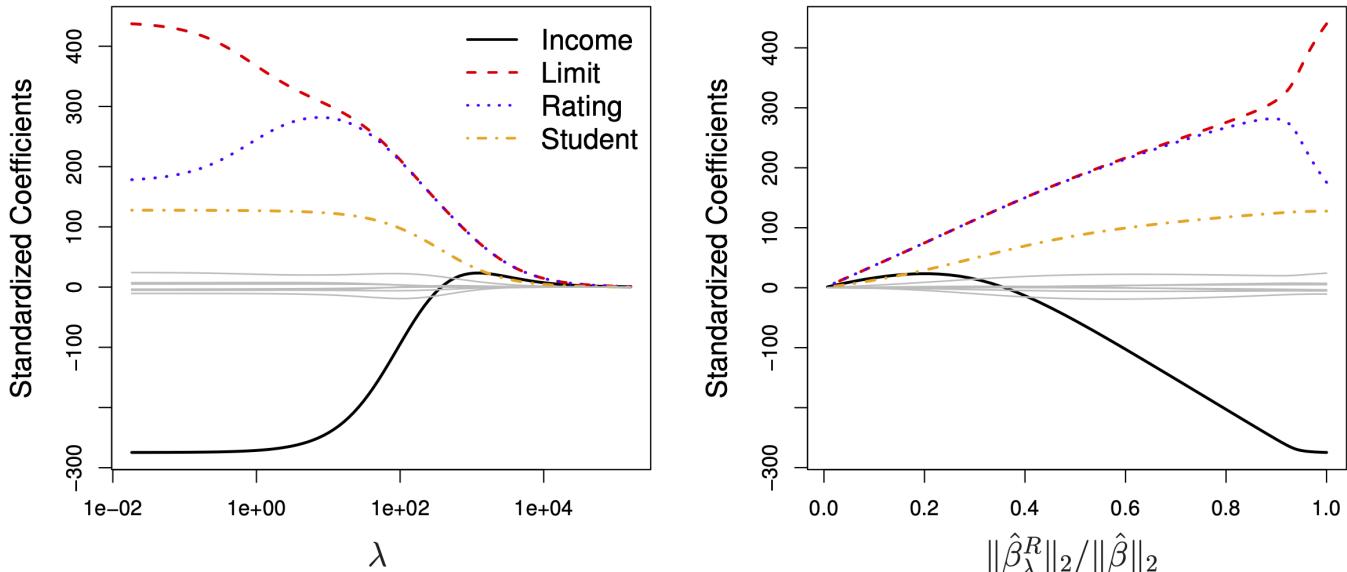


Abbildung 11.2: Links: Regressionskoeffizienten als Funktion von lambda. Rechts: L2-Norm der Ridge-Regression im Verhältnis zur OLS-Regression

## 11.4.2 Standardisierung

Die Straftermformel sagt uns, dass die Ridge-Regression abhängig von der Skalierung der Prädiktoren ist. Daher sollten die Prädiktoren vor der Ridge-Regression zunächst auf  $sd = 1$  standardisiert werden. Da wir  $\beta_0$  nicht schrumpfen wollen, sondern nur die Koeffizienten der Prädiktoren, bietet es sich an, die Prädiktoren dazu noch zu zentrieren. Kurz: Die z-Transformation bietet sich als Vorverarbeitung zur Ridge-Regression an.

# 11.5 Lasso, L1

## 11.5.1 Strafterm

Der Strafterm in der "Lasso-Variante" der regularisierten Regression lautet so:

$$\text{Strafterm} = \lambda \sum_{j=1}^p |\beta_j|,$$

ist also analog zur Ridge-Regression konzipiert.

Genau wie bei der L2-Norm-Regularisierung ist ein "guter" Wert von lambda entscheidend. Dieser Wert wird, wie bei der Ridge-Regression, durch Tuning bestimmt.

Der Unterschied ist, dass die L1-Norm (Absolutwerte) und nicht die L2-Norm (Quadratwerte) verwendet werden.

Die L1-Norm eines Vektors ist definiert durch  $\|\beta\|_1 = \sum |\beta_j|$ .

## 11.5.2 Variablenelektion

Genau wie die Ridge-Regression führt ein höhere lambda-Wert zu einer Regularisierung (Schrumpfung) der Koeffizienten. Im Unterschied zur Ridge-Regression hat das Lasso die Eigenschaft, einzelne Parameter auf exakt Null zu schrumpfen und damit faktisch als Prädiktor auszuschließen. Anders gesagt hat das Lasso die praktische Eigenschaft, Variablenelektion zu ermöglichen.

Abb. 11.3 verdeutlicht den Effekt der Variablenelektion, vgl. James et al. (2021), Kap. 6.2. Die Ellipsen um  $\beta$  herum nennt man Kontourlinien. Alle Punkte einer Kontourlinie haben den gleichen RSS-Wert, stehen also für eine gleichwertige OLS-Lösung.

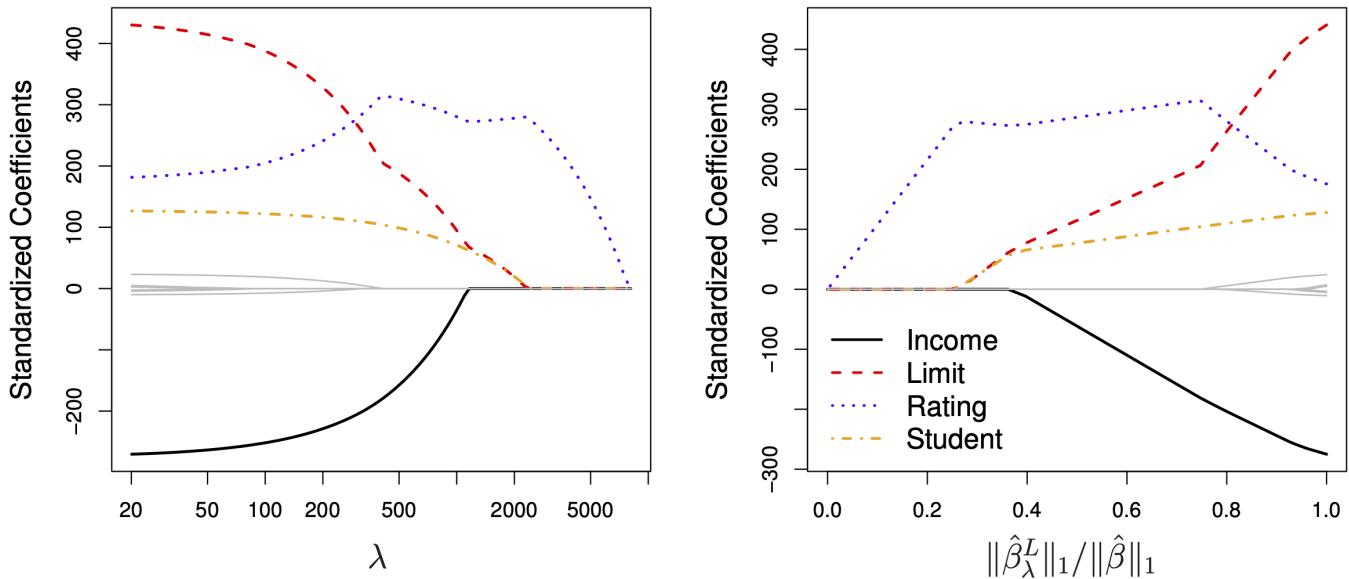


Abbildung 11.3: lambda in der Lasso-Regression

Warum erlaubt die L1-Norm Variablenelektion, die L2-Norm aber nicht? Abb. 11.4 verdeutlicht den Unterschied zwischen L1- und L2-Norm. Es ist eine Regression mit zwei Prädiktoren, also den zwei Koeffizienten  $\beta_1, \beta_2$  dargestellt.

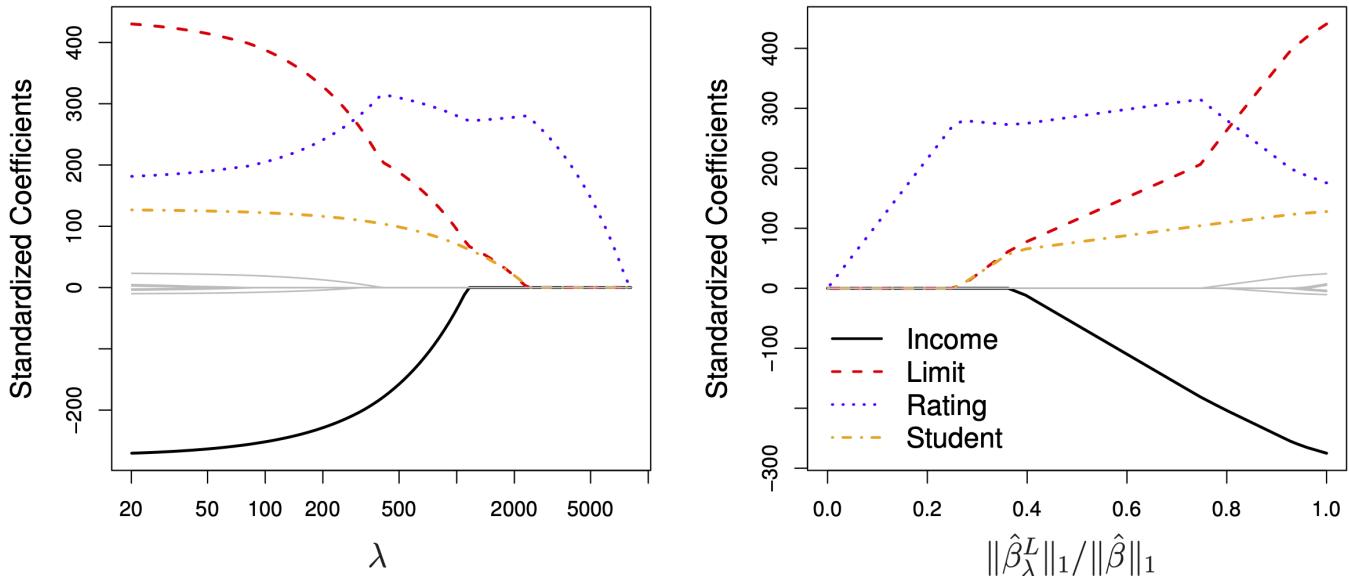


Abbildung 11.4: Verlauf des Strafterms bei der L1-Norm (links) und der L2-Norm (rechts)

Betrachten wir zunächst das rechte Teilbild für die L2-Norm aus Abb. 11.4, das in Abb. 11.5 in den Fokus gerückt wird (Rhys 2020).

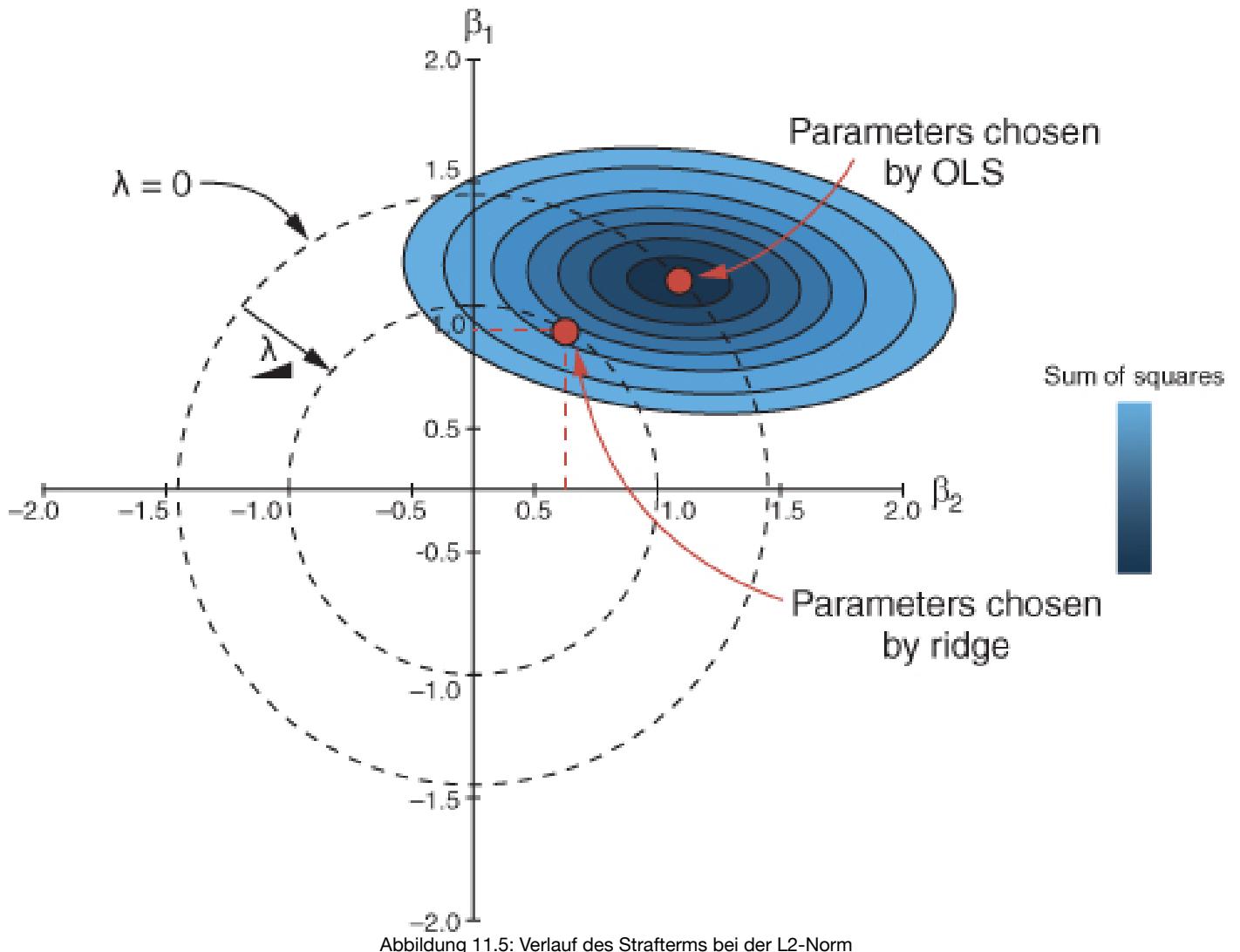
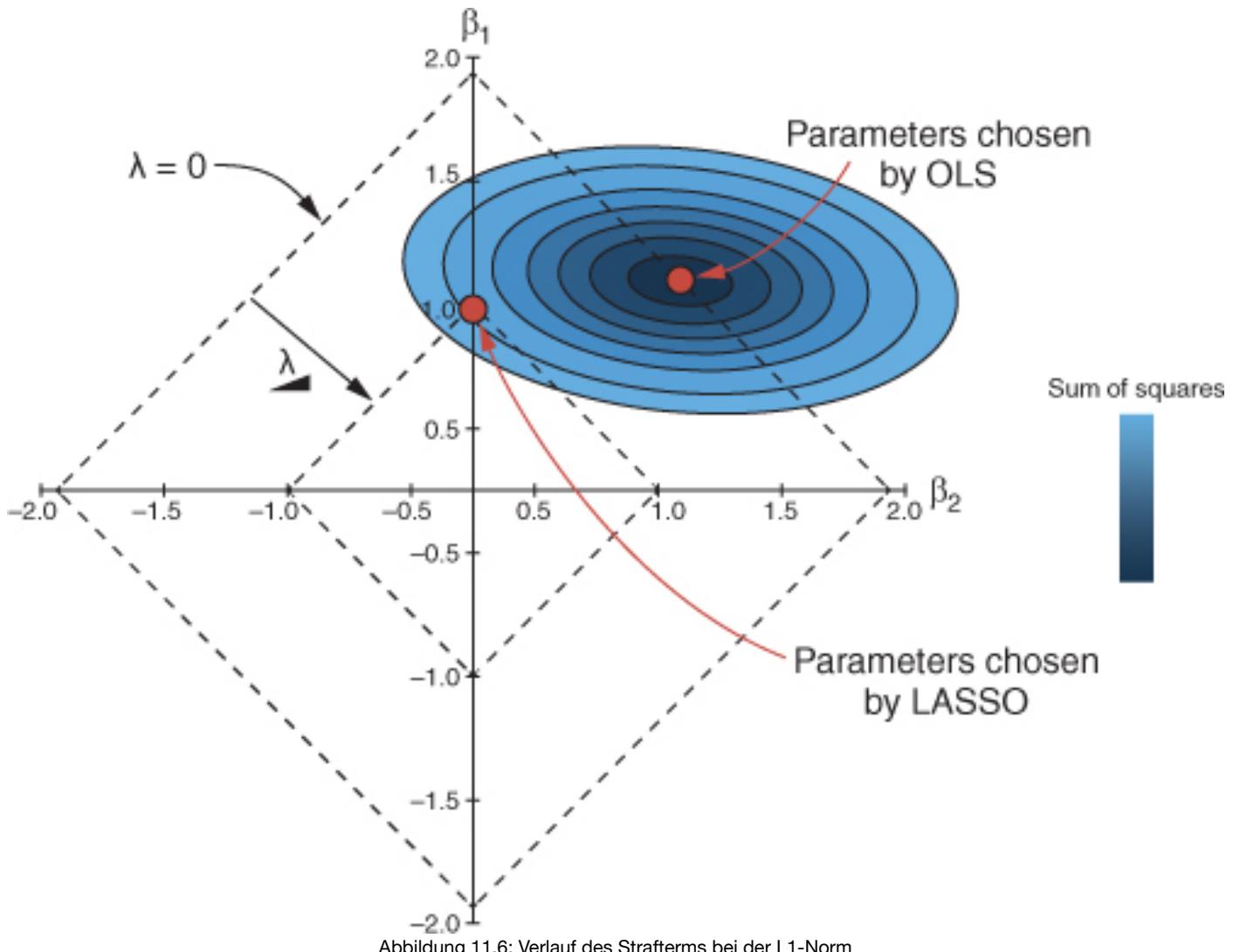


Abbildung 11.5: Verlauf des Strafterms bei der L2-Norm

Wenn lambda gleich Null ist, entspricht  $L_{L2}$  genau der OLS-Lösung. Vergrößert man lambda, so liegt  $L_{L2}$  dem Schnittpunkt des OLS-Kreises mit dem zugehörigen lambda-Kreis. Wie man sieht, führt eine Erhöhung von lambda zu einer Reduktion der Absolutwerte von  $\beta_1$  und  $\beta_2$ . Allerdings werden, wie man im Diagramm sieht, auch bei hohen lambda-Werten die Regressionskoeffizienten nicht exakt Null sein.

Warum lässt die L2-Norm für bestimmte lambda-Werte den charakteristischen Kreis entstehen? Die Antwort ist, dass die Lösungen für  $\beta_1^2 + \beta_2^2 = 1$  (mit  $\lambda = 1$ ) graphisch als Kreis dargestellt werden können.

Anders ist die Situation bei der L1-Norm, dem Lasso, vgl. Abb. 11.6 (Rhys 2020).



Eine Erhöhung von  $\lambda$  führt aufgrund der charakteristischen Kontourlinie zu einem Schnittpunkt (von OLS-Lösung und lambda-Wert), der - wenn lambda groß genug ist, stets auf einer der beiden Achsen liegt, also zu einer Nullsetzung des Parameters führt.

Damit kann man argumentieren, dass das Lasso implizit davon ausgeht, dass einige Koeffizienten in Wirklichkeit exakt Null sind, die L2-Norm aber nicht.

## 11.6 L1 vs. L2

### 11.6.1 Wer ist stärker?

Man kann nicht sagen, dass die L1- oder die L2-Norm strikt besser sei. Es kommt auf den Datensatz an. Wenn man einen Datensatz hat, in dem es einige wenige starke Prädiktoren gibt und viele sehr schwache (oder exakt irrelevante) Prädiktoren gibt, dann wird L1 tendenziell zu besseren Ergebnissen führen (James et al. 2021, S. 246). Das Lasso hat noch den Vorteil der Einfachheit, da weniger Prädiktoren im Modell verbleiben.

Ridge-Regression wird dann besser abschneiden (tendenziell), wenn die Prädiktoren etwa alle gleich stark sind.

### 11.6.2 Elastic Net als Kompromiss

Das Elastic Net (EN) ist ein Kompromiss zwischen L1- und L2-Norm.  $\lambda$  wird auf einen Wert zwischen 1 und 2 eingestellt; auch hier wird der Wert für  $\lambda$  wieder per Tuning gefunden.

$$L_{EN} = RSS + \lambda((1 - \alpha)) \cdot L2\text{-Strafterm} + \alpha \cdot L1\text{-Strafterm}$$

$\alpha$  ist ein Tuningparameter, der einstellt, wie sehr wir uns Richtung L1- vs. L2-Norm bewegen. Damit wird sozusagen die "Mischung" eingestellt (von L1- vs. L2).

Spezialfälle:

- Wenn  $\alpha = 0$  resultiert die Ridge-Regression (L1-Strafterm wird Null)
- Wenn  $\alpha = 1$  resultiert die Lasso-Regression (L2-Strafterm wird Null)

## 11.7 Aufgaben

- Fallstudie Serie The Office (<https://juliasilge.com/blog/lasso-the-office/>)
- Fallstudie NBER Papers (<https://juliasilge.com/blog/nber-papers/>)

# 12 Kaggle

## 12.1 Vorbereitung

### 12.1.1 Lernsteuerung

### 12.1.2 Lernziele

- Sie wissen, wie man einen Datensatz für einen Prognosewettbewerb bei Kaggle einreicht
- Sie kennen einige Beispiele von Notebooks auf Kaggle (für die Sprache R)
- Sie wissen, wie man ein Workflow-Set in Tidymodels berechnet
- Sie wissen, dass Tidymodels im Rezept keine Transformationen im Test-Sample berücksichtigt und wie man damit umgeht

### 12.1.3 Hinweise

- Machen Sie sich mit Kaggle vertraut. Als Übungs-Wettbewerb dient uns `TMDB Box-office Revenue` (s. Aufgaben)

### 12.1.4 R-Pakete

In diesem Kapitel werden folgende R-Pakete benötigt:

```
library(tidyverse)
library(tidymodels)
library(tictoc) # Rechenzeit messen
library(lubridate) # Datumsangaben
library(VIM) # fehlende Werte
library(visdat) # Datensatz visualisieren
```

## 12.2 Was ist Kaggle?

Kaggle, a subsidiary of Google LLC, is an online community of data scientists and machine learning practitioners. Kaggle allows users to find and publish data sets, explore and build models in a web-based data-science environment, work with other data scientists and machine learning engineers, and enter competitions to solve data science challenges.

Quelle (<https://en.wikipedia.org/wiki/Kaggle>)

Kaggle as AirBnB for Data Scientists?! (<https://www.kaggle.com/getting-started/44916>)

## 12.3 Fallstudie TMDB

Wir bearbeiten hier die Fallstudie TMDB Box Office Prediction - Can you predict a movie's worldwide box office revenue? (<https://www.kaggle.com/competitions/tmdb-box-office-prediction/overview>), ein Kaggle (<https://www.kaggle.com/>)-Prognosewettbewerb.

Ziel ist es, genaue Vorhersagen zu machen, in diesem Fall für Filme.

### 12.3.1 Aufgabe

Reichen Sie bei Kaggle eine Submission für die Fallstudie ein! Berichten Sie den Score!

### 12.3.2 Hinweise

- Sie müssen sich bei Kaggle ein Konto anlegen (kostenlos und anonym möglich); alternativ können Sie sich mit einem Google-Konto anmelden.
- Halten Sie das Modell so einfach wie möglich. Verwenden Sie als Algorithmus die lineare Regression ohne weitere Schnörkel.
- Logarithmieren Sie `budget` und `revenue`.
- Minimieren Sie die Vorverarbeitung (`steps`) so weit als möglich.
- Verwenden Sie `tidymodels`.
- Die Zielgröße ist `revenue` in Dollars; nicht in "Log-Dollars". Sie müssen also rücktransformieren, wenn Sie `revenue` logarithmiert haben.

## 12.3.3 Daten

Die Daten können Sie von der Kaggle-Projektseite beziehen oder so:

```
d_train_path <- "https://raw.githubusercontent.com/sebastiansauer/Lehre/main/data/tmdb-box-office-prediction/train.csv"
d_test_path <- "https://raw.githubusercontent.com/sebastiansauer/Lehre/main/data/tmdb-box-office-prediction/test.csv"
```

Wir importieren die Daten von der Online-Quelle:

```
d_train_raw <- read_csv(d_train_path)
d_test <- read_csv(d_test_path)
```

Mal einen Blick werfen:

```
glimpse(d_train_raw)
```

```
## Rows: 3,000
## Columns: 23
## $ id <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 1...
## $ belongs_to_collection <chr> "[{'id': 313576, 'name': 'Hot Tub Time Machine C...
## $ budget <dbl> 1.40e+07, 4.00e+07, 3.30e+06, 1.20e+06, 0.00e+00...
## $ genres <chr> "[{'id': 35, 'name': 'Comedy'}]", "[{'id': 35, '...
## $ homepage <chr> NA, NA, "http://sonyclassics.com/whiplash/", "ht...
## $ imdb_id <chr> "tt2637294", "tt0368933", "tt2582802", "tt182148...
## $ original_language <chr> "en", "en", "en", "hi", "ko", "en", "en", "en", ...
## $ original_title <chr> "Hot Tub Time Machine 2", "The Princess Diaries ...
## $ overview <chr> "When Lou, who has become the \\"father of the In...
## $ popularity <dbl> 6.575393, 8.248895, 64.299990, 3.174936, 1.14807...
## $ poster_path <chr> "/tQtWuvvMf0hCc2QR2tkolwl7c3c.jpg", "/w9Z7A0GHEh...
## $ production_companies <chr> "[{'name': 'Paramount Pictures', 'id': 4}, {'nam...
## $ production_countries <chr> "[{'iso_3166_1': 'US', 'name': 'United States of...
## $ release_date <chr> "2/20/15", "8/6/04", "10/10/14", "3/9/12", "2/5/...
## $ runtime <dbl> 93, 113, 105, 122, 118, 83, 92, 84, 100, 91, 119...
## $ spoken_languages <chr> "[{'iso_639_1': 'en', 'name': 'English'}]", "[{'...
## $ status <chr> "Released", "Released", "Released", "Released", ...
## $ tagline <chr> "The Laws of Space and Time are About to be Viol...
## $ title <chr> "Hot Tub Time Machine 2", "The Princess Diaries ...
## $ Keywords <chr> "[{'id': 4379, 'name': 'time travel'}], {'id': 96...
## $ cast <chr> "[{'cast_id': 4, 'character': 'Lou', 'credit_id': ...
## $ crew <chr> "[{'credit_id': '59ac067c92514107af02c8c8', 'dep...
## $ revenue <dbl> 12314651, 95149435, 13092000, 16000000, 3923970,...
```

```
glimpse(d_test)
```

```
## Rows: 4,398
## Columns: 22
## $ id <dbl> 3001, 3002, 3003, 3004, 3005, 3006, 3007, 3008, ...
## $ belongs_to_collection <chr> "[{'id': 34055, 'name': 'Pokémon Collection', 'p...
## $ budget <dbl> 0.00e+00, 8.80e+04, 0.00e+00, 6.80e+06, 2.00e+06...
## $ genres <chr> "[{'id': 12, 'name': 'Adventure'}, {'id': 16, 'n...
## $ homepage <chr> "http://www.pokemon.com/us/movies/movie-pokemon-...
## $ imdb_id <chr> "tt1226251", "tt0051380", "tt0118556", "tt125595...
## $ original_language <chr> "ja", "en", "en", "fr", "en", "en", "de", "en", ...
## $ original_title <chr> "ディアルガvs/パルキアvsダークライ", "Attack of t...
## $ overview <chr> "Ash and friends (this time accompanied by newco...
## $ popularity <dbl> 3.851534, 3.559789, 8.085194, 8.596012, 3.217680...
## $ poster_path <chr> "/tnftmLMemPlduW6MRyZE0ZUD19z.jpg", "/9MgBNBqlH1...
## $ production_companies <chr> NA, "[{'name': 'Woolner Brothers Pictures Inc.',..."
## $ production_countries <chr> "[{'iso_3166_1': 'JP', 'name': 'Japan'}, {'iso_3...
## $ release_date <chr> "7/14/07", "5/19/58", "5/23/97", "9/4/10", "2/11...
## $ runtime <dbl> 90, 65, 100, 130, 92, 121, 119, 77, 120, 92, 88, ...
## $ spoken_languages <chr> "[{'iso_639_1': 'en', 'name': 'English'}, {'iso_...
## $ status <chr> "Released", "Released", "Released", "Released", ...
## $ tagline <chr> "Somewhere Between Time & Space... A Legend Is B...
## $ title <chr> "Pokémon: The Rise of Darkrai", "Attack of the 5...
## $ Keywords <chr> "[{'id': 11451, 'name': 'pokémon'}, {'id': 1155...
## $ cast <chr> "[{'cast_id': 3, 'character': 'Tonio', 'credit_i...
## $ crew <chr> "[{'credit_id': '52fe44e7c3a368484e03d683', 'dep...
```

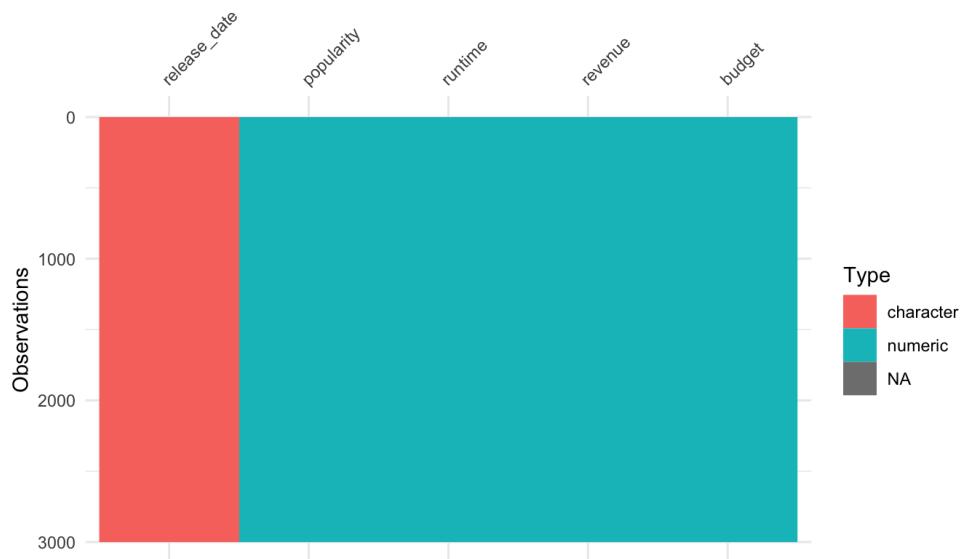
## 12.3.4 Train-Set verschlanken

Da wir aus Gründen der Einfachheit einige Spalten nicht berücksichtigen, entfernen wir diese Spalten, was die Größe des Datensatzes massiv reduziert.

```
d_train <-
d_train_raw %>%
  select(popularity, runtime, revenue, budget, release_date)
```

## 12.3.5 Datensatz kennenlernen

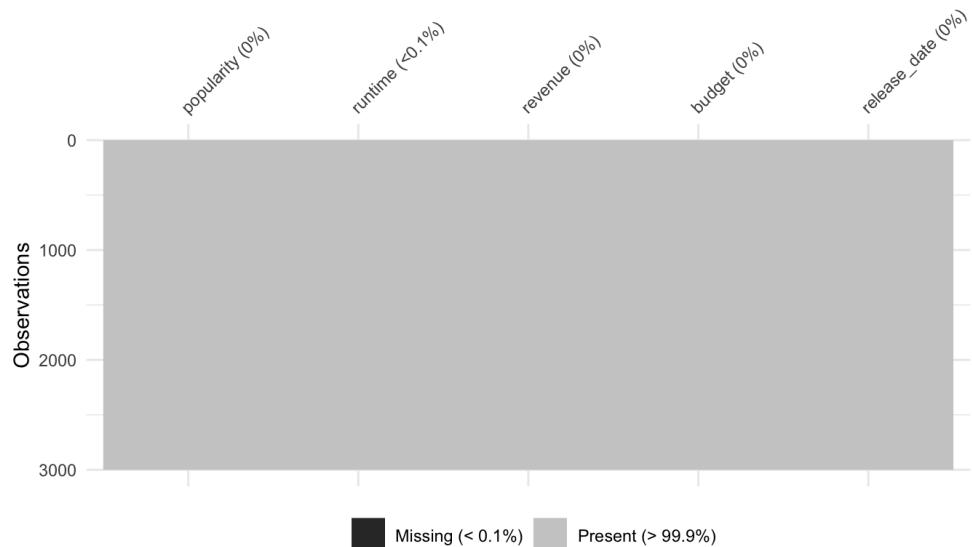
```
library(visdat)
vis_dat(d_train)
```



## 12.3.6 Fehlende Werte prüfen

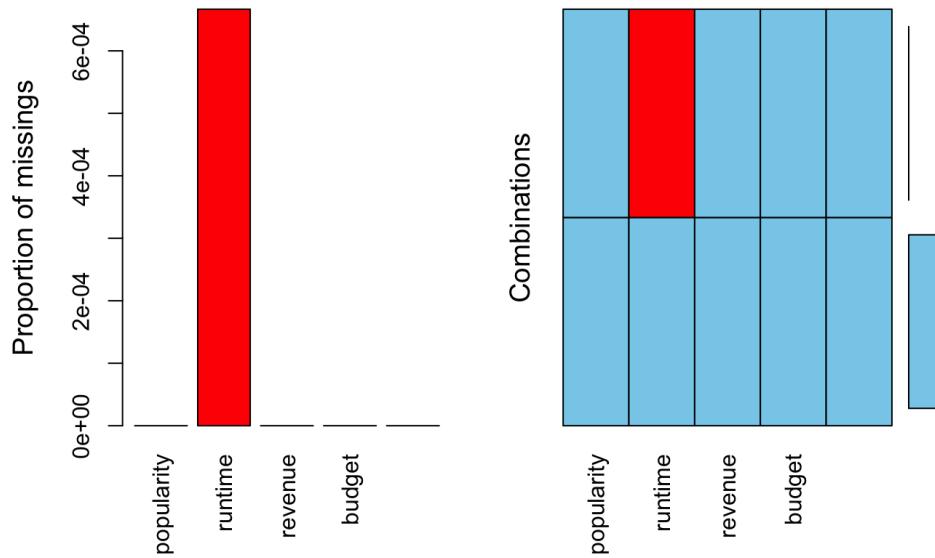
Welche Spalten haben viele fehlende Werte?

```
vis_miss(d_train)
```



Mit `{VIM}` kann man einen Datensatz gut auf fehlende Werte hin untersuchen:

```
aggr(d_train)
```



## 12.4 Rezept

### 12.4.1 Rezept definieren

```
rec1 <-
  recipe(revenue ~ ., data = d_train) %>%
  #update_role(all_predictors(), new_role = "id") %>%
  #update_role(popularity, runtime, revenue, budget, original_language) %>%
  #update_role(revenue, new_role = "outcome") %>%
  step_mutate(budget = if_else(budget < 10, 10, budget)) %>%
  step_log(budget) %>%
  step_mutate(release_date = mdy(release_date)) %>%
  step_date(release_date, features = c("year", "month"),
            keep_original_cols = FALSE) %>%
  step_impute_knn(all_predictors()) %>%
  step_dummy(all_nominal())

rec1
```

```
## Recipe
##
## Inputs:
##
##       role #variables
##   outcome          1
## predictor         4
##
## Operations:
##
## Variable mutation for if_else(budget < 10, 10, budget)
## Log transformation on budget
## Variable mutation for mdy(release_date)
## Date features from release_date
## K-nearest neighbor imputation for all_predictors()
## Dummy variables from all_nominal()
```

```
tidy(rec1)
```

```
## # A tibble: 6 × 6
##   number operation type      trained skip   id
##   <int>    <chr>    <chr>    <lgl>   <lgl> <chr>
## 1       1 step     mutate    FALSE    FALSE  mutate_f9k21
## 2       2 step     log      FALSE    FALSE  log_s63NX
## 3       3 step     mutate    FALSE    FALSE  mutate_0aThF
## 4       4 step     date     FALSE    FALSE  date_MMqil
## 5       5 step     impute_knn FALSE    FALSE  impute_knn_NH3ii
## 6       6 step     dummy    FALSE    FALSE  dummy_qKzk3
```

## 12.4.2 Check das Rezept

```
prep(rec1, verbose = TRUE)
```

```
## oper 1 step mutate [training]
## oper 2 step log [training]
## oper 3 step mutate [training]
## oper 4 step date [training]
## oper 5 step impute knn [training]
## oper 6 step dummy [training]
## The retained training set is ~ 0.38 Mb in memory.
```

```
## Recipe
##
## Inputs:
##
##       role #variables
##   outcome          1
## predictor         4
##
## Training data contained 3000 data points and 2 incomplete rows.
##
## Operations:
##
## Variable mutation for ~if_else(budget < 10, 10, budget) [trained]
## Log transformation on budget [trained]
## Variable mutation for ~mdy(release_date) [trained]
## Date features from release_date [trained]
## K-nearest neighbor imputation for runtime, budget, release_date_year, release_da... [trained]
## Dummy variables from release_date_month [trained]
```

```
prep(rec1) %>%
  bake(new_data = NULL)
```

```
## # A tibble: 3,000 × 16
##   popularity runtime budget revenue release_date_year release_date_month_Feb
##   <dbl>     <dbl>    <dbl>    <dbl>           <dbl>                <dbl>
## 1       6.58      93    16.5  12314651        2015                  1
## 2       8.25     113    17.5  95149435        2004                  0
## 3      64.3      105   15.0  13092000        2014                  0
## 4      3.17      122   14.0  16000000        2012                  0
## 5      1.15      118   2.30  3923970         2009                  1
## 6      0.743     83   15.9  3261638          1987                  0
## 7      7.29      92   16.5  85446075        2012                  0
## 8      1.95      84   2.30  2586511         2004                  0
## 9      6.90      100   2.30  34327391        1996                  1
## 10     4.67      91   15.6  18750246        2003                  0
## # ... with 2,990 more rows, and 10 more variables: release_date_month_Mar <dbl>,
## #   release_date_month_Apr <dbl>, release_date_month_May <dbl>,
## #   release_date_month_Jun <dbl>, release_date_month_Jul <dbl>,
## #   release_date_month_Aug <dbl>, release_date_month_Sep <dbl>,
## #   release_date_month_Oct <dbl>, release_date_month_Nov <dbl>,
## #   release_date_month_Dec <dbl>
```

Wir definieren eine Helper-Funktion:

```
sum_isna <- function(x) {sum(is.na(x))}
```

Und wenden diese auf jede Spalte an:

```
prep(rec1) %>%
  bake(new_data = NULL) %>%
  map_df(sum_isna)
```

```
## # A tibble: 1 × 16
##   popularity runtime budget revenue release_date_year release_date_month_Feb
##   <int>     <int>    <int>    <int>           <int>                <int>
## 1       0       0       0       0            0                  0
## # ... with 10 more variables: release_date_month_Mar <int>,
## #   release_date_month_Apr <int>, release_date_month_May <int>,
## #   release_date_month_Jun <int>, release_date_month_Jul <int>,
## #   release_date_month_Aug <int>, release_date_month_Sep <int>,
## #   release_date_month_Oct <int>, release_date_month_Nov <int>,
## #   release_date_month_Dec <int>
```

Keine fehlenden Werte mehr *in den Prädiktoren*.

Nach fehlenden Werten könnte man z.B. auch so suchen:

```
datawizard::describe_distribution(d_train)
```

## Variable	Mean	SD	IQR	Range	Skewness	Kurtosis	n	n_Missing
## popularity	8.46	12.10	6.88	[1.00e-06, 294.34]	14.38	280.10	3000	0
## runtime	107.86	22.09	24.00	[0.00, 338.00]	1.02	8.19	2998	2
## revenue	6.67e+07	1.38e+08	6.66e+07	[1.00, 1.52e+09]	4.54	27.78	3000	0
## budget	2.25e+07	3.70e+07	2.90e+07	[0.00, 3.80e+08]	3.10	13.23	3000	0

So bekommt man gleich noch ein paar Infos über die Verteilung der Variablen. Praktische Sache.

## 12.4.3 Check Test-Sample

Das Test-Sample backen wir auch mal.

Wichtig: Wir preppen den Datensatz mit dem *Train-Sample*.

```
bake(prep(rec1), new_data = d_test) %>%
  head()
```

```
## # A tibble: 6 × 15
##   popularity runtime budget release_date_year release_date_mon... release_date_mo...
##   <dbl>     <dbl>    <dbl>        <dbl>           <dbl>           <dbl>
## 1      3.85     90    2.30       2007            0             0
## 2      3.56     65   11.4       2058            0             0
## 3      8.09    100    2.30       1997            0             0
## 4      8.60    130   15.7       2010            0             0
## 5      3.22     92   14.5       2005            1             0
## 6      8.68    121    2.30       1996            1             0
## # ... with 9 more variables: release_date_month_Apr <dbl>,
## #   release_date_month_May <dbl>, release_date_month_Jun <dbl>,
## #   release_date_month_Jul <dbl>, release_date_month_Aug <dbl>,
## #   release_date_month_Sep <dbl>, release_date_month_Oct <dbl>,
## #   release_date_month_Nov <dbl>, release_date_month_Dec <dbl>
```

## 12.5 Kreuzvalidierung

```
cv_scheme <- vfold_cv(d_train,
                      v = 5,
                      repeats = 3)
```

## 12.6 Modelle

### 12.6.1 Baum

```
mod_tree <-
  decision_tree(cost_complexity = tune(),
                tree_depth = tune(),
                mode = "regression")
```

### 12.6.2 Random Forest

```
doParallel::registerDoParallel()
```

```
mod_rf <-
  rand_forest(mtry = tune(),
              min_n = tune(),
              trees = 1000,
              mode = "regression") %>%
  set_engine("ranger", num.threads = 4)
```

### 12.6.3 XGBoost

```
mod_boost <- boost_tree(mtry = tune(),
                         min_n = tune(),
                         trees = tune()) %>%
  set_engine("xgboost", nthreads = parallel::detectCores()) %>%
  set_mode("regression")
```

### 12.6.4 LM

```
mod_lm <-
  linear_reg()
```

## 12.7 Workflows

```
preproc <- list(rec1 = rec1)
models <- list(tree1 = mod_tree, rf1 = mod_rf, boost1 = mod_boost, lm1 = mod_lm)

all_workflows <- workflow_set(preproc, models)
```

## 12.8 Fitten und tunen

```
if (file.exists("objects/tmdb_model_set.rds")) {
  tmdb_model_set <- read_rds("objects/tmdb_model_set.rds")
} else {
  tic()
  tmdb_model_set <-
    all_workflows %>%
    workflow_map(
      resamples = cv_scheme,
      grid = 10,
      # metrics = metric_set(rmse),
      seed = 42, # reproducibility
      verbose = TRUE)
  toc()
}
```

Man könnte sich das Ergebnisobjekt abspeichern, um künftig Rechenzeit zu sparen:

```
write_rds(tmdb_model_set, "objects/tmdb_model_set.rds")
```

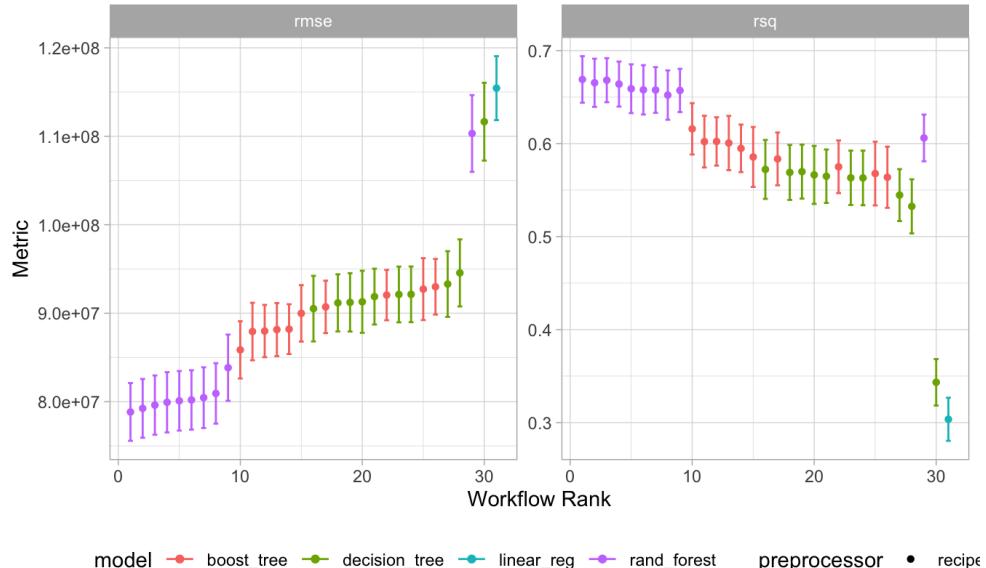
Aber Achtung: Wenn Sie vergessen, das Objekt auf der Festplatte zu aktualisieren, haben Sie eine zusätzliche Fehlerquelle. Gefahr im Verzug. Professioneller ist der Ansatz mit dem R-Paket target (<https://books.ropensci.org/targets/>).

## 12.9 Finalisieren

### 12.9.1 Welcher Algorithmus schneidet am besten ab?

Genauer geagt, welches Modell, denn es ist ja nicht nur ein Algorithmus, sondern ein Algorithmus plus ein Rezept plus die Parameterinstatiierung plus ein spezifischer Datensatz.

```
tune::autoplot(tmdb_model_set) +
  theme(legend.position = "bottom")
```



R-Quadrat ist nicht entscheidend; rmse ist wichtiger.

Die Ergebnislage ist nicht ganz klar, aber einiges spricht für das Boosting-Modell, `rec1_boost1`.

```
tmdb_model_set %>%
  collect_metrics() %>%
  arrange(-mean) %>%
  head(10)
```

```
## # A tibble: 10 × 9
##   wflow_id .config    preproc model .metric .estimator  mean     n std_err
##   <chr>     <chr>      <chr> <chr> <chr>    <dbl> <int>  <dbl>
## 1 rec1_lm1 Preprocess... recipe line... rmse  standard 1.15e8   15  2.20e6
## 2 rec1_tree1 Preprocess... recipe deci... rmse  standard 1.12e8   15  2.67e6
## 3 rec1_rf1  Preprocess... recipe rand... rmse  standard 1.10e8   15  2.64e6
## 4 rec1_tree1 Preprocess... recipe deci... rmse  standard 9.46e7   15  2.30e6
## 5 rec1_tree1 Preprocess... recipe deci... rmse  standard 9.33e7   15  2.26e6
## 6 rec1_boost1 Preprocess... recipe boos... rmse  standard 9.30e7   15  1.91e6
## 7 rec1_boost1 Preprocess... recipe boos... rmse  standard 9.27e7   15  2.13e6
## 8 rec1_tree1 Preprocess... recipe deci... rmse  standard 9.21e7   15  1.91e6
## 9 rec1_tree1 Preprocess... recipe deci... rmse  standard 9.21e7   15  1.91e6
## 10 rec1_boost1 Preprocess... recipe boos... rmse standard 9.21e7  15  1.73e6
```

```
best_model_params <-
extract_workflow_set_result(tmdb_model_set, "rec1_boost1") %>%
  select_best()

best_model_params
```

```
## # A tibble: 1 × 4
##   mtry trees min_n .config
##   <int> <int> <int> <chr>
## 1     6    100     4 Preprocessor1_Model04
```

```
best_wf <-
all_workflows %>%
  extract_workflow("rec1_boost1")

#best_wf
```

```
best_wf_finalized <-
  best_wf %>%
  finalize_workflow(best_model_params)

best_wf_finalized
```

```
## == Workflow ==
## Preprocessor: Recipe
## Model: boost_tree()
##
## — Preprocessor —
## 6 Recipe Steps
##
## • step_mutate()
## • step_log()
## • step_mutate()
## • step_date()
## • step_impute_knn()
## • step_dummy()
##
## — Model —
## Boosted Tree Model Specification (regression)
##
## Main Arguments:
##   mtry = 6
##   trees = 100
##   min_n = 4
##
## Engine-Specific Arguments:
##   nthreads = parallel::detectCores()
##
## Computational engine: xgboost
```

## 12.9.2 Final Fit

```
fit_final <-
best_wf_finalized %>%
  fit(d_train)
```

```
## [18:18:00] WARNING: amalgamation/../src/learner.cc:627:
## Parameters: { "nthreads" } might not be used.
##
## This could be a false alarm, with some parameters getting used by language bindings but
## then being mistakenly passed down to XGBoost core, or some parameter actually being used
## but getting flagged wrongly here. Please open an issue if you find any such cases.
```

```
fit_final
```

```
## == Workflow [trained] ==
## Preprocessor: Recipe
## Model: boost_tree()
##
## — Preprocessor ——————
## 6 Recipe Steps
##
## • step_mutate()
## • step_log()
## • step_mutate()
## • step_date()
## • step_impute_knn()
## • step_dummy()
##
## — Model ——————
## ##### xgb.Booster
## raw: 269.9 Kb
## call:
##   xgboost::xgb.train(params = list(eta = 0.3, max_depth = 6, gamma = 0,
##     colsample_bytree = 1, colsample_bynode = 0.4, min_child_weight = 4L,
##     subsample = 1, objective = "reg:squarederror"), data = x$data,
##     nrounds = 100L, watchlist = x$watchlist, verbose = 0, nthreads = 8L,
##     nthread = 1)
##   params (as set within xgb.train):
##     eta = "0.3", max_depth = "6", gamma = "0", colsample_bytree = "1",
##     colsample_bynode = "0.4", min_child_weight = "4", subsample = "1",
##     objective = "reg:squarederror", nrounds = "100", nthread = "1",
##     validate_parameters = "TRUE"
##   xgb.attributes:
##     niter
##   callbacks:
##     cb.evaluation.log()
##   # of features: 15
##   niter: 100
##   nfeatures : 15
##   evaluation_log:
##     iter training_rmse
##       1      120931645
##       2      101209046
##   ---
##       99      26964007
##      100      26851702
```

```
d_test$revenue <- NA

final_preds <-
  fit_final %>%
  predict(new_data = d_test) %>%
  bind_cols(d_test)
```

## 12.10 Submission

### 12.10.1 Submission vorbereiten

```
submission_df <-  
final_preds %>%  
select(id, revenue = .pred)
```

Abspeichern und einreichen:

```
write_csv(submission_df, file = "objects/submission.csv")
```

Diese CSV-Datei reichen wir dann bei Kaggle ein.

## 12.10.2 Kaggle Score

Diese Submission erzielte einen Score von **4.79227** (RMSLE).

## 12.11 Aufgaben

- Arbeiten Sie sich so gut als möglich durch diese Analyse zum Verlauf von Covid-Fällen (<https://github.com/sebastiansauer/covid-icu>)
- Fallstudie zur Modellierung einer logististischen Regression mit tidymodels (<https://onezero.blog/modelling-binary-logistic-regression-using-tidymodels-library-in-r-part-1/>)
- Fallstudie zu Vulkanausbrüchen (<https://juliasilge.com/blog/multinomial-volcano-eruptions/>)
- Fallstudie Himalaya (<https://juliasilge.com/blog/himalayan-climbing/>)

## 12.12 Vertiefung

- Fields arranged by purity, xkcd 435 (<https://xkcd.com/435/>)

# 13 Der rote Faden

Mittlerweile haben wir einiges zum Thema Data Science bzw. maschinelles Lernen behandelt (und sie hoffentlich viel gelernt).

Da ist es an der Zeit, einen Schritt zurück zu treten, um sich einen Überblick über den gegangenen Weg zu verschaffen, den berühmten “roten Faden” zu sehen, den zurückgelegten Weg nachzuzeichnen in den groben Linien, um einen (klarer) Überblick über das Terrain zu bekommen.

In diesem Kapitel werden wir verschiedene “Aussichtspfade” suchen, um im Bild zu bleiben, die uns einen Überblick über das Gelände versprechen.

### 13.0.1 Lernziele

- Sie erarbeiten sich einen Überblick über den bisher gelernten Stoff bzw. verfeinern Ihren bestehenden Überblick

### 13.0.2 Literatur

- Rhys im Überblick

## 13.1 Aussichtspunkt 1: Blick vom hohen Berg

Und so zeigt sich ein “Flussbild”<sup>9</sup> (Abb. 13.1).

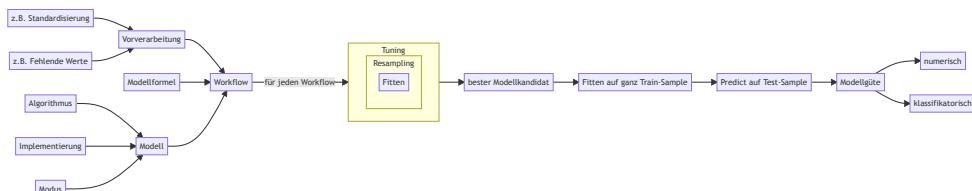


Abbildung 13.1: Ein Flussbild des maschinellen Lernens

Der Reiseführer erzählt uns zu diesem Bild folgende Geschichte:

Video-Geschichte (<https://youtu.be/PO-Urf5hGtY>)

## 13.2 Aussichtspunkt 2: Blick in den Hof der Handwerker

Wenn man auf einem hohen Berg gestanden ist, hat man zwar einen guten Überblick über das Land bekommen, aber das konkrete Tun bleibt auf solchen Höhen verborgen.

Möchte man wissen, wie das geschäftige Leben abläuft, muss man also den tätigen Menschen über die Schulter schauen. Werfen wir also einen Blick in den "Hof der Handwerker", wo grundlegende Werkstücke gefertigt werden, und wir jeden Handgriff aus der Nähe mitverfolgen können.

### 13.2.1 Ein maximale einfaches Werkstück mit Tidymodels

Weniger blumig ausgedrückt: Schauen wir uns ein maximal einfaches Beispiel an, wie man mit Tidymodels Vorhersagen tätigt. Genauer gesagt bearbeiten wir einen sehr einfachen Ansatz (<https://www.kaggle.com/code/ssauer/simple-linear-model-tidymodels>) für einen Kaggle-Prognosewettbewerb.

### 13.2.2 Ein immer noch recht einfaches Werkstück mit Tidymodels

Dieses Beispiel ist nur wenig aufwändiger als das vorherige.

## 13.3 Aussichtspunkt 3: Der Nebelberg (Quiz)

Da der "Nebelberg" zumeist in Wolken verhüllt ist, muss man, wenn man ihn ersteigt und ins Land hinunterschaut, erraten, welche Teile zu sehen sind. Sozusagen eine Art Landschafts-Quiz.

Voilà, hier ist es, das Quiz zum maschinellen Lernen:

## Data-Science-Quiz

In Google anmelden, um den Fortschritt zu speichern. [Weitere Informationen](#)

Decision Trees (Baummodelle) sind Overfitting (Überanpassung) 1 Punkt  
mehr ausgesetzt als lineare Modelle.

- Richtig
- Falsch

Ein Resampling-Schema mit  $v=10$  Faltungen und  $r=5$  Wiederholungen ist identisch zu einem Schema mit  $v=50$  Faltungen und  $r=1$  Wiederholungen. 1 Punkt

- Richtig
- Falsch

"Normale" (nicht regularisierte) lineare Modelle sind besser interpretierbar als L1-regularisierte lineare Modelle. 1 Punkt

- Richtig
- Falsch

"Normale" lineare Modelle verfügen nicht über Tuningparameter. 1 Punkt

- Richtig
- Falsch

## 13.4 Aussichtspunkt 4: Der Exerzitien-Park

Wir stehen vor dem Eingang zu einem Park, in dem sich viele Menschen an merkwürdigen Übungen, Exerzitien, befleißigen. Vielleicht wollen Sie sich auch an einigen Übungen abhärten? Bitte schön, lassen Sie sich nicht von mir aufhalten.

YACSDA: Yet Another Case Study on Data Analysis

...

### NUR EXPLORATIVE DATENANALYSE

- Datenjudo mit Pinguinen (<https://allisonhorst.shinyapps.io/dplyr-learnr/#section-welcome>)
- Data-Wrangling-Aufgaben zur Lebenserwartung (<https://data-se.netlify.app/2021/02/24/exercises-to-data-wrangling-with-the-tidyverse/>)
- Aufgabe zur Datenvisualisierung des Diamantenpreises (<https://data-se.netlify.app/2020/12/07/ex-visualizing-diamonds/>)
- Fallstudie Flugverspätungen - EDA (<https://data-se.netlify.app/2021/03/08/eda-zu-flugversp%C3%A4tungen/>)
- Fallstudie zur EDA: Top-Gear (<https://data-se.netlify.app/2021/02/11/yacda-topgear/>)

- Fallstudie zur EDA: OECD-Wellbeing-Studie (<https://data-se.netlify.app/2021/02/11/explorative-datenanalyse-zum-datensatz-oecd-wellbeing/>)
- Fallstudie zur EDA: Movie Rating (<https://minimaxir.com/2018/07/imdb-data-analysis/>)
- Fallstudie zur EDA: Women in Parliament (<https://github.com/saghirb/WiP-tidyverse/blob/master/doc/WiP-tidyverse.pdf>)
- Finde den Tag mit den meisten Flugverspätungen, Datensatz ‘nycflights13’ (<https://data-se.netlify.app/2021/05/27/datensatz-flights-finde-den-tag-mit-den-meisten-abfl%C3%BCgen/>)

## NUR LINEARE MODELL

- Beispiel für Prognosemodellierung 1, grundlegender Anspruch, Video (<https://youtu.be/5pBTHrnRIZY>)
- Beispiel für Ihre Prognosemodellierung 2, mittlerer Anspruch (<https://data-se.netlify.app/2020/11/13/fallstudie-zur-regressionsanalyse-ggplot2movies/>)
- Beispiel für Ihre Prognosemodellierung 3, hoher Anspruch (<https://data-se.netlify.app/2021/03/10/fallstudie-modellierung-von-flugversp%C3%A4tungen/>)
- Fallstudie: Modellierung von Flugverspätungen (<https://data-se.netlify.app/2021/03/10/fallstudie-modellierung-von-flugversp%C3%A4tungen/>)
- Movies (<https://data-se.netlify.app/2020/11/13/fallstudie-zur-regressionsanalyse-ggplot2movies/>)
- Fallstudie Einfache lineare Regression in Base-R, Anfängerniveau, Kaggle-Competition TMDB (<https://www.kaggle.com/code/ssauer/tmdb-simple-regression-beginners>)
- Fallstudie Sprit sparen (<https://data-se.netlify.app/2022/05/02/fallstudie-spritverbrauch/>)

## YouTube-PLAYLISTS

- Playlist YACSDAs ([https://youtube.com/playlist?list=PLRR4REmBgplGet\\_IcNf2wOd0W8j4c9hYN](https://youtube.com/playlist?list=PLRR4REmBgplGet_IcNf2wOd0W8j4c9hYN))
- Playlist zur Prüfungsleistung Prognosewettbewerb (<https://youtube.com/playlist?list=PLRR4REmBgplH6uG8LZWPTSMReX1OFxfUx>)
- Kaggle-Fallstudie TMDB: einfache lineare Regression (<https://youtu.be/vR9l-k50I1M>)
- Playlist zum statistischen Modellieren (<https://www.youtube.com/playlist?list=PLRR4REmBgplGWcSjrtt0m36aXHLaiTgdF>)

## MASCHINELLES LERNEN MIT TIDYMODELS

- Experimenting with machine learning in R with tidymodels and the Kaggle titanic dataset (<https://www.r-bloggers.com/2021/08/experimenting-with-machine-learning-in-r-with-tidymodels-and-the-kaggle-titanic-dataset/>)
- Tutorial on tidymodels for Machine Learning (<https://hansjoerg.me/2020/02/09/tidymodels-for-machine-learning/>)
- Classification with Tidymodels, Workflows and Recipes (<https://www.kirenz.com/post/2021-02-17-r-classification-tidymodels/>)
- A (mostly!) tidyverse tour of the Titanic (<https://www.kaggle.com/code/varimp/a-mostly-tidyverse-tour-of-the-titanic/report>)
- Personalised Medicine - EDA with tidy R (<https://www.kaggle.com/code/headsortails/personalised-medicine-eda-with-tidy-r/report>)
- Tidy TitaRnic (<https://www.kaggle.com/code/headsortails/tidy-titanic/report>)
- Fallstudie Seegurken (<https://www.tidymodels.org/start/models/>)
- Sehr einfache Fallstudie zur Modellierung einer Regression mit tidymodels (<https://juliasilge.com/blog/student-debt/>)
- Fallstudie zur linearen Regression mit Tidymodels (<https://www.gmudatamining.com/lesson-10-r-tutorial.html>)
- Analyse zum Verlauf von Covid-Fällen (<https://github.com/sebastiansauer/covid-icu>)
- Fallstudie zur Modellierung einer logistischen Regression mit tidymodels (<https://onezero.blog/modelling-binary-logistic-regression-using-tidymodels-library-in-r-part-1/>)
- Fallstudie zu Vulkanausbrüchen (<https://juliasilge.com/blog/multinomial-volcano-eruptions/>)
- Fallstudie Himalaya (<https://juliasilge.com/blog/himalayan-climbing/>)
- Fallstudien zu Studiengebühren (<https://juliasilge.com/blog/tuition-resampling/>)
- 1. Modell der Fallstudie Hotel Bookings (<https://www.tidymodels.org/start/case-study/>)
- Aufgaben zur logistischen Regression, PDF (<https://github.com/sebastiansauer/datascience1/blob/main/Aufgaben/Thema8-Loesungen1.pdf>)
- Fallstudie Oregon Schools (<https://bcullen.rbind.io/post/2020-06-02-tidymodels-decision-tree-learning-in-r/>)
- Fallstudie Windturbinen (<https://juliasilge.com/blog/wind-turbine/>)
- Fallstudie Churn (<https://www.gmudatamining.com/lesson-13-r-tutorial.html>)
- Einfache Durchführung eines Modellierungen mit XGBoost (<https://data-se.netlify.app/2020/12/14/titanic-tidymodels-boost/>)
- Fallstudie Oregon Schools (<https://bcullen.rbind.io/post/2020-06-02-tidymodels-decision-tree-learning-in-r/>)

- Fallstudie Churn (<https://www.gmudatamining.com/lesson-13-r-tutorial.html>)
- Fallstudie Ikea (<https://juliasilge.com/blog/ikea-prices/>)
- Fallstudie Wasserquellen in Sierra Leone (<https://juliasilge.com/blog/water-sources/>)
- Fallstudie Bäume in San Francisco (<https://dev.to/juliasilge/tuning-random-forest-hyperparameters-in-r-with-tidyTuesday-trees-data-4lh>)
- Fallstudie Vulkanausbrüche (<https://juliasilge.com/blog/multinomial-volcano-eruptions/>)
- Fallstudie Brettspiele mit XGBoost (<https://juliasilge.com/blog/board-games/>)
- Fallstudie Serie The Office (<https://juliasilge.com/blog/lasso-the-office/>)
- Fallstudie NBER Papers (<https://juliasilge.com/blog/nber-papers/>)
- Fallstudie Einfache lineare Regression mit Tidymodels, Kaggle-Competition TMDB (<https://www.kaggle.com/ssauer/simple-linear-model-tidymodels>)
- Fallstudie Einfaches Random-Forest-Modell mit Tidymodels, Kaggle-Competition TMDB (<https://www.kaggle.com/code/ssauer/simple-rf-tuned>)
- Fallstudie Workflow-Set mit Tidymodels, Kaggle-Competition TMDB (<https://www.kaggle.com/ssauer/tmdb-xgboost-tidymodels>)
- Fallstudie Titanic mit Tidymodels bei Kaggle (<https://www.kaggle.com/code/modesty520/a-tutorial-with-tidymodels/report#modeling>)
- Einfache Fallstudie mit Tidymodels bei Kaggle (<https://www.kaggle.com/code/benthecoder/tidymodels-in-r-using-measles-data/notebook>)

## 13.5 Aussichtspunkt 5: In der Bibliothek

Einen Überblick über eine Landschaft gewinnt man nicht nur von ausgesetzten Wegpunkten aus, sondern auch, manchmal, aus Schriftstücken. Hier ist eine Auswahl an Literatur, die Grundlagen zu unserem Landstrich erläutert.

- Rhys (2020)
- Silge and Kuhn (2022)

Etwas weiter leiten uns diese Erzähler:

- James et al. (2021)
- Kuhn and Johnson (2013)

## 13.6 Krafttraining

Um die Aussicht genießen zu können, muss man manchmal ausgesetzte Plätze in ~~schwindelerregenden einigermaßen steilen~~ als Hügel erkennbaren Höhen erreichen...

Sportliche Leistungen erreicht nur, wer trainiert ist. Das ist im Land des Data Science nicht anders.

Hier ist eine Liste von Übungen, die Ihre Datenkraft stählen soll:

1. **Lerngruppe:** Den Wert einer Lerngruppe kann man kaum unterschätzen. Die Motivation, der Austausch, der Zwang seine Gedanken geordnet darzustellen, das wechselseitige Abfragen - diese Dinge machen eine Lerngruppe zu einem der wichtigsten Erfolgsgarant in Ihren Lernbemühungen.
2. **Exzerpte:** Exzerpte, Zusammenfassungen also, sind nötig, um von einer vermeintlichen "Jaja, easy, versthe ich alles" Oberflächen-Verarbeitung zu einem (ausgeprägterem) Tiefenverständnis vorzudringen.
3. **Aufgaben:** Manchmal stellt ein Dozent Aufgaben ein. Die Chance sollte man nutzen, denn zwar ist vieles in der Didaktikforschung noch unsicher, aber dass Aufgaben lösen beim Lernen hilft, und zwar viel, ist eines der wenigen unstrittigen Erkenntnisse.
4. **Fallstudien:** Ähnliches wie Aufgaben, die oft kleinteilig-akademisch angelegt sind, hilft die große Schwester der schnöden Aufgabe, die Fallstudie, beim Vordringen in Verständnistiefen.
5. **Lesen:** Ja, Lesen ist voll Old School. Aber so was Ähnliches wie Updaten der Brain-Software. Nützlich, weil die alte Software irgendwann nicht mehr supported wird.
6. **Forum:** Sie haben eine Frage, aber Sie können unmöglich ein paar Tage warten, bis Sie den Dozenten im Unterricht sprechen? Posten Sie die Frage in einem Forum! Vielleicht im Forum des Moduls oder aber in einem geeigneten Forum im Internet.
7. **Youtube:** Zwar wettern Dozenten gerne über die mangelnde Verarbeitungstiefe beim Fern schauen. Außerdem sind Lehrvideos didaktisch echt asbachuralt. Aber okay, manchmal und in überschaubarer Dosis ist ein Lehrvideo eine nützliche Ergänzung zu den übrigen Maßnahmen.

## 13.7 Aufgaben

- Einfache Random-Forest-Modellierung bei Kaggle (TMDB) (<https://www.kaggle.com/code/ssauer/simple-rf-tuned>)
- Einfache Workflow-Set-Modellierung bei Kaggle (TMDB) (<https://www.kaggle.com/code/ssauer/tmdb-xgboost-tidymodels>)
- Bearbeiten Sie so viele Fallstudien der Fallstudiensammlung wie nötig, um den Stoff flüssig zu beherrschen

## 13.8 Vertiefung

- Mathematische Grundlagen können Sie z.B. hier vertiefen (<https://deisenroth.cc/publication/deisenroth-2020/>)
- Gute Fallstudie bei Kaggle für Regressionsprobleme: House Prices (<https://www.kaggle.com/competitions/house-prices-advanced-regression-techniques>)
- Sie möchten schnell ein Code-Schnipsel (öffentlich sichtbar) teilen? Probieren Sie Github Gists aus (<https://gist.github.com/>)

# 14 Fallstudien

## 14.0.1 Lernziele

- Sie können die Techniken des Maschinellen Lernens mit dem Tidymodels-Ansatz flüssig anbringen

## 14.0.2 Literatur

- Rhys, Kap. 12

## 14.1 Fallstudien zur explorativen Datenanalyse

### FALLSTUDIEN - NUR EXPLORATIVE DATENANALYSE

- Datenjudo mit Pinguinen (<https://allisonhorst.shinyapps.io/dplyr-learnr/#section-welcome>)
- Data-Wrangling-Aufgaben zur Lebenserwartung (<https://data-se.netlify.app/2021/02/24/exercises-to-data-wrangling-with-the-tidyverse/>)
- Aufgabe zur Datenvisualisierung des Diamantenpreises (<https://data-se.netlify.app/2020/12/07/ex-visualizing-diamonds/>)
- Fallstudie Flugverspätungen - EDA (<https://data-se.netlify.app/2021/03/08/eda-zu-flugversp%C3%A4tungen/>)
- Fallstudie zur EDA: Top-Gear (<https://data-se.netlify.app/2021/02/11/yacda-topgear/>)
- Fallstudie zur EDA: OECD-Wellbeing-Studie (<https://data-se.netlify.app/2021/02/11/explorative-datenanalyse-zum-datensatz-oecd-wellbeing/>)
- Fallstudie zur EDA: Movie Rating (<https://minimaxir.com/2018/07/imdb-data-analysis/>)
- Fallstudie zur EDA: Women in Parliament (<https://github.com/saghirb/WiP-tidyverse/blob/master/doc/WiP-tidyverse.pdf>)
- Finde den Tag mit den meisten Flugverspätungen, Datensatz 'nycflights13' (<https://data-se.netlify.app/2021/05/27/datensatz-flights-findest-den-tag-mit-den-meisten-abfl%C3%BCgen/>)

## 14.2 Fallstudien zu linearen Modellen

### FALLSTUDIEN - NUR LINEARE MODELLE

- Beispiel für Prognosemodellierung 1, grundlegender Anspruch, Video (<https://youtu.be/5pBTHrnRIZY>)
- Beispiel für Ihre Prognosemodellierung 2, mittlerer Anspruch (<https://data-se.netlify.app/2020/11/13/fallstudie-zur-regressionsanalyse-ggplot2movies/>)
- Beispiel für Ihre Prognosemodellierung 3, hoher Anspruch (<https://data-se.netlify.app/2021/03/10/fallstudie-modellierung-von-flugversp%C3%A4tungen/>)
- Fallstudie: Modellierung von Flugverspätungen (<https://data-se.netlify.app/2021/03/10/fallstudie-modellierung-von-flugversp%C3%A4tungen/>)
- Movies (<https://data-se.netlify.app/2020/11/13/fallstudie-zur-regressionsanalyse-ggplot2movies/>)
- Fallstudie Einfache lineare Regression in Base-R, Anfängerniveau, Kaggle-Competition TMDB (<https://www.kaggle.com/code/ssauer/tmdb-simple-regression-beginners>)
- Fallstudie Sprit sparen (<https://data-se.netlify.app/2022/05/02/fallstudie-spritverbrauch/>)

## 14.3 Fallstudien zum maschinellen Lernen mit Tidymodels

### FALLSTUDIEN - MASCHINELLES LERNEN MIT TIDYMODELS

- Experimenting with machine learning in R with tidymodels and the Kaggle titanic dataset (<https://www.r-bloggers.com/2021/08/experimenting-with-machine-learning-in-r-with-tidymodels-and-the-kaggle-titanic-dataset/>)
- Tutorial on tidymodels for Machine Learning (<https://hansjoerg.me/2020/02/09/tidymodels-for-machine-learning/>)
- Classification with Tidymodels, Workflows and Recipes (<https://www.kirenz.com/post/2021-02-17-r-classification-tidymodels/>)
- A (mostly!) tidyverse tour of the Titanic (<https://www.kaggle.com/code/varimp/a-mostly-tidyverse-tour-of-the-titanic/report>)
- Personalised Medicine - EDA with tidy R (<https://www.kaggle.com/code/headsortails/personalised-medicine-eda-with-tidy-r/report>)

- Tidy TitaRnic (<https://www.kaggle.com/code/headsoretails/tidy-titarnic/report>)
- Fallstudie Seegurken (<https://www.tidymodels.org/start/models/>)
- Sehr einfache Fallstudie zur Modellierung einer Regression mit tidymodels (<https://juliasilge.com/blog/student-debt/>)
- Fallstudie zur linearen Regression mit Tidymodels (<https://www.gmudatamining.com/lesson-10-r-tutorial.html>)
- Analyse zum Verlauf von Covid-Fällen (<https://github.com/sebastiansauer/covid-icu>)
- Fallstudie zur Modellierung einer logististischen Regression mit tidymodels (<https://onezero.blog/modelling-binary-logistic-regression-using-tidymodels-library-in-r-part-1/>)
- Fallstudie zu Vulkanausbrüchen (<https://juliasilge.com/blog/multinomial-volcano-eruptions/>)
- Fallstudie Himalaya (<https://juliasilge.com/blog/himalayan-climbing/>)
- Fallstudien zu Studiengebühren (<https://juliasilge.com/blog/tuition-resampling/>)
- 1. Modell der Fallstudie Hotel Bookings (<https://www.tidymodels.org/start/case-study/>)
- Aufgaben zur logistischen Regression, PDF (<https://github.com/sebastiansauer/datasience1/blob/main/Aufgaben/Thema8-Loesungen1.pdf>)
- Fallstudie Oregon Schools (<https://bcullen.rbind.io/post/2020-06-02-tidymodels-decision-tree-learning-in-r/>)
- Fallstudie Windturbinen (<https://juliasilge.com/blog/wind-turbine/>)
- Fallstudie Churn (<https://www.gmudatamining.com/lesson-13-r-tutorial.html>)
- Einfache Durchführung eines Modellierung mit XGBoost (<https://data-se.netlify.app/2020/12/14/titanic-tidymodels-boost/>)
- Fallstudie Oregon Schools (<https://bcullen.rbind.io/post/2020-06-02-tidymodels-decision-tree-learning-in-r/>)
- Fallstudie Churn (<https://www.gmudatamining.com/lesson-13-r-tutorial.html>)
- Fallstudie Ikea (<https://juliasilge.com/blog/ikea-prices/>)
- Fallstudie Wasserquellen in Sierra Leone (<https://juliasilge.com/blog/water-sources/>)
- Fallstudie Bäume in San Francisco (<https://dev.to/juliasilge/tuning-random-forest-hyperparameters-in-r-with-tidytuesday-trees-data-4ih>)
- Fallstudie Vulkanausbrüche (<https://juliasilge.com/blog/multinomial-volcano-eruptions/>)
- Fallstudie Brettspiele mit XGBoost (<https://juliasilge.com/blog/board-games/>)
- Fallstudie Serie The Office (<https://juliasilge.com/blog/lasso-the-office/>)
- Fallstudie NBER Papers (<https://juliasilge.com/blog/nber-papers/>)
- Fallstudie Einfache lineare Regression mit Tidymodels, Kaggle-Competition TMDB (<https://www.kaggle.com/ssauer/simple-linear-model-tidymodels>)
- Fallstudie Einfaches Random-Forest-Modell mit Tidymodels, Kaggle-Competition TMDB (<https://www.kaggle.com/code/ssauer/simple-rf-tuned>)
- Fallstudie Workflow-Set mit Tidymodels, Kaggle-Competition TMDB (<https://www.kaggle.com/ssauer/tmdb-xgboost-tidymodels>)
- Fallstudie Titanic mit Tidymodels bei Kaggle (<https://www.kaggle.com/code/modesty520/a-tutorial-with-tidymodels/report#modeling>)
- Einfache Fallstudie mit Tidymodels bei Kaggle (<https://www.kaggle.com/code/benthecoder/tidymodels-in-r-using-measles-data/notebook>)

## 14.4 Aufgaben

- Bearbeiten Sie eine Auswahl von Fallstudien Ihrer Wahl aus dieser Sammlung (<https://sebastiansauer.github.io/Lehre/Material/yacsdas.html>)

## 14.5 Vertiefung

- Wie man eine Data-Science-Projekt strukturiert (<https://medium.com/swlh/how-to-structure-a-python-based-data-science-project-a-short-tutorial-for-beginners-7e00bff14f56>)
- Hausmeisterarbeit mit {{janitor}} (<https://albert-rapp.de/post/2022-01-12-janitor-showcase/>)

## References

- Baumer, Benjamin S., Daniel T. Kaplan, and Nicholas J. Horton. 2017. *Modern Data Science with r* (Chapman & Hall/CRC Texts in Statistical Science). Boca Raton, Florida: Chapman; Hall/CRC.
- Chen, Tianqi, and Carlos Guestrin. 2016. “XGBoost: A Scalable Tree Boosting System.” In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 785–94. KDD ’16. New York, NY, USA: Association for Computing Machinery.

- <https://doi.org/10.1145/2939672.2939785> (<https://doi.org/10.1145/2939672.2939785>).
- Friedman, J. 2001. "Greedy Function Approximation: A Gradient Boosting Machine." <https://doi.org/10.1214/AOS/1013203451> (<https://doi.org/10.1214/AOS/1013203451>).
- Hvitfeldt, Emil. 2022. *ISLR Tidymodels Labs*. <https://emilhvitfeldt.github.io/ISLR-tidymodels-labs/index.html> (<https://emilhvitfeldt.github.io/ISLR-tidymodels-labs/index.html>).
- James, Gareth, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2021. *An Introduction to Statistical Learning: With Applications in r*. Second edition. Springer Texts in Statistics. New York: Springer. <https://link.springer.com/book/10.1007/978-1-0716-1418-1> (<https://link.springer.com/book/10.1007/978-1-0716-1418-1>).
- Kuhn, Max, and Kjell Johnson. 2013. *Applied Predictive Modeling*. Vol. 26. Springer.
- Rhys, Hefin. 2020. *Machine Learning with r, the Tidyverse, and Mlr*. Shelter Island, NY: Manning publications.
- Sauer, Sebastian. 2019. *Moderne Datenanalyse Mit r: Daten Einlesen, Aufbereiten, Visualisieren Und Modellieren*. 1. Auflage 2019. FOM-Edition. Wiesbaden: Springer. <https://www.springer.com/de/book/9783658215866> (<https://www.springer.com/de/book/9783658215866>).
- Silge, Julia, and Max Kuhn. 2022. *Tidy Modeling with R*. <https://www.tmwr.org/> (<https://www.tmwr.org/>).
- Spurzem, Lothar. 2017. *VW 1303 von Wiking in 1:87*. [https://de.wikipedia.org/wiki/Modellautomobil#/media/File:Wiking-Modell\\_VW\\_1303\\_\(um\\_1975\).JPG](https://de.wikipedia.org/wiki/Modellautomobil#/media/File:Wiking-Modell_VW_1303_(um_1975).JPG) ([https://de.wikipedia.org/wiki/Modellautomobil#/media/File:Wiking-Modell\\_VW\\_1303\\_\(um\\_1975\).JPG](https://de.wikipedia.org/wiki/Modellautomobil#/media/File:Wiking-Modell_VW_1303_(um_1975).JPG)).
- Taleb, Nassim Nicholas. 2019. *The Statistical Consequences of Fat Tails, Papers and Commentaries*. Monograph. <https://nassimtaleb.org/2020/01/final-version-fat-tails/> (<https://nassimtaleb.org/2020/01/final-version-fat-tails/>).
- Timbers, Tiffany-Anne, Trevor Campbell, and Melissa Lee. 2022. *Data Science: An Introduction*. First edition. Statistics. Boca Raton: CRC Press.
- Wickham, Hadley, and Garrett Grolemund. 2016. *R for Data Science: Visualize, Model, Transform, Tidy, and Import Data*. O'Reilly Media. <https://r4ds.had.co.nz/index.html> (<https://r4ds.had.co.nz/index.html>).

1. <https://link.springer.com/book/10.1007/978-3-658-21587-3> (<https://link.springer.com/book/10.1007/978-3-658-21587-3>) ↵
2. <https://www.tmwr.org/> (<https://www.tmwr.org/>) ↵
3. Das klappt bei randlastigen Verteilungen nicht ↵
4. Übrigens gehört zu den weiteren Vorteilen von Bäumen, dass sie die Temperatur absenken; zu Zeiten von Hitzewellen könnte das praktisch sein. Ansonsten erzeugen sie aber nur Luft und haben auch sonst kaum erkennbaren Nutzen. ↵
5. bei Fat-Tails-Variablen muss man diese Aussage einschränken ↵
6. Wenn es einen No-Free-Lunch-Satz gibt, müsste es auch einen Too-Good-to-be-True-Satz geben, den wir hiermit postulieren. ↵
7. Schlimmes Denglisch ↵
8. Streng genommen ist er eine Funktion der L2-Norm bzw. mit Lambda-Gewichtet und ohne die Wurzel, die zur Vektornorm gehört ↵
9. Wem das Bild zu klein gezeichnet ist, der nehme entweder eine Lupe oder öffne das Bild per Rechtsklick in einem neuen Tab. ↵