Matthias Thimm

Maschinelles Lernen

hagen text books Band 2

Hagen
University
Press

Matthias Thimm

Maschinelles Lernen

hagen text books

Hrsg. von Eva Cendon und Felicitas Schmieder

Band 2

Maschinelles Lernen

Matthias Thimm

HAGEN
UNIVERSITY
PRESS

Impressum

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Druck: CPI Druckdienstleistungen GmbH, Ferdinand-Jühlke-Straße 7, 99095 Erfurt

1. Auflage 2026

ISSN 3051-9993 (Print)
ISSN 3052-0002 (Online)
ISBN 978-3-98767-509-6 (Print)
ISBN 978-3-98767-038-1 (E-PDF)
DOI 10.57813/20260220-092408-0

Hagen UP (Hagen University Press)
FernUniversität in Hagen
Feithstraße 152
58097 Hagen

Inhaltsverzeichnis

1 Einleitung und Grundlagen

In diesem Kapitel geben wir zunächst eine kurze Einleitung und einen Überblick über das Thema dieses Buches (Unterkapitel 1.1) und wiederholen anschließend einige wichtige mathematische Grundlagen (Unterkapitel 1.2).

1.1 Einleitung und Überblick

Maschinelles Lernen ist eine Teildisziplin der künstlichen Intelligenz und beschäftigt sich mit der Entwicklung von Methoden zur automatischen Extraktion von Mustern und Regelmäßigkeiten aus Daten, um neue ähnliche Daten in intelligenter Weise bearbeiten zu können. Es gibt viele Definitionen des maschinellen Lernens, wir werden uns hier an [Tom M. Mitchell. Machine Learning. McGraw-Hill Science, 1997] orientieren:

> A computer program is said to *learn* from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, measured by P, improves with experience E.

Je nach Instantiierung der Komponenten E, T und P können verschiedene spezielle Ansätze des maschinellen Lernens charakterisiert werden. Eine typische Aufgabe des *überwachten Lernens* ist die *Klassifikation*, beispielsweise die Klassifikation von Bildern als „Katze“ oder „Hund“. In diesem Fall ist die Erfahrung (Komponente E) gegeben durch eine Menge von bereits klassifizierten Bildern und die Aufgabe (Komponente T) besteht darin, Muster in E zu erlernen, sodass neue Bilder entsprechend klassifiziert werden können. Ein Maß für die Güte dieses Ansatzes (Komponente P) ist dann beispielsweise

die Genauigkeit in der Klassifizierung neuer Bilder. Die klassische Unterteilung von Ansätzen des maschinellen Lernens erfolgt nach der Natur von E in das *überwachte* Lernen, das *unüberwachte* Lernen und das *Reinforcement Learning*. Wir werden in diesem Kurs dieser Unterteilung folgen und zusätzlich noch einen Fokus auf das sogenannte *Deep Learning* setzen, einer Familie von Ansätzen, die grundsätzlich alle zuvor genannten Problembereiche des maschinellen Lernens abdeckt und in den letzten 10 Jahren zunehmend an Bedeutung gewonnen hat.

Das Buch „Maschinelles Lernen“ ist in 6 Kapitel gegliedert:

1. Einleitung und Grundlagen
2. Überwachtes Lernen
3. Unüberwachtes Lernen
4. Reinforcement Learning
5. Deep Learning
6. Zusammenfassung und Ausblick

Im Rest von Kapitel 1 werden wir einige wichtige mathematische Grundlagen, die für das Verständnis des restlichen Materials notwendig sind, wiederholen. Kapitel 2 führt in das überwachte maschinelle Lernen ein, insbesondere werden dort lineare Regression, logistische Regression, *Support Vector Machines*, die Nächste-Nachbarn-Klassifikation, die Bayes-Klassifikation und die Entscheidungsbäume diskutiert. Kapitel 3 behandelt das unüberwachte Lernen und insbesondere K-Means-Clustering, hierarchisches Clustering, Assoziationsregellernen, Anomalieerkennung und die Hauptkomponentenanalyse. Kapitel 4 gibt einen Überblick über das *Reinforcement*

Learning, insbesondere Markov-Entscheidungsprozesse, sowie das aktive und passive *Reinforcement Learning*. Kapitel 5 behandelt das *Deep Learning* und gibt insbesondere einen Überblick über künstliche neuronale Netzwerke, *Convolutional Neural Networks*, *Recurrent Neural Networks* und das Lernen von Repräsentationen. Kapitel 6 schließt dieses Buch mit einer Zusammenfassung und einem kurzen Ausblick ab.

1.2 Grundlagen

In diesem Unterkapitel werden einige mathematische Grundlagen, die zum weiteren Verständnis benötigt werden, wiederholt. Die Darstellung in diesem Unterkapitel ist absichtlich sehr knapp und dient nicht der Vermittlung der tatsächlichen mathematischen Inhalte. Alle in diesem Unterkapitel aufgeführten Techniken sollten der Leserin/dem Leser prinzipiell bekannt sein und die Wiederholung dient ausschließlich der Fixierung von Begriffen und Notation. Es ist der Leserin und dem Leser geraten, sich fehlendes Wissen in diesen Bereichen gegebenenfalls anderweitig anzueignen.

1.2.1 Grundlegende Notationen

Die Menge der natürlichen Zahlen ohne 0 wird mit $\mathbb{N}$ bezeichnet, die Menge der natürlichen Zahlen mit 0 wird mit $\mathbb{N}_0$ bezeichnet. $\mathbb{Z}$ ist die Menge der ganzen Zahlen. Weiterhin bezeichnen wir die Menge der reellen Zahlen mit $\mathbb{R}$, die Menge der nicht-negativen reellen Zahlen mit $\mathbb{R}^{\geq 0}$ und die Menge der positiven reellen Zahlen mit $\mathbb{R}^{>0}$. Zur Darstellung von Dezimalzahlen verwenden wir die englische Schreibweise und benutzen den Punkt als Dezimaltrennzeichen, d. h., wir schreiben beispielsweise 1.23 für die Zahl $\frac{123}{100}$.

Eine *Multimenge* ist eine Menge, bei der Elemente mehrfach auftreten können, beispielsweise ist $\{1,3,3,5,6\}$ eine Multimenge natürlicher Zahlen. Mengenoperationen auf Multimengen sind analog zu normalen Mengen definiert, wobei Mehrfachvorkommen entsprechend berücksichtigt werden. Beispielsweise ist also $\{1,3,3,4\} \cup \{3,4,5\} = \{1,3,3,3,4,4,5\}$ und $\{3,3,4,5\} \setminus \{3,4\} = \{3,5\}$.

1.2.2 O-Notation

Für eine Funktion $f : \mathbb{N} \to \mathbb{N}$ bezeichnet $O(f)$ die Menge der Funktionen, die asymptotisch nicht schneller als f wachsen:

$$O(f) = \{g : \mathbb{N} \to \mathbb{N} \mid \exists c > 0 : \exists n_0 \in \mathbb{N} : \forall n > n_0 : g(n) \leq cf(n)\}$$

1.2.3 Lineare Algebra

Ein (Spalten-)Vektor (oder Punkt) x ist ein Tupel $x \in \mathbb{R}^n$ (für $n \in \mathbb{N}$). Sind $x_1, \ldots, x_n \in \mathbb{R}$ die Komponenten von x so schreiben wir

$$x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = (x_1, \ldots, x_n)^T$$

wobei „T“ hierbei für „transponiert“ steht. Beachten Sie, dass wir für die Darstellung von Vektoren sowohl runde $(\cdot)$ als auch eckige Klammern $[\cdot]$ benutzen. $\vec{0}$ bezeichne den Nullvektor, d. h., $\vec{0} = (0, \ldots, 0)^T$ (der Kontext gibt dabei an, wie viele Nullen der Vektor enthält). Die *Euklidische Norm* $\|\cdot\|$ bildet einen Vektor $x = (x_1, \ldots, x_n)^T \in \mathbb{R}^n$ auf seine Länge im Raum ab, d. h.,

$$\|x\| = \sqrt{x_1^2 + \ldots + x_n^2}$$

Eine Matrix X ist eine zwei-dimensionale Struktur $X \in \mathbb{R}^{n \times m}$ mit $n, m \in \mathbb{N}$, wobei n die Anzahl der Zeilen und m die Anzahl der Spalten bezeichnet, also

$$X = \begin{pmatrix} x_{1,1} & \dots & x_{1,m} \\ \vdots & \ddots & \vdots \\ x_{n,1} & \dots & x_{n,m} \end{pmatrix} = \begin{bmatrix} x_{1,1} & \dots & x_{1,m} \\ \vdots & \ddots & \vdots \\ x_{n,1} & \dots & x_{n,m} \end{bmatrix}$$

Auch für die Darstellung von Matrizen benutzen wir sowohl runde $(\cdot)$ als auch eckige Klammern $[\cdot]$. Die transponierte Matrix $X^T \in \mathbb{R}^{m \times n}$ einer Matrix $X \in \mathbb{R}^{n \times m}$ ist definiert durch

$$X^T = \begin{pmatrix} x_{1,1} & \dots & x_{n,1} \\ \vdots & \ddots & \vdots \\ x_{1,m} & \dots & x_{n,m} \end{pmatrix}$$

Ist $x \in \mathbb{R}^m$ und $X \in \mathbb{R}^{n \times m}$, so ist

$$\begin{aligned} Xx &= \begin{pmatrix} x_{1,1} & \dots & x_{1,m} \\ \vdots & \ddots & \vdots \\ x_{n,1} & \dots & x_{n,m} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_m \end{pmatrix} \\ &= \begin{pmatrix} x_{1,1}x_1 + \ldots + x_{1,m}x_m \\ \vdots \\ x_{n,1}x_1 + \ldots + x_{n,m}x_m \end{pmatrix} \in \mathbb{R}^n \end{aligned}$$

Ist $X \in \mathbb{R}^{n \times m}$ und $Y \in \mathbb{R}^{m \times k}$, so ist

$$\begin{aligned} XY &= \begin{pmatrix} x_{1,1} & \dots & x_{1,m} \\ \vdots & \ddots & \vdots \\ x_{n,1} & \dots & x_{n,m} \end{pmatrix} \begin{pmatrix} y_{1,1} & \dots & y_{1,k} \\ \vdots & \ddots & \vdots \\ y_{m,1} & \dots & y_{m,k} \end{pmatrix} \\ &= \begin{pmatrix} x_{1,1}y_{1,1} + \ldots + x_{1,m}y_{m,1} & \dots & x_{1,1}y_{1,k} + \ldots + x_{1,m}y_{m,k} \\ \vdots & \ddots & \vdots \\ x_{n,1}y_{1,1} + \ldots + x_{n,m}y_{m,1} & \dots & x_{n,1}y_{1,k} + \ldots + x_{n,m}y_{m,k} \end{pmatrix} \\ &\in \mathbb{R}^{n \times k} \end{aligned}$$

Beachten Sie noch zwei Spezialfälle für Vektoren $x = (x_1, \ldots, x_n)^T, y = (y_1, \ldots, y_n)^T \in \mathbb{R}^n$. Es gilt

$$x^T y = (x_1, \ldots, x_n) \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} = x_1 y_1 + \ldots + x_n y_n \in \mathbb{R}$$

und $x^T y$ heißt auch (Standard-)*Skalarprodukt* von x und y. Andererseits ist

$$xy^T = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} (y_1, \ldots, y_n) = \begin{pmatrix} x_1 y_1 & \ldots & x_1 y_n \\ \vdots & \ddots & \vdots \\ x_n y_1 & \ldots & x_n y_n \end{pmatrix} \in \mathbb{R}^{n \times n}$$

Die *Inverse* einer quadratischen Matrix $X \in \mathbb{R}^{n \times n}$ wird (falls sie existiert) mit X^{-1} bezeichnet. Ist $I \in \mathbb{R}^{n \times n}$ die *Einheitsmatrix* der Größe n, also

$$I = \begin{pmatrix} 1 & 0 & \ldots & \ldots & 0 \\ 0 & \ddots & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & 0 \\ 0 & \ldots & \ldots & 0 & 1 \end{pmatrix}$$

so gilt

$$XX^{-1} = X^{-1}X = I$$

Eine Matrix $X \in \mathbb{R}^{n \times n}$ mit

$$X = \begin{pmatrix} x_{1,1} & \ldots & x_{1,n} \\ \vdots & \ddots & \vdots \\ x_{n,1} & \ldots & x_{n,n} \end{pmatrix}$$

heißt *symmetrisch*, wenn $x_{i,j} = x_{j,i}$ für alle $i, j = 1, \ldots, n$. Eine symmetrische Matrix X heißt *positiv definit*, wenn für alle $x \in \mathbb{R}^n \setminus \{\vec{0}\}$, $x^T X x > 0$ gilt.

Ist $X \in \mathbb{R}^{n\times n}$, so ist $X|_{i,j}$ mit $i,j \in \{1,\ldots,n\}$ die Matrix, die aus X entsteht, wenn man die i-te Zeile und j-te Spalte entfernt, d.,h.,

$$X|_{i,j} = \begin{pmatrix} x_{1,1} & \cdots & x_{1,j-1} & x_{1,j+1} & \cdots & x_{1,n} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ x_{i-1,1} & \cdots & x_{i-1,j-1} & x_{i-1,j+1} & \cdots & x_{i-1,n} \\ x_{i+1,1} & \cdots & x_{i+1,j-1} & x_{i+1,j+1} & \cdots & x_{i+1,n} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ x_{n,1} & \cdots & x_{n,j-1} & x_{n,j+1} & \cdots & x_{n,n} \end{pmatrix}$$

Die *Determinante* $\det X$ von $X \in \mathbb{R}^{n\times n}$ ist definiert durch

$$\det X = \sum_{i=1}^{n} (-1)^{i+1} x_{i,1} \det X|_{i,1}$$

wobei $\det X = x_{1,1}x_{2,2} - x_{2,1}x_{1,2}$ für $X \in \mathbb{R}^{2\times 2}$ gilt.

Eine Zahl $\lambda \in \mathbb{R}$ heißt *Eigenwert* einer symmetrischen Matrix $X \in \mathbb{R}^{n\times n}$, wenn es $x \in \mathbb{R}^n \setminus \{\vec{0}\}$ gibt mit

$$Xx = \lambda x$$

Der Vektor x heißt dann auch *Eigenvektor* von X zu λ. Jeder Eigenwert λ ist Lösung der Gleichung

$$\det(X - \lambda I) = 0$$

Die *algebraische Vielfachheit* von λ ist die Zahl, wie oft λ Nullstelle von $\det(X - \lambda I)$ ist. Ist die algebraische Vielfachheit größer 1, so sagen wir auch, dass λ mehrfacher Eigenwert von X ist. Ist v Eigenvektor von X zu λ, so ist auch αv für alle $\alpha \in \mathbb{R} \setminus \{0\}$ Eigenvektor von X zu λ.

Ein Vektor $w \in \mathbb{R}^n$ ist eine *Linearkombination* von Vektoren $v_1,\ldots,v_m \in \mathbb{R}^n$ mit *Koeffizienten* $\alpha_1,\ldots,\alpha_m \in \mathbb{R}$ wenn

$$w = \alpha_1 v_1 + \ldots + \alpha_m v_m$$

Die Menge aller Linearkombinationen w von $V = \{v_1, \ldots, v_m\}$ heißt der durch V *aufgespannte Unterraum* von $\mathbb{R}^n$. Eine Menge $V = \{v_1, \ldots, v_m\}$ heißt *linear unabhängig*, wenn kein Vektor $v \in V$ als Linearkombination von $V \setminus \{v\}$ dargestellt werden kann. Solch eine Menge V heißt *Basis* von $\mathbb{R}^n$, wenn $|V| = n$. Eine Basis V von $\mathbb{R}^n$ heißt *orthogonal*, wenn $v^T w = 0$ für alle $v, w \in V$ mit $v \neq w$. Eine orthogonale Basis V heißt *orthonormal* wenn $\|v\| = 1$ für alle $v \in V$.

1.2.4 Analysis

Ist $f : \mathbb{R} \to \mathbb{R}$ eine Funktion, so bezeichnet f' die *Ableitung* von f. Für eine Funktion mit mehreren Parametern, also $g : \mathbb{R}^n \to \mathbb{R}$ mit $x_1, \ldots, x_n \mapsto g(x_1, \ldots, x_n)$, ist

$$\frac{\partial g}{\partial x_j}$$

für alle $j = 1, \ldots, n$ die *partielle Ableitung* von g bzgl. x_j. Der *Gradient* $\nabla_{x_1,\ldots,x_n} g$ von g ist dann ein Vektor bestehend aus allen partiellen Ableitungen, also

$$\nabla_{x_1,\ldots,x_n} g = \begin{pmatrix} \frac{\partial g}{\partial x_1} \\ \vdots \\ \frac{\partial g}{\partial x_n} \end{pmatrix}$$

Wir interpretieren $\nabla_{x_1,\ldots,x_n} g$ auch als Funktion $\nabla_{x_1,\ldots,x_n} g : \mathbb{R}^n \to \mathbb{R}^n$ mit

$$\nabla_{x_1,\ldots,x_n} g(x) = \begin{pmatrix} \frac{\partial g}{\partial x_1}(x) \\ \vdots \\ \frac{\partial g}{\partial x_n}(x) \end{pmatrix}$$

für alle $x \in \mathbb{R}^n$. Sind die Argumente $x_1, \ldots, x_n$ aus dem Kontext ersichtlich, so schreiben wir auch verkürzt ∇g statt $\nabla_{x_1,\ldots,x_n} g$.

1.2.5 Optimierung

Ist $f : \mathbb{R}^n \to \mathbb{R}$ eine reellwertige Funktion, so bezeichnet

$$\min_{x \in \mathbb{R}^n} f(x)$$

das Problem, einen Punkt $\hat{x} \in \mathbb{R}^n$ zu finden, so dass $f(\hat{x}) \leq f(x)$ für alle $x \in \mathbb{R}^n$ gilt (und die Lösung dieses Problems ist dann der Funktionswert $f(\hat{x})$ des gefundenen Punktes $\hat{x}$). Diese Art der Optimierungsprobleme nennt man *Minimierungsproblem ohne Nebenbedingungen*. Eine numerische Methode zur Lösung solcher Probleme ist *Gradient Descent* (dt. *Gradientenabstiegsverfahren*). Sei dazu ∇f wohldefiniert und $\gamma \in \mathbb{R}^{>0}$. Sei $x_0 \in \mathbb{R}^n$ beliebig und definiere eine Folge $x_1, x_2, \ldots \in \mathbb{R}^n$ via

$$x_i = x_{i-1} - \gamma \nabla f(x_{i-1}) \tag{1}$$

für alle $i \in \mathbb{N}$. Ist γ (die *Lernrate*) klein genug gewählt, so gilt $f(x_0) \geq f(x_1) \geq \ldots$ und $x_0, x_1, \ldots$ konvergiert gegen ein lokales Minimum. Varianten von *Gradient Descent* verkleinern γ nach jeder Anwendung von (1) oder bestimmen stets den optimalen Wert (zum Beispiel durch das Liniensuchverfahren).

Ein *Minimierungsproblem mit Nebenbedingungen* hat die Form

$$\begin{aligned} &\min_{x \in \mathbb{R}^n} f(x) \\ \text{so dass } \; &x \in X \end{aligned}$$

wobei die Bedingung $x \in X$ beispielsweise in Form von Gleichungen und/oder Ungleichungen gegeben werden kann, z. B.

$$\begin{aligned} &\min_{x_1, x_2 \in \mathbb{R}} x_1^2 + x_2^2 \\ \text{so dass } \; &x_1 + x_2 = 2 \end{aligned} \tag{2}$$

Minimierungsprobleme mit Nebenbedingungen können grundsätzlich in Minimierungsprobleme ohne Nebenbedingungen durch das *Lagrange-Verfahren* umgeformt werden. Für das obige Beispiel (2), sei $\lambda \in \mathbb{R}$ (der *Langrange*-Multiplikator) ausreichend groß, dann ist jede Lösung von

$$\min_{x_1,x_2\in\mathbb{R}} x_1^2 + x_2^2 + \lambda(x_1 + x_2 - 2)^2 \tag{3}$$

auch eine Lösung von (2) und andersherum.

Alles oben gesagte gilt natürlich analog für Maximierungsprobleme.

1.2.6 Stochastik

Für eine endliche Menge $S = \{x_1, \ldots, x_n\} \subseteq \mathbb{R}$ heißt

$$\tilde{x} = \frac{x_1 + \ldots + x_n}{n}$$

Mittelwert von S und

$$\sigma = \sqrt{\frac{(x_1 - \tilde{x})^2 + \ldots + (x_n - \tilde{x})^2}{n}}$$

heißt *Standardabweichung* von S. Die quadrierte Standardabweichung σ^2 heißt auch *Varianz*.

Eine (diskrete) Wahrscheinlichkeitsverteilung P über einer Menge Ω ist eine Funktion $P : 2^\Omega \to [0,1]$ mit

1. $P(\Omega) = 1$.
2. $P(X_1 \cup X_2) = P(X_1) + P(X_2)$ für alle $X_1, X_2 \subseteq \Omega$ mit $X_1 \cap X_2 = \emptyset$.

Für $X_1, X_2 \subseteq \Omega$ ist die *bedingte Wahrscheinlichkeit* $P(X_1 \mid X_2)$ definiert durch

$$P(X_1 \mid X_2) = \frac{P(X_1 \cap X_2)}{P(X_2)}$$

Die *Gleichverteilung* P_0 über Ω ist die Wahrscheinlichkeitsverteilung definiert durch

$$P_0(X) = \frac{|X|}{|\Omega|} \qquad \text{für alle } X \subseteq \Omega$$

Eine *Wahrscheinlichkeitsdichtefunktion* p ist eine integrierbare Funktion $p : \mathbb{R} \to \mathbb{R}^{\geq 0}$ mit

$$\int_{-\infty}^{\infty} p(x)dx = 1$$

Für $\mu, \sigma^2 \in \mathbb{R}$ ist die *Normalverteilung* $\mathcal{N}(\mu, \sigma^2)$ mit Erwartungswert μ und Varianz σ^2 definiert durch

$$\mathcal{N}(\mu, \sigma^2)(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

für alle $x \in \mathbb{R}$.

2 Überwachtes Lernen

Überblick über dieses Kapitel

Überwachtes Lernen beschreibt eine Familie von Problemen des maschinellen Lernens, bei der Modelle mit Hilfe von Daten gelernt werden, die schon mit ihren „Zielwerten" gekennzeichnet sind. Diese Form des Lernens heißt *überwacht*, da diese Kennzeichnungen eine Art Steuerung oder „Überwachung" des Lernprozesses realisieren. Die beiden wichtigsten Probleme beim überwachten Lernen sind *Regression* und *Klassifikation*. Bei der Regression geht es um die Vorhersage kontinuierlicher Werte, beispielsweise die Vorhersage der morgigen Temperatur aus Daten der Temperaturverläufe der vorherigen Tage. Bei der Klassifikation geht es um die Vorhersage diskreter Klassen, beispielsweise die Vorhersage, ob ein Bild eine Katze oder einen Hund darstellt, gegeben eine Menge von anderen Bildern, die schon mit „Hund" oder „Katze" gekennzeichnet sind.

Wir werden uns in diesem Kapitel mit sechs Methoden des überwachten Lernens beschäftigten. In Abschnitt 2.1 schauen wir uns mit der linearen Regression einen der einfachsten Ansätze zum maschinellen Lernen und insbesondere dem Regressionsproblem an. In Abschnitt 2.2 geht es um die logistische Regression, die trotz ihres Namens eigentlich eine Methode für das Klassifikationsproblem ist. In Abschnitt 2.3 diskutieren wir *Support Vektor Machines*, die eine sehr effektive und weit verbreitete Methode für das überwachte Lernen darstellen. Abschnitt 2.4 stellt den Ansatz der Nächste-Nachbarn-Klassifikation, eine konzeptuell sehr einfache Methode, vor. Abschnitt 2.5 behandelt die Bayes-Klassifikation, ei-

ne auf probabilistischen Grundsätzen fundierte Methode zur Klassifikation. Schließlich diskutieren wir Entscheidungsbäume in Abschnitt 2.6, die eine regelbasierte Sichtweise auf Klassifikation repräsentieren.

Bibliographische Anmerkungen

Einen einfachen Einstieg in allgemeine Grundlagen des maschinellen Lernens bietet Kapitel 5 von [13] und einen fokussierten technischeren Überblick gibt Kapitel 5 von [5]. Der Coursera-Kurs „Supervised Machine Learning: Regression and Classification" von Andrew Ng [11] ist gerade für das überwachte Lernen auch sehr zu empfehlen.

Für eine kurze Zusammenfassung der linearen Regression (Abschnitt 2.1) ist [4, Kapitel 3.1] zu empfehlen. Gleiches gilt für die logistische Regression (Abschnitt 2.2), hier findet sich eine Zusammenfassung in [4, Kapitel 3.2]. *Support Vector Machines* werden in [4, Kapitel 3.4] diskutiert. Für die Nächste-Nachbarn-Klassifikation (Abschnitt 2.4) siehe [4, Kapitel 3.5]. Weiterführende Literatur zur Bayes-Klassifikation, auf die sich auch die Darstellung in Abschnitt 2.5 stützt, findet sich beispielsweise in [8, Kapitel 6]. Entscheidungsbäume werden ausführlich in [8, Kapitel 3] und [3, Kapitel 5] diskutiert.

2.1 Lineare Regression

Lineare Regression ist die prinzipiell simpelste Form des maschinellen Lernens und wird üblicherweise auch eher als statistische Methode und nicht als Methode des maschinellen Lernens verstanden. Die Prinzipien der linearen Regression sind allerdings archetypisch für viele Methoden des maschinellen Lernens und es lassen sich viele komplexe Konzepte einfach am Beispiel der linearen Regression erläutern. Aus diesem Grund sind viele der in diesem Unterkapitel eingeführten Konzepte für den gesamten Bereich des maschinellen Lernens relevant.

2.1.1 Grundlagen

In seiner einfachsten Form besteht das Problem der linearen Regression in der *optimalen* Anpassung einer Geraden an eine gegebene Menge von Punkten. Insbesondere beschäftigen wir uns hier mit der Anwendung der linearen Regression für das Problem der *Funktionsapproximation*, d. h., der Vorhersage des Funktionswerts einer Instanz, gegeben gewisser *Merkmalsausprägungen* der Instanz. Sei $n \in \mathbb{N}$ fix.

Definition 1. Ein *Datenpunkt* $x = (x_1, \ldots, x_n)^T$ ist ein Punkt $x \in \mathbb{R}^n$.

Wir nennen n die *Dimension* und jedes $i = 1, \ldots, n$ ein *Merkmal* (engl. *feature*). Ist $x \in \mathbb{R}^n$ ein Datenpunkt mit $x = (x_1, \ldots, x_n)^T$, so stellt jedes x_i, $i = 1, \ldots, n$, eine *Ausprägung* des Merkmals i dar. Sei $m \in \mathbb{N}$ fix. Ein Tupel (x, y) mit $y \in \mathbb{R}$ (der Wert der *Zielvariablen*) ist ein *Beispiel* und eine Menge $D = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ mit Beispielen $(x^{(i)}, y^{(i)})$, $i = 1, \ldots, m$, heißt *Datensatz*.

Beispiel 1. Stellen Sie sich vor, Sie wollen Ihrem oder Ihrer Liebsten einen Ring schenken und möchten, dass

dies eine Überraschung wird. Weiterhin ergibt sich nicht die Möglichkeit den Umfang des Ringfingers der Person (heimlich) abzumessen, so dass Sie andere Wege finden müssen, um an diese Information zu kommen. Wie es der Zufall so will, verfügen Sie allerdings über Daten Ihrer besten Freunde, die den Umfang des Ringfingers mit der Körpergröße der jeweiligen Person in Beziehung setzt, siehe Tabelle 1. Abbildung 1 visualisiert die Daten entsprechend. Ignorieren wir die Bezeichner (Personennamen) der Daten (diese sind für die Lernaufgabe nicht von Relevanz) so lassen sich die Daten entsprechend als Datensatz D_{ring} via

$$\begin{aligned} D_{\text{ring}} = \{&((153.3), 47.1), ((158.9), 46.8), ((160.8), 49.3), \\ &((179.6), 53.2), ((156.6), 47.7), ((165.1), 49.0), \\ &((165.9), 50.6), ((156.7), 47.1), ((167.8), 51.7), \\ &((160.8), 47.8)\} \end{aligned}$$

repräsentieren. Hierbei ist beispielsweise $((153.3), 47.1)$ ein Beispiel mit Datenpunkt $(153.3) \in \mathbb{R}^1$, wobei 153.3 die Ausprägung des Merkmals „Körpergröße“ ist.

Beispiel 2. Um Ihren persönlichen Aufwand für den Kurs „Maschinelles Lernen“ möglichst gering zu halten, haben Sie wieder Ihre Freunde (die den Kurs alle schon besucht haben) befragt, wie viele Stunden sie für den Kurs investiert haben, wie viele Sprechstunden sie besucht haben und welche Note am Ende dabei herausgekommen ist, siehe Tabelle 2. Abbildung 2 visualisiert die Daten entsprechend. Ignorieren wir die Bezeichner (Personennamen) der Daten (diese sind für die Lernaufgabe nicht von Relevanz) so lassen sich die Daten entspre-

Person	Körpergröße (in cm)	Ringfingerumfang (in mm)
Anna	153.3	47.1
Bernd	158.9	46.8
Catharina	160.8	49.3
David	179.6	53.2
Edith	156.6	47.7
Frank	165.1	49.0
Gertrud	165.9	50.6
Hans	156.7	47.1
Irma	167.8	51.7
Jochen	160.8	47.8

Tabelle 1: Datensatz zu Beispiel 1. Bitte beachten Sie, dass die Daten rein fiktiv sind.

chend als Datensatz D_{note} via

$$\begin{aligned} D_{\text{note}} = \{&((253,3),2.0),((301,6),1.0),((211,1),3.3),\\ &((103,3),5.0),((353,4),1.0),((250,2),2.3),\\ &((98,4),4.0),((150,4),3.7),((63,1),5.0),\\ &((282,5),1.3)\} \end{aligned}$$

repräsentieren. Hierbei ist beispielsweise $((253,3),2.0)$ ein Beispiel mit Datenpunkt $(253,3) \in \mathbb{R}^2$, wobei 253 die Ausprägung des Merkmals „Lernaufwand“ und 3 die Ausprägung des Merkmals „Anzahl Teilnahmen Sprechstunden“ ist.

Wir benutzen lineare Regression zur Funktionsapproximation, d. h., die Aufgabe des zu erlernenden Modells ist die Vorhersage des Funktionswertes $y \in \mathbb{R}$ zu einem beliebigen Datenpunkt $x \in \mathbb{R}^n$. Wir nehmen dazu an, dass die zu suchende Funktion *ähnlich* zu der

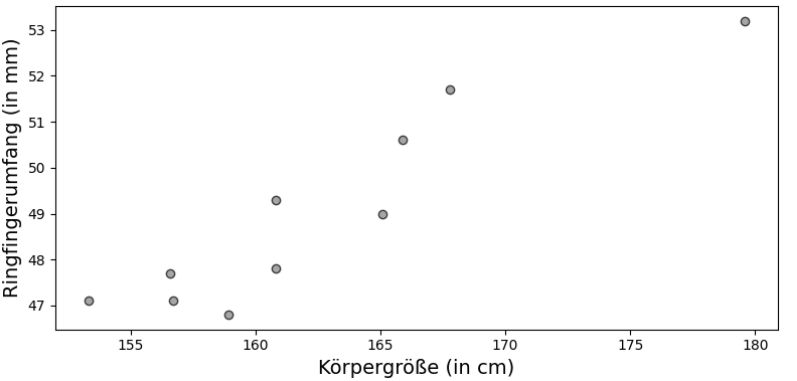


Abbildung 1: Datensatz zu Beispiel 1 (visualisiert).

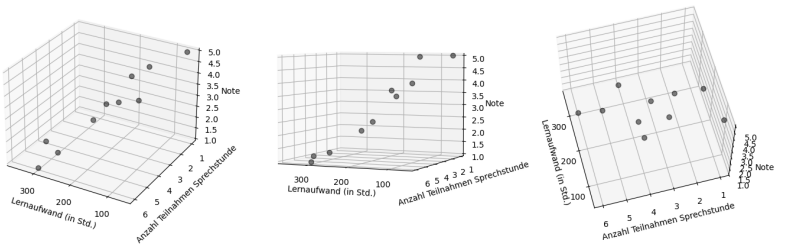


Abbildung 2: Datensatz zu Beispiel 2 (visualisiert).

Funktion ist, die einen gegebenen *Trainingsdatensatz* $D = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ generiert hat. Bei der linearen Regression nehmen wir zusätzlich an, dass der Zusammenhang zwischen x und y (die *Zielvariable*) *linear* ist.

Definition 2. Sei $\theta = (\theta_0, \theta_1, \ldots, \theta_n)^T \in \mathbb{R}^{n+1}$. Eine *lineares Modell* auf $\mathbb{R}^n$ ist eine Funktion $h_\theta : \mathbb{R}^n \to \mathbb{R}$ mit

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \ldots + \theta_n x_n$$

für alle $x = (x_1, \ldots, x_n)^T \in \mathbb{R}^n$.

Person	Lernaufwand (in Std.)	Anzahl Teilnahmen Sprechstunde	Note
Anna	253	3	2.0
Bernd	301	6	1.0
Catharina	211	1	3.3
David	103	3	5.0
Edith	353	4	1.0
Frank	250	2	2.3
Gertrud	98	4	4.0
Hans	150	4	3.7
Irma	63	1	5.0
Jochen	282	5	1.3

Tabelle 2: Datensatz zu Beispiel 2. Bitte beachten Sie, dass die Daten rein fiktiv sind.

Die Werte $\theta = (\theta_0, \ldots, \theta_n)$ sind die *Parameter* des Modells h_θ. Der Parameter θ_0 stellt hierbei den konstanten Teil der Funktion h_θ dar.

Gegeben ein Trainingsdatensatz $D = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$, ist es nun die Aufgabe, konkrete Werte für die Parameter θ zu finden, so dass $h_\theta(x^{(i)}) \approx y^{(i)}$, für alle $i = 1, \ldots, m$, ist. Dann gilt unter der Annahme, dass zukünftige Beispiele aus der gleichen Verteilung entstammen, auch, dass für einen neuen Datenpunkt $x \in \mathbb{R}^n$ der Wert $h_\theta(x)$ „nahe“ am tatsächlichen Funktionswert liegt.

Beispiel 3. Abbildung 3 visualisiert drei verschiedene mögliche lineare Modelle (unterschiedlicher Güte), die die Daten aus Beispiel 1 (siehe auch Tabelle 1 und Abbil-

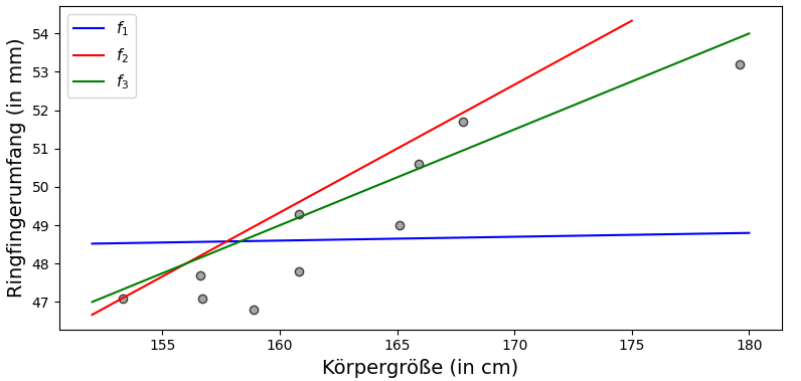


Abbildung 3: Illustration dreier möglicher linearer Modelle zu den Daten aus Beispiel 1.

dung 1) erklären. Die dargestellten Funktionen sind

$$f_1(z) = 47 + \frac{1}{100}z$$
$$f_2(z) = -4 + \frac{1}{3}z$$
$$f_3(z) = 9 + \frac{1}{4}z$$

Betrachten wir uns f_1 etwas genauer. Die Funktion f_1 kann äquivalent repräsentiert werden durch die Parameter

$$\theta = (47, \frac{1}{100})$$

eines linearen Modells h_θ. Gegeben ein Datenpunkt $x = (x_1)$ ist

$$h_\theta(x) = \theta_0 + \theta_1 x_1$$

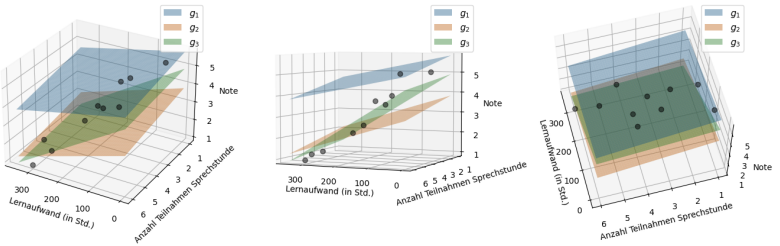


Abbildung 4: Illustration dreier möglicher linearer Modelle zu den Daten aus Beispiel 2.

Beispiel 4. Abbildung 4 visualisiert drei verschiedene mögliche lineare Modelle (unterschiedlicher Güte), die die Daten aus Beispiel 2 (siehe auch Tabelle 2 und Abbildung 2) erklären. Die dargestellten Funktionen sind

$$g_1(y,z) = 6 - \frac{1}{300}y - \frac{1}{6}z$$
$$g_2(y,z) = 4 - \frac{1}{200}y - \frac{1}{6}z$$
$$g_3(y,z) = 5 - \frac{1}{100}y - \frac{1}{10}z$$

Betrachten wir uns g_1 etwas genauer. Die Funktion g_1 kann äquivalent repräsentiert werden durch die Parameter

$$\theta = (6, -\frac{1}{300}, -\frac{1}{6})$$

eines linearen Modells h_θ. Gegeben ein Datenpunkt $x = (x_1, x_2)$ ist

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2$$

Das Finden der optimalen Parameter θ, so dass h_θ möglichst gut an die Beispiele in D angepasst ist, kann als Optimierungsproblem modelliert werden. Dazu wählt man zunächst ein geeignetes *Abstandsmaß* (oder *Fehlermaß* oder *Kostenfunktion*), das bewertet, wie nah eine Funktion an die Beispiele D angepasst ist. Das üblichste Fehlermaß im maschinellen Lernen ist der *quadratische Fehler*. Um die Notation zu vereinfachen, repräsentieren wir $D = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ durch eine Matrix X_D und einen Vektor y_D wie folgt

$$X_D = \begin{bmatrix} 1 & x_1^{(1)} & \ldots & x_n^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(m)} & \ldots & x_n^{(m)} \end{bmatrix} \qquad y_D = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

Mit anderen Worten, $X_D \in \mathbb{R}^{m \times (n+1)}$ enthält als Zeilenvektoren alle Datenpunkte der Beispiele aus D und eine zusätzlich vorangestellte 1. Letztere dient der einfacheren Darstellung als Vektorprodukt und wird auch *Bias* genannt. Da die Zeilen von X_D später mit $\theta \in \mathbb{R}^{n+1}$ multipliziert werden sollen, stimmen hier nun die Dimensionen und der konstante Teil der Funktion h_θ wird hier einfach mit 1 multipliziert. Der Vektor $y_D \in \mathbb{R}^m$ enthält die zugehörigen Funktionswerte.

Definition 3. Sei D ein Datensatz und $f : \mathbb{R}^n \to \mathbb{R}$ eine beliebige Funktion. Der *quadratische Fehler* L von f bzgl. D ist definiert durch

$$L(D, f) = \sum_{i=1}^{m} (f(x^{(i)}) - y^{(i)})^2$$

Ist $f = h_\theta$ eine lineare Funktion h_θ mit Parametern θ, so ist dies äquivalent zu

$$L(D, \theta) = \|X_D \theta - y_D\|^2$$

wobei $\|\cdot\|$ die *Euklidische Norm* ist.

Beispiel 5. Wir setzen Beispiel 3 fort und wiederholen noch einmal die drei genannten linearen Modelle f_1, f_2 und f_3:

$$f_1(z) = 47 + \frac{1}{100}z$$
$$f_2(z) = -4 + \frac{1}{3}z$$
$$f_3(z) = 9 + \frac{1}{4}z$$

Wenden wir den quadratischen Fehler auf diese Funktionen bzgl. des Datensatzes D_{ring} (siehe Tabelle 1) an, erhalten wir

$$L(D_{\text{ring}}, f_1) \approx 41.705$$
$$L(D_{\text{ring}}, f_2) \approx 21.349$$
$$L(D_{\text{ring}}, f_3) \approx 9.778$$

Wie man sieht erhält die optisch am besten angepasste Funktion f_3 den niedrigsten quadratischen Fehler.

Beispiel 6. Wir setzen Beispiel 4 fort und wiederholen noch einmal die drei genannten linearen Modelle g_1, g_2 und g_3:

$$g_1(y,z) = 6 - \frac{1}{300}y - \frac{1}{6}z$$
$$g_2(y,z) = 4 - \frac{1}{200}y - \frac{1}{6}z$$
$$g_3(y,z) = 5 - \frac{1}{100}y - \frac{1}{10}z$$

Wenden wir den quadratischen Fehler auf diese Funktionen bzgl. des Datensatzes D_{note} (siehe Tabelle 2) an,

erhalten wir

$$L(D_{\text{note}}, g_1) \approx 47.0454$$
$$L(D_{\text{note}}, g_2) \approx 9.958$$
$$L(D_{\text{note}}, g_3) \approx 3.397$$

Auch hier sieht man, dass die optisch am besten angepasste Funktion g_3 den niedrigsten quadratischen Fehler erhält.

Damit h_θ die Beispiele in D bestmöglich approximiert, suchen wir Parameter θ, die den quadratischen Fehler L bzgl. D *minimieren*, d h., wir suchen eine Lösung für das folgende Optimierungsproblem:

$$\min_\theta L(D, \theta) = \min_\theta \|X_D\theta - y_D\|^2 \tag{4}$$

Für lineare Regression ist das obige Optimierungsproblem stets eindeutig lösbar, d. h., ein lokales Minimum ist stets das globale Minimum. Methoden wie *Gradient Descent* können hier also das obige Problem beliebig genau lösen. Bei großen Trainingsdatensätzen und/oder vielen Merkmalen ist dies auch die praktikabelste Methode. Allerdings kann das Minimum in (4) durch einfache Methoden der Differentialrechnung auch in geschlossener Form dargestellt und somit direkt berechnet werden.[1] Da das lokale/globale Minimum von $L(D, \theta)$ eindeutig bestimmt ist, kann dieses durch die Nullstelle des Gradienten bzgl. θ von $L(D, \theta)$ charakterisiert werden, d. h., θ ist die optimale Parameterkombination gdw. $\nabla_\theta L(D, \theta) = 0$ ist. Es

[1] Beachten Sie bitte, dass im Folgenden und auch in späteren Unterkapiteln, einige mathematische Details abstrahiert werden und nicht immer alle notwendigen Vorbedingungen zur Anwendung mathematischer Gleichheiten explizit überprüft werden. Für weitere Details verweisen wir auf weiterführende Literatur.

folgt:

$$\begin{aligned}
& \nabla_\theta L(D,\theta) = 0 \\
\Longrightarrow \quad & \nabla_\theta \|X_D\theta - y_D\|^2 = 0 \\
\Longrightarrow \quad & \nabla_\theta (X_D\theta - y_D)^T (X_D\theta - y_D) = 0 \\
\Longrightarrow \quad & \nabla_\theta (\theta^T X_D^T X_D \theta - 2\theta^T X_D^T y_D + y_D^T y_D) = 0 \\
\Longrightarrow \quad & 2X_D^T X_D \theta - 2X_D^T y_D = 0 \\
\Longrightarrow \quad & (X_D^T X_D)^{-1} X_D^T y_D = \theta
\end{aligned}$$

Da X_D und y_D gegeben ist, kann θ mit obiger Gleichung direkt berechnet werden. Es sollte aber beachtet werden, dass diese Methode die rechenaufwändige Invertierung der Matrix $X_D^T X_D$ beinhaltet. Gerade bei großen Werten für m und n ist die Verwendung von numerischen Optimierungsmethoden (wie *Gradient Descent*) zu bevorzugen.

Beispiel 7. Wir setzen Beispiel 5 fort. Das optimale lineare Modell in diesem Beispiel ist gegeben durch

$$f(z) \approx 5.362 + 0.269z$$

Die Funktion f ist visualisiert in Abbildung 5 und verfügt mit

$$L(D_{\text{ring}}, f) \approx 5.91$$

über einen geringeren quadratischen Fehler als die Funktionen aus Beispiel 3.

Beispiel 8. Wir setzen Beispiel 6 fort. Das optimale lineare Modell in diesem Beispiel ist gegeben durch

$$g(y,z) \approx 6.321 - 0.0141y - 0.166z$$

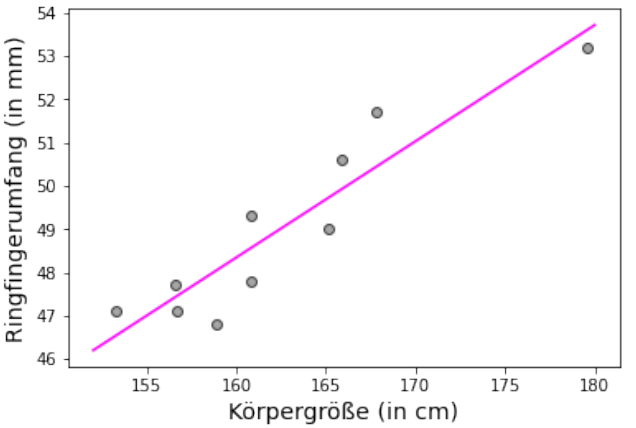


Abbildung 5: Optimal angepasstes lineares Modell für den Datensatz D_{ring}.

Die Funktion g ist visualisiert in Abbildung 6 und verfügt mit

$$L(D_{\text{note}}, g) \approx 0.83$$

über einen geringeren quadratischen Fehler als die Funktionen aus Beispiel 4.

2.1.2 Evaluation

Lineare Regression ist nur eine Methode von vielen im Bereich des maschinellen Lernens. Eine wichtiger Punkt bei der Entwicklung und Anwendung von Methoden des maschinellen Lernens ist daher die *Evaluation*, d. h., die Analyse und Feststellung, wie gut eine bestimmte Methode ein konkretes Problem des maschinellen Lernens löst. Dies hilft insbesondere dabei, die für das konkrete Problem beste Methode auszuwählen.

Um eine Methode des maschinellen Lernens zu evaluieren, teilt man üblicherweise die für das Lernen zur

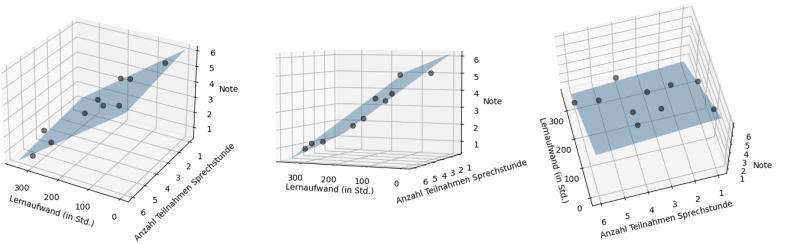


Abbildung 6: Optimal angepasstes lineares Modell für den Datensatz D_{note}.

Verfügung stehenden Daten in einen *Trainingsdatensatz* (wie zuvor schon genutzt) und einen *Testdatensatz* auf. Anschließend trainiert man die Methode ausschließlich auf dem Trainingsdatensatz und benutzt den Testdatensatz, um zu evaluieren wie gut das gelernte Modell auf zuvor ungesehenen Daten generalisiert. Üblicherweise geschieht diese Aufteilung in Trainings- und Testdatensatz *zufällig* und so, dass der Trainingsdatensatz ungefähr 75%-90% des Gesamtdatensatzes ausmacht. Als Evaluationsmaß benutzen wir eine normalisierte Variante des quadratischen Fehlers: das *Bestimmtheitsmaß*.

Definition 4. Sei $D = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ ein Datensatz und $f : \mathbb{R}^n \to \mathbb{R}$ eine beliebige Funktion. Das *Bestimmtheitsmaß* R^2 von f bzgl. D ist definiert durch

$$R^2(D, f) = \left(1 - \frac{L(D, f)}{\sum_{i=1}^{m}(y^{(i)} - \tilde{y})^2}\right) = \left(1 - \frac{\sum_{i=1}^{m}(f(x^{(i)}) - y^{(i)})^2}{\sum_{i=1}^{m}(y^{(i)} - \tilde{y})^2}\right)$$

wobei $\tilde{y}$ der Mittelwert der $y^{(i)}$ ist:

$$\tilde{y} = \frac{1}{m}\sum_{i=1}^{m} y^{(i)}$$

Der Wert $R^2(D, f)$ kann maximal 1 betragen, dies entspricht dem Fall, daß f den Datensatz D perfekt modelliert ($f(x^{(i)}) = y^{(i)}$ für alle $i = 1, \ldots, m$). Je kleiner der Wert $R^2(D, f)$, desto schlechter die Vorhersagequalität von f bezüglich D. Für ein naives Modell f_{naive}, dass stets den Mittelwert $\tilde{y}$ vorhersagen würde, hätten wir $R^2(D, f_{\text{naive}}) = 0$.[2] Modelle f mit $R^2(D, f) < 0$ sind somit noch schlechter als dieses naive Modell.

Gegeben einen Datensatz D und einer Aufteilung von D in einen Trainingsdatensatz D^{train} und einen Testdatensatz D^{test}, so erwarten wir, dass $R^2(D^{\text{train}}, f)$ für das auf D^{train} gelernte Modell f relativ nahe an 1 liegt (da f durch Minimierung des quadratischen Fehlers gelernt wurde und R^2 prinzipiell nur eine skalierte Variante des quadratischen Fehlers ist). $R^2(D^{\text{test}}, f)$ ist üblicherweise kleiner als $R^2(D^{\text{train}}, f)$. Je näher $R^2(D^{\text{test}}, f)$ allerdings an $R^2(D^{\text{train}}, f)$ liegt, desto besser ist die Fähigkeit von f, auf ungesehene Daten zu generalisieren.

Beispiel 9. Wir setzen Beispiel 7 fort. Tabelle 3 enthält noch einmal den Datensatz D_{ring}, diesmal mit Indizes anstatt Personennamen. Wir definieren den Trainingsdatensatz $D^{\text{train1}}_{\text{ring}}$ und den Testdatensatz $D^{\text{test1}}_{\text{ring}}$ via

$$\begin{aligned} D^{\text{train1}}_{\text{ring}} = \{&(x^{(1)}, y^{(1)}), (x^{(3)}, y^{(3)}), (x^{(4)}, y^{(4)}), (x^{(5)}, y^{(5)}), \\ &(x^{(6)}, y^{(6)}), (x^{(7)}, y^{(7)}), (x^{(8)}, y^{(8)}), (x^{(10)}, y^{(10)})\} \\ D^{\text{test1}}_{\text{ring}} = \{&(x^{(2)}, y^{(2)}), (x^{(9)}, y^{(9)})\} \end{aligned}$$

Mit anderen Worten, $D^{\text{test1}}_{\text{ring}}$ enthält die 2. und 9. Zeile aus Tabelle 3 und $D^{\text{train1}}_{\text{ring}}$ enthält alle übrigen Zeilen. Lernen

[2] Im Szenario der Vorhersage der Ringgröße hätten wir beispielsweise $f_{\text{naive}}(x) = 49$, also ein Modell, dass konstant den Mittelwert aller beobachteten Ringgrößen (49) vorhersagt.

wir ein lineares Modell ausschließlich auf $D_{\text{ring}}^{\text{train1}}$, so erhalten wir das in Abbildung 7 dargestellte Modell f_1 (hier sind die Trainingsdaten auch grau dargestellt, wohingegen die Testdaten rot dargestellt sind). Beachten Sie, dass sich dieses Modell f_1 von dem Modell f aus Beispiel 7 geringfügig unterscheidet, da es nicht auf den exakt selben Daten trainiert wurde. Wollen wir nun evaluieren, wie gut dieses Modell generalisiert, können wir es auf die Testdaten anwenden und vergleichen dazu die Vorhersagegenauigkeit des Modells f_1 mit den vorhandenen Werten. Dazu nutzen wir das Bestimmtheitsmaß als Messgröße, d. h., wir berechnen

$$R^2(D_{\text{ring}}^{\text{test1}}, f_1) \approx 0.691$$

Der Wert 0.691 ist durchaus hoch, insbesondere bei dem recht kleinen Trainingsdatensatz, den wir genutzt haben. Vergleichen wir diesen Wert nun mit dem entsprechenden Wert auf den Trainingsdaten

$$R^2(D_{\text{ring}}^{\text{train1}}, f_1) \approx 0.919$$

so sehen wir, dass das Modell f_1 weitaus besser auf dem Trainingsdatensatz abschneidet (auf dem es ja auch gelernt wurde). Der Abstand $R^2(D_{\text{ring}}^{\text{test1}}, f_1)$ zu $R^2(D_{\text{ring}}^{\text{train1}}, f_1)$ ist relativ groß und man könnte vermuten, dass das Modell f_1 zu stark an die Trainingsdaten angepasst ist und eher schlecht generalisiert (siehe das Problem der *Überanpassung* im nächsten Abschnitt). Vermutlich haben wir hier aber nur Pech mit der Aufteilung in Trainings- und Testdaten gehabt. Wir probieren eine andere Aufteilung:

$$\begin{aligned} D_{\text{ring}}^{train2} = \{&(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), (x^{(3)}, y^{(3)}), (x^{(4)}, y^{(4)}), \\ &(x^{(5)}, y^{(5)}), (x^{(6)}, y^{(6)}), (x^{(7)}, y^{(7)}), (x^{(10)}, y^{(10)})\} \\ D_{\text{ring}}^{test2} = \{&(x^{(8)}, y^{(8)}), (x^{(9)}, y^{(9)})\} \end{aligned}$$

Nr.	Körpergröße (in cm)	Ringfingerumfang (in mm)
1	153.3	47.1
2	158.9	46.8
3	160.8	49.3
4	179.6	53.2
5	156.6	47.7
6	165.1	49.0
7	165.9	50.6
8	156.7	47.1
9	167.8	51.7
10	160.8	47.8

Tabelle 3: Datensatz zu Beispiel 7, reproduziert von Tabelle 1.

Mit anderen Worten, $D_{\text{ring}}^{\text{test2}}$ enthält die 8. und 9. Zeile aus Tabelle 3 und $D_{\text{ring}}^{\text{train2}}$ enthält alle übrigen Zeilen. Lernen wir ein lineares Modell ausschließlich auf $D_{\text{ring}}^{\text{train2}}$, so erhalten wir das in Abbildung 8 dargestellte Modell f_2 (wieder sind die Trainingsdaten grau und die Testdaten rot dargestellt). Wir berechnen das Bestimmtheitsmaß auf $D_{\text{ring}}^{\text{train2}}$ und $D_{\text{ring}}^{\text{test2}}$:

$$R^2(D_{\text{ring}}^{train2}, f_2) \approx 0.877$$
$$R^2(D_{\text{ring}}^{test2}, f_2) \approx 0.783$$

Hier sind beide Werte recht hoch und auch recht nah beieinander.

Um das im vorherigen Beispiel auftretende Problem der variierenden Abstände zwischen Bestimmtheit der Trainings- und Testdaten zu adressieren, benutzt man

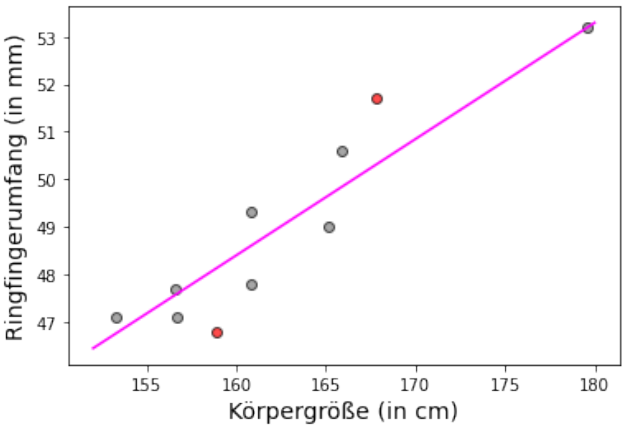


Abbildung 7: Optimal angepasstes lineares Modell für den Datensatz $D_{\text{ring}}^{\text{train1}}$.

oft die sogenannte *Kreuzvalidierung* (engl. *cross validation*). Gegeben ein Datensatz D und k eine natürliche Zahl (übliche Werte sind 5 oder 10), so teilen wir zunächst D in k ungefähr gleich große Teildatensätze $D_1, \ldots, D_k$ auf. Anschliessend führen wir eine Evaluation auf k verschiedenen Paaren von Trainings- und Testdaten $(D_1^{\text{train}}, D_1^{\text{test}}), \ldots, (D_k^{\text{train}}, D_k^{\text{test}})$ aus, die wie folgt definiert sind:

$$
\begin{aligned}
D_1^{\text{train}} &= D_1 \cup \ldots \cup D_{k-1} & D_1^{\text{test}} &= D_k \\
D_2^{\text{train}} &= D_1 \cup \ldots \cup D_{k-2} \cup D_k & D_2^{\text{test}} &= D_{k-1} \\
D_3^{\text{train}} &= D_1 \cup \ldots \cup D_{k-3} \cup D_{k-1} \cup D_k & D_3^{\text{test}} &= D_{k-2} \\
D_4^{\text{train}} &= D_1 \cup \ldots \cup D_{k-4} \cup D_{k-2} \cup \ldots \cup D_k & D_4^{\text{test}} &= D_{k-3} \\
&\vdots & & \\
D_k^{\text{train}} &= D_2 \cup \ldots \cup D_k & D_k^{\text{test}} &= D_1
\end{aligned}
$$

Die verschiedenen Paare von Trainings- und Testdaten erhalten wir also, indem wir einen Teildatensatz D_i als Test-

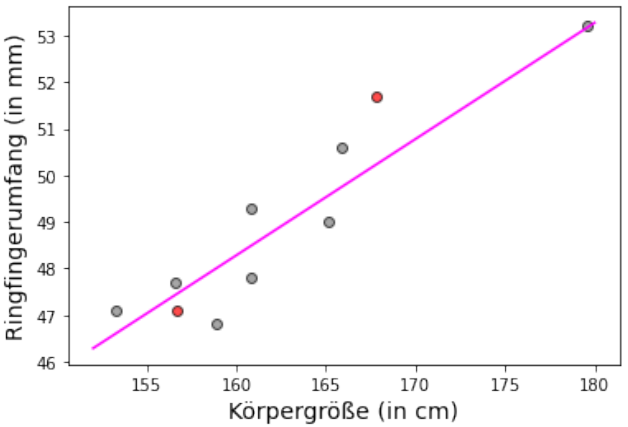


Abbildung 8: Optimal angepasstes lineares Modell für den Datensatz $D_{\text{ring}}^{\text{train2}}$.

datensatz und den Rest als Trainingsdatensatz deklarieren. Auf jedem Paar können dann die entsprechenden R^2-Werte berechnet werden und als finale Evaluationsmetrik berechnet man den Durchschnitt von $R^2(D_1^{\text{test}}, \cdot), \ldots, R^2(D_k^{\text{test}}, \cdot)$.

2.1.3 Nichtlineare Modelle

Lineare Regression ist eine mächtige Methode des maschinellen Lernens, aber fundamental durch die Linearitätsannahme in ihrer Anwendbarkeit eingeschränkt. In vielen realen Anwendungsfällen stehen Merkmale und Zielvariable nicht notwendigerweise in einem linearen Zusammenhang.

Beispiel 10. Betrachten Sie den in Tabelle 4 dargestellten und in Abbildung 9 visualisierten Datensatz D_{houses}, der die Größe eines Hauses zu seinem Kaufpreis in Relation

Nr.	Fläche (in m^2)	Preis (in EUR)
1	100	210000
2	120	270000
3	160	290000
4	170	470000
5	200	620000
6	210	680000
7	310	1600000
8	330	1900000
9	370	2570000
10	400	3300000

Tabelle 4: Datensatz D_{houses} zu Beispiel 10. Die Zahlen sind rein fiktiv.

setzt (die Daten sind rein fiktiv). Die Abbildung visualisiert auch das lineare Modell f, das optimal an die Daten angepasst ist (hier wurde der gesamte Datensatz D_{houses} als Trainingsdatensatz benutzt). Wie man sieht, ist eine Anpassung an die Daten nur sehr schlecht möglich, da diese offensichtlich nicht in einem linearen Zusammenhang zueinander stehen.

Die Einbeziehung nichtlinearer Zusammenhänge zwischen Merkmalen und der Zielvariablen kann bei der linearen Regression realisiert werden, indem in einem Vorbereitungsschritt die Beispiele um zusätzliche (nichtlineare) Merkmale ergänzt werden, die aus den schon existierenden Merkmalen berechnet werden.

Beispiel 11. Eine visuelle Inspektion der Datenpunkte in Abbildung 9 legt nahe, dass ein quadratischer Zusammenhang zwischen der Fläche des Hauses und dem Kaufpreis besteht. Wir erweiterten aus diesem Grund den

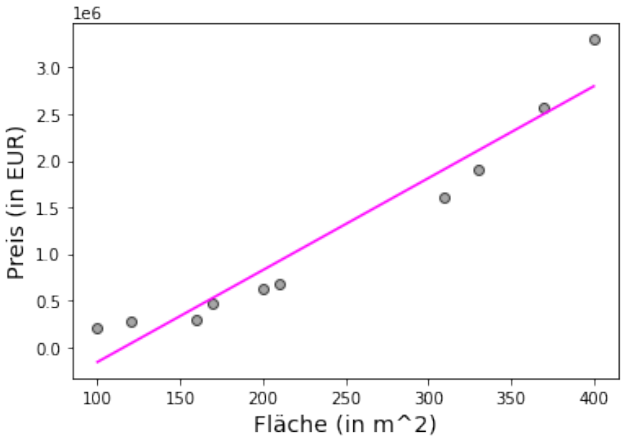


Abbildung 9: Optimal angepasstes lineares Modell für den Datensatz D_{houses}.

Datensatz D_{houses} um ein weiteres Merkmal, dass das Quadrat des Merkmals „Fläche“ ist, d. h. jedes Beispiel des Datensatzes mit Merkmalsausprägung x des Merkmals „Fläche“ erhält ein weiteres Merkmal „Fläche^2“ mit Ausprägung x^2. Der erweiterte Datensatz D'_{houses} ist in Tabelle 5 dargestellt. Das neue Merkmal „Fläche^2“ hat keine intuitive Bedeutung und deshalb haben wir diesem Merkmal in Tabelle 5 auch keine Einheit zugewiesen.

Die eigentliche formale Maschinerie der linearen Regression kann nun auf einen erweiterten Datensatz genauso angewendet werden wie auf dem originalen Datensatz.

Beispiel 12. Wir lösen das Optimierungsproblem aus Gleichung (4) bezüglich D'_{houses} und erhalten Parameter θ, die folgender linearen Funktion f_{houses} entsprechen:

$$f_{\text{houses}}(x_1, x_2) \approx -7297.676x_1 + 34.189x_2 + 647307.78$$

Nr.	Fläche (in m^2)	Fläche^2	Preis (in EUR)
1	100	10000	210000
2	120	14400	270000
3	160	25600	290000
4	170	28900	470000
5	200	40000	620000
6	210	44100	680000
7	310	96100	1600000
8	330	108900	1900000
9	370	136900	2570000
10	400	160000	3300000

Tabelle 5: Erweiterter Datensatz D'_{houses} zu Beispiel 11.

wobei x_1 eine Ausprägung des Merkmals „Fläche“ und x_2 eine Ausprägung des Merkmals „Fläche^2“ ist. Da $x_2 = x_1^2$ für jedes Beispiel ist, können wir f_{houses} auch vereinfacht darstellen durch

$$f_{\text{houses}}(x_1) \approx -7297.676x_1 + 34.189x_1^2 + 647307.78$$

und somit ist f auch keine lineare Funktion mehr, sondern ein Polynom. Abbildung 10 visualisiert f_{houses}.

Im vorherigen Beispiel haben wir nur ein einziges polynomielles Merkmal ergänzt. Bei größeren Regressionsproblemen (oder auch Klassifikationsproblemen) ist es meist nicht klar, welche polynomiellen Merkmale hinzugefügt werden und bis zu welchem Grade. Üblicherweise wählt man einen Maximalgrad als Parameter aus, fügt alle möglichen Polynome bis zu diesem Grad als zusätzliche Merkmale ein und „hofft“, dass überflüssige Merkmale bei der Optimierung ignoriert werden (also dass der entsprechende Parameter in θ nahe bei 0 liegt; wir werden

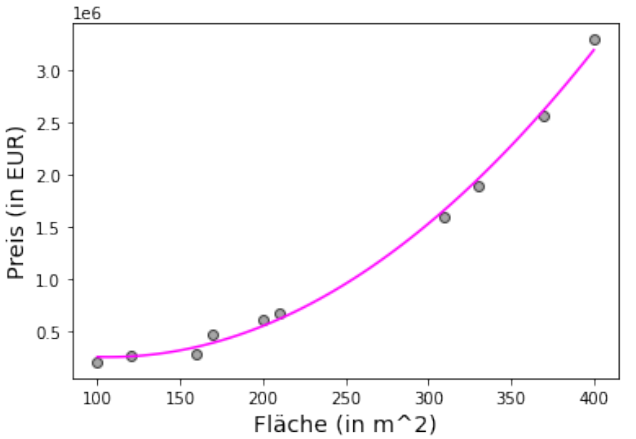


Abbildung 10: Optimal angepasstes „lineares“ Modell für den Datensatz D'_{houses}.

uns mit dieser Problematik aber in den nächsten beiden Abschnitten noch genauer beschäftigen).

Beispiel 13. Angenommen wir haben einen Datensatz D mit Beispielen der Form $(x^{(i)}, y^{(i)})$, wobei

$$x^{(i)} = (x_1^{(i)}, x_2^{(i)}, x_3^{(i)}) \in \mathbb{R}^3$$

und $y^{(i)} \in \mathbb{R}$. Die *polynomielle Merkmalserweiterung* von D für den Maximalgrad 2 besteht entsprechend aus den Beispielen $(\hat{x}^{(i)}, y^{(i)})$ mit

$$\hat{x}^{(i)} = (x_1^{(i)}, x_2^{(i)}, x_3^{(i)}, (x_1^{(i)})^2, x_1^{(i)}x_2^{(i)}, x_1^{(i)}x_3^{(i)}, (x_2^{(i)})^2, x_2^{(i)}x_3^{(i)}, (x_3^{(i)})^2)$$

Der Datensatz D'_{houses} aus Beispiel 12 ist also eine polynomielle Merkmalserweiterung für den Maximalgrad 2. Da dort die Beispiele nur über ein Merkmal verfügten, musste auch nur ein weiteres polynomielles Merkmal bei der Erweiterung hinzugefügt werden. Beispiel 13 macht

deutlich, dass bei mehreren Merkmalen bei der polynomiellen Erweiterung eine ganze Reihe zusätzlicher Merkmale hinzukommen können.

Lineare Regression auf polynomiellen Erweiterungen nennt man auch direkt *polynomielle Regression*. Es ist wichtig hervorzuheben, dass wir auch nach Einführung polynomieller Merkmale zur Berechnung des am besten angepassten Modells immer noch Gleichung (4) benutzen und damit ein *lineares Modell* berechnen. Dieses Modell ist allerdings nur linear bezüglich des erweiterten Merkmalsraums und erscheint polynomiell bei Projektion auf den Ursprungsraum (wie in Abbildung 10).

2.1.4 Über- und Unteranpassung

Die Möglichkeit, zusätzliche Merkmale durch beliebige Verknüpfung vorhandener Merkmale zu generieren, erlaubt es, beliebig komplexe Modelle zu konstruieren und damit beliebig komplexe Funktionen zu approximieren. Dies führt zu der Frage, warum man statt eines linearen Modells nicht direkt ein maximal komplexes Modell nimmt, das damit auch beliebig genau an den Trainingsdatensatz angepasst werden kann. Zunächst ergeben sich dadurch ressourcenspezifische Probleme, da das Lernen unter Umständen signifikant mehr Zeit benötigt. Viel signifikanter dabei ist allerdings das Problem der *Überanpassung* (engl. *overfitting*), d. h., das gelernte Modell ist aufgrund seiner Komplexität so stark an die Trainingsdaten angepasst, dass es nicht mehr gut auf ungesehene Daten generalisiert. Das dazu entgegengesetzte Problem ist die sogenannte *Unteranpassung* (engl. *underfitting*), d. h., das gelernte Modell ist nicht ausdrucksstark genug, um sowohl die Trainings- als auch die Testdaten vernünftig zu modellieren. Das Erkennen von Unter- und Überanpassung eines Modells und das richtige Abwägen

zwischen diesen beiden Aspekten ist eine der wichtigsten Aufgaben in der Entwicklung eines adäquaten Modells für eine bestimmte (überwachte) Lernaufgabe.

Beispiel 14. Betrachten wir den Datensatz D^* in Tabelle 6. Abbildung 11 zeigt lineare Modelle für den Datensatz D^* und dessen polynomielle Erweiterungen bis Grad 9. Das Modell für $d = 1$ entspricht dabei dem normalen linearen Regressionsmodell auf dem Ursprungsdatensatz D^* und die Modelle der übrigen Abbildungen nehmen je ein höheres Polynom auf dem Merkmal x mit auf, bis zu x^9 in der letzten Abbildung, d. h. die Funktion f^9 in der letzten Abbildung ist von der Form

$$f^9(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \ldots + \theta_9 x^9$$

Die Komplexität der Modelle nimmt mit der Hinzunahme von Termen höherer Grade weiter zu und komplexere Modelle sind somit in der Lage, die Trainingsdaten beliebig genau zu approximieren. Offensichtlich ist das Modell für $d = 1$ zu einfach, um die Trainingsdaten korrekt zu erklären: dieses Modell ist *unterangepasst*. Auf der anderen Seite ist das Modell für $d = 9$ *überangepasst*. Es repräsentiert die Trainingsdaten zwar perfekt, ist allerdings nicht robust gegenüber möglichen Störungseinflüssen in den Trainingsdaten und modelliert die Daten in unintuitiver Weise. Gleiches gilt für die Modelle für $d = 4$ bis $d = 8$. Die beiden Modelle für $d = 2$ und $d = 3$ erscheinen am passendsten, auch wenn diese die Trainingsdaten nicht perfekt modellieren.

Das Problem der Abwägung von Über- und Unteranpassung wird auch *Verzerrung-Varianz-Dilemma* genannt (engl. *bias-variance tradeoff*). Im Allgemeinen ist der *Verzerrungsfehler* eines Modells (engl. *bias error*) der Fehler, der durch zu einfache Annahmen in der Modellstruktur

x	y
1	17
2	27
3	43
4	73
5	80
6	82
7	95
8	92
9	99
10	104

Tabelle 6: Datensatz D^* zu Beispiel 14.

entsteht und damit eventuell zu Unteranpassung führt. Der *Varianzfehler* eines Modells (engl. *variance error*) ist der Fehler, der durch zu starke Anpassung an potentiell verzerrte Trainingsdaten entsteht und damit eventuell zu Überanpassung führt.

Um ein Modell zu bestimmen, das weder zu stark unterangepasst noch zu stark überangepasst ist, hilft es, die Verläufe der Kostenfunktionswerte (oder des Bestimmtheitsmaßes) bei Trainings- und Testdaten mit steigender Modellkomplexität zu betrachten. Abbildung 12 zeigt typische Verläufe des Bestimmtheitsmaßes in diesem Kontext. Wie wir bereits zuvor gesehen haben, nimmt die Bestimmtheit eines Modells mit steigender Komplexität auf den Trainingsdaten zu: je ausdrucksstärker das Modell ist, desto besser wird es an die Trainingsdaten angepasst. Bei den Testdaten ist der Verlauf etwas komplexer. Üblicherweise nimmt die Bestimmtheit zunächst zu: solange das Modell unterangepasst ist, können weder Trainings- noch Testdaten gut modelliert werden, aber

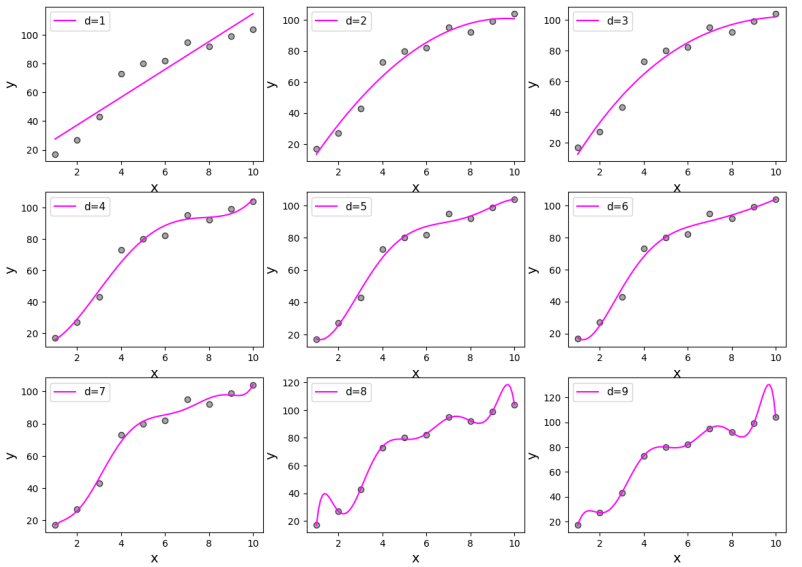


Abbildung 11: Optimal angepasste lineare Modelle für den Datensatz D^* (links oben) und dessen polynomielle Erweiterungen bis Grad 9.

je näher man an das „korrekte“ Modell kommt, desto besser werden insbesondere auch die Vorhersagen auf den Testdaten. Steigt die Modellkomplexität aber weiter, so wird das Modell überangepasst und die Vorhersagequalität auf den Testdaten sinkt wieder. Die optimale Modellkomplexität liegt also in diesem Fall am Scheitelpunkt des Kurvenverlaufs zu den Testdaten. Hier ist die Bestimmtheit sowohl bei den Trainings- als auch bei den Testdaten relativ hoch und beide Werte relativ nah beieinander. Modelle, deren Komplexität links davon liegen sind unterangepasst und Modelle, die rechts von diesem Punkt liegen, sind überangepasst.

Beispiel 15. Wir führen Beispiel 14 fort und benutzen

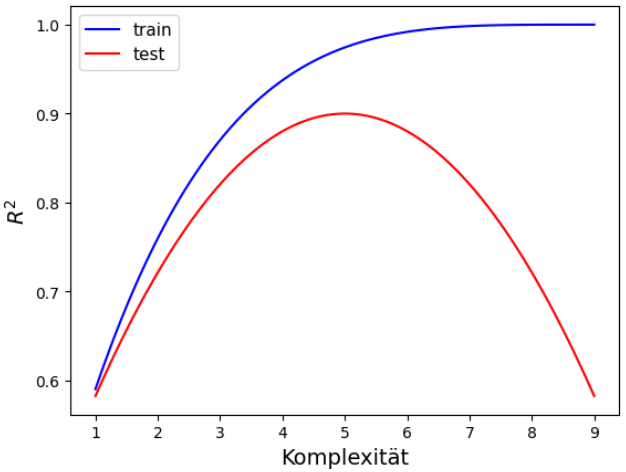


Abbildung 12: Typischer Zusammenhang der Bestimmtheit eines Regressors auf Trainings- und Testdaten bei steigender Komplexität.

weiterhin Datensatz D^* als Trainingsdaten. Wir betrachten noch einen weiteren Datensatz D^*_{test}, den wir als Testdatensatz benutzen und der zusammen mit Datensatz D^* in Abbildung 13 visualisiert ist. Es sollte offensichtlich sein, dass D^* und D^*_{test} derselben Verteilung entstammen. Die Berechnung der einzelnen Bestimmtheitswerte auf D^* und D^*_{test} bezüglich der verschiedenen Regressionsmodelle aus Beispiel 14 ergibt die in Abbildung 14 dargestellten Verläufe. Die Verläufe bestätigen den Eindruck aus Beispiel 14. Das Modell mit $d = 1$ ist unterangepasst, wohingegen die Modelle mit $d = 4$ bis $d = 9$ überangepasst sind. Die Modelle mit $d = 2$ und $d = 3$ erscheinen optimal.

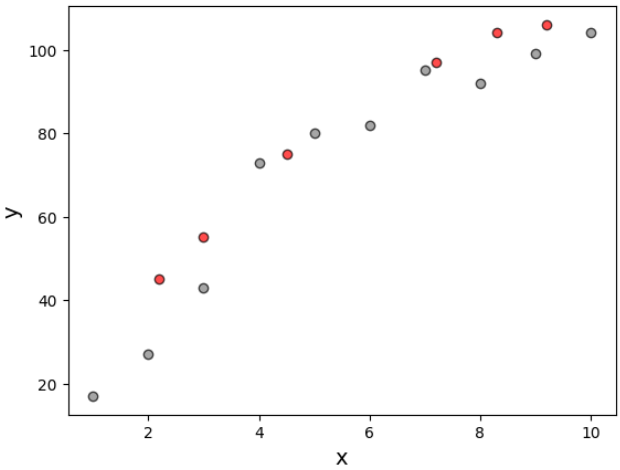


Abbildung 13: Trainingsdatensatz D^* (in grau) und Testdatensatz D^*_{test} (in rot) aus Beispiel 15.

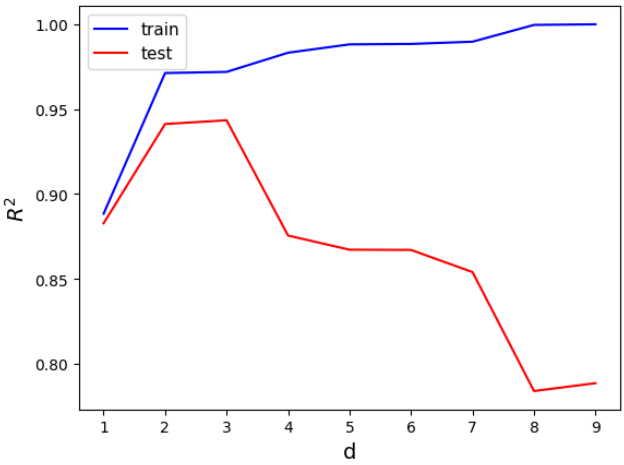


Abbildung 14: Bestimmtheit verschiedener polynomieller Regressionsmodelle auf Trainings- und Testdaten aus Beispiel 15.

2.1.5 Regularisierung

Eine Möglichkeit um bei der linearen Regression das Verzerrung-Varianz-Dilemma (semi-automatisch) zu lösen, ist die *Regularisierung*. Regularisierung beschreibt eine Technik, die dafür sorgt, dass ein zu komplexes Modell beim Lernvorgang bestraft wird.

Definition 5. Sei $L(D, f)$ eine beliebige Kostenfunktion mit $f : \mathbb{R}^n \to \mathbb{R}$. Eine Funktion[3] $R : (\mathbb{R}^n \to \mathbb{R}) \to \mathbb{R}^{\geq 0}$ heißt *Regularisierer* und für $\lambda \in \mathbb{R}$ mit $\lambda > 0$ heißt

$$L_{R,\lambda}(D, f) = L(D, f) + \lambda R(f)$$

die mit R und λ *regularisierte Kostenfunktion*.

Im allgemeinen soll ein Regularisierer R ein Maß für die Komplexität von f implementieren und durch die Einbeziehung des Terms $\lambda R(f)$ in die Kostenfunktion L wird bei Erlernen der Funktion f dessen Komplexität entsprechend berücksichtigt. Je nach Größe von λ (der *Regularisierungsparameter*) werden komplexere Funktionen mehr oder weniger stark bei der Minimierung von $L_{R,\lambda}(D, f)$ berücksichtigt. Schauen wir uns einen konkreten Regularisierer für die lineare Regression an.

Definition 6. Sei $\theta \in \mathbb{R}^{n+1}$ und $h_\theta : \mathbb{R}^n \to \mathbb{R}$ mit

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \ldots + \theta_n x_n$$

für alle $x = (x_1, \ldots, x_n)^T \in \mathbb{R}^n$ ein lineares Modell und $L(D, \theta)$ mit

$$L(D, \theta) = ||X_D \theta - y_D||^2$$

[3] $\mathbb{R}^{\geq 0}$ bezeichne die Menge der nicht-negativen reellen Zahlen.

die entsprechende Kostenfunktion (der quadratische Fehler). Der *Tikhonov-Regularisierer* $R_T : \mathbb{R}^n \to \mathbb{R}^{\geq 0}$ is definiert via

$$R_T(\theta_1, \ldots, \theta_n) = \sum_{i=1}^{n} \theta_i^2$$

und für $\lambda \in \mathbb{R}$ mit $\lambda > 0$

$$L_T(D, \theta) = \|X_D\theta - y_D\|^2 + \lambda \sum_{i=1}^{n} \theta_i^2$$

die entsprechende regularisierte Kostenfunktion.[4]

Die lineare Regression mit Kostenfunktion L_T nennt man allgemein auch *Ridge-Regression* (engl. *ridge regression*). Der Term $\lambda \sum_{i=1}^{n} \theta_i^2$ bestraft hohe Werte der einzelnen Elemente von θ und ist minimal, wenn $\theta_1 = \ldots = \theta_n = 0$ ist, also die Funktion h_θ eine Parallele zur ersten Achse darstellt. Solch eine Funktion beschreibt natürlich das einfachste Modell für eine gegebene Lernaufgabe. Der Term $\lambda \sum_{i=1}^{n} \theta_i^2$ wird umso größer, je mehr Elemente von θ ungleich 0 sind und je größer sie sind. Ist dies der Fall, so ist die gelernte Funktion h_θ umso komplexer, da viele Merkmale (insbesondere solche, die zusätzlich durch die in Abschnitt 2.1.3 angesprochenen Techniken hinzugefügt wurden) in die Anpassung einbezogen werden. Mit anderen Worten erzwingt die Einbeziehung des Terms $\lambda \sum_{i=1}^{n} \theta_i^2$ eine Fokussierung auf möglichst einfache Funktionen. Durch den Parameter λ wird gesteuert, ob die gelernte Funktion eher überangepasst wird (bei kleinen Werten von λ) oder eher unterangepasst wird (bei großen Werten von λ). Das Finden eines geeignetes Wertes von λ ist hier also die Kernaufgabe.

[4] Beachten Sie, dass der konstante Term θ_0 des linearen Modells nicht bei der Regularisierung beachtet wird.

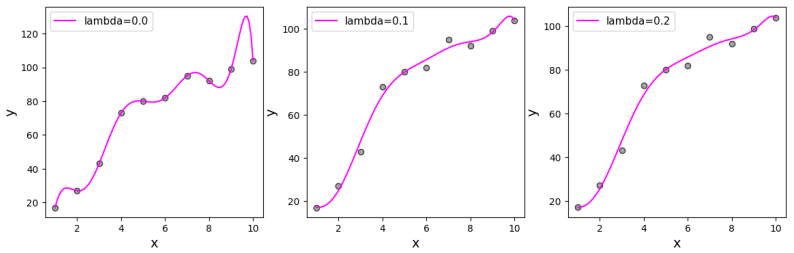


Abbildung 15: Ridge-Regressionsmodelle aus Beispiel 16.

Beispiel 16. Abbildung 15 zeigt drei Ridge-Regressionsmodelle für den mit Maximalgrad 9 erweiterten Datensatz D^* aus Tabelle 6 mit drei verschiedenen Parametern $\lambda \in \{0, 0.1, 0.2\}$. Das Modell für $\lambda = 0$ ist somit das normale polynomielle Regressionsmodell und identisch mit dem letzten Modell in Abbildung 11. Bei den anderen Modellen sieht man einen eindeutigen Trend der Überanpassung des ersten Modells entgegenzuwirken. Mit anderen Worten, obwohl die anderen Modelle über den selben Merkmalen definiert sind, werden bei der Optimierung die Parameter vieler (hoch-polynomieller) Merkmale sehr klein gewählt, um den Regularisierungsterm nicht zu groß werden zu lassen.

2.2 Logistische Regression

Nachdem wir uns im vorherigen Unterkapitel mit einem Regressionsproblem beschäftigt haben, betrachten wir nun das Problem der *Klassifikation*.

2.2.1 Klassifikation und Logistische Regression

Prinzipiell ist das Lernproblem der Klassifikation ähnlich wie bei der Regression. Wir haben einen Trainingsdatensatz $D = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ als Eingabe und das Ziel ist das Erlernen einer Funktion f mit $f(x^{(i)}) \approx y^{(i)}$, $i = 1, \ldots, m$ (und der allgemeinen Anforderung, dass f auch gut auf ungesehene Beispiele generalisiert). Im Gegensatz zur Regression ist der Wertebereich der Zielvariablen $y^{(i)}$ allerdings *diskret*, üblicherweise sogar *endlich*, und oft auch nur *binär* ($y^{(i)} \in \{0, 1\}$). Bei der Klassifikation besteht also die Aufgabe, einen Datenpunkt einer bestimmten *Klasse* zuzuweisen.

Beispiel 17. Sie sind Obstproduzent und möchten Ihren jährlichen Umsatz maximieren, indem Sie Ihr Produkt (den fiktiven *Rapidapfel*) so früh wie möglich ernten. Sie nehmen eine Reihe von Proben nach verschiedenen Tagen und testen, ob der Rapidapfel *genießbar* (Klasse 1) oder (noch) *ungenießbar* (Klasse 0) ist. Tabelle 7 und Abbildung 16 zeigen die erhobenen Daten D_{rapid}.

Beispiel 18. Sie sind auf der Suche nach einer neuen Wohnung, die möglichst groß und nahe zum Arbeitsplatz (AP) liegt. Tabelle 8 und Abbildung 17 zeigen Ihre bisherigen Wohnungsbesichtigungen und ob die Wohnung für Sie in Ordnung (OK= 1) oder nicht (OK= 0) war (Datensatz $D_{\text{apartment}}$).

Die Unterscheidung zwischen Regressions- und Klassifikationsproblem (und damit auch die anzuwendende

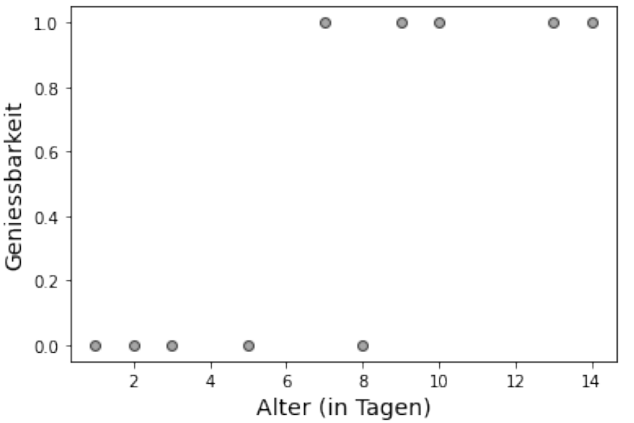


Abbildung 16: Datensatz D_{rapid} zu Beispiel 17 (visualisiert). Bitte beachten Sie, dass die Daten rein fiktiv sind.

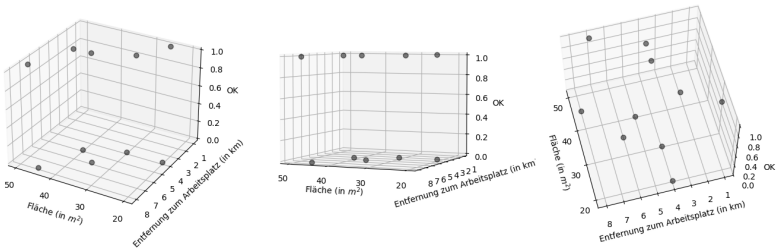


Abbildung 17: Datensatz $D_{\text{apartment}}$ zu Beispiel 18 (visualisiert). Bitte beachten Sie, dass die Daten rein fiktiv sind.

Nr.	Alter (in Tagen)	Genießbar (= 1), Ungenießbar (= 0)
1	5	0
2	14	1
3	10	1
4	7	1
5	2	0
6	1	0
7	9	1
8	8	0
9	3	0
10	13	1

Tabelle 7: Datensatz D_{rapid} zu Beispiel 17. Bitte beachten Sie, dass die Daten rein fiktiv sind.

Lernmethode) ist nicht immer eindeutig. Streng genommen war Beispiel 2 aus Unterkapitel 2.1 zur Notenbestimmung auch ein Klassifikationsproblem mit Klassen $\{1.0, 1.3, 1.7, \ldots, 3.7, 4.0, 5.0\}$. Im Gegensatz zu Beispiel 17 besitzen Noten jedoch eine natürliche *Ordnung* (1.0 ist besser als 1.3, 1.3 ist besser als 1.7, usw.). Bei Klassifikationsproblemen haben die Klassen üblicherweise keine (eindeutige) Ordnung. Neben der einfachen binären Klassifikation in eine positive Klasse und eine negative Klasse (wie in Beispiel 17) ist ein weiteres klassisches Klassifikationsproblem die Klassifikation von Objekten in Bildern beispielsweise in „Auto“, „Flugzeug“ und „Schiff“. Wir werden uns mit der *Mehrklassenklassifizierung* (engl. *multi-class classification*) in Abschnitt 2.2.3 genauer beschäftigen.

Wir schauen uns nun ein einfaches Modell und den dazugehörigen Lernalgorithmus für binäre Klassifikati-

Nr.	Fläche (in m^2)	Entfernung zum AP (in km)	OK
1	40	4	1
2	30	3	1
3	45	8	0
4	35	6	0
5	20	4	0
6	50	7	1
7	30	4	0
8	45	4	1
9	40	5	0
10	25	1	1

Tabelle 8: Datensatz $D_{\text{apartment}}$ zu Beispiel 18. Bitte beachten Sie, dass die Daten rein fiktiv sind.

on an: die *logistische Regression*. Trotz ihres Namens ist die logistische Regression eine Methode zur Klassifikation (und nicht zur Regression). Leider ist die Bezeichnung historisch bedingt und auch heutzutage üblich, wir werden sie deshalb in gleicher Weise verwenden. Das Modell der logistischen Regression ist die *Sigmoid*-Funktion (oder auch *Logit*-Funktion).

Definition 7. Sei $\theta \in \mathbb{R}^{n+1}$. Die *Sigmoid-Funktion* h_θ^{logit} auf $\mathbb{R}^n$ ist die Funktion $h_\theta^{\text{logit}} : \mathbb{R}^n \to (0,1)$ mit

$$h_\theta^{\text{logit}}(x) = \frac{1}{1 + e^{-(\theta_0 + \theta_1 x_1 + \ldots + \theta_n x_n)}}$$

Abbildung 18 visualisiert die Sigmoid-Funktion für den einfachsten Parameter $\theta = (0,1)$. Die Funktion h_θ^{logit} besteht prinzipiell aus dem Einsetzen eines linearen Mo-

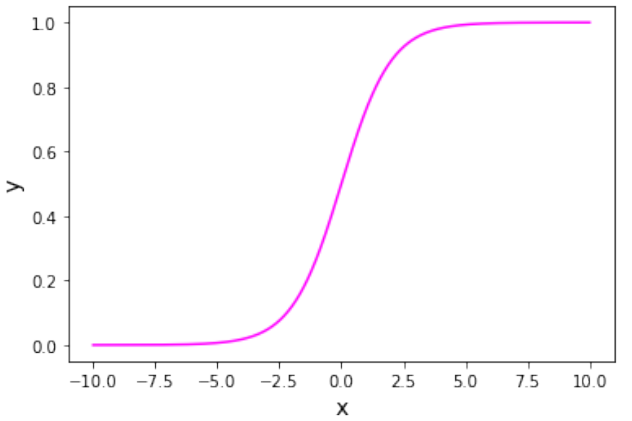


Abbildung 18: Die Sigmoid-Funktion $h^{\text{logit}}_{(0,1)}(x)$

dells h_θ in die Funktion $g(z) = \frac{1}{1+e^{-z}}$ und bildet diese damit in den Zielbereich $(0,1)$ ab.

Die eigentliche Klassifikation mit h^{logit}_θ geschieht dann mit einem Vergleich zum Schwellenwert 0.5. Für eine beliebige Funktion $f : \mathbb{R}^n \to [0,1]$ definiere:

$$\text{clf}_f(x) = \begin{cases} 1 & \text{falls } f(x) \geq 0.5 \\ 0 & \text{falls } f(x) < 0.5 \end{cases}$$

Wie bei der linearen Regression besteht die Lernaufgabe nun darin, den Parametervektor θ so zu ermitteln, dass h^{logit}_θ (bzw. $\text{clf}_{h^{\text{logit}}_\theta}$) einen gegebenen Datensatz gut erklärt. Wie bei der linearen Regression formulieren wir dies als ein Minimierungsproblem bezüglich einer Kostenfunktion.

Definition 8. Sei D ein Datensatz und $f : \mathbb{R}^n \to (0,1)$ eine beliebige Funktion. Die *logistische Kostenfunktion* L^{logit}

von f bzgl. D ist definiert durch[5]

$$L^{\text{logit}}(D, f) = -\sum_{i=1}^{m} y^{(i)} \ln f(x^{(i)}) + (1 - y^{(i)}) \ln(1 - f(x^{(i)}))$$

Die einzelnen Summanden der Funktion L^{logit} bestehen aus zwei Termen ($y^{(i)} \ln f(x^{(i)})$ und $(1 - y^{(i)}) \ln(1 - f(x^{(i)}))$) von denen stets wegen $y^{(i)} \in \{0, 1\}$ nur einer ungleich Null ist. Für $y^{(i)} = 1$ reduziert sich der entsprechende Summand zu $\ln f(x^{(i)})$. Ist $f(x^{(i)}) \approx 1$ (stimmt also die Funktion f auf $x^{(i)}$ mit der Grundwahrheit $y^{(i)}$ überein), so entfällt der Summand wegen $\ln 1 = 0$ vollständig (das Beispiel ist korrekt klassifiziert). Tendiert hingegen $f(x^{(i)})$ gegen 0, so tendiert $\ln f(x^{(i)})$ gegen $-\infty$ und unter Betrachtung des dem Summationszeichen vorangestellten $-$ gegen $+\infty$, die Fehlklassifikation bewirkt also eine Erhöhung der Kostenfunktion. Bei der logistischen Regression suchen wir Parameter θ, so dass h_θ^{logit} optimal an den Trainingsdatensatz D angepasst ist, d. h., wir suchen eine Lösung für das folgende Optimierungsproblem:

$$\min_{\theta} L^{\text{logit}}(D, h_\theta^{\text{logit}}) \tag{5}$$

Das Optimierungsproblem (5) ist *konvex* und verfügt daher über eine eindeutige Lösung. Mithilfe numerischer Methoden wie *Gradient Descent* kann es auch effizient gelöst werden.

[5] Wir benutzen hier zur Definition der logistischen Kostenfunktion den natürlichen Logarithmus ln, da dies manche Rechnungen mit der Exponentialfunktion e^x vereinfacht. Prinzipiell kann ln aber durch den Logarithmus $\log_b$ zu jeder beliebigen Basis b ersetzt werden, ohne das globale Verhalten von L^{logit} zu ändern.

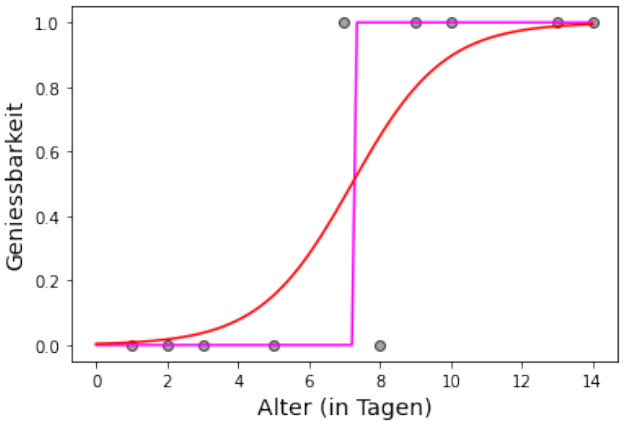


Abbildung 19: Optimal angepasstes logistisches Modell f (in rot) und die zugehörige Klassifikationsfunktion clf_f (in magenta) für den Datensatz D_{rapid}.

Beispiel 19. Wir setzen Beispiel 17 fort. Das optimale logistische Modell in diesem Beispiel ist gegeben durch

$$f(z) \approx \frac{1}{1+e^{5.559-0.771z}}$$

Die Funktion f ist visualisiert in Abbildung 19.

Beispiel 20. Wir setzen Beispiel 18 fort. Das optimale logistische Modell in diesem Beispiel ist gegeben durch

$$g(y,z) \approx \frac{1}{1+e^{3.763-0.26y+1.232z}}$$

Die Funktion g ist visualisiert in Abbildung 20. Abbildung 21 zeigt eine 2D-Projektion der Daten $D_{\text{apartment}}$ und die Klassifikationsgrenze der Klassifikationsfunktion clf_g. Alle Punkte oberhalb dieser Grenze werden als Klasse 0 (nicht ok) und alle Punkte unterhalb dieser Grenze werden als Klasse 1 (ok) klassifiziert.

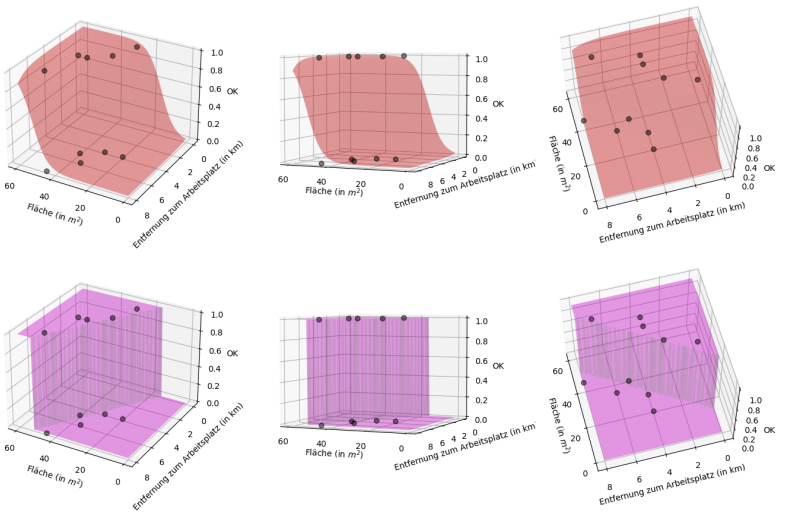


Abbildung 20: Optimal angepasstes logistisches Modell g (oben, in rot) und die zugehörige Klassifikationsfunktion clf_g (unten, in magenta) für den Datensatz $D_{\text{apartment}}$.

2.2.2 Evaluation

Um einen Klassifikationsalgorithmus zu evaluieren, geht man ähnlich vor wie bei der Regression (siehe dazu Abschnitt 2.1.1). Insbesondere teilt man auch hier den zur Verfügung stehenden Datensatz D in einen *Trainingsdatensatz* D^{train} und einen *Testdatensatz* D^{test} auf. Den Trainingsdatensatz D^{train} benutzt man dann zum Erlernen des Klassifikators und den Testdatensatz D^{test} zur Evaluierung. Für die Evaluierung eines Regressionsalgorithmus haben wir in Abschnitt 2.1.1 das *Bestimmtheitsmaß* benutzt, das prinzipiell den Abstand der erwarteten Funktionswerte zur Vorhersage als Gütemaß benutzt hat. Bei der Klassifikation ist eine Abstandsmessung allerdings nicht sinnvoll, da die Vorhersagen diskrete Klas-

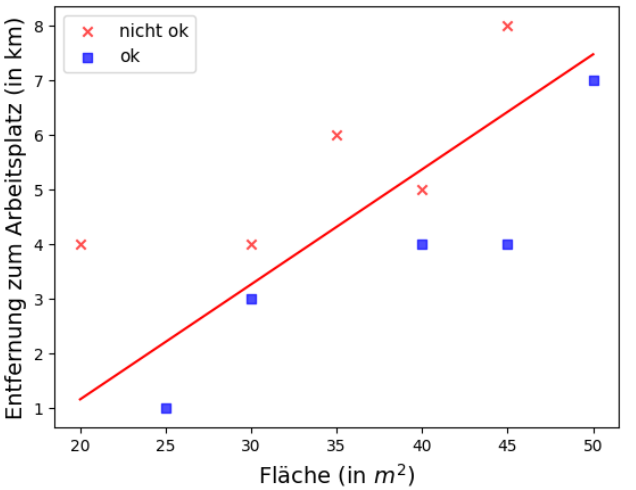


Abbildung 21: 2D-Projektion der Daten $D_{\text{apartment}}$ und Klassifikationsgrenze der Klassifikationsfunktion clf_g (in rot).

sen sind. Das einfachste Gütekriterium für einen Klassifikator ist die *Genauigkeit* (engl. *accuracy*). Wir betrachten hier nur den Fall der binären Klassifikation mit Klassen 0 und 1.

Definition 9. Sei $D = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ ein Datensatz und $\text{clf} : \mathbb{R}^n \to \{0,1\}$ ein (binärer) Klassifikator. Das *Genauigkeitsmaß* acc von c bzgl. D ist definiert durch

$$\text{acc}(D,c) = \frac{1}{m}\sum_{i=1}^{m}(1 - |y^{(i)} - \text{clf}(x^{(i)})|)$$

Der Wert $\text{acc}(D,c)$ ist also einfach das Verhältnis der korrekt klassifizierten Beispiele zu allen Beispielen.

Beispiel 21. Wir setzen Beispiel 19 fort, siehe auch Abbildung 19, und evaluieren clf_f auf dem gesamten Datensatz

D_{rapid}. Wie Abbildung 19 zu entnehmen ist, klassifiziert clf_f alle bis auf 2 Beispiele korrekt (der Datenpunkt Nr. 8 wird fälschlicherweise als 1 und der Datenpunkt Nr. 4 wird fälschlicherweise als 0 klassifiziert, siehe Tabelle 7). Da 8 von 10 Beispielen korrekt klassifiziert wurden erhalten wir

$$\text{acc}(D_{\text{rapid}}, \text{clf}_f) = \frac{8}{10} = 0.8$$

Beispiel 22. Wir setzen Beispiel 20 fort, siehe auch Abbildung 20, und evaluieren clf_g auf dem gesamten Datensatz $D_{\text{apartment}}$. Wie Abbildung 20 zu entnehmen ist, klassifiziert clf_g alle bis auf 1 Beispiel korrekt (der Datenpunkt Nr. 9 wird fälschlicherweise als 1 klassifiziert, siehe Tabelle 8). Da 9 von 10 Beispielen korrekt klassifiziert wurden erhalten wir

$$\text{acc}(D_{\text{apartment}}, \text{clf}_g) = \frac{9}{10} = 0.9$$

Das Genauigkeitsmaß hat den Nachteil, dass es bei ungleicher Klassenrepräsentation eine falsche Einschätzung der Qualität eines Klassifikators liefern kann. Betrachten wir dazu das Beispiel der Klassifikation eines Tumors als *bösartig* (Klasse 1) oder *gutartig* (Klasse 0), beispielsweise aus Computertomographiebildern. Üblicherweise hat man in diesem Szenario signifikant mehr Trainings- und Testdaten für die Klasse 0. Hat man beispielsweise einen Testdatensatz D_{tumor}, der aus insgesamt 100 Beispielen besteht, wobei davon 99 Beispiele der Klasse 0 zugehörig sind und 1 Beispiel der Klasse 1 zugehörig ist, so erreicht der triviale Klassifikator $\text{clf}_{trivial}$ definiert als $\text{clf}_{trivial} = 0$ (der also stets Gutartigkeit des Tumors vorhersagt, unabhängig von der tatsächlichen Beobachtung) eine Genauigkeit von 0.99. Um eine bessere Einschätzung eines Klassifikators zu erhalten betrachtet

	$y = 1$	$y = 0$
$\text{clf}_f = 1$	4	1
$\text{clf}_f = 0$	1	4

Tabelle 9: Konfusionsmatrix von clf_f bzgl. D_{rapid}.

man zusätzlich oft die sogenannte *Konfusionsmatrix* (engl. *confusion matrix*).

Definition 10. Sei $D = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ ein Datensatz und $\text{clf} : \mathbb{R}^n \to \{0, 1\}$ ein (binärer) Klassifikator. Definiere

$$TP(D, \text{clf}) = |\{i \mid y^{(i)} = 1, \text{clf}(x^{(i)}) = 1\}|$$
$$TN(D, \text{clf}) = |\{i \mid y^{(i)} = 0, \text{clf}(x^{(i)}) = 0\}|$$
$$FP(D, \text{clf}) = |\{i \mid y^{(i)} = 0, \text{clf}(x^{(i)}) = 1\}|$$
$$FN(D, \text{clf}) = |\{i \mid y^{(i)} = 1, \text{clf}(x^{(i)}) = 0\}|$$

Die *Konfusionsmatrix* von clf bzgl. D stellt die vier obigen Werte tabellarisch wie folgt dar:

	$y = 1$	$y = 0$
$\text{clf} = 1$	$TP(D, \text{clf})$	$FP(D, \text{clf})$
$\text{clf} = 0$	$FN(D, \text{clf})$	$TN(D, \text{clf})$

Den Wert $TP(D, c)$ nennt man *Richtig-positiv-Wert* (engl. *true positives*), den Wert $TN(D, c)$ *Richtig-negativ-Wert* (engl. *true negatives*), den Wert $FP(D, c)$ *Falsch-positiv-Wert* (engl. *false positives*) und den Wert $FN(D, c)$ *Falsch-negativ-Wert* (engl. *false negatives*).

Beispiel 23. Wir setzen Beispiel 21 fort. Die Konfusionsmatrix von clf_f bzgl. D_{rapid} ist in Tabelle 9 dargestellt.

Beispiel 24. Wir setzen Beispiel 22 fort. Die Konfusionsmatrix von clf_g bzgl. $D_{\text{apartment}}$ ist in Tabelle 10 dargestellt.

	$y = 1$	$y = 0$
$\text{clf}_g = 1$	5	1
$\text{clf}_g = 0$	0	4

Tabelle 10: Konfusionsmatrix von clf_g bzgl. $D_{\text{apartment}}$.

Die Konfusionsmatrix gibt einen kompakten Überblick über die Qualität eines Klassifikators über alle Klassen. Die vier Werte *TP*, *TN*, *FP* und *FN* kann man für die Bildung einer Reihe weiterer Evaluationsmaße verwenden.[6] Die drei wichtigsten weiteren Maße sind *Präzision* (engl. *precision*), *Sensitivität* (engl. *recall*) und das *F*1-Maß.

Definition 11. Sei $D = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ ein Datensatz und $\text{clf} : \mathbb{R}^n \to \{0, 1\}$ ein (binärer) Klassifikator. Definiere

$$\text{prec}(D, \text{clf}) = \frac{TP(D, \text{clf})}{TP(D, \text{clf}) + FP(D, clf)}$$

$$\text{rec}(D, \text{clf}) = \frac{TP(D, \text{clf})}{TP(D, \text{clf}) + FN(D, \text{clf})}$$

$$\text{F1}(D, \text{clf}) = \frac{2\text{prec}(D, \text{clf})\text{rec}(D, \text{clf})}{\text{prec}(D, \text{clf}) + \text{rec}(D, \text{clf})}$$

Das Maß $\text{prec}(D, \text{clf})$ (Präzision) misst also, wie viele der als positiv klassifizierten Beispiele tatsächlich positiv sind. Das Maß $\text{rec}(D, \text{clf})$ (Sensivität) misst, wie viele der positiven Beispiele tatsächlich als positiv klassifiziert wurden. Das F1-Maß ist das harmonische Mittel von

[6] Beachten Sie auch, dass das Genauigkeitsmaß über diese vier Werten definiert werden kann: $\text{acc}(D, \text{clf}) = \frac{TP(D,\text{clf})+TN(D,\text{clf})}{TP(D,\text{clf})+TN(D,\text{clf})+FP(D,\text{clf})+FN(D,\text{clf})}$

Präzision und Sensivität und ist üblicherweise ein geeignetes Maß für die Gesamtqualität eines Klassifikators.

Beispiel 25. Wir setzen Beispiel 23 fort. Hier gilt

$$\begin{aligned}\mathrm{prec}(D_{\mathrm{rapid}}, \mathrm{clf}_f) &= \frac{4}{5}\\ \mathrm{rec}(D_{\mathrm{rapid}}, \mathrm{clf}_f) &= \frac{4}{5}\\ \mathrm{F1}(D_{\mathrm{rapid}}, \mathrm{clf}_f) &= \frac{4}{5}\end{aligned}$$

Beispiel 26. Wir setzen Beispiel 24 fort. Hier gilt

$$\begin{aligned}\mathrm{prec}(D_{\mathrm{apartment}}, \mathrm{clf}_g) &= \frac{5}{6}\\ \mathrm{rec}(D_{\mathrm{apartment}}, \mathrm{clf}_g) &= \frac{5}{5} = 1\\ \mathrm{F1}(D_{\mathrm{apartment}}, \mathrm{clf}_g) &= \frac{10}{11}\end{aligned}$$

Betrachten wir abschließend noch einmal das Beispiel der Tumorklassifikation mit einem Testdatensatz D_{tumor} und dem Klassifikator $\mathrm{clf}_{trivial}$ definiert als $\mathrm{clf}_{trivial} = 0$. Betrachten wir weiterhin den alternativen trivialen Klassifikator $\mathrm{clf}_{trivial'}$ definiert als $\mathrm{clf}_{trivial'} = 1$ (dieser Klassifikator sagt also stets einen bösartigen Tumor vorher, unabhängig von der tatsächlichen Beobachtung). Die entsprechenden Konfusionsmatrizen sind in Tabelle 11 abgebildet. Hier haben wir

	$y=1$	$y=0$
$\text{clf}_{trivial}=1$	0	0
$\text{clf}_{trivial}=0$	1	99

	$y=1$	$y=0$
$\text{clf}_{trivial'}=1$	1	99
$\text{clf}_{trivial'}=0$	0	0

Tabelle 11: Konfusionsmatrizen von $\text{clf}_{trivial}$ (links) und $\text{clf}_{trivial'}$ (rechts) bzgl. D_{tumor}.

$$\begin{aligned}\text{prec}(D_{\text{tumor}}, \text{clf}_{trivial}) &= 1\\ \text{rec}(D_{\text{tumor}}, \text{clf}_{trivial}) &= 0\\ \text{F1}(D_{\text{tumor}}, \text{clf}_{trivial}) &= 0\end{aligned}$$

$$\begin{aligned}\text{prec}(D_{\text{tumor}}, \text{clf}_{trivial'}) &= \frac{1}{100}\\ \text{rec}(D_{\text{tumor}}, \text{clf}_{trivial'}) &= \frac{1}{1} = 1\\ \text{F1}(D_{\text{tumor}}, \text{clf}_{trivial'}) &= \frac{2}{101}\end{aligned}$$

Beachten Sie, dass $\text{prec}(D_{\text{tumor}}, \text{clf}_{trivial}) = \frac{0}{0}$ und damit strenggenommen undefiniert ist. Bei diesem Spezialfall wird die Präzision aber überlicherweise als 1 definiert. Bei beiden Klassifikatoren ist ersichtlich, dass einer der beiden Werte Präzision und Sensitivität sehr gut ist, der andere aber sehr schlecht. Das F1-Maß harmonisiert beide Werte und bewertet beide Klassifikatoren als sehr schlecht (wie gewünscht).

2.2.3 Mehrklassenklassifizierung

Klassifikationsprobleme mit mehr als zwei Klassen können auf einfache Art und Weise in eine Reihe von binären Klassifikationsproblemen transformiert werden. Seien c_1, $\ldots, c_k$ mit $k > 2$ die diskreten Klassen eines Klassifikationsproblems, dann lernt man k verschiedene Klassifikatoren

Nr.	Alter (in Jahren)	Gewicht (in kg)	Tierart
1	12	12	Hund
2	13	34	Hund
3	16	30	Wolf
4	14	15	Dingo
5	15	39	Wolf
6	15	13	Dingo
7	10	5	Hund
8	17	56	Wolf
9	14	11	Dingo
10	11	23	Hund

Tabelle 12: Datensatz D_{animals} zu Beispiel 27. Bitte beachten Sie, dass die Daten rein fiktiv sind.

$f_1, \ldots, f_k$, so dass f_i daran angepasst ist, Klasse c_i von Klasse nicht-$c_i = \{c_1, \ldots, c_k\} \setminus \{c_i\}$, für $i = 1, \ldots, k$, zu unterscheiden.

Beispiel 27. Wir klassifizieren Tiere in die drei Klassen *Hund* (Klasse 0), *Wolf* (Klasse 1) und Dingo (Klasse 2) und benutzen dazu die Merkmale *Alter* (im Sinne von erreichtes Lebensalter des Tieres) und *Gewicht*. Tabelle 12 zeigt einen beispielhaften Datensatz D_{animals} und Abbildung 22 visualisiert die Daten (beachten Sie, dass wir hier die Daten als zweidimensionale Projektion visualisieren).
Wir trainieren nun separat drei logistische Modelle für die Klassen *Hund*, *Wolf* und *Dingo*. Mit anderen Worten, wir erstellen drei Versionen des Datensatzes D_{animals} mit binären Klassen, nämlich $D^{\text{Hund}}_{\text{animals}}$, $D^{\text{Wolf}}_{\text{animals}}$ und $D^{\text{Dingo}}_{\text{animals}}$. Tabelle 13 zeigt beispielhaft den Datensatz $D^{\textit{Hund}}_{\text{animals}}$, bei der alle vorherigen *Hund*-Instanzen die Klasse 1 und alle übrigen Instanzen die Klasse 0 erhalten. Für dieser drei

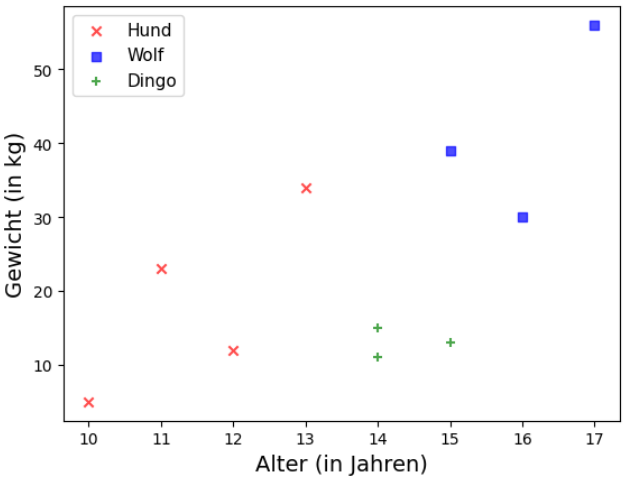


Abbildung 22: Datensatz D_{animals} zu Beispiel 27 (visualisiert). Bitte beachten Sie, dass die Daten rein fiktiv sind.

Datensätze erhalten wir die folgenden logistischen Modelle:

$$f_{\text{Hund}}(y,z) \approx \frac{1}{1+e^{-17.131+1.386y-0.0519z}}$$
$$f_{\text{Wolf}}(y,z) \approx \frac{1}{1+e^{20.99-0.928y-0.239z}}$$
$$f_{\text{Dingo}}(y,z) \approx \frac{1}{1+e^{11.707-1.194y+0.304z}}$$

Die Klassifikationsgrenzen dieser drei logistischen Modelle sind in Abbildung 23 dargestellt.

Ein neues Beispiel x klassifiziert man nun, indem man diesem Beispiel diejenige Klasse c_i zuweist, für die $f_i(x)$ maximal unter allen Klassifikatoren $f_1,\ldots,f_k$ ist. Für beliebige Funktionen $f_1,\ldots,f_k : \mathbb{R}^{n+1} \to [0,1]$ definiere:

$$\text{clf}_{f_1,\ldots,f_k}(x) = \arg_i \max f_i(x)$$

Nr.	Alter (in Jahren)	Gewicht (in kg)	Ist Hund?
1	12	12	1
2	13	34	1
3	16	30	0
4	14	15	0
5	15	39	0
6	15	13	0
7	10	5	1
8	17	56	0
9	14	11	0
10	11	23	1

Tabelle 13: Datensatz $D^{Hund}_{\text{animals}}$ zu Beispiel 27.

Für den (in der Realität selten auftretenden) Fall, dass zwei (oder mehr) Klassifikatoren f_i, f_j denselben maximalen Wert haben, sei $\text{clf}_{f_1,\ldots,f_k}$ beliebig (aber fix) definiert, beispielsweise indem der kleinere Index von i und j gewählt wird.

Beispiel 28. Wir setzen Beispiel 27 fort. Betrachten wir ein neues Beispiel

$$x = (12, 40)$$

so errechnen wir

$$f_{Hund}(12,40) \approx \frac{1}{1+e^{-17.131+1.386\cdot 12-0.0519\cdot 40}} \approx 0.929$$

$$f_{Wolf}(12,40) \approx \frac{1}{1+e^{20.99-0.928\cdot 12-0.239\cdot 40}} \approx 0.427$$

$$f_{Dingo}(12,40) \approx \frac{1}{1+e^{11.707-1.194\cdot 12+0.304\cdot 40}} \approx 0$$

und es ergibt sich

$$\text{clf}_{f_{Hund},f_{Wolf},f_{Dingo}}(x) = \text{Hund}$$

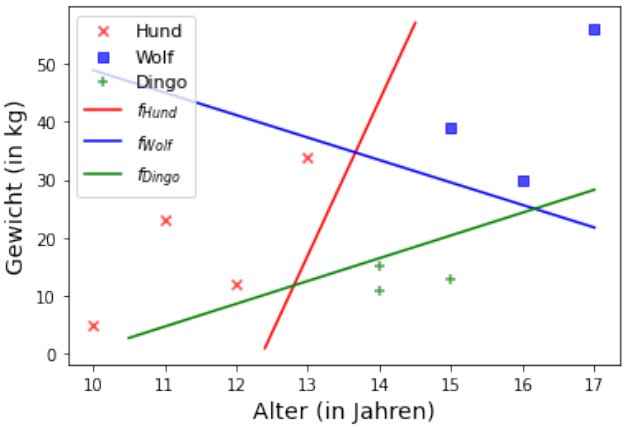


Abbildung 23: Klassifikationsgrenzen der optimal angepassten logistische Modelle für die drei Klassen *Hund*, *Wolf* und *Dingo* auf dem Datensatz D_{animals}.

Abbildung 24 visualisiert die Klassifikationsgrenzen des Klassifikators $\text{clf}_{f_{Hund}, f_{Wolf}, f_{Dingo}}$. Punkte im blauen Bereich werden als Hunde klassifiziert, Punkte im hellbraunen Bereich als Wölfe und Punkte im dunkelbraunen Bereich als Dingos.

Zur Evaluation von Modellen zur Klassifikation mit mehreren Klassen nutzen wir wie zuvor die Metriken Genauigkeit, Präzision, Sensitivität und F1. Da es bei mehr als zwei Klassen (und prinzipiell auch schon bei zwei Klassen) keine eindeutige „positive“ und keine eindeutige „negative“ Klasse geben muss, benutzt man hier üblicherweise klassenspezifische Varianten von Präzision, Sensitivität und F1. Bei k Klassen bekommen wir also jeweils k verschiedene Werte für Präzision, Sensitivität und F1. Für eine Klasse c schreiben wir dann prec_c, rec_c und F1_c für die entsprechende klassenspezifische Variante.

Beispiel 29. Angenommen, ein Klassifikator f klassifi-

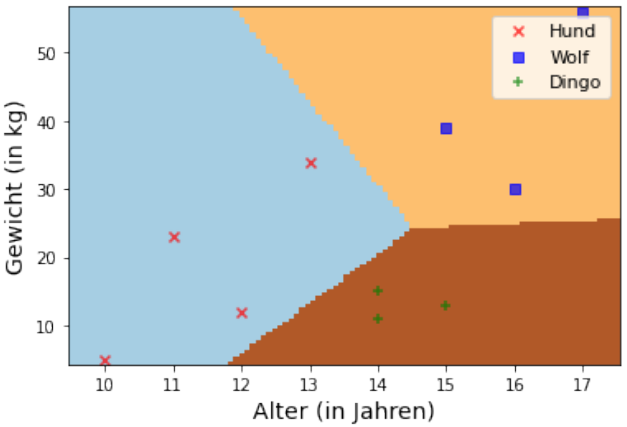


Abbildung 24: Klassifikationsgrenzen von $\text{clf}_{f_{Hund}, f_{Wolf}, f_{Dingo}}$ auf dem Datensatz D_{animals}.

ziert Bilder in Klassen *Hund*, *Katze* und *Maus* und ein Testdatensatz D ist gegeben als

$$D = \{(x^{(1)}, Hund), (x^{(2)}, Katze), (x^{(3)}, Katze), (x^{(4)}, Hund), \\ (x^{(5)}, Katze), (x^{(6)}, Maus), (x^{(7)}, Katze), (x^{(8)}, Hund)\}$$

Der Klassifikator f macht die folgenden Vorhersagen

$$\begin{array}{lll} f(x^{(1)}) = Hund & f(x^{(2)}) = Katze & f(x^{(3)}) = Katze \\ f(x^{(4)}) = Hund & f(x^{(5)}) = Katze & f(x^{(6)}) = Maus \\ f(x^{(7)}) = Katze & f(x^{(8)}) = Katze & \end{array}$$

Dann erhalten wir

$$\begin{aligned}
\text{prec}_{Hund}(D, f) &= \frac{2}{2} = 1 \\
\text{rec}_{Hund}(D, f) &= \frac{2}{3} \\
\text{F1}_{Hund}(D, f) &= \frac{4}{5}
\end{aligned}$$

$$\begin{aligned}
\text{prec}_{Katze}(D, f) &= \frac{4}{5} \\
\text{rec}_{Katze}(D, f) &= \frac{4}{4} = 1 \\
\text{F1}_{Katze}(D, f) &= \frac{8}{9}
\end{aligned}$$

$$\begin{aligned}
\text{prec}_{Maus}(D, f) &= 1 \\
\text{rec}_{Maus}(D, f) &= 1 \\
\text{F1}_{Maus}(D, f) &= 1
\end{aligned}$$

2.2.4 Nichtlineare Modelle

Die in Abschnitt 2.1.3 vorgestellten Techniken zur Merkmalserweiterung bei Regressionsproblemen können in gleicher Weise auch auf Klassifikationsprobleme und insbesondere bei der logistischen Regression angewendet werden.

Beispiel 30. In einer Produktionsstätte für Schrauben soll die Qualitätssicherung erhöht und deshalb ein Modell erlernt werden, das automatisch Schrauben außerhalb der Norm aussortiert. Dazu benutzen wir die Merkmale *Länge* und *Gewicht* der Schraube und die Klassen 1 (Qualität ausreichend) und 0 (Qualität nicht ausreichend.

Nr.	Gewicht (in g)	Länge (in mm)	Ok?
1	11	27	1
2	10	28	1
3	13	29	0
4	11	28	1
5	11	30	0
6	12	26	0
7	10	29	1
8	10	30	0
9	9	27	0
10	8	28	0

Tabelle 14: Datensatz D_{screws} zu Beispiel 30. Bitte beachten Sie, dass die Daten rein fiktiv sind.

Tabelle 14 zeigt einen beispielhaften Datensatz D_{screws} und Abbildung 25 visualisiert die Daten. Abbildung 25 zeigt auch die Klassifikationsgrenze des optimalen logistischen Modells f_{screws}, das in keinster Weise die Daten adäquat repräsentiert (es klassifiziert insbesondere alle Daten aus f_{screws} als „nicht ok“). Es gilt

$$\begin{aligned}
\text{prec}_0(D_{\text{screws}}, \text{clf}_{f_{screws}}) &= 0.6 \\
\text{rec}_0(D_{\text{screws}}, \text{clf}_{f_{screws}}) &= 1.0 \\
\text{F1}_0(D_{\text{screws}}, \text{clf}_{f_{screws}}) &= 0.75 \\
\text{prec}_1(D_{\text{screws}}, \text{clf}_{f_{screws}}) &= 1 \\
\text{rec}_1(D_{\text{screws}}, \text{clf}_{f_{screws}}) &= 0 \\
\text{F1}_1(D_{\text{screws}}, \text{clf}_{f_{screws}}) &= 0
\end{aligned}$$

Wie man an den Werte erkennen kann, klassifiziert f_{screws} sehr einseitig. Insbesondere zeigt die Diskrepanz zwischen F1_0 und F1_1 und der Wert $\text{F1}_1 = 0$, dass f_{screws} die Daten nicht adäquat repräsentiert.

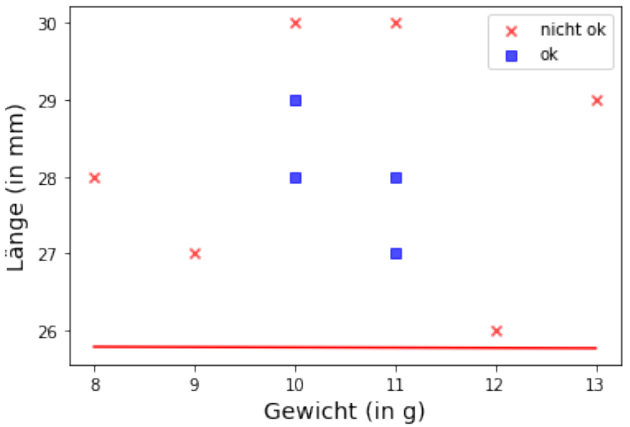


Abbildung 25: Datensatz D_{screws} zu Beispiel 30 (visualisiert) und die Klassifikationsgrenze des optimalen logistischen Modells (in rot). Bitte beachten Sie, dass die Daten rein fiktiv sind.

Betrachten wir nun zusätzlich zu den Merkmalen *Länge* und *Gewicht* alle polynomiellen Merkmale mit Maximalgrad 2, so erhalten wir den erweiterten Datensatz D^{ext}_{screws} dargestellt in Tabelle 15. Das optimale logistische Modell f^{ext}_{screws} ist weitaus besser an die Daten angepasst, die Klassifikationsgrenze zu f^{ext}_{screws} ist in Abbildung 26 dargestellt. Insbesondere gilt

$$\begin{aligned}
\text{prec}_0(D^{ext}_{\text{screws}}, \text{clf}_{f^{ext}_{screws}}) &\approx 0.83 \\
\text{rec}_0(D^{ext}_{\text{screws}}, \text{clf}_{f^{ext}_{screws}}) &\approx 0.83 \\
\text{F1}_0(D^{ext}_{\text{screws}}, \text{clf}_{f^{ext}_{screws}}) &\approx 0.83 \\
\text{prec}_1(D^{ext}_{\text{screws}}, \text{clf}_{f^{ext}_{screws}}) &= 0.75 \\
\text{rec}_1(D^{ext}_{\text{screws}}, \text{clf}_{f^{ext}_{screws}}) &= 0.75 \\
\text{F1}_1(D^{ext}_{\text{screws}}, \text{clf}_{f^{ext}_{screws}}) &= 0.7
\end{aligned}$$

Nr.	x_1 (=Gewicht)	x_2 (=Länge)	x_1^2	$x_1 x_2$	x_2^2	Ok?
1	11	27	121	297	729	1
2	10	28	100	280	784	1
3	13	29	169	377	841	0
4	11	28	121	308	784	1
5	11	30	121	330	900	0
6	12	26	144	312	676	0
7	10	29	100	290	841	1
8	10	30	100	300	900	0
9	9	27	81	243	729	0
10	8	28	64	224	784	0

Tabelle 15: Erweiterter Datensatz D^{ext}_{screws} mit polynomiellen Merkmalen des Maximalgrads 2 zu Beispiel 30.

Die Einbeziehung nicht-linearer Merkmale bei der logistischen Regression bringt dieselben Vor- und Nachteile wie bei der linearen Regression mit sich. Auf der einen Seite erlaubt die Einbeziehung nicht-linearer Merkmale die Repräsentation komplexerer Klassifikationsmodelle und damit die bessere Anpassung an Daten, die nicht geeignet mit einem linearen Modell angepasst werden können (wie im vorherigen Beispiel). Auf der anderen Seite erhöht die Einbeziehung weiterer Merkmale den Rechenaufwand beim Lernen und es entsteht wiederum das Problem der *Überanpassung* (siehe auch Abschnitt 2.1.3). Um letzterem entgegenzuwirken kann das logistische Modell um einen Regularisierungsterm erweitert werden (in gleicher Weise wie wir das bereits für die lineare Regression in Abschnitt 2.1.4 getan haben).

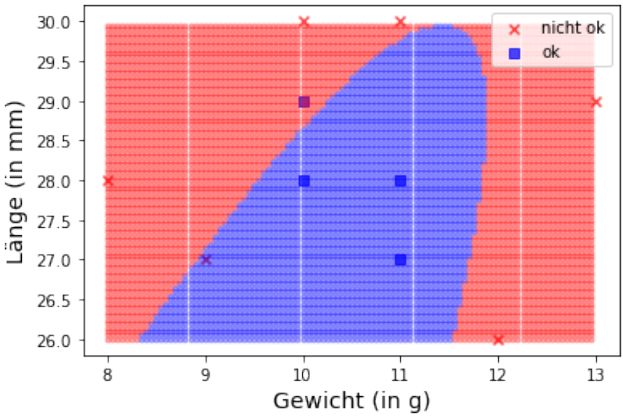


Abbildung 26: Klassifikationsgrenze des optimalen logistischen Modells f^{ext}_{screws} zu Datensatz D^{ext}_{screws} aus Beispiel 30.

2.3 Support Vector Machines

Die *Support Vector Machine*[7] (SVM) ist eine sehr robuste und oft genutzte Methode zur Klassifikation (wobei es auch Varianten zur Regression gibt).

2.3.1 Grundlagen

Die Kernidee von SVMs, die in ähnlicher Form auch schon bei der logistischen Regression vorhanden ist, ist die Bestimmung einer *Hyperebene* im Merkmalsraum, die die Beispiele der unterschiedlichen Klassen voneinander trennt. Abbildung 6 aus Unterkapitel 2.2 veranschaulicht die Intuition hinter SVMs schon sehr gut. Im 2-dimensionalen sind Hyperebenen einfache Geraden und die in Abbildung 6 aus Unterkapitel 2.2 visualisierten Klassifikationsgrenzen des logistischen Modells entsprechen somit diesen separierenden Hyperebenen. Während bei der logistischen Regression die Klassifikationsgrenzen nur *implizit* berechnet werden und der Lernalgorithmus der logistischen Regression auf der Minimierung der logistischen Kostenfunktion basiert, so wird bei der SVM die Klassifikationsgrenze *explizit* gelernt. Es wird dabei (bei binärer Klassifikation) diejenige Hyperebene gesucht, die die beiden Klassen voneinander trennt und dabei den größtmöglichen Abstand zu den Beispielen der beiden Klassen hat.

Definition 12. Sei $\theta \in \mathbb{R}^n$, $b \in \mathbb{R}$. Eine *Hyperebene*[8] $h^{\text{SVM}}_{\theta,b}$ im $\mathbb{R}^n$ bzgl. θ und b ist die Menge aller Punkte $x \in \mathbb{R}^n$, die

[7] Es gibt zu dem Begriff *Support Vector Machine* keine gebräuchliche deutsche Übersetzung.

[8] Beachten Sie bitte, dass bei der Definition einer Hyperebene der Parameter θ aus dem $\mathbb{R}^n$ kommt und nicht aus dem $\mathbb{R}^{n+1}$ wie in vorherigen Unterkapiteln. Die Rolle des konstanten Anteils (zuvor θ_0) übernimmt nun der Bezeichner b.

die Gleichung

$$h^{\text{SVM}}_{\theta,b}: \quad \theta^T x - b = 0$$

erfüllen.

Beispiel 31. Wir betrachten wieder den fiktiven *Rapidapfel* und möchten gerne eine automatische Qualitätskontrolle vor der Ausfuhr durchführen. Dazu nehmen wir eine Reihe von Proben und setzen den *Durchmesser* und das *Gewicht* der Äpfel mit dem Ergebnis der Qualitätskontrolle (*ok* und *nicht ok*) in Verbindung. Tabelle 16 und Abbildung 27 zeigen die erhobenen Daten D_{rapid2}. Auch gezeigt werden zwei separierende Geraden h_1 und h_2, jeweils definiert als die Punkte $x = (x_1, x_2)^T$, die die folgenden Gleichungen erfüllen:

$$h_1 = h^{\text{SVM}}_{(-0.9,-1),-140}: \quad -0.9x_1 - x_2 + 140 = 0$$
$$h_2 = h^{\text{SVM}}_{(-0.3,-1),-106}: \quad -0.3x_1 - x_2 + 106 = 0$$

Die eigentliche Klassifikation eines neuen Beispiels x geschieht nun, indem man berechnet, auf welcher Seite der Hyperebene $h^{\text{SVM}}_{\theta,b}$ sich x befindet und die entsprechende Klasse ausgibt. Für Parameter θ und b definiere:

$$\text{clf}_{\theta,b}(x) = \begin{cases} 1 & \text{falls } \theta^T x - b \geq 0 \\ 0 & \text{falls } \theta^T x - b < 0 \end{cases} \tag{6}$$

Wie zuvor besteht die Lernaufgabe nun darin, die Parameter θ und b so zu ermitteln, dass $h^{\text{SVM}}_{\theta,b}$ (bzw. $\text{clf}_{\theta,b}$) einen gegebenen Datensatz gut erklärt. Wir nehmen dazu zunächst an, dass die Klassen *linear separierbar* sind, d. h., dass es tatsächlich eine Hyperebene gibt, die die Klassen voneinander trennen kann (also kein Beispiel

Nr.	Gewicht (in g)	Durchmesser (in mm)	Ok?
1	61	92	1
2	58	93	1
3	58	87	0
4	60	91	1
5	61	89	1
6	56	86	0
7	57	88	0
8	58	92	1
9	57	85	0
10	59	92	1

Tabelle 16: Datensatz D_{rapid2} zu Beispiel 31. Bitte beachten Sie, dass die Daten rein fiktiv sind.

fehlklassifiziert wird). Den Fall der nicht-linear separierbaren Daten schauen wir uns in Abschnitt 2.3.2 an. Bei linear separierbaren Daten suchen wir dazu nach der eindeutig bestimmten Hyperebene, die die Klassen separiert und den (Euklidischen) Abstand zum nächsten Beispiel maximiert. Für einen Datensatz D sind die optimalen Parameter θ und b daher bestimmt durch die Lösung des folgenden Minimierungsproblems:

$$\begin{aligned} &\min \|\theta\| \qquad (7)\\ \text{so dass } &\theta^T x - b \geq 1 \quad \text{für alle } (x,1) \in D\\ &\theta^T x - b \leq -1 \text{ für alle } (x,0) \in D \end{aligned}$$

Bei der binären Klassifikation mit SVMs benennt man die beiden Klassen üblicherweise 1 und -1 (statt 1 und 0).[9]

[9] In diesem Fall wird (6) zu

$$\text{clf}_{\theta,b}(x) = \begin{cases} 1 & \text{falls } \theta^T x - b \geq 0 \\ -1 & \text{falls } \theta^T x - b < 0 \end{cases}$$

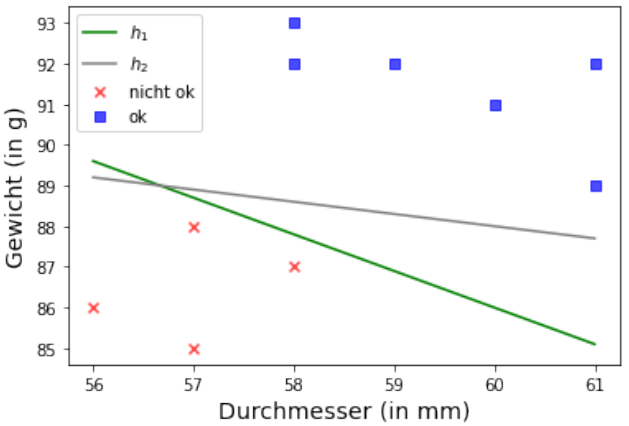


Abbildung 27: Datensatz D_{rapid2} zu Beispiel 31 (visualisiert) und zwei separierende Graden h_1 und h_2.

Mit dieser Notation lässt sich das Optimierungsproblem (7) auch einfacher durch

$$\begin{aligned} &\min \|\theta\| \\ &\text{so dass } y(\theta^T x - b) \geq 1 \text{ für alle } (x, y) \in D \end{aligned} \tag{8}$$

ausdrücken.

Beispiel 32. Wir setzen Beispiel 31 fort. Die optimale SVM für dieses Beispiel ist gegeben durch

$$h_{\text{opt}} \approx h^{\text{SVM}}_{(-0.4,-0.4),-59.01}: \quad -0.4x_1 - 0.4x_2 + 59.01 = 0$$

Die Funktion h_{opt} ist visualisiert in Abbildung 28.

Schauen wir uns die Konstruktion der optimalen separierenden Hyperebene etwas genauer an. Abbildung 29 zeigt einen zweidimensionalen Datensatz (mit weißen

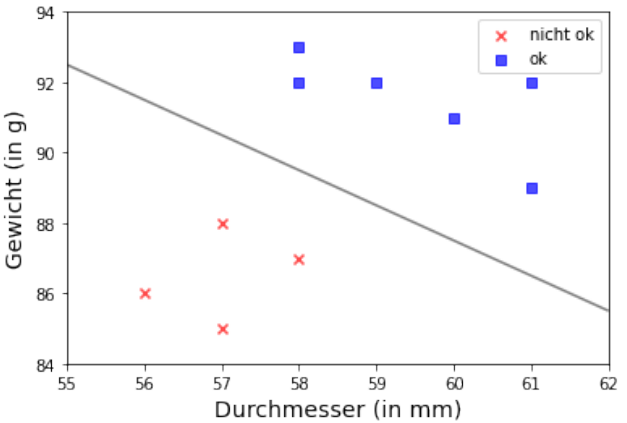


Abbildung 28: Optimal angepasste SVM h_{opt} für Datensatz D_{rapid2} zu Beispiel 32.

bzw. schwarzen Punkten, die die beiden Klassen symbolisieren), sowie die optimal angepasste Hyperebene h_0 mit

$$h_0 = h^{\text{SVM}}_{\theta,b}: \quad \theta^T x - b = 0$$

Weiterhin dargestellt sind zwei zu h_0 parallele Hyperebenen h_1 und h_{-1}, die definiert sind durch

$$\begin{aligned} h_1: &\quad \theta^T x - b = 1 \\ h_{-1}: &\quad \theta^T x - b = -1 \end{aligned}$$

Mit anderen Worten, h_1 und h_{-1} sind die beiden im Abstand von jeweils $\frac{1}{\|\theta\|}$ liegenden Hyperebenen zu h_0. Die Lösung des Optimierungsproblems (8) arrangiert h_0, h_1, und h_{-1} gerade so, dass h_1 und h_{-1} ihre jeweils nächstgelegene Klasse gerade berühren und h_0 genau in der der Mitte zwischen h_1 und h_{-1} liegt. Eine Interpretation des Optimierungsproblems (8) ist daher, dass Parameter θ

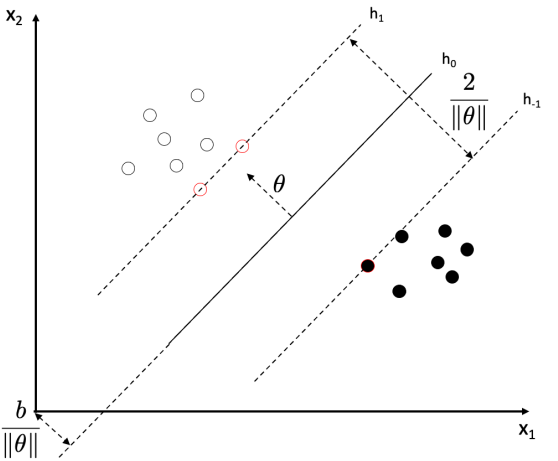


Abbildung 29: Optimal angepasste SVM h_0 mit parallelen Geraden h_1 und h_{-1} und Stützvektoren (in rot).

und b gefunden werden sollen, so dass der Abstand zwischen h_1 und h_{-1} maximal ist (dieser beträgt genau $\frac{2}{\|\theta\|}$). Der Parameter θ ist damit der *Normalenvektor* der Hyperebenen h_0, h_1 und h_{-1} und der Abstand von h_0 zum Ursprung ist genau $\frac{b}{\|\theta\|}$. Wie man in Abbildung 29 erkennen kann, ist die Lösung des Optimierungsproblems (8) einzig durch diejenigen Datenpunkte bestimmt, die genau auf den Hyperebenen h_1 und h_{-1} liegen (alle „dahinter" liegenden Datenpunkte haben keinen Einfluss auf die Lösung). Diese Datenpunkte nennt man daher *Stützvektoren* (engl. *support vectors*), von denen sich der Name *Support Vector Machine* ableitet.

Beispiel 33. Wir setzen Beispiel 32 fort. Abbildung 30 zeigt noch einmal die optimale separierende Hyperebene für D_{rapid2}, sowie die beiden parallel liegenden Hyperebenen im Abstand $\frac{1}{\|\theta\|}$. Die entsprechenden Stützvektoren sind markiert.

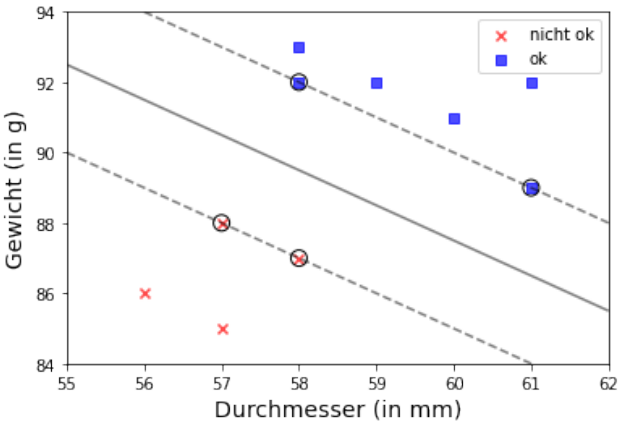


Abbildung 30: Optimal angepasste SVM h_{opt} und zugehörige Stützvektoren (umkreist) für Datensatz D_{rapid2} zu Beispiel 33.

2.3.2 Nicht-linear-separierbare Daten

In realistischen Anwendungsfällen für Klassifikationsalgorithmen ist die lineare Separierbarkeit der Daten in den wenigsten Fällen gegeben. Bei nicht vorhandener linearer Separierbarkeit besitzt das Optimierungsproblem (8) keine zulässige Lösung und die lineare SVM bleibt undefiniert.

Beispiel 34. Wir betrachten wieder $D_{\text{apartment}}$ aus Unterkapitel 2.2, Beispiel 2. Der Datensatz ist noch einmal in Tabelle 17 und Abbildung 31 dargestellt. Beachten Sie, dass es keine Parameter θ und b gibt, so dass $y(\theta^T x - b) \geq 1$ für alle $(x, y) \in D_{\text{apartment}}$ gilt. Damit gibt es keine separierende Hyperebene.

Um auch nicht-linear separierende Datensätze mit (linearen) SVMs zu klassifizieren, modifizieren wir das Optimierungsproblem (8) etwas. Anstatt $y(\theta^T x - b) \geq 1$ für

Nr.	Fläche (in m^2)	Entfernung zum AP (in km)	OK
1	40	4	1
2	30	3	1
3	45	8	0
4	35	6	0
5	20	4	0
6	50	7	1
7	30	4	0
8	45	4	1
9	40	5	0
10	25	1	1

Tabelle 17: Datensatz $D_{\text{apartment}}$ zu Beispiel 34 (siehe auch Unterkapitel 2.2, Beispiel 2).

alle $(x, y) \in D$ strikt zu fordern, soll der Term $1 - y(\theta^T x - b)$ nur so klein wie möglich werden (die Verletzung der Bedingungen soll also so gering wie möglich sein). Realisiert wird dies durch die *Hinge*-Kostenfunktion[10].

Definition 13. Sei D ein Datensatz und $f : \mathbb{R}^n \to \mathbb{R}$ eine beliebige Funktion.[11] Die *Hinge-Kostenfunktion* L^{hinge} von f bzgl. D ist definiert durch

$$L^{\text{hinge}}(D, f) = \sum_{i=1}^{m} \max\{0, 1 - y^{(i)} f(x^{(i)})\}$$

Beachten Sie, dass wir für den Fall $f(x^{(i)}) = y^{(i)} \in \{-1, 1\}$ für $i = 1, \ldots, m$ (die Funktion f erklärt die Daten D also

[10] Der deutsche Begriff „Scharnier"-Kostenfunktion ist nicht üblich.

[11] Beachten Sie, dass wir wie bei SVMs üblich die beiden Klassen mit 1 und -1 bezeichnen und nicht mit 1 und 0. Ist $f(x)$ negativ, so wird x der Klasse -1 zugeordnet, ist $f(x)$ nicht-negativ, so wird x der Klasse 1 zugeordnet.

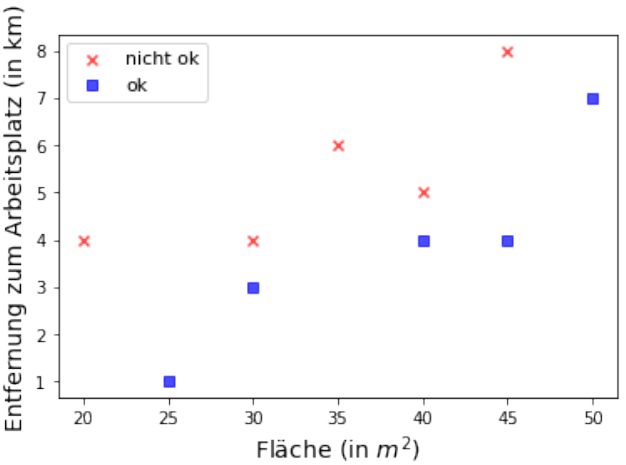


Abbildung 31: Datensatz $D_{\text{apartment}}$ zu Beispiel 34 (siehe auch Unterkapitel 2.2, Beispiel 2).

perfekt) $L^{\text{hinge}}(D, f) = 0$ erhalten, die Kosten also minimal sind. Je stärker die Vorhersagen von f von den Zielwerten abweichen, desto größer wird $L^{\text{hinge}}(D, f)$.

Wir wenden die Hinge-Kostenfunktion auf lineare SVMs an, indem wir sie in die Zielfunktion des Optimierungsproblems (8) aufnehmen (und dafür die harten Nebenbedingungen weglassen). Weiterhin fügen wir einen *Regularisierungsparameter* $C \in \mathbb{R}^{>0}$ ein.[12] Für einen Datensatz D sind damit die optimalen Parameter θ und b bestimmt durch die Lösung des folgenden Minimierungsproblems:

$$\min C\|\theta\|^2 + \frac{1}{m}\sum_{i=1}^{m} \max\{0, 1 - y^{(i)}(\theta^T x^{(i)} - b)\} \qquad (9)$$

Beachten Sie die folgenden Anmerkungen zum Optimie-

[12] $\mathbb{R}^{>0}$ bezeichne die Menge der positiven reellen Zahlen.

rungsproblem (9):

- Die Hinge-Kostenfunktion wird hier mit dem Faktor $\frac{1}{m}$ multipliziert, d. h., der zweite Term der Zielfunktion ist die *durchschnittliche* Abweichung der Datenpunkte anstatt die Summe aller Abweichungen. Bei der Minimierung in (9) spielt dies keine Rolle, aber dies wirkt sich auf die Skalierung des Parameters C aus.

- Im ersten Term wird $\|\theta\|$ quadriert. Dies war bei (8) nicht der Fall und auch nicht notwendig, da dort $\|\theta\|$ der einzige Term in der Zielfunktion war und die Quadrierung keinen Unterschied in der Lösung bewirkt. In (9) sind allerdings zwei Terme gegeneinander abzuwägen und die Quadrierung bewirkt, dass der Term $\|\theta\|$ eine zunächst höhere Priorität erhält (auch in Abhängigkeit von Parameter C).

- Der Term $C\|\theta\|^2$ ist identisch zum *Tikhonov-Regularisierer* (siehe Definition 6 in Unterkapitel 2.1) und der Parameter C übt eine ähnliche Funktion aus wie bei der Regularisierung. Bei kleinem C verhält sich das Optimierungsproblem (9) ähnlich wie das Optimierungsproblem (8). Insbesondere erhält man bei linear separierbaren Daten D und genügend kleinem C dieselbe separierende Hyperebene. Bei größeren Werten von C wird der gelernte Klassifikator toleranter gegenüber Abweichungen von linearer Separierbarkeit von D.

Die Form (9) ist die in der Praxis geläufige Form, um eine lineare SVM für nicht-linear separierbare Daten zu erlernen. Eine solche SVM nennt man auch *soft-margin* SVM (im Gegensatz dazu nennt man SVMs nach (8) auch *hard-margin* SVMs).

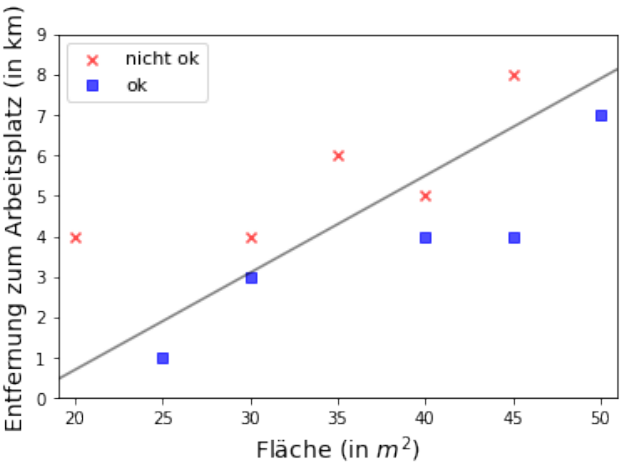


Abbildung 32: Datensatz $D_{\text{apartment}}$ und *soft-margin* SVM für $C = 1$ zu Beispiel 35.

Beispiel 35. Wir setzen Beispiel 34 fort. Abbildung 32 zeigt die *soft-margin* SVM für $C = 1$.

2.3.3 Kernelfunktionen

Lineare *soft-margin* SVMs können sinnvoll angewendet werden, wenn die Daten grundsätzlich in einem linearen Zusammenhang stehen, aber einzelne Datenpunkte verrauschte Informationen darstellen. Ähnlich wie die lineare und die logistische Regression (siehe Abschnitte 2.1.2 und 2.2.3) können sowohl *hard-margin* als auch *soft-margin* SVMs generalisiert werden, um nicht-lineare Zusammenhänge in den Daten zu erkennen.

Beispiel 36. Wir betrachten wieder den Datensatz D_{screws} aus Abschnitt 2.2.3, Beispiel 14. Der Datensatz ist noch einmal in Tabelle 18 und Abbildung 33 dargestellt. Es

Nr.	Gewicht (in g)	Länge (in mm)	Ok?
1	11	27	1
2	10	28	1
3	13	29	0
4	11	28	1
5	11	30	0
6	12	26	0
7	10	29	1
8	10	30	0
9	9	27	0
10	8	28	0

Tabelle 18: Datensatz D_{screws} zu Beispiel 36 (siehe auch Abschnitt 2.2.3, Beispiel 14).

sollte offensichtlich sein, dass weder eine lineare *hard-margin* noch eine lineare *soft-margin* SVM die verschiedenen Klassen erkennen kann.

Um SVMs auf Daten wie aus dem vorherigen Beispiel anwenden zu können, benutzen wir eine ähnliche Technik, wie wir es bereits für lineare und logistische Regression getan haben: wir transformieren den Ursprungs-Merkmalsraum in einen höherdimensionalen Raum, indem wir neue Merkmale durch Kombination existierender Merkmale erzeugen. Bei SVMs können wir allerdings einen Trick (den sogenannten *Kernel*-Trick) anwenden, der uns eine explizite Merkmalserweiterung (und den damit einhergehenden rechnerischen Aufwand) erspart. Bevor wir genauer auf diesen Trick eingehen, beschäftigen wir uns erst einmal damit, wie genau die (soft-margin) SVM berechnet wird. Dazu hier noch einmal das Opti-

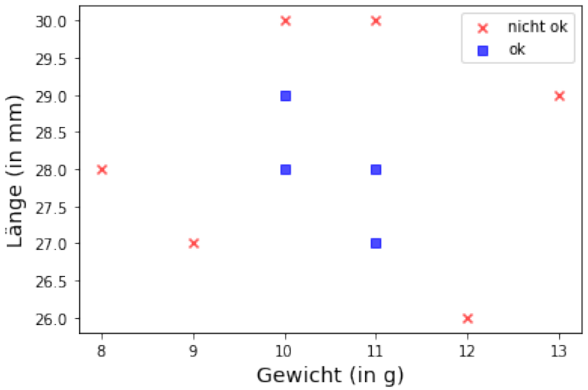


Abbildung 33: Datensatz D_{screws} zu Beispiel 36 (siehe auch Abschnitt 2.2.3, Beispiel 14).

mierungsproblem (9):

$$\min C\|\theta\|^2 + \frac{1}{m}\sum_{i=1}^{m} \max\{0, 1 - y^{(i)}(\theta^T x^{(i)} - b)\}$$

Das obige Optimierungsproblem kann direkt mit Methoden wie *Gradient Descent* gelöst werden. Eine andere Möglichkeit besteht darin, das zum obigen Problem *duale* Problem zu lösen (seien dazu $\lambda_1, \ldots, \lambda_m$ die dazugehörigen Lagrange-Multiplikatoren)[13]:

$$\max \sum_{i=1}^{m} \lambda_i - \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} y^{(i)} y^{(j)} \lambda_i \lambda_j \left((x^{(i)})^T x^{(j)}\right) \quad (10)$$

$$\text{so dass } \sum_{i=1}^{m} \lambda_i y^{(i)} = 0$$

$$0 \le \lambda_i \le \frac{1}{2mC} \quad i = 1, \ldots, m$$

[13] Wir werden im Rahmen dieses Buches nicht auf die Herleitung der dualen Form eingehen, siehe hierzu entsprechende Textbücher zu Optimierung.

Sei $\lambda_1^*, \ldots, \lambda_m^*$ die (eindeutig bestimmte) Lösung von (10). Sei k der kleinste Index, so dass $\lambda_k^* > 0$ (es muss mindestens einen solchen Index geben). Definiere

$$\theta^* = \sum_{i=1}^{m} \lambda_i^* y^{(i)} x^{(i)}$$
$$b^* = (\theta^*)^T x^{(k)} - y^{(k)}$$

Es gilt dann, dass θ^*, b^* Lösung von (9) ist. Das Optimierungsproblem (10) ist quadratisch und konvex und kann mit Algorithmen zur quadratischen Optimierung effizient gelöst werden.

Beachten Sie, dass das Optimierungsproblem (10) über m (=Anzahl Beispiele im Trainingsdatensatz) Variablen $\lambda_1, \ldots, \lambda_m$ definiert ist. Insbesondere ist die Form des Optimierungsproblems unabhängig von der Anzahl der Merkmale n. Das einzige Vorkommen von Merkmalen in (10) ist $\left((x^{(i)})^T x^{(j)}\right)$, wo alle paarweisen Skalarprodukte der Beispiele berechnet werden. Das Ergebnis dieser Skalarprodukte ist stets eine reelle Zahl und muss nur ein einziges Mal während der Optimierung berechnet werden. Werden die Daten nun in einen höherdimensionalen Merkmalsraum transformiert, so müssen wir ausschließlich an dieser Stelle ein anderes Skalarprodukt verwenden. Sei $k : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ eine beliebige Funktion. Wir schreiben (10) allgemeiner als

$$\max \sum_{i=1}^{m} \lambda_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} y^{(i)} y^{(j)} \lambda_i \lambda_j k(x^{(i)}, x^{(j)}) \quad (11)$$

$$\text{so dass } \sum_{i=1}^{m} \lambda_i y^{(i)} = 0$$

$$0 \leq \lambda_i \leq \frac{1}{2mC} \quad i = 1, \ldots, m$$

Beachten Sie, dass wir für

$$k_{\text{linear}}(x, x') = x^T x'$$

die exakt gleiche Form wie (10) erhalten. Funktionen k heißen *Kernelfunktionen* (oder einfach nur *Kernel*) und k_{linear} heißt die *lineare Kernelfunktion*. Sei $\phi : \mathbb{R}^n \to \mathbb{R}^{n'}$ mit $n' > n$ eine beliebige Transformation des Merkmalsraums in einen höherdimensionalen Merkmalsraum. Eine Kernelfunktion k_ϕ realisiert dann das Skalarprodukt im $\mathbb{R}^{n'}$:

$$k_\phi(x, x') = \phi(x)^T \phi(x')$$

Der eigentliche „Kernel-Trick" besteht dann darin, dass k_ϕ direkt über x und x' definiert werden kann, ohne die Transformationen $\phi(x)$ und $\phi(x')$ explizit durchzuführen. Schauen wir uns dies an einem Beispiel an.

Beispiel 37. Sei $\phi : \mathbb{R}^2 \to \mathbb{R}^3$ definiert durch

$$\phi(x_1, x_2) = (x_1^2, \sqrt{2}x_1 x_2, x_2^2)$$

Die Transformation ϕ realisiert also (in etwa) eine polynomielle Merkmalserweiterung. Beispielsweise gilt $\phi(2,3) \approx (4, 8.48, 9)^T$. Betrachte nun

$$k_\phi(x, x') = \phi(x)^T \phi(x')$$

Sei $a = (2,3)^T$ und $b = (4,5)^T$. Um $k_\phi(a, b)$ zu berechnen, müssen also zunächst beide Merkmalsvektoren a und b in den erweiterten Merkmalsraum transformiert werden:

$$\phi(a) \approx (4, 8.48, 9)^T$$
$$\phi(b) \approx (16, 28.28, 25)^T$$

Dann erhalten wir $k_\phi(a,b)$ durch das Skalarprodukt von $\phi(a)$ mit $\phi(b)$:

$$k_\phi(a,b) \approx (4, 8.48, 9) \begin{pmatrix} 16 \\ 28.28 \\ 25 \end{pmatrix} \approx 64 + 240 + 225 = 529$$

Man kann allerdings leicht beweisen[14], dass k_ϕ charakterisiert werden kann durch

$$k_\phi(x,x') = (x^T x')^2 \tag{12}$$

Insbesondere gilt

$$k_\phi(a,b) = (a^T b)^2 = \left((2,3) \begin{pmatrix} 4 \\ 5 \end{pmatrix} \right)^2 = (8+15)^2 = 529$$

Wie man hoffentlich leicht einsieht, ist die Berechnung von k_ϕ über die Charakterisierung (12) rechentechnisch einfacher als via $k_\phi(x,x') = \phi(x)^T \phi(x')$.

Der Kernel k_ϕ aus dem vorherigen Beispiel ist der *homogene polynomielle* Kernel zu Grad 2. Gebräuchliche Kernel sind die folgenden:

$$k^d_{\text{poly-h}}(x,x') = (x^T x')^d$$
(homogener polynomieller Kernel zu Grad $d > 0$)

$$k^{d,r}_{\text{poly-i}}(x,x') = (x^T x' + r)^d$$
(inhomogener polynomieller Kernel zu Grad $d > 0$ mit $r \in \mathbb{R}$)

$$k^\gamma_{\text{rbf}}(x,x') = e^{-\gamma \|x - x'\|^2}$$
(Radiale Basisfunktion mit $\gamma > 0$)

Beachten Sie auch, dass der lineare Kernel ein Spezialfall des homogenen polynomiellen Kernels ist: $k_{\text{linear}} = k^1_{\text{poly-h}}$.

[14] Dies ist eine Übungsaufgabe.

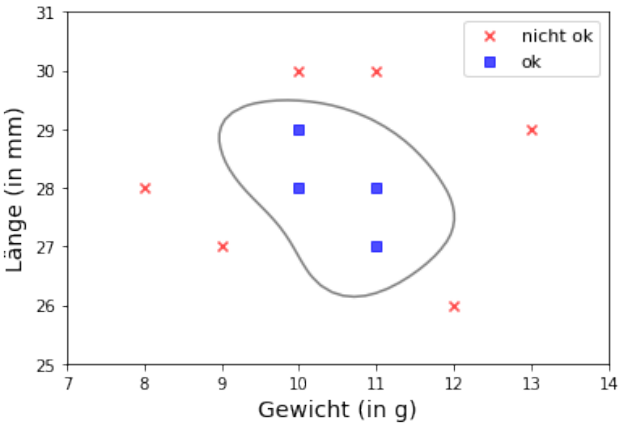


Abbildung 34: Datensatz D_{screws} und Entscheidungsgrenze zur SVM mit Kernel k^1_{rbf} und $C = 500$ zu Beispiel 38.

Der Kernel k^{γ}_{rbf} wird gelegentlich auch Gaußscher Kernel genannt und wird recht oft verwendet, da er in der Lage ist auch komplexere Klassenunterscheidungen zu modellieren.

Beispiel 38. Wir setzen Beispiel 36 fort. Abbildung 34 zeigt die Entscheidungsgrenze zur SVM basierend auf der Radialen Basisfunktion mit Parametern $\gamma = 1$ und $C = 500$. Punkte innerhalb der Grenze werden als „ok“ klassifiziert und Punkte außerhalb der Grenze werden als „nicht ok“ klassifiziert.

2.4 Nächste-Nachbarn Klassifikation

Nächste-Nachbarn-Klassifikation (engl. *nearest-neighbour classification*) ist ein konzeptuell sehr einfacher Algorithmus zur Klassifikation, der (in der einfachsten Ausprägung) im Gegensatz zu den bisher diskutierten Ansätzen kein Training benötigt, sondern direkt auf dem Trainingsdatensatz basierend Klassifikationen vornimmt.

2.4.1 Grundlagen

Die Grundidee des KNN-Algorithmus (*k-nearest neighbour*) besteht darin, dass zur Vorhersage der Klasse einfach die vorherrschende Klasse der $k \in \mathbb{N}$ nächsten Nachbarn (aus dem Trainingsdatensatz) im Merkmalsraum gewählt wird.

Definition 14. Sei $D = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ ein Datensatz, $k \in \mathbb{N}$ und $x \in \mathbb{R}^n$. Definiere

$$\mathrm{clf}_{D,k}(x) = \mathrm{maj}(\mathrm{nearest}_k(D, x))$$

mit

$$\begin{aligned}\mathrm{nearest}_k(D, x) = \{y_1, \ldots, y_k\} \subseteq \{y^{(1)}, \ldots, y^{(m)}\} \\ \text{so dass } x_i \neq x_j \text{ für alle } i, j = 1, \ldots, k \\ \text{und } \|x_i - x\| < \|\hat{x} - x\| \text{ für alle} \\ i = 1, \ldots, k \text{ und } \hat{x} \in \{x^{(1)}, \ldots, x^{(m)}\} \setminus \{x_1, \ldots, x_k\}\end{aligned}$$

und

$$\begin{aligned}\mathrm{maj}(\{y_1, \ldots, y_k\}) = y \in \{y_1, \ldots, y_k\} \\ \text{so dass } |\{i \mid y_i = y\}| \text{ maximal unter allen} \\ y \in \{y_1, \ldots, y_k\} \text{ ist}\end{aligned}$$

Mit anderen Worten, $\text{nearest}_k(D, x)$ bestimmt die Klassen derjenigen k Beispiele des Datensatzes D, die sich bezüglich der Euklidischen Distanz am nächsten zu x befinden. Die Funktion maj bestimmt dann diejenige Klasse, die unter diesen k nächsten Nachbarn am häufigsten auftaucht. Beachten Sie bitte die folgenden Annahmen zu der obigen Definition:

- Bei $\text{nearest}_k(D, x)$ nehmen wir an, dass alle Beispiele des Datensatzes D eine unterschiedliche Distanz zu x haben. Für den Fall, dass zwei oder mehr Beispiele in D die gleiche Distanz zu x haben (und diese eventuell beide dann zu den k nächsten Nachbarn zählen würden), werden wir dennoch nur immer genau k nächste Nachbarn betrachten und damit ggfs. ein (zufällig gewähltes) Beispiel mit der gleichen Distanz wie der zuletzt gewählte nächste Nachbar ignorieren.

- Bei maj nehmen wir an, dass es immer eine eindeutige Klasse in der Mehrheit gibt. Gibt es mehrere maximal vorhandene Klassen, wählen wir eine Klasse zufällig aus.

Der KNN-Algorithmus in Definition 14 ist an einigen Stellen parametrisierbar. Zum einen muss die Bestimmung der k nächsten Nachbarn nicht notwendigerweise mithilfe der Euklidischen Norm $\|\cdot\|$ erfolgen, prinzipiell kann hier eine beliebige Norm verwendet werden. Zum anderen kann die Funktion maj durch andere Selektionsfunktionen ausgetauscht werden, beispielsweise eine Funktion, die der Klasse der näheren Nachbarn (innerhalb der k ausgewählten Nachbarn) eine höhere Gewichtung gibt. Im Folgenden werden wir aber ausschließlich die in Definition 14 vorgestellte Variante benutzen, die als einzigen Parameter die Anzahl der auszuwählenden

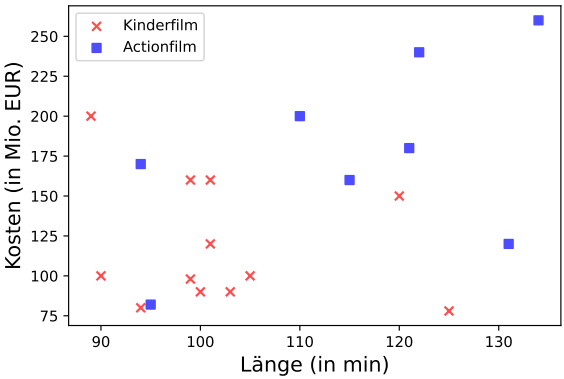


Abbildung 35: Datensatz D_{movies} zu Beispiel 39 (visualisiert).

Nachbarn k besitzt. Typische Werte für k liegen hier im Bereich $1,\ldots,10$, wobei Klassifikatoren mit kleineren Werten für k zu Überanpassung und höheren Werten für k zu Unteranpassung neigen.

Beispiel 39. Wir klassifizieren Filme in die beiden Klassen „Kinderfilm“ und „Actionfilm“ anhand der beiden Merkmale „Länge“ und „Kosten“. Tabelle 19 und Abbildung 35 zeigen die erhobenen Daten D_{movies}. Abbildung 36 zeigt die Entscheidungsgrenzen von drei verschiedenen KNN-Klassifikatoren für $k = 1,3,5$.

2.4.2 Nächste-Nachbarn-Regression

Der KNN-Algorithmus kann mit einer einfachen Modifikation auch auf Regressionsprobleme angewendet werden. Anstatt als Klasse eines neuen Beispiels die vorherrschende Klasse der Nachbarschaft zu wählen, wird hier als Zielwert der Mittelwert der Nachbarschaft gewählt.

Nr.	Länge (in Min.)	Kosten (in Mio. EUR)	Klasse
1	90	100	Kinderfilm
2	101	120	Kinderfilm
3	103	90	Kinderfilm
4	89	200	Kinderfilm
5	122	240	Actionfilm
6	100	90	Kinderfilm
7	131	120	Actionfilm
8	94	170	Actionfilm
9	99	98	Kinderfilm
10	125	78	Kinderfilm
11	134	260	Actionfilm
12	121	180	Actionfilm
13	105	100	Kinderfilm
14	94	80	Kinderfilm
15	99	160	Kinderfilm
16	120	150	Kinderfilm
17	110	200	Actionfilm
18	115	160	Actionfilm
19	95	82	Actionfilm
20	101	160	Kinderfilm

Tabelle 19: Datensatz D_{movies} zu Beispiel 39. Bitte beachten Sie, dass die Daten rein fiktiv sind.

Definition 15. Sei $D = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ ein Datensatz (für Regression, also $y^{(i)} \in \mathbb{R}$, $i = 1, \ldots, m$), $k \in \mathbb{N}$ und $x \in \mathbb{R}^n$. Definiere

$$\text{regr}_{D,k}(x) = \frac{\sum_{y \in \text{nearest}_k(D,x)} y}{k}$$

mit $\text{nearest}_k(D, x)$ wie in Definition 14.

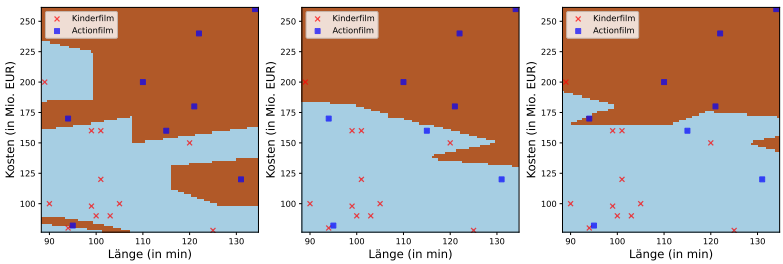


Abbildung 36: Drei KNN-Klassifikatoren für den Datensatz D_{movies} zu Beispiel 39 (links: $k = 1$, mitte: $k = 3$, rechts: $k = 5$). Beispiele in braunen Regionen werden als Actionfilm und Beispiele in blauen Regionen werden als Kinderfilm klassifiziert.

Beispiel 40. Wir betrachten wieder den Datensatz D_{houses} aus Abschnitt 2.1.2, Beispiel 11. Der Datensatz ist nochmal in Tabelle 20 und Abbildung 37 dargestellt. Abbildung 38 zeigt drei verschiedene KNN-Regressoren für $k = 1, 3, 5$.

2.4.3 Merkmalskalierung

Ein besonderer Nachteil des KNN-Algorithmus (insbesondere bei der Verwendung der Euklidischen Norm) ist eine Empfindlichkeit bzgl. verschiedener Skalen der Merkmale.

Beispiel 41. Wir betrachten wieder Beispiel 11 aus Abschnitt 2.2.2 zur Klassifikation von Tieren in die drei Klassen *Hund* (Klasse 0), *Wolf* (Klasse 1) und Dingo (Klasse 2) und benutzen dazu die Merkmale *Alter* (im Sinne von erreichtem Lebensalter des Tieres) und *Gewicht*. Tabelle 21 und Abbildung 39 zeigen noch einmal die Daten. Ab-

Nr.	Fläche (in m^2)	Preis (in EUR)
1	100	210000
2	120	270000
3	160	290000
4	170	470000
5	200	620000
6	210	680000
7	310	1600000
8	330	1900000
9	370	2570000
10	400	3300000

Tabelle 20: Datensatz D_{houses} zu Beispiel 40, siehe auch Beispiel 11 aus Abschnitt 2.1.2.

bildung 40 zeigt die Entscheidungsgrenzen von drei verschiedenen KNN-Klassifikatoren für $k = 1,3,5$. Betrachten wir zunächst den Fall für $k = 1$. Hier ist schon klar zu sehen, dass die Entscheidungsgrenzen des Klassifikators etwas unintuitiv sind. Betrachten Sie beispielsweise ein Tier x, dass sich auf der Verbindungslinie der Tiere 2 und 10 befindet (in der Abbildung die oberen beiden Hunde). Da beide Tiere 2 und 10 als Hunde klassifiziert wurden, würde man davon ausgehen, dass auch x als Hund klassifiziert werden sollte, da sich die Merkmalsausprägungen von x genau zwischen Tier 2 und Tier 10 befinden. Allerdings gibt es einen Bereich zwischen Tier 2 und Tier 10, der als Wolf klassifiziert wird. Der Grund dafür ist der größere Einfluss des Merkmals „Gewicht“, das auf einer größeren Skala (5–56) als das Merkmal „Alter“ (10–17) auf den Trainingsdaten verteilt ist. Durch Nutzung der Euklidischen Distanz fallen vergleichsweise geringe Abweichungen beim Merkmal „Gewicht“ dadurch stärker

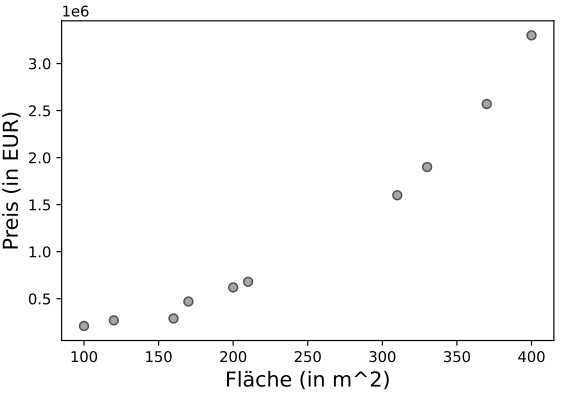


Abbildung 37: Datensatz D_{houses} zu Beispiel 40, siehe auch Beispiel 11 aus Abschnitt 2.1.2.

in die Abstandsberechnung ein. Bei den Klassifikatoren zu $k = 3$ und $k = 5$ sieht man, dass Abweichungen im Merkmal „Alter“ immer weniger eine Rolle spielen.

Das Problem der unterschiedlichen Skalen bei Merkmalen tritt nicht nur beim KNN-Algorithmus auf, die meisten Verfahren des maschinellen Lernens können (zu unterschiedlichen Graden) hier unintuitive Ergebnisse liefern. Ein wichtiger Schritt in der *Datenvorverarbeitung* besteht deshalb darin, die Merkmalsausprägungen entsprechend zu *normieren*. Es gibt dazu verschiedene Ansätze, wir stellen hier die gebräuchlichste *z-Transformation* (oder einfach nur *Standardisierung*) vor.

Definition 16. Sei $D = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ ein Datensatz. Der *z-transformierte* Datensatz $\hat{D} = \{(\hat{x}^{(1)}, \hat{y}^{(1)}), \ldots,$

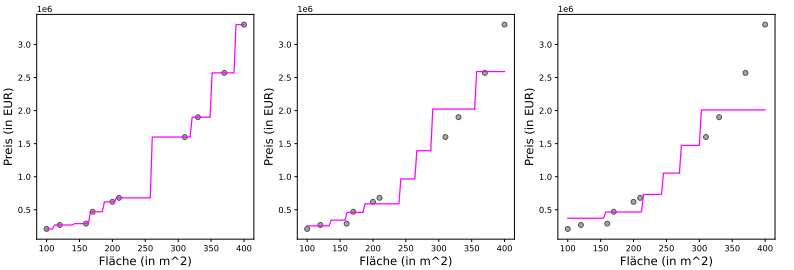


Abbildung 38: Drei KNN-Regressoren für den Datensatz D_{houses} zu Beispiel 40 (links: $k = 1$, mitte: $k = 3$, rechts: $k = 5$).

$(\hat{x}^{(m)}, \hat{y}^{(m)})\}$ ist definiert als

$$\hat{x}_j^{(i)} = \frac{x_j^{(i)} - \tilde{x}_j}{\sigma_j}$$
$$\hat{y}^{(i)} = y^{(i)}$$

wobei

$$\tilde{x}_j = \frac{x_j^{(1)} + \ldots + x_j^{(m)}}{m}$$

die Mittelwert von Merkmal j und

$$\sigma_j = \sqrt{\frac{(x_j^{(1)} - \tilde{x}_j)^2 + \ldots + (x_j^{(m)} - \tilde{x}_j)^2}{m}}$$

die Standardabweichung von Merkmal j ist, für $j = 1, \ldots, m$, $i = 1, \ldots, n$.

Nach der z-Transformation ist der Erwartungswert (über dem gegebenen Datensatz) jeder Merkmalsausprägung 0 und die Varianz jeder Merkmalsausprägung 1.

Nr.	Alter (in Jahren)	Gewicht (in kg)	Tierart
1	12	12	Hund
2	13	34	Hund
3	16	30	Wolf
4	14	15	Dingo
5	15	39	Wolf
6	15	13	Dingo
7	10	5	Hund
8	17	56	Wolf
9	14	11	Dingo
10	11	23	Hund

Tabelle 21: Datensatz D_{animals} zu Beispiel 41, siehe auch Beispiel 11 aus Abschnitt 2.2.2.

Unterschiede zwischen verschiedenen Merkmalsausprägungen sind dabei besser vergleichbar.

Beispiel 42. Wir führen Beispiel 41 fort. Als Mittelwerte $\tilde{x}_{\text{Alter}}$ und $\tilde{x}_{\text{Gewicht}}$ sowie Standardabweichungen σ_{Alter} und σ_{Gewicht} der beiden Merkmale „Alter“ und „Gewicht“ erhalten wir

$$\begin{aligned}\tilde{x}_{\text{Alter}} &= \frac{12+13+16+14+15+15+10+17+14+11}{10}\\ &= \frac{137}{10} = 13.7\\ \sigma_{\text{Alter}} &\approx 2.1\end{aligned}$$

$$\begin{aligned}\tilde{x}_{\text{Gewicht}} &= \frac{12+34+30+15+39+13+5+56+11+23}{10}\\ &= \frac{238}{10} = 23.8\\ \sigma_{\text{Gewicht}} &\approx 15.039\end{aligned}$$

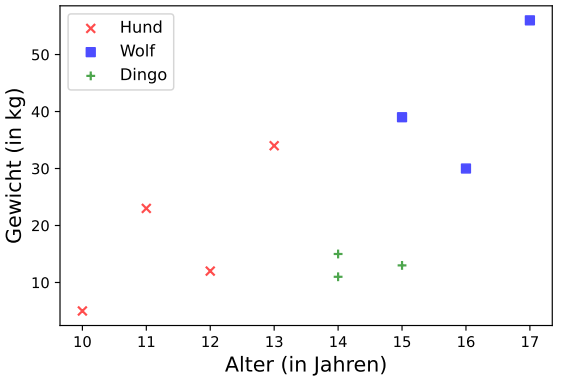


Abbildung 39: Datensatz D_{animals} zu Beispiel 41, siehe auch Beispiel 11 aus Abschnitt 2.2.2.

Daraus ergibt sich der *z-transformierte* Datensatz $\hat{D}_{\text{animals}}$, wie in Tabelle 22 und Abbildung 42 dargestellt. Abbildung 42 zeigt die Entscheidungsgrenzen von drei verschiedenen KNN-Klassifikatoren für $k = 1, 3, 5$.

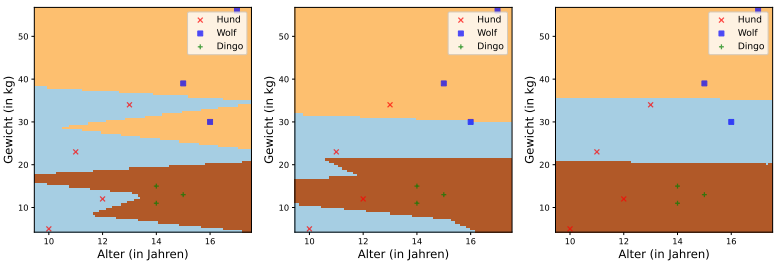


Abbildung 40: Drei KNN-Klassifikatoren für den Datensatz D_{animals} zu Beispiel 41 (links: $k = 1$, mitte: $k = 3$, rechts: $k = 5$). Beispiele in dunkelbraunen Regionen werden als „Dingo“, Beispiele in hellbraunen Regionen werden als „Wolf“ und Beispiele in blauen Regionen werden als „Hund“ klassifiziert.

Nr.	Alter (normalisiert)	Gewicht (normalisiert)	Tierart
1	-0.81	-0.785	Hund
2	-0.333	0.678	Hund
3	1.095	0.412	Wolf
4	0.143	-0.585	Dingo
5	0.619	1.011	Wolf
6	0.619	-0.718	Dingo
7	-1.762	-1.25	Hund
8	1.571	2.141	Wolf
9	0.143	-0.851	Dingo
10	-1.286	-0.053	Hund

Tabelle 22: Z-Transformierter Datensatz $\hat{D}_{\text{animals}}$ zu Beispiel 42.

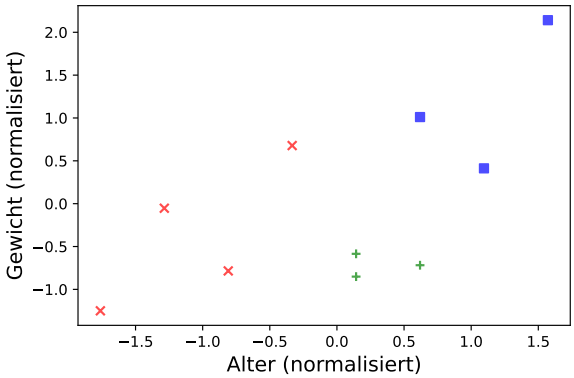


Abbildung 41: Z-Transformierter Datensatz $\hat{D}_{\text{animals}}$ zu Beispiel 42.

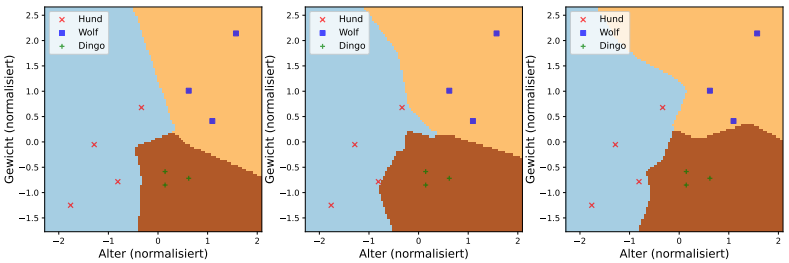


Abbildung 42: Drei KNN-Klassifikatoren für den Datensatz $\hat{D}_{\text{animals}}$ zu Beispiel 42 (links: $k = 1$, mitte: $k = 3$, rechts: $k = 5$). Beispiele in dunkelbraunen Regionen werden als „Dingo“, Beispiele in hellbraunen Regionen werden als „Wolf“ und Beispiele in blauen Regionen werden als „Hund“ klassifiziert.

2.5 Bayes Klassifikator

Bevor wir uns mit dem eigentlichen Thema dieses Unterkapitels beschäftigen (Klassifikation mit Bayes'schen Methoden), blicken wir noch einmal auf die bisher betrachteten Techniken des maschinellen Lernens zurück. In den Unterkapiteln 2.1–2.3 haben wir uns mit linearer und logistischer Regression, sowie mit *Support Vector Machines* beschäftigt. Was diese Ansätze gemeinsam haben, ist das Anpassen eines bestimmten *Modells* an Trainingsdaten mit anschließender Nutzung dieses Modells für neue Vorhersagen (sei es Regression oder Klassifikation). Die Kernidee bestand also darin, *Parameter* für den entsprechenden Modelltyp zu finden, sodass das gelernte Modell die Trainingsdaten am besten erklärt und das Modell „gut" generalisiert. In Unterkapitel 2.4 haben wir uns die Nächste-Nachbarn-Klassifikation angeschaut. Bei diesem Ansatz wird kein explizites Modell gelernt und die Grundidee bestand darin, dass eine neue Instanz so klassifiziert wird, wie es gegeben durch die Nachbarschaft der neuen Instanz am *wahrscheinlichsten* ist. Wir werden uns in diesem Unterkapitel mit einer allgemeinen Methode beschäftigen, die die beiden Konzepte *Modell* und *Wahrscheinlichkeit* beim Lernprozess expliziert.

2.5.1 Das Maximum-Likelihood-Prinzip

Wie bereits oben beschrieben, besteht das Problem des maschinellen Lernens darin, zu einem gegebenen Trainingsdatensatz D ein optimales Modell h zu bestimmen. In diesem Unterkapitel werden wir zu einem Modell h auch *Hypothese* sagen und üblicherweise nimmt man an, dass h aus einem gegebenen Hypothesenraum H entstammt (z. B. der Menge aller linearen Funktionen im

Fall der linearen Regression). Der Begriff der Optimalität kann so interpretiert werden, dass wir die *wahrscheinlichste* Hypothese h suchen, gegeben dass wir die Trainingsdaten D beobachtet haben. Sei P die Wahrscheinlichkeitsverteilung, die unserem Szenario unterliegt (und üblicherweise nur partiell bekannt ist). Wir suchen also eine Hypothese h, die den Wert $P(h \mid D)$ maximiert, wobei $P(h \mid D)$ die Wahrscheinlichkeit von h gegeben der Beobachtung D beschreibt. Der Wert $P(h \mid D)$, den wir auch „a posteriori"-Wahrscheinlichkeit von h nennen, kann im Allgemeinen nicht direkt bestimmt werden, aber das *Bayes-Theorem* kann helfen, den Wert $P(h \mid D)$ anzunähern. Es gilt

$$P(h \mid D) = \frac{P(D \mid h)P(h)}{P(D)} \tag{13}$$

Der Wert $P(h)$ ist die „a priori" Wahrscheinlichkeit der Hypothese h, d. h., die eventuell durch Hintergrundwissen angereicherte Wahrscheinlichkeit, wie genau die Hypothese auszusehen hat. Üblicherweise besteht unser vollständiges Hintergrundwissen nur aus dem Datensatz D und wir haben keine weiteren Informationen zum Hypothesenraum H. Deshalb nimmt man hier üblicherweise eine *Gleichverteilung* auf H an, d. h., alle Hypothesen sind gleich wahrscheinlich (im Falle der linearen Regression sind also alle Parameter θ gleich wahrscheinlich). Der Wert $P(D)$ ist die Wahrscheinlichkeit, dass der Datensatz D beobachtet wurde. Der Wert $P(D \mid h)$ ist die Wahrscheinlichkeit, D zu beobachten, gegeben dass D von h „generiert" wurde. Im Gegensatz zu $P(h \mid D)$ ist $P(D \mid h)$ einfacher abzuschätzen, da $P(D \mid h)$ prinzipiell beschreibt, wie gut h den Datensatz D erklärt. Wir sind daran interessiert, eine Hypothese h^* zu finden, die (13) maximiert:

$$h^* = \arg\max_{h \in H} P(h \mid D) \tag{14}$$

Eine Hypothese h^*, die (14) erfüllt, nennen wir *Maximum-A-Posteriori*-Hypothese (MAP-Hypothese). Nach (13) und dem Umstand, dass der Term $P(D)$ für die Maximierung irrelevant ist[15], folgt:

$$\begin{aligned} h^* &= \arg\max_{h \in H} P(h \mid D) \\ &= \arg\max_{h \in H} \frac{P(D \mid h)P(h)}{P(D)} \\ &= \arg\max_{h \in H} P(D \mid h)P(h) \end{aligned} \tag{15}$$

Ist $P(h)$ gleichverteilt, so können wir auch den Term $P(h)$ aus der obigen Optimierungsaufgabe streichen und wir erhalten

$$h^* = \arg\max_{h \in H} P(D \mid h) \tag{16}$$

Eine Hypothese h^*, die (16) erfüllt, nennen wir *Maximum-Likelihood*-Hypothese (ML-Hypothese), da sie nur auf der maximalen Wahrscheinlichkeit, die Daten D bzgl. h zu beobachten, basiert. Den Wert $P(D \mid h)$ kann man bei einem konkreten Modelltyp bzw. Hypothesenraum H relativ gut abschätzen und wir werden im nächsten Abschnitt zeigen, dass lineare Regression genau eine ML-Hypothese lernt. Im Allgemeinen ist zu beachten, dass die obigen Herleitungen grundsätzliche Motivationen für bestimmte Formen von Hypothesen, aber noch keine konkrete Rechenvorschrift liefern. Viele Lernmethoden (wie für die lineare Regression im nächsten Abschnitt gezeigt wird) lassen sich jedoch als eine spezielle Form von MAP- oder ML-Lernen klassifizieren und sind damit durch Wahrscheinlichkeitstheorie und die obigen Überlegungen sinnvoll begründet.

[15] Der Term $P(D)$ ist irrelevant, da er konstant für alle Hypothesen h ist.

2.5.2 Bayes-Klassifikation und lineare Regression

Führen wir uns noch einmal das Problem der (linearen) Regression in Erinnerung, siehe auch Unterkapitel 2.1. Gegeben ein Datensatz $D = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ mit $x^{(i)} \in \mathbb{R}^{n+1}$ und $y^{(i)} \in \mathbb{R}$ für $i = 1, \ldots, m$ gilt es, Parameter $\theta^* \in \mathbb{R}^{n+1}$ zu finden mit[16]

$$\begin{aligned} \theta^* &= \arg\min_{\theta} L(D, \theta) \\ &= \arg\min_{\theta} \|X_D\theta - y_D\|^2 \\ &= \arg\min_{\theta} \sum_{i=1}^{m} (\theta^T x^{(i)} - y^{(i)})^2 \end{aligned} \tag{17}$$

Das gelernte Modell h_{θ^*} mit $h_{\theta^*}(x) = \theta_0^* + \theta_1^* x_1 + \ldots \theta_n^* x_n$ beschreibt dann eine Gerade im $\mathbb{R}^n$, die den funktionalen Zusammenhang zwischen den Merkmalen $x \in \mathbb{R}^n$ und der Zielvariablen $y \in \mathbb{R}$ modelliert. Abbildung 43 zeigt dazu noch einmal das optimal angepasste lineare Modell h_{θ^*} für den Datensatz D_{ring} aus Unterkapitel 2.1.

Betrachten wir das Problem der linearen Regression nun aus der Perspektive der Bayes'schen Klassifikation. Wir gehen davon aus, dass unser Datensatz *D verrauscht* ist, d. h., es wird keine Gerade geben, auf der alle Trainingsbeispiele liegen. Mit anderen Worten, die Werte $y^{(i)}$ für $i = 1, \ldots, m$ unseres Datensatzes haben die Form

$$y^{(i)} = \hat{y}^{(i)} + \epsilon^{(i)} \tag{18}$$

wobei $\hat{y}^{(i)}$ der wahre Wert der Merkmalsausprägung $x^{(i)}$ ist und $\epsilon^{(i)}$ der *Fehler*. Was für eine Art Fehler können wir in der Praxis erwarten? Eine durchaus übliche Annahme

[16] Beachten Sie, dass wir hier die in Unterkapitel 2.1 eingeführte Konvention benutzen, dass für $x^{(i)} = (x_0^{(i)}, x_1^{(i)}, \ldots, x_n^{(i)}) \in \mathbb{R}^{n+1}$ stets $x_0^{(i)} = 1$ gilt, $i = 1, \ldots, m$.

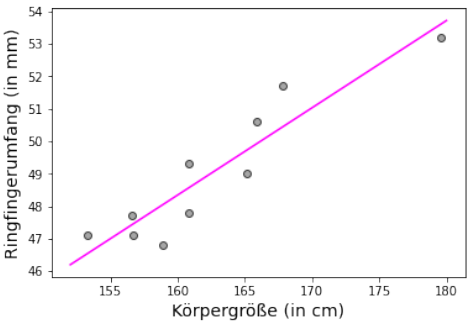


Abbildung 43: Optimal angepasstes lineares Modell h_{θ^*} für den Datensatz D_{ring} aus Unterkapitel 2.1

hier ist, dass die Fehlerwerte im Datensatz *normalverteilt* sind. Seien Sie daran erinnert, dass für eine Zufallsvariable Z mit $Z \sim \mathcal{N}(\mu, \sigma^2)$ (Z ist normalverteilt mit Erwartungswert μ und Varianz σ^2) und einer Ausprägung $z \in \mathbb{R}$ gilt[17]

$$p(z \mid \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(z-\mu)^2}{2\sigma^2}}$$

Wir gehen davon aus, dass die Fehlerwerte $\epsilon^{(i)}$ den Erwartungswert 0 und eine unbekannte Varianz σ^2 haben. Weiterhin gehen wir davon aus, dass alle Beispiele des Trainingsdatensatzes unabhängig voneinander sind und die Verteilung der Fehlerwerte bei jedem Beispiel identisch ist. Es folgt, dass die Wahrscheinlichkeitsdichte, den Wert $y^{(i)}$ anstatt $\hat{y}^{(i)}$ zu beobachten, gegeben ist durch

$$p(y^{(i)} \mid \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y^{(i)}-\hat{y}^{(i)})^2}{2\sigma^2}}$$

[17] Beachten Sie, dass wir hier Wahrscheinlichkeitsdichten statt Wahrscheinlichkeitsverteilungen betrachten müssen, da die Merkmale kontinuierlich sind.

Der Wert $y^{(i)} - \hat{y}^{(i)}$ ist wegen (18) gleich $\epsilon^{(i)}$, also genau der Fehler in der Vorhersage. Ist nun θ unsere Hypothese, so gilt

$$\epsilon^{(i)} = h_\theta(x^{(i)}) - y^{(i)}$$

Insgesamt folgt also, dass die Wahrscheinlichkeitsdichte, ein Beispiel $d^{(i)} = (x^{(i)}, y^{(i)})$ unter der Hypothese θ zu beobachten, gegeben ist durch

$$p(d^{(i)} \mid \sigma^2, \theta) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(h_\theta(x^{(i)}) - y^{(i)})^2}{2\sigma^2}}$$

Da wir davon ausgehen, dass alle Beispiele des Datensatzes D unabhängig voneinander sind, folgt für die Wahrscheinlichkeitsdichte, den Datensatz D unter der Hypothese θ zu beobachten, dass

$$p(D \mid \sigma^2, \theta) = \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(h_\theta(x^{(i)}) - y^{(i)})^2}{2\sigma^2}}$$

Verfolgen wir nun den Bayes'schen Ansatz und wünschen, eine ML-Hypothese θ^* auszuwählen, so ist diese gegeben durch

$$\begin{aligned} \theta^* &= \arg\max_\theta p(D \mid \sigma^2, \theta) \\ &= \arg\max_\theta \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(h_\theta(x^{(i)}) - y^{(i)})^2}{2\sigma^2}} \end{aligned}$$

Der obige Term ist aufgrund der Exponentialfunktion etwas sperrig. Anstatt den Likelihood wie oben zu maximieren, können wir äquivalent auch den *Log-Likelihood* maximieren. Dazu wenden wir einfach den natürlichen Logarithmus auf den gesamten Ausdruck an. Da dieser

eine monotone Funktion ist, sind die Lösungen der jeweiligen Maximierungsprobleme identisch. Wir erhalten

$$\begin{aligned}
\theta^* &= \arg\max_\theta \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(h_\theta(x^{(i)})-y^{(i)})^2}{2\sigma^2}} \\
&= \arg\max_\theta \ln\left(\prod_{i=1}^{m} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(h_\theta(x^{(i)})-y^{(i)})^2}{2\sigma^2}} \right) \\
&= \arg\max_\theta \sum_{i=1}^{m} \ln\left(\frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(h_\theta(x^{(i)})-y^{(i)})^2}{2\sigma^2}} \right) \\
&= \arg\max_\theta \sum_{i=1}^{m} \ln\left(\frac{1}{\sqrt{2\pi\sigma^2}} \right) - \frac{(h_\theta(x^{(i)})-y^{(i)})^2}{2\sigma^2}
\end{aligned}$$

Der Term $\ln(1/\sqrt{2\pi\sigma^2})$ ist konstant und kann bei der Maximierung ignoriert werden. Gleiches gilt für den Faktor $1/2\sigma^2$ und da die Maximierung eines Terms x der Minimierung des Terms $-x$ entspricht, erhalten wir

$$\begin{aligned}
\theta^* &= \arg\max_\theta \sum_{i=1}^{m} -\frac{(h_\theta(x^{(i)})-y^{(i)})^2}{2\sigma^2} \\
&= \arg\max_\theta \sum_{i=1}^{m} -(h_\theta(x^{(i)})-y^{(i)})^2 \\
&= \arg\min_\theta \sum_{i=1}^{m} (h_\theta(x^{(i)})-y^{(i)})^2 \\
&= \arg\min_\theta \sum_{i=1}^{m} (\theta^T x^{(i)}-y^{(i)})^2
\end{aligned}$$

Das letzte Optimierungsproblem ist offensichtlich identisch zu (17). Beachten Sie insbesondere auch, dass die tatsächliche (unbekannte) Varianz σ^2 keine Rolle mehr bei der Bestimmung von θ^* spielt.

Wir haben in diesem Abschnitt also gezeigt, dass die lineare Regression (unter Benutzung des quadratischen Fehlers als Kostenfunktion) eine Maximum-Likelihood-Hypothese bestimmt, die lineare Regression also nach Bayes'schen Grundsätzen plausibel ist.

2.5.3 Naive Bayes-Klassifikation

In diesem Abschnitt benutzen wir nun unsere bisher gewonnenen theoretischen Erkenntnisse, um einen neuen Klassifikationsalgorithmus zu definieren. Wir betrachten dazu zunächst ein Klassifikationsszenario, bei dem die Merkmale in einem *endlichen* Merkmalsraum definiert sind (den allgemeinen Fall diskutieren wir kurz in Abschnitt 2.5.4). Für jedes Merkmal $i = 1, \ldots, n$ sei $Z_i = \{z_{i,1}, \ldots, z_{i,n_i}\}$ der entsprechende Merkmalsraum für ein $n_i \in \mathbb{N}$ und sei $Z = \{c_1, \ldots, c_k\}$ die Menge der Klassen. Sei also ein Datensatz $D = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ gegeben mit $x^{(i)} \in Z_1 \times \ldots \times Z_n$ und $y^{(i)} \in Z$ für $i = 1, \ldots, m$. Im Folgenden gehen wir davon aus, dass D als *Multimenge* repräsentiert ist, d. h., wir erlauben, dass Elemente mehrfach in D vorkommen können. Die Grundidee der Bayes-Klassifikation ist, dass diejenige Hypothese ausgewählt wird, die am wahrscheinlichsten, gegeben den Daten, zu erwarten ist. Angewandt auf das obige Klassifikationsproblem können wir dieses Prinzip folgendermaßen *direkt* interpretieren: Gegeben ein neuer Datenpunkt x, wählen wir diejenige Klasse $c \in Z$, die aufgrund von D am wahrscheinlichsten ist. Hierbei interpretieren wir „am wahrscheinlichsten" einfach durch „am häufigsten".

Definition 17. Sei $D = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ wie zuvor gegeben und $x \in Z_1 \times \ldots \times Z_n$ ein neuer Datenpunkt.

Definiere den *Bayes-Klassifikator* $\text{clf}_D^{\text{Bayes}}$ via

$$\text{clf}_D^{\text{Bayes}}(x) = \arg\max_{c \in Z} P(c \mid x, D) \tag{19}$$

mit

$$P(c \mid x, D) = \frac{|\{(x, c) \in D\}|}{\sum_{c' \in Z} |\{(x, c') \in D\}|}$$

Mit anderen Worten, einem neuen Datenpunkt x wird diejenige Klasse $c \in Z$ zugewiesen, die am häufigsten x in D zugewiesen wird. Dem aufmerksamen Leser sollten nun direkt einige Probleme mit dieser Definition auffallen, wir werden diese weiter unten diskutieren. Zunächst schauen wir uns aber ein Beispiel an.

Beispiel 43. Wir betrachten ein Empfehlungssystem für Filme und klassifizieren Filme anhand der binären Merkmale „Hat der Film einen Oskar gewonnen?“ und „Wurde der Film in Europa gedreht?“ als „gut“ und „schlecht“. Es gilt also $Z_1 = \{0, 1\}$ („Oskar nicht gewonnen“ =0, „Oskar gewonnen“ =1), $Z_2 = \{0, 1\}$ („nicht in Europa gedreht“ =0, „in Europa gedreht“ =1) und $Z = \{0, 1\}$ („Film schlecht“ =0, „Film gut“ =1). Tabelle 23 zeigt unseren Trainingsdatensatz D_{movies2}. Betrachten wir nun einen neuen zu klassifizierenden Datenpunkt $x^* = (0, 1)$ (ein in Europa gedrehter Film, der keinen Oskar erhalten hat). Wir berechnen für beide Klassen 0 und 1 die Wahrscheinlichkeit, dass x^* der entsprechenden Klasse zugeordnet wird. Beachten Sie, dass es drei Beispiele in D_{movies2} gibt, die mit x^* in der Merkmalsausprägung übereinstimmen (Nr. 1, 4 und 5), zwei davon sind als 1 und eins als 0 klassifiziert. Wir erhalten also

$$P(c = 0 \mid x = x^*, D = D_{\text{movies2}}) = \frac{1}{3}$$

$$P(c = 1 \mid x = x^*, D = D_{\text{movies2}}) = \frac{2}{3}$$

Nr.	Z_1	Z_2	Z
1	0	1	1
2	1	0	1
3	0	0	0
4	0	1	1
5	0	1	0
6	0	0	1
7	0	0	0
8	1	0	0
9	1	0	1
10	1	0	1

Tabelle 23: Datensatz D_{movies2} zu Beispiel 43. Bitte beachten Sie, dass die Daten rein fiktiv sind.

Die Klasse 1 kommt also häufiger unter der Merkmalsausprägung x^* vor und deshalb erhalten wir

$$\text{clf}^{\text{Bayes}}_{D_{\text{movies2}}}(x^*) = 1$$

Der Film x^* wird also als „gut“klassifiziert.

Ein offensichtliches Problem des Bayes-Klassifikators aus Definition 17 ist, dass wir für jede zu erwartende Merkmalsausprägung x^* Beispiele im Trainingsdatensatz D haben müssen, um x^* überhaupt klassifizieren zu können. Beispielsweise können wir bzgl. des Datensatzes D_{movies2} aus Beispiel 43 für den Datenpunkt $x^{**} = (1,1)$ (d. h., ein in Europa gedrehter Film, der einen Oskar gewonnen hat) keine Klassifikation vorhersagen, da alle Wahrscheinlichkeiten undefiniert (d. h., gleich $\frac{0}{0}$) sind. Ein anderes Problem (das auch prinzipiell schon beim KNN-Algorithmus aufgetreten ist) ist, dass zur Berechnung von $\text{clf}^{\text{Bayes}}_{D}(x)$ stets der gesamte Datensatz D vorge-

halten werden muss. Insbesondere bei großen Datensätzen kann dies signifikant zu einer langen Berechnungszeit beitragen.

Beide oben genannten Probleme können gelöst werden, wenn man von der allgemeinen Bayes-Klassifikation aus Definition 17 zur *Naiven Bayes-Klassifikation* wechselt. Der Unterschied dabei liegt darin, dass wir bei der Berechnung von $P(c \mid x, D)$ eine Unabhängigkeitsannahme über die Verteilung der einzelnen Merkmale vornehmen und dabei den wahren Wert $P(c \mid x, D)$ nur abschätzen. Dazu interpretieren wir die Beobachtung x zunächst als die simultane Beobachtung der Merkmalsausprägungen von x, d. h., wir ersetzen x durch $x_1, \ldots, x_n$. Nun schreiben wir den Ausdruck (19) wieder unter Verwendung des Bayes-Theorem etwas um:

$$\begin{aligned}\arg\max_{c \in Z} P(c \mid x, D) &= \arg\max_{c \in Z} \frac{P(c \mid D) P(x \mid c, D)}{P(x \mid D)} \\ &= \arg\max_{c \in Z} P(c \mid D) P(x \mid c, D) \\ &= \arg\max_{c \in Z} P(c \mid D) P(x_1, \ldots, x_n \mid c, D)\end{aligned}$$

Nun nehmen wir an, dass

$$P(x_1, \ldots, x_n \mid c, D) = P(x_1 \mid c, D) P(x_2 \mid c, D) \ldots P(x_n \mid c, D)$$

gilt, d. h., die Verteilungen der Merkmalsausprägungen sind bedingt unabhängig voneinander, gegeben die Klasse (und die Daten). Den dazugehörigen Klassifikator nennen wir den *Naiven Bayes-Klassifikator*.

Definition 18. Sei $D = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ wie zuvor gegeben und $x = (x_1, \ldots, x_n)^T \in Z_1 \times \ldots \times Z_n$ ein neuer Datenpunkt. Definiere den *Naiven Bayes-Klassifikator*

$\mathrm{clf}_D^{\mathrm{NaiveBayes}}$ via

$$\begin{aligned}&\mathrm{clf}_D^{\mathrm{NaiveBayes}}(x)\\ &= \arg\max_{c\in Z} P(c \mid D)P(x_1 \mid c,D)P(x_2 \mid c,D)\ldots P(x_n \mid c,D) \quad (20)\end{aligned}$$

mit

$$P(c \mid D) = \frac{|\{(z,c) \in D\}|}{|D|}$$
$$P(x_i \mid c,D) = \frac{|\{(z',c) \in D \mid z' = (z_1,\ldots,z_n), z_i = x_i\}|}{|\{(z,c) \in D\}|}$$

für $i = 1,\ldots,n$.

Die Wahrscheinlichkeit $P(c \mid D)$ ist also die relative Frequenz der Klasse c im Datensatz D und $P(x_i \mid c,D)$ die relative Frequenz von Beispielen der Klasse c, die x_i als i-te Merkmalsausprägung haben.

Beispiel 44. Wir führen Beispiel 43 fort und betrachten wieder den Datenpunkt $x^* = (0,1)$ (ein in Europa gedrehter Film, der keinen Oskar erhalten hat). Zunächst erhalten wir für die Verteilung der Klassen die Wahrscheinlichkeiten

$$P(c = 0 \mid D = D_{\mathrm{movies2}}) = \frac{4}{10}$$
$$P(c = 1 \mid D = D_{\mathrm{movies2}}) = \frac{6}{10}$$

da Beispiele 3, 5, 7 und 8 als Klasse 0 klassifiziert sind und die übrigen Beispiele als Klasse 1 klassifiziert sind. Nun schauen wir uns die Wahrscheinlichkeitsverteilungen der einzelnen Merkmalsausprägungen bzgl. der verschiede-

nen Klassen an. Wir erhalten

$$P(x_1 = 0 \mid c = 0, D = D_{\text{movies2}}) = \frac{3}{4}$$
$$P(x_1 = 1 \mid c = 0, D = D_{\text{movies2}}) = \frac{1}{4}$$
$$P(x_2 = 0 \mid c = 0, D = D_{\text{movies2}}) = \frac{3}{4}$$
$$P(x_2 = 1 \mid c = 0, D = D_{\text{movies2}}) = \frac{1}{4}$$

für Klasse 0 und

$$P(x_1 = 0 \mid c = 1, D = D_{\text{movies2}}) = \frac{3}{6}$$
$$P(x_1 = 1 \mid c = 1, D = D_{\text{movies2}}) = \frac{3}{6}$$
$$P(x_2 = 0 \mid c = 1, D = D_{\text{movies2}}) = \frac{4}{6}$$
$$P(x_2 = 1 \mid c = 1, D = D_{\text{movies2}}) = \frac{2}{6}$$

für Klasse 1. Damit gilt für $x^* = (0,1)$:

$$\begin{aligned}
&P(c = 0 \mid D = D_{\text{movies2}}) P(x_1 = 0 \mid c = 0, D = D_{\text{movies2}}) \\
&\quad P(x_2 = 1 \mid c = 0, D = D_{\text{movies2}}) \\
=&\frac{4}{10} \cdot \frac{3}{4} \cdot \frac{1}{4} = \frac{3}{40}
\end{aligned}$$

$$\begin{aligned}
&P(c = 1 \mid D = D_{\text{movies2}}) P(x_1 = 0 \mid c = 1, D = D_{\text{movies2}}) \\
&\quad P(x_2 = 1 \mid c = 1, D = D_{\text{movies2}}) \\
=&\frac{6}{10} \cdot \frac{3}{6} \cdot \frac{2}{6} = \frac{1}{10}
\end{aligned}$$

und wegen $1/10 > 3/40$ damit

$$\text{clf}_D^{\text{NaiveBayes}}(x^*) = 1$$

Beachten Sie, dass die Klassifikation mit dem Naiven Bayes-Ansatz hier mit der Bayes-Klassifikation (siehe Beispiel 43) übereinstimmt. Betrachten wir nun den neuen Datenpunkt $x^{**} = (1,1)$ (d. h., ein in Europa gedrehter Film, der einen Oskar gewonnen hat), so erhalten wir

$$\begin{aligned}
&P(c = 0 \mid D = D_{\text{movies2}})P(x_1 = 1 \mid c = 0, D = D_{\text{movies2}}) \\
&\quad P(x_2 = 1 \mid c = 0, D = D_{\text{movies2}}) \\
=&\frac{4}{10} \cdot \frac{1}{4} \cdot \frac{1}{4} = \frac{1}{40}
\end{aligned}$$

$$\begin{aligned}
&P(c = 1 \mid D = D_{\text{movies2}})P(x_1 = 1 \mid c = 1, D = D_{\text{movies2}}) \\
&\quad P(x_2 = 1 \mid c = 1, D = D_{\text{movies2}}) \\
=&\frac{6}{10} \cdot \frac{3}{6} \cdot \frac{2}{6} = \frac{1}{10}
\end{aligned}$$

und wegen $1/10 > 1/40$ damit

$$\text{clf}_D^{\text{NaiveBayes}}(x^{**}) = 1$$

Beachten Sie, dass für den Datenpunkt x^{**} zuvor keine Bayes-Klassifikation möglich war.

Auch wenn die Unabhängigkeitsannahme des Naiven Bayes-Klassifikators nicht bei jedem Klassifikationsproblem gerechtfertigt ist, so wird er in der Praxis dennoch häufig erfolgreich eingesetzt. Im Unterschied zur Bayes-Klassifikation muss nicht der gesamte Trainingsdatensatz für die Klassifikation vorgehalten werden, es genügt hier, die Werte $P(c \mid D)$ und $P(x_i \mid c, D)$ für alle Kombinationen von Merkmalen x_i und Klassen $c \in Z$ zu berechnen und einzig diese vorzuhalten. Zumindest bei größeren Datensätzen ergibt sich hier ein signifikanter Speicherplatzvorteil gegenüber der Bayes-Klassifikation. Wie im vorherigen Beispiel deutlich gemacht wurde, müssen auch nicht alle Kombinationen von Merkmalsausprägungen im Trainingsdatensatz vorhanden sein, um auch

für beliebige (ungesehene) Datenpunkte plausible Klassifikationen vorherzusagen.

2.5.4 Kontinuierliche Merkmale

Gibt es in dem betrachteten Lernszenario eins oder mehrere kontinuierliche Merkmale, so ist die Maschinerie des Naiven Bayes-Klassifikators prinzipiell in gleicher Weise anwendbar. Wir müssen dazu nur von einer Wahrscheinlichkeitsverteilung über den Ausprägungen des Merkmals zu einer Wahrscheinlichkeitsdichte wechseln, d. h., in (20) anstatt die direkt aus den Daten berechenbare Verteilung $P(x_i \mid c, D)$ eine abgeschätzte Dichte $p(x_i \mid c, D)$ benutzen. Die Hauptschwierigkeit hierbei liegt darin, diese Dichte geeignet abzuschätzen, da (im Gegensatz zu einer Verteilung) gewisse zusätzliche Annahmen gemacht werden müssen.

Beispiel 45. Betrachten Sie den Datensatz D_{kont} in Tabelle 24. Wir haben hier ein kontinuierliches Merkmal Z_1 mit Merkmalsraum $\mathbb{R}$. Eine genaue Inspektion der Verteilung der Merkmalsausprägungen legt nahe, dass x_1 für die Klasse 0 normalverteilt mit Erwartungswert 2 und für die Klasse 1 normalverteilt mit Erwartungswert 4 ist.

Kann aus den Daten beispielsweise die Dichte eines Merkmals x_i für eine Klasse c als Normalverteilung mit Erwartungswert μ und Varianz σ^2 abgeschätzt werden, so können wir den entsprechenden Term $P(x_i \mid c, D)$ in (20) ersetzen durch

$$p(x_i \mid c, D) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x_i-\mu)^2}{2\sigma^2}}$$

Das Problem der Dichteabschätzung werden wir an dieser Stelle nicht weiter vertiefen, sondern noch einmal in Unterkapitel 3.4 „Anomalieerkennung“ aufgreifen.

Nr.	Z_1	Z
1	4.0	1
2	2.2	0
3	3.9	1
4	4.0	1
5	1.9	0
6	4.1	1
7	2.0	0
8	1.9	0
9	2.0	0
10	3.8	1

Tabelle 24: Datensatz D_{kont} zu Beispiel 45.

2.6 Entscheidungsbäume

Entscheidungsbäume sind Modelle für überwachtes Lernen, die sehr intuitiv die gelernten Regeln zur Klassifikation (oder auch Regression) darstellen und deswegen recht beliebt sind.

2.6.1 Modell

Ein Entscheidungsbaum ist ein gerichteter Baum, bei dem jeder Knoten eine Fallunterscheidung für den Wert eines Merkmals eines gegebenen Datenpunktes darstellt. Für jeden Fall gibt es einen eindeutigen Nachfolgerknoten und die Blätter des Baumes enthalten die möglichen Klassifikation des betrachteten Datenpunkts. Ein Pfad von der Wurzel des Entscheidungsbaums zu einem Blatt stellt somit eine Reihe von Entscheidungen bzgl. eines Datenpunktes dar, die zu einer Klassifikation führen.

Definition 19. Sei $X = X_1 \times \ldots \times X_n$ der Merkmalsraum von n Merkmalen und Y die Menge der Klassen (oder der Zielraum bei Regressionsproblemen). Ein *Entscheidungsbaum* T für X und Y ist ein Tupel $T = (V, E, r)$ mit

1. (V, E) ist ein (abwärts) gerichteter Baum mit Wurzel r.[18]

2. Ein Blattknoten $v \in V$ heißt auch *Klassifikationsknoten* mit $\mathrm{cl}(v) \in Y$.

3. Ein innerer Knoten $v \in V$ heißt auch *Entscheidungsknoten* mit (sei $\{v_1, \ldots, v_k\}$ die Menge der Nachfolger von v in T):

[18] Hierbei bezeichnet V die Menge der Knoten und $E \subseteq V \times V$ die Menge der (gerichteten) Kanten, die im folgenden implizit definiert werden.

(a) $\text{att}(v) \in \{1,\ldots,n\}$ ist das *Entscheidungsmerkmal* von v und

(b) $\text{succ}(v)$ ist eine *Entscheidungsfunktion* $\text{succ}(v) : X_{att(v)} \to \{v_1,\ldots,v_k\}$.

Ist $x = (x_1,\ldots,x_n)^T \in X_1 \times \ldots \times X_n$ ein Datenpunkt, so ist die Klassifikation von x bzgl. eines Entscheidungsbaums $T = (V,E,r)$ definiert via

$$\text{clf}_T(x) = \text{clf}_T(x,r)$$

mit (für alle $v \in V$)

$$\text{clf}_T(x,v) = \begin{cases} \text{cl}(v) & \text{falls } v \text{ Klassifikationsknoten} \\ \text{clf}_T(x,\text{succ}(v)(x_{\text{att}(v)})) & \text{sonst} \end{cases}$$

Die Entscheidungsfunktion $\text{succ}(v)$ (für engl. *successor*) eines Entscheidungsknotens v wird üblicherweise als einfacher Vergleich mit einer Konstanten realisiert (wie „$x_i < 10$“) und v besitzt dann zwei Nachfolger: einen falls der Vergleich positiv ausfällt und einen falls der Vergleich negativ ausfällt. Für Merkmale mit endlich vielen Ausprägungen erhält v einen Nachfolger für jede dieser Ausprägungen.

Beispiel 46. Wir betrachten wieder den $D_{\text{apartment}}$ von Beispiel 2 aus Unterkapitel 2.2. Der Datensatz ist noch einmal in Tabelle 25 dargestellt. Ein zu $D_{\text{apartment}}$ passender Entscheidungsbaum $T_{\text{apartment}}$ ist definiert durch

$T_{\text{apartment}} = (V_{\text{apartment}}, E_{\text{apartment}}, v_1)$ mit

$$V_{\text{apartment}} = \{v_1, \ldots, v_9\}$$
$$E_{\text{apartment}} = \{(v_1, v_2), (v_1, v_3), (v_3, v_4), (v_3, v_5), (v_5, v_6), (v_5, v_7), (v_7, v_8), (v_7, v_9)\}$$

$$\begin{aligned}
\text{cl}(v_2) &= 1 \\
\text{cl}(v_4) &= 0 \\
\text{cl}(v_6) &= 1 \\
\text{cl}(v_8) &= 0 \\
\text{cl}(v_9) &= 1
\end{aligned}$$

$$\text{att}(v_1) = 2$$
$$\text{succ}(v_1)(z) = \begin{cases} v_2 & \text{falls } z < 4 \\ v_3 & \text{sonst} \end{cases}$$

$$\text{att}(v_3) = 1$$
$$\text{succ}(v_3)(z) = \begin{cases} v_4 & \text{falls } z < 37 \\ v_5 & \text{sonst} \end{cases}$$

$$\text{att}(v_5) = 2$$
$$\text{succ}(v_5)(z) = \begin{cases} v_6 & \text{falls } z < 5 \\ v_7 & \text{sonst} \end{cases}$$

$$\text{att}(v_7) = 1$$
$$\text{succ}(v_7)(z) = \begin{cases} v_8 & \text{falls } z < 47 \\ v_9 & \text{sonst} \end{cases}$$

Eine Visualisierung des Entscheidungsbaums $T_{\text{apartment}}$ ist in Abbildung 44 zu finden. Es ist zu beobachten, dass der Entscheidungsbaum $T_{\text{apartment}}$ den Datensatz $D_{\text{apartment}}$ perfekt klassifiziert, d. h, es gilt beispielsweise

$$F1_0(D_{\text{apartment}}, \text{clf}_{T_{\text{apartment}}}) = F1_1(D_{\text{apartment}}, \text{clf}_{T_{\text{apartment}}}) = 1$$

Nr.	Fläche (in m^2, Merkmal 1)	Entfernung zum AP (in km, Merkmal 2)	OK
1	40	4	1
2	30	3	1
3	45	8	0
4	35	6	0
5	20	4	0
6	50	7	1
7	30	4	0
8	45	4	1
9	40	5	0
10	25	1	1

Tabelle 25: Datensatz $D_{\text{apartment}}$ zu Beispiel 46, siehe auch Beispiel 2 aus Unterkapitel 2.2.

Beispiel 47. Wir betrachten das klassische „Tennis"-Beispiel zu Entscheidungsbäumen [8] , bei dem es darum geht, anhand der Wetterlage (insbesondere bzgl. *Wetteransage*, *Temperatur*, *Luftfeuchtigkeit* und *Windstärke*) vorherzusagen, ob wir Tennis spielen sollen. Tabelle 26 zeigt unseren Datensatz D_{tennis}. Ein Entscheidungsbaum T_{tennis}, der D_{tennis} perfekt klassifiziert, ist in Abbildung 47 dargestellt (wir verzichten hier auf die formale Darstellung des Entscheidungsbaums). Beachten Sie in diesem Beispiel, dass nicht alle Merkmale für die Klassifikation erforderlich sind (es gibt keinen Entscheidungsknoten für *Temperatur*).

Das Lernen eines Entscheidungsbaumes aus einem Trainingsdatensatz geschieht üblicherweise nicht durch Lösen eines Optimierungsproblems (wie wir es bisher beispielsweise bei der logistischen Regression und den *Support Vector Machines* gesehen haben), sondern durch

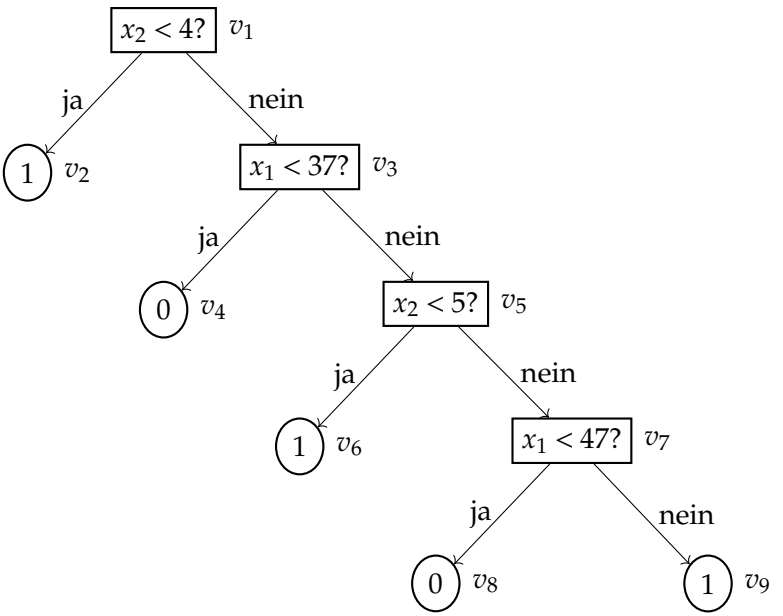


Abbildung 44: Entscheidungsbaum $T_{\text{apartment}}$ aus Beispiel 46.

dedizierte *prozedurale* Algorithmen. Die beiden bekanntesten Algorithmen dazu sind der *ID3-Algorithmus* und der *C4.5-Algorithmus*, die wir in den Abschnitten 2.6.2 bzw. 2.6.3 diskutieren werden.

2.6.2 Der ID3-Algorithmus

Wir betrachten zunächst ein binäres Klassifikationsszenario, bei dem die Merkmale in einem *endlichen* Merkmalsraum definiert sind. Für jedes Merkmal $i = 1, \ldots, n$ sei $Z_i = \{z_{i,1}, \ldots, z_{i,n_i}\}$ der entsprechende Merkmalsraum für ein $n_i \in \mathbb{N}$ und sei $Z = \{c_1, \ldots, c_k\}$ die Menge der Klassen. Sei also ein Datensatz $D = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ gegeben mit $x^{(j)} \in Z_1 \times \ldots \times Z_n$ und $y^{(j)} \in \{0, 1\}$ für $j = 1, \ldots, m$.

Nr.	Wetter-ansage	Tem-peratur	Luftfeuch-tigkeit	Wind-stärke	Tennis?
1	sonnig	heiß	hoch	schwach	Nein
2	sonnig	heiß	hoch	stark	Nein
3	bewölkt	heiß	hoch	schwach	Ja
4	regnerisch	gemäßigt	hoch	schwach	Ja
5	regnerisch	kalt	normal	schwach	Ja
6	regnerisch	kalt	normal	stark	Nein
7	bewölkt	kalt	normal	stark	Ja
8	sonnig	gemäßigt	hoch	schwach	Nein
9	sonnig	kalt	normal	schwach	Ja
10	regnerisch	gemäßigt	normal	schwach	Ja
11	sonnig	gemäßigt	normal	stark	Ja
12	bewölkt	gemäßigt	hoch	stark	Ja
13	bewölkt	heiß	normal	schwach	Ja
14	regnerisch	gemäßigt	hoch	stark	Nein

Tabelle 26: Datensatz D_{tennis} zu Beispiel 26.

Wie zuvor, gehen wir bei einem solchen Szenario auch davon aus, dass D als *Multimenge* repräsentiert ist, d. h., wir erlauben, dass Elemente mehrfach in D vorkommen können.

Sowohl der ID3-Algorithms (engl. *Iterative Dichotomiser 3*) als auch der im nächsten Abschnitt diskutierte C4.5-Algorithmus basieren auf dem Prinzip der *top-down induction of decision trees* (TDIDT). Dieses Prinzip baut einen Entscheidungsbaum auf Grundlage eines gegebenen Trainingsdatensatzes rekursiv von der Wurzel beginnend auf. Für einen Knoten v und einen Datensatz D, wird dazu zunächst das Entscheidungsmerkmal $i \in \{1,\ldots,n\}$ für v ausgewählt. Dabei soll das Merkmal gewählt werden, das den Datensatz D „am besten“ partitioniert. Wir lassen an dieser Stelle diese Auswahlfunktion zunächst abstrakt, wir werden uns weiter unten mit diesem Thema etwas detaillierter beschäftigen. Ist das

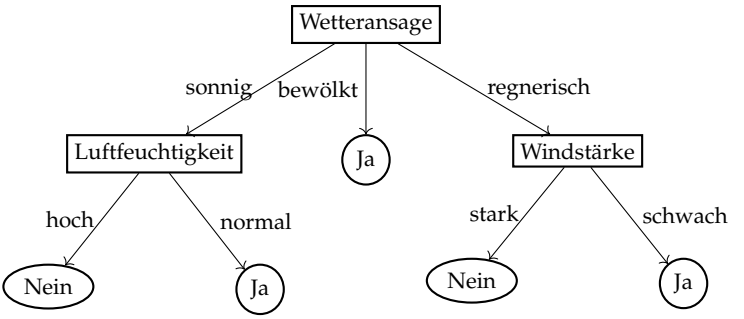


Abbildung 45: Entscheidungsbaum T_{tennis} aus Beispiel 47.

Merkmal i ausgewählt, so erzeugen wir für jede Merkmalsausprägung $z_{i,1}, \ldots, z_{i,n_i}$ einen entsprechenden Nachfolgeknoten $v_1, \ldots, v_{n_i}$. Dann partitionieren wir den Datensatz D entsprechend der Merkmalsausprägungen in Datensätze $D_1, \ldots, D_{n_i}$ mit

$$D_j = \{(x, y) \in D \mid x_i = z_j\}$$

für $j = 1, \ldots, n_i$. Dieses Verfahren wird rekursiv nun für alle Knoten $v_1, \ldots, v_{n_i}$ mit den entsprechenden Datensätzen $D_1, \ldots, D_{n_i}$ solange fortgeführt, bis alle Beispiele eines Datensatzes D_j eindeutig als 0 oder 1 klassifiziert sind oder die Beispiele nicht mehr unterschieden werden können (dieser Fall tritt ein, wenn die Beispiele identische Merkmalsausprägungen aber verschiedene Klassen haben). In beiden Fällen fügen wir an der entsprechenden Stelle einen Klassifikationsknoten ein (im zweiten Fall wählen wir dazu die in D am häufigsten vorkommende Klasse). Der Algorithmus TDIDT ist in dieser allgemeinen Form in Algorithmus 1 als Pseudocode noch einmal dargestellt (beachten Sie, dass der zurückgegebene Entscheidungsbaum $T = (V, E, r)$ durch den Wurzelknoten r charakteri-

Algorithmus 1 Der allgemeine TDIDT-Algorithmus für das Lernen von Entscheidungsbäumen

Eingabe: Datensatz D
Ausgabe: Wurzelknoten r des gelernten Baumes

TDIDT(D)

1: **if** „Alle Beispiele in D sind als 0 klassifiziert“ **then**
2: **return** „Neuer Klass.-knoten v mit $\mathrm{cl}(v) = 0$“
3: **if** „Alle Beispiele in D sind als 1 klassifiziert“ **then**
4: **return** „Neuer Klass.-knoten v mit $\mathrm{cl}(v) = 1$“
5: **if** „Alle Beispiele in D haben identische Merkmalsausprägungen“ **then**
6: **return** „Neuer Klass.-knoten v mit $\mathrm{cl}(v) = x$ wobei x die am häufigsten in D vorkommende Klasse ist“
7: $i = \mathrm{SelectFeature}(D)$
8: $v =$„Neuer Entscheidungsknoten mit $\mathtt{att}(v) = i$“
9: **for** $z \in Z_i$ **do**
10: $D' = \{(x, y) \in D \mid x_i = z\}$
11: $v' = \mathrm{TDIDT}(D')$
12: $\mathrm{succ}(v)(z) = v'$
13: **return** v

siert ist: die Mengen V und E werden nur implizit im Algorithmus gebildet).

Sei SelectFeature eine Funktion, sodass $D \neq D'$ für alle D' aus Zeile 10 von Algorithmus 1 gilt. Mit anderen Worten, SelectFeature wählt niemals ein Merkmal aus, das D nicht in wenigstens zwei Mengen partitioniert. Eine solche Auswahlfunktion nennen wir in diesem Kontext *rational*. Es sollte offensichtlich sein, dass der Algorithmus TDIDT für jede rationale Instanz der Funktion SelectFeature *korrekt* ist, d. h., die Ausgabe von TDIDT ist stets ein Entscheidungsbaum, der den Datensatz D vollständig korrekt klassifiziert (für den Fall, dass es

keine Beispiele mit identischen Merkmalsausprägungen aber verschiedenen Klassen gibt). Die konkrete Auswahl der Funktion SelectFeature kann aber durchaus Einfluss auf die Größe des gelernten Entscheidungsbaumes haben.

Beispiel 48. Wir führen Beispiel 47 fort und betrachten den Ablauf des Algorithmus 1 bei Aufruf TDIDT(D_{tennis}):

1. Zunächst ist zu beobachten, dass nicht alle Beispiele in D_{tennis} als 0 (*Nein*) oder 1 (*Ja*) klassifiziert sind und weiterhin die Beispiele in D_{tennis} nicht alle identische Merkmalsausprägungen haben. Die **if**-Bedingungen in Zeilen 1, 3 und 5 evaluieren also alle zu *falsch*.

2. Nehmen wir an, dass SelectFeature(D_{tennis}) in Zeile 7 das Merkmal *Wetteransage* auswählt.

3. In den Zeilen 9–12 läuft die **for**-Schleife über die Werte $z \in \{sonnig, bewölkt, regnerisch\}$.

   3.1 Für $z = sonnig$ ergibt sich in Zeile 10

   $$D' = \{1,2,8,9,11\}$$

   3.2 Wir rufen rekursiv TDIDT(D') auf:

   3.2.1 Die **if**-Bedingungen in Zeilen 1, 3 und 5 evaluieren für D' alle zu *falsch*.

   3.2.2 Nehmen wir an, dass SelectFeature(D') in Zeile 7 das Merkmal *Luftfeuchtigkeit* auswählt.

   3.2.3 In den Zeilen 9–12 läuft die **for**-Schleife über die Werte $z \in \{hoch, normal\}$.

   3.2.3.1 Für $z = hoch$ ergibt sich in Zeile 10

   $$D' = \{1,2,8\}$$

3.2.3.2 Wir rufen rekursiv TDIDT(D') auf: da alle Beispiele in D' als *Nein* klassifiziert sind, geben wir einen entsprechenden Klassifikationsknoten zurück.

3.2.3.3 Für z = *normal* ergibt sich in Zeile 10

$$D' = \{9, 11\}$$

3.2.3.4 Wir rufen rekursiv TDIDT(D') auf: da alle Beispiele in D' als *Ja* klassifiziert sind, geben wir einen entsprechenden Klassifikationsknoten zurück.

3.2.4 Wir geben einen Entscheidungsknoten für *Luftfeuchtigkeit* zurück, der entsprechend auf die beiden zuvor genannten Knoten weist.

3.3 Für z = *bewölkt* ergibt sich in Zeile 10

$$D' = \{3, 7, 12, 13\}$$

3.4 Wir rufen rekursiv TDIDT(D') auf: da alle Beispiele in D' als *Ja* klassifiziert sind, geben wir einen entsprechenden Klassifikationsknoten zurück.

3.5 Für z = *regnerisch* ergibt sich in Zeile 10

$$D' = \{4, 5, 6, 10, 14\}$$

3.6 Wir rufen rekursiv TDIDT(D') auf:

3.6.1 Die **if**-Bedingungen in Zeilen 1, 3 und 5 evaluieren für D' alle zu *falsch*.

3.6.2 Nehmen wir an, dass SelectFeature(D') in Zeile 7 das Merkmal *Windstärke* auswählt.

3.6.3 In den Zeilen 9–12 läuft die **for**-Schleife über die Werte $z \in \{stark, schwach\}$.

3.6.3.1 Für z = *stark* ergibt sich in Zeile 10

$$D' = \{6, 14\}$$

3.6.3.2 Wir rufen rekursiv TDIDT(D') auf: da alle Beispiele in D' als *Nein* klassifiziert sind, geben wir einen entsprechenden Klassifikationsknoten zurück.

3.6.3.3 Für z = *schwach* ergibt sich in Zeile 10

$$D' = \{4, 5, 6\}$$

3.6.3.4 Wir rufen rekursiv TDIDT(D') auf: da alle Beispiele in D' als *Ja* klassifiziert sind, geben wir einen entsprechenden Klassifikationsknoten zurück.

3.7 Wir geben einen Entscheidungsknoten für das Merkmal *Windstärke* zurück, der entsprechend auf die beiden zuvor genannten Knoten weist.

4. Wir geben einen Entscheidungsknoten für *Wetteransage* zurück, der entsprechend auf die drei zuvor genannten Knoten weist.

Beachten Sie, dass der oben konstruierte Baum dem Entscheidungsbaum aus Abbildung 45 entspricht.

Wir kommen nun zu dem noch offenen Punkt der Merkmalsauswahl in Schritt 7 von Algorithmus 1. Das Ziel der Funktion SelectFeature ist es, ein Merkmal so auszuwählen, dass der finale Entscheidungsbaum möglichst klein ist. Dies ist darin begründet, dass ein kleiner Entscheidungsbaum ein einfacheres Modell zur Erklärung der Daten darstellt als ein größerer Entscheidungsbaum (gegeben, dass beide Bäume die Daten gleich gut klassifizieren). Der ID3-Algorithmus wählt daher das Merkmal

$i \in \{1, \ldots, n\}$ aus, das den *höchsten Informationsgewinn* bei der Klassifizierung bietet. Formal wird dies durch die *Entropie* definiert.

Definition 20. Sei D ein Datensatz über zwei Klassen 0 und 1. Die *Entropie*[19] $H(D)$ von D ist definiert als

$$H(D) = -\frac{|\{(x,y) \in D \mid y = 0\}|}{|D|} \log_{10} \frac{|\{(x,y) \in D \mid y = 0\}|}{|D|} - \frac{|\{(x,y) \in D \mid y = 1\}|}{|D|} \log_{10} \frac{|\{(x,y) \in D \mid y = 1\}|}{|D|}$$

Ist D ein Datensatz über Klassen $Z = \{c_1, \ldots, c_k\}$, so ist die *Entropie* $H(D)$ von D allgemein definiert als

$$H(D) = -\sum_{i=1}^{k} \frac{|\{(x,c) \in D \mid c = c_i\}|}{|D|} \log_{10} \frac{|\{(x,c) \in D \mid c = c_i\}|}{|D|}$$

Beispiel 49. Wir setzen Beispiel 48 fort. Die Entropie von Datensatz D_{tennis} ist definiert als (Beispiele 1,2,6,8,14 werden als *Nein* und die übrigen Beispiele als *Ja* klassifiziert):

$$H(D_{\text{tennis}}) = -\frac{5}{14} \log_{10} \frac{5}{14} - \frac{9}{14} \log_{10} \frac{9}{14} \approx 0.283$$

Beachten Sie, dass $H(D) = 0$ gdw. alle Beispiele in D gleich klassifiziert sind und dass $H(D)$ maximal ist, wenn die Verteilung einer Gleichverteilung entspricht. Die Intuition hinter der Entropie $H(D)$ ist, dass diese das Maß der Unordnung bzgl. der Klassenverteilung angibt. Bei einem hohen Entropiewert ist es relativ schwierig, die richtige Klasse bzgl. der Daten D vorherzusagen, bei einem niedrigeren Wert ist dies einfacher.

[19] Wir definieren stets $0 \log_{10} 0 = 0$; für die folgenden Rechenbeispiele benutzen wir den Logarithmus zur Basis 10 (die konkrete Wahl der Basis ist allerdings irrelevant, solange sie bei allen Berechnungen dieselbe ist).

Definition 21. Sei D ein Datensatz und $i \in \{1,\dots,n\}$ ein Merkmal. Die *bedingte Entropie* $H(D \mid i)$ von D bzgl. i ist definiert als

$$H(D \mid i) = \sum_{j=1}^{n_i} \frac{|\{(x,y) \in D \mid x_i = z_{i,j}\}|}{|D|} H(\{(x,y) \in D \mid x_i = z_{i,j}\})$$

Die bedingte Entropie $H(D \mid i)$ gibt an, wie gut die einzelnen durch das Merkmal i entstandenen Teildatensätze klassifiziert sind. Ein hoher Wert bedeutet, dass die entstandenen Teildatensätze (gewichtet nach deren Größe) recht ungeordnet sind, ein niedriger Wert bedeutet, dass die Klassen durch das Merkmal i gut getrennt werden.

Beispiel 50. Wir setzen Beispiel 49 fort und berechnen die bedingten Entropien bzgl. der vier Merkmale *Wetteransage*, *Temperatur*, *Luftfeuchtigkeit* und *Windstärke*:

$$\begin{aligned}
&H(D_{\text{tennis}} \mid \textit{Wetteransage}) \\
=&\frac{5}{14} H(\{1,2,8,9,11\}) + \frac{4}{14} H(\{3,7,12,13\}) + \\
&\frac{5}{14} H(\{4,5,6,10,14\}) \\
=&\frac{5}{14}\left(-\frac{3}{5}\log_{10}\frac{3}{5} - \frac{2}{5}\log_{10}\frac{2}{5}\right) + \frac{4}{14}\left(-\frac{0}{4}\log_{10}\frac{0}{4} - \frac{4}{4}\log_{10}\frac{4}{4}\right) + \\
&\frac{5}{14}\left(-\frac{2}{5}\log_{10}\frac{2}{5} - \frac{3}{5}\log_{10}\frac{3}{5}\right) \\
\approx&0.104 + 0 + 0.104 = 0.208
\end{aligned}$$

$$
\begin{aligned}
&H(D_{\text{tennis}} \mid \textit{Temperatur}) \\
=&\frac{4}{14}H(\{1,2,3,13\}) + \frac{6}{14}H(\{4,8,10,11,12,14\}) + \\
&\frac{4}{14}H(\{5,6,7,9\}) \\
=&\frac{4}{14}\left(-\frac{2}{4}\log_{10}\frac{2}{4} - \frac{2}{4}\log_{10}\frac{2}{4}\right) + \frac{6}{14}\left(-\frac{2}{6}\log_{10}\frac{2}{6} - \frac{4}{6}\log_{10}\frac{4}{6}\right) + \\
&\frac{4}{14}\left(-\frac{1}{4}\log_{10}\frac{1}{4} - \frac{3}{4}\log_{10}\frac{3}{4}\right) \\
\approx&0.086 + 0.118 + 0.07 = 0.274
\end{aligned}
$$

$$
\begin{aligned}
&H(D_{\text{tennis}} \mid \textit{Luftfeuchtigkeit}) \\
=&\frac{7}{14}H(\{1,2,3,4,8,12,14\}) + \frac{7}{14}H(\{5,6,7,9,10,11,13\}) \\
=&\frac{7}{14}\left(-\frac{4}{7}\log_{10}\frac{4}{7} - \frac{3}{7}\log_{10}\frac{3}{7}\right) + \frac{7}{14}\left(-\frac{1}{7}\log_{10}\frac{1}{7} - \frac{6}{7}\log_{10}\frac{6}{7}\right) \\
\approx&0.148 + 0.089 = 0.237
\end{aligned}
$$

$$
\begin{aligned}
&H(D_{\text{tennis}} \mid \textit{Windstärke}) \\
=&\frac{8}{14}H(\{1,3,4,5,8,9,10,13\}) + \frac{6}{14}H(\{2,6,7,11,12,14\}) \\
=&\frac{8}{14}\left(-\frac{2}{8}\log_{10}\frac{2}{8} - \frac{6}{8}\log_{10}\frac{6}{8}\right) + \frac{6}{14}\left(-\frac{3}{6}\log_{10}\frac{3}{6} - \frac{3}{6}\log_{10}\frac{3}{6}\right) \\
\approx&0.14 + 0.129 = 0.269
\end{aligned}
$$

Definition 22. Sei D ein Datensatz und $i \in \{1,\ldots,n\}$ ein Merkmal. Der *Informationsgewinn* $IG(D,i)$ von D bzgl. i ist definiert als

$$IG(D,i) = H(D) - H(D \mid i)$$

Der Informationsgewinn $IG(D,i)$ gibt an, wie sehr die Merkmalsauswahl i die Daten in D nach den Klas-

sen (vor-)sortiert. Je höher der Informationsgewinn, desto besser klassifiziert das Merkmal i den Datensatz D. Beim ID3-Algorithmus wird die Funktion SelectFeature so definiert, dass dasjenige Merkmal ausgewählt, das den höchsten Informationsgewinn verspricht:

$$\text{SelectFeature}_{ID3}(D) = \arg \max_{i \in \{1,\ldots,n\}} IG(D,i)$$

Beachten Sie, dass wegen $IG(D,i) = H(D) - H(D \mid i)$ wir bei $\text{SelectFeature}_{ID3}(D)$ eigentlich nur $H(D \mid i)$ minimieren müssen, da der Term $H(D)$ merkmalsunabhängig ist. Wegen der Interpretierbarkeit von IG als Informationsgewinn, wird die Funktion $\text{SelectFeature}_{ID3}(D)$ aber üblicherweise wie oben definiert.

Beispiel 51. Wir setzen Beispiel 50 fort und berechnen

$$\begin{aligned}
&IG(D_{\text{tennis}}, \textit{Wetteransage}) \\
&= H(D_{\text{tennis}}) - H(D_{\text{tennis}} \mid \textit{Wetteransage}) \\
&\approx 0.283 - 0.208 = 0.075
\end{aligned}$$

$$\begin{aligned}
&IG(D_{\text{tennis}}, \textit{Temperatur}) \\
&= H(D_{\text{tennis}}) - H(D_{\text{tennis}} \mid \textit{Temperatur}) \\
&\approx 0.283 - 0.274 = 0.009
\end{aligned}$$

$$\begin{aligned}
&IG(D_{\text{tennis}}, \textit{Luftfeuchtigkeit}) \\
&= H(D_{\text{tennis}}) - H(D_{\text{tennis}} \mid \textit{Luftfeuchtigkeit}) \\
&\approx 0.283 - 0.237 = 0.046
\end{aligned}$$

$$\begin{aligned}
&IG(D_{\text{tennis}}, \textit{Windstärke}) \\
&= H(D_{\text{tennis}}) - H(D_{\text{tennis}} \mid \textit{Windstärke}) \\
&\approx 0.283 - 0.269 = 0.014
\end{aligned}$$

Die Funktion $\text{SelectFeature}_{ID3}$ würde also hier das Merkmal *Wetteransage* auswählen.

2.6.3 Der C4.5-Algorithmus

Der C4.5-Algorithmus ist eine Weiterentwicklung des ID3-Algorithmus. Der allgemeine Aufbau dieses Algorithmus ist identisch zum TDIDT-Algorithmus aus Algorithmus 1. Der C4.5-Algorithmus enthält allerdings eine Reihe von Optimierungen, drei der wichtigsten sind die folgenden:

1. Der ID3-Algorithmus tendiert bei leicht verrauschten Daten schnell zur Überanpassung, d. h., es wird ein Entscheidungsbaum mit sehr langen Pfaden von der Wurzel bis zu den Blättern gelernt. Der C4.5-Algorithmus enthält einen zusätzlichen Nachbearbeitungsschritt, der den Entscheidungsbaum *kürzt*. Dazu werden zunächst die im Entscheidungsbaum implizit vorhandenen Klassifikationsregeln[20] extrahiert. Anschließend wird versucht, „wenig relevante“ Vorbedingungen der Regeln zu erkennen und zu entfernen (dies geschieht beim C4.5-Algorithmus mit proprietären Heuristiken). Anschließend wird aus den Regeln ein neuer (kleinerer) Entscheidungsbaum konstruiert.

2. Um mit Datensätzen mit kontinuierlichen Merkmalen zu arbeiten, werden die entsprechenden Merkmalsräume in zwei Intervalle aufgeteilt und so jedes kontinuierliche Merkmal in ein Merkmal mit endlich vielen Ausprägungen konvertiert. Die Hauptschwierigkeit besteht darin, die Intervallsgrenzen festzulegen. Schauen wir uns den Datensatz D_{tennis2} in Tabelle 27 an, bei dem das Merkmal *Temperatur*

[20] Ein Beispiel für eine solche Klassifikationsregel für den Entscheidungsbaum in Abbildung 45 ist etwa „Wenn Wetteransage=sonnig und Luftfeuchtigkeit=hoch, dann Tennis=Ja.“

Nr.	Temperatur (in °C)	Tennis?
1	4	Nein
2	9	Nein
3	16	Ja
4	22	Ja
5	27	Ja
6	32	Nein

Tabelle 27: Datensatz D_{tennis2}.

nun kontinuierlich ist. Sinnvolle Intervallgrenzen liegen stets zwischen zwei Merkmalsausprägungen, bei denen die zugehörigen Beispiele unterschiedliche Klassen haben (es macht im Beispiel also keinen Sinn, die Intervallgrenze auf 8 zu setzen, da die Grenze 10 immer besser partitionieren wird). Wir wählen dann stets die Mitte der jeweiligen Merkmalsausprägungen und erhalten damit eine Reihe neuer Merkmale. Im Beispiel erhalten wir zwei Merkmale $\mathit{Temperatur}_{\leq 12.5}$ und $\mathit{Temperatur}_{\leq 29.5}$. Das Merkmal $\mathit{Temperatur}_{\leq 12.5}$ partitioniert dabei den Datensatz D_{tennis2} in $\{1,2\}$ (Beispiele mit Temperatur kleiner oder gleich 12.5 °C) und $\{3,4,5,6\}$ (Beispiele mit Temperatur größer 12.5 °C). Das Merkmal $\mathit{Temperatur}_{\leq 29.5}$ partitioniert den Datensatz D_{tennis2} in $\{1,2,3,5\}$ (Beispiele mit Temperatur kleiner oder gleich 29.5 °C) und $\{6\}$ (Beispiele mit Temperatur größer 29.5 °C). Diese beiden neuen Merkmale ersetzen nun das Merkmal *Temperatur* in einem Vorverarbeitungsschritt und Algorithmus 1 kann in gleicher Weise ausgeführt werden.

3. Der C4.5-Algorithmus benutzt eine optimierte Version des Informationsgewinns als Auswahlfunkti-

on für Merkmale. Ein Problem mit der ursprünglichen Version ist, dass sie Merkmale mit einer hohen Anzahl an Merkmalsausprägungen bevorzugt. Betrachten Sie dazu in unserem Tennis-Beispiel im Extremfall ein Merkmal *Datum*. Unser Datensatz wird höchstwahrscheinlich pro Tag maximal ein Beispiel enthalten, das Merkmal *Datum* klassifiziert also direkt jedes Beispiel perfekt in einem Schritt und hat damit maximalen Informationsgewinn. Der gelernte Entscheidungsbaum besteht also nur aus dem Wurzelknoten mit Nachfolgern für jedes im Trainingsdatensatz vorkommenden Datum. Natürlich ist dieser Entscheidungsbaum extrem an den Trainingsdatensatz überangepasst. Beim C4.5-Algorithmus wird der Informationsgewinn entsprechend skaliert, um dieses Phänomen zu vermeiden.

Weitere Optimierungen des C4.5-Algorithmus beinhalten die Behandlung von unvollständiger Information (beispielsweise einzelne fehlende Merkmalsausprägungen der Beispiele) und Einbeziehung von Gewichtungen der Merkmale.

2.6.4 Entscheidungswälder

In den Unterkapiteln 2.1–2.6 haben wir eine Reihe verschiedener Modelle und Algorithmen zum überwachten maschinellen Lernen kennengelernt. Diese haben unterschiedliche Stärken und Schwächen und eignen sich für verschiedene Anwendungsszenarien bzw. Typen von Datensätzen mal besser und mal schlechter. Eine allgemeine Methodik, um die Vorteile verschiedener Modelltypen (bzw. Modelle des selben Typs mit verschiedenen Parametern) zu kombinieren, ist das sogenannte *Ensemble*-Lernen. Bei dieser Methodik werden viele verschiede-

ne Modelle gleichzeitig gelernt und die Resultate bei der Anwendung geeignet aggregiert. Ensemble-Lernen kann insbesondere auch dazu genutzt werden, der Überanpassung einzelner Modelle entgegenzuwirken. Dazu werden die verschiedenen Modelle nur mit einer Teilmenge der zur Verfügung stehenden Daten trainiert. In diesem Abschnitt schauen wir uns ein Ensemble-Lernverfahren für Entscheidungsbäume an, das diesen letzten Punkt adressiert.

Ein *Entscheidungswald* F (engl. *random forest*) ist eine Menge von Entscheidungsbäumen $F = \{T_1, \ldots, T_l\}$. Ist x ein Datenpunkt, so ist die Klassifikation von x bzgl. des Entscheidungswalds F definiert via

$$\mathrm{clf}_F(x) = \begin{cases} 0 & |\{j \mid \mathrm{clf}_{T_j}(x) = 0\}| > |\{j \mid \mathrm{clf}_{T_j}(x) = 1\}| \\ 1 & \text{sonst} \end{cases}$$

Mit anderen Worten, F klassifiziert x als 0 gdw. eine Mehrheit der Bäume in T den Punkt x als 0 klassifiziert. Bei einem nicht-binären Klassifikationsproblem wird die Klasse ausgewählt, die häufiger als jede andere Klasse auftritt. Bei Anwendung auf ein Regressionsproblem wird der Durchschnittswert aller Vorhersagen bestimmt.

Um einen Entscheidungswald F aus einem Datensatz $D = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ zu lernen, benutzt man l-mal einen Algorithmus zum Lernen eines Entscheidungsbaums (wie ID3 oder C4.5), allerdings stets auf einem leicht modifizierten Datensatz $D_1, \ldots, D_l$. Der übliche Ansatz zum Erstellen von D_j, $j = 1, \ldots, l$ ist dabei wie folgt. Wir starten mit einer leeren Menge und wählen ein Element von D zufällig gleichverteilt aus. Dies wiederholen wir bis D_j genau m Elemente enthält. Da dies *Ziehen mit Zurücklegen* entspricht, können Beispiele aus D in einem D_j mehrfach auftreten oder gar nicht. Mit hoher Wahrscheinlichkeit unterscheiden sich dann die ge-

lernten Bäume $T_1, \ldots, T_l$ voneinander und die Klassifikation mit clf_F ist dadurch robuster. Dieser Ansatz eines Ensemble-Lernverfahrens nennt man auch *bootstrap aggregating* (oder *bagging*) und kann prinzipiell in gleicher Weise mit jedem überwachten Lernverfahren (oder einer Mischung verschiedener überwachter Lernverfahren) genutzt werden.

Beim Entscheidungswald ändert man den obigen Algorithmus noch ein wenig ab, um weitere Randomisierung in das Lernverfahren zu bringen. Dies geschieht dadurch, dass man beim Lernen eines einzelnen Entscheidungsbaum in Zeile 7 (siehe Algorithmus 1) das Merkmal für den aktuellen Entscheidungsknoten nicht aus der gesamten Anzahl verfügbarer Merkmale auswählt, sondern aus einer zufällig gewählten Teilmenge. Dadurch ist sichergestellt, dass ein evtl. vorhandenes dominantes Merkmal (also ein Merkmal, das einen sehr hohen Informationsgewinn im Trainingsdatensatz hat) nicht immer in den gelernten Bäumen in der Wurzel gewählt wird. Dadurch wird der Überanpassung weiter entgegengewirkt. Die Größe dieser Teilmenge, die bei jeder Wahl eines Merkmals neu bestimmt wird, ist ein Parameter des Lernverfahrens, genau wie die Anzahl l der Entscheidungsbäume. Oft in der Praxis auftretende Werte sind $\sqrt{n}$ für die Anzahl zufälliger Merkmale, aus denen ein Entscheidungsmerkmal gewählt wird und $l = 100$.

3 Unüberwachtes Lernen

Überblick über dieses Kapitel

Im Gegensatz zum überwachten Lernen, stehen bei Problemen des *unüberwachten Lernens* keine annotierten Trainingsdaten zur Verfügung. Mit anderen Worten besteht die Datengrundlage nicht aus Beispielen $D = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$, zu denen wir die Klasse oder den Funktionswert schon kennen, sondern nur aus Datenpunkten $E = \{x^{(1)}, \ldots, x^{(m)}\}$. Die Aufgabe eines unüberwachten Lernalgorithmus ist es dann, eigenständig Struktur in den Daten zu erkennen. Das prototypische Problem des unüberwachten Lernens ist die *Clusteranalyse*, bei der E in ähnliche Gruppen aufzuteilen ist. Andere Probleme sind beispielsweise die *Anomalieerkennung* oder das Lernen von Regeln.

Wir werden uns in diesem Kapitel mit fünf Methoden des unüberwachten Lernens beschäftigten. In Unterkapitel 3.1 schauen wir uns mit dem K-Means-Algorithmus einen klassischen Ansatz für die Clusteranalyse an. In Unterkapitel 3.2 beschäftigen wir uns mit einem weiteren Clusteringverfahren, dem hierarchischen Clustering. In Unterkapitel 3.3 diskutieren wir das Assoziationsregellernen und in Unterkapitel 3.4 die Anomaliedetektion. Schließlich schauen wir uns in Unterkapitel 3.5 noch die Grundelemente der *Principal Component Analysis* an.

Bibliographische Anmerkungen

Weiterführende Literatur zu unüberwachtem Lernen (und auch allgemeines maschinelles Lernen) bietet das Buch [15]. Weiterhin ist der Coursera-Kurs „Unsupervised Lear-

ning, Recommenders, Reinforcement Learning“ von Andrew Ng [12] sehr zu empfehlen.

Eine allgemeine Einführung in die Clusteranalyse bietet [7]. Einen Überblick über K-Means-Clustering (Unterkapitel 3.1) findet sich insbesondere in Kapitel 2 in [7] und eine Kurzusammenfassung gibt es in [4, Kapitel 9.2.1]. Näheres zum *Divisive Analysis Clustering* aus Unterkapitel 3.2 zum hierarchischen Clustering findet sich in Kapitel 6 von [7], sowie in [15, Kapitel 13]. Assoziationsregellernen (Unterkapitel 3.3) wird in [3, Kapitel 5] detailliert diskutiert, wobei auch die Originalarbeiten zum Apriori-Algorithmus [2] und zum FP-Growth-Algorithmus [6] zu empfehlen sind. Anomalieerkennung (Unterkapitel 3.4) wird in [4, Kapitel 7.3] behandelt. Einen kurzen Überblick zur *Principal Component Analysis* (Unterkapitel 3.5) findet sich beispielsweise in [5, Kapitel 2.12].

3.1 K-Means-Clustering

Bei der *Clusteranalyse* geht es darum, $E = \{x^{(1)}, \ldots, x^{(m)}\}$ in Teilmengen $E_1, \ldots, E_k$ zu partitionieren[21], sodass die Datenpunkte in jedem E_i *ähnlich* zueinander sind ($i = 1, \ldots, k$), wohingegen Datenpunkte in verschiedenen E_i, E_j *unähnlich* zueinander sind ($i, j = 1, \ldots, k$, $i \neq j$). Da wir zu den Datenpunkten in E keine Klassifikation haben, besteht also die Hauptaufgabe darin, eine Menge von Klassen und eine Zuordnung von Klassen zu Datenpunkten zu finden.

3.1.1 Clusteranalyse und K-Means

Wir beginnen die Diskussion der Clusteranalyse mit zwei Beispielen.

Beispiel 52. Sie betreiben ein Geschäft, in dem Sie unter anderem Film-DVDs verkaufen. Sie haben dazu genau drei Regale reserviert und möchten gerne die Filme so auf die drei Regale aufteilen, dass Filme gleichen Genres zusammen stehen. Allerdings haben Sie außer der Spielzeit und der Filmkosten keine weiteren Informationen zu den Filmen (und nicht die Zeit, sich alle anzuschauen). Tabelle 28 und Abbildung 46 zeigen den vorhandenen Datensatz E_{movies}.

Beispiel 53. Wir schauen uns noch einmal die Daten aus Beispiel 1 von Unterkapitel 2.1 an, diesmal aber unter einer anderen Fragestellung. Unsere Daten enthalten Informationen zu Körpergröße und Ringfingerumfang einiger Personen, aber anders als in Beispiel 1 von Unterkapitel 2.1 interpretieren wir den Ringfingerumfang nicht als Zielvariable, sondern als ein weiteres Merkmal. Wir nehmen die Rolle eines Juweliers an, möchten

[21] Das heißt, $E_1 \cup \ldots \cup E_k = E$ und $E_i \cap E_j = \emptyset$ für $i, j = 1, \ldots, k$ mit $i \neq j$.

Nr.	Länge (in Min.)	Kosten (in Mio. EUR)
1	120	250
2	80	12
3	125	230
4	115	40
5	85	24
6	118	255
7	117	35
8	82	9
9	85	15
10	130	280
11	124	290
12	110	30
13	90	26
14	118	240

Tabelle 28: Datensatz E_{movies} aus Beispiel 52.

aber zu unseren Ringen nicht alle möglichen Größen anbieten, sondern nur 2, 3 oder maximal 4 verschiedene Standardgrößen (unter der Annahme, dass Personen mit „ähnlicher" Ringgröße zu der passenden Standardgröße greifen). Die Fragestellung ist hier: was ist eine geeignete Unterteilung der Ringgrößen und auf wie viele Standardgrößen kann ich mich einschränken, sodass meine Kunden immer noch „zufrieden" sind?

Das letzte Beispiel ist ein klassisches Beispiel für eine der wichtigsten Anwendungsdomänen des Clusterings, nämlich die *Marktsegmentierung*.

Der klassische Algorithmus der Clusteranalyse ist das *K-Means-Clustering*, das sowohl namentlich als auch konzeptuell große Ähnlichkeiten zum KNN-Algorithmus für überwachtes Lernen hat (siehe Unterkapitel 2.4). Neben

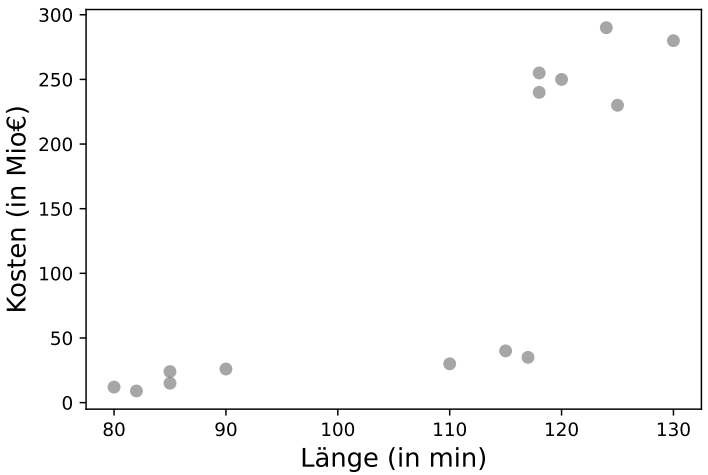


Abbildung 46: Datensatz E_{movies} aus Beispiel 52.

Nr.	Körpergröße (in cm)	Ringfingerumfang (in mm)
1	153.3	47.1
2	158.9	46.8
3	160.8	49.3
4	179.6	53.2
5	156.6	47.7
6	165.1	49.0
7	165.9	50.6
8	156.7	47.1
9	167.8	51.7
10	160.8	47.8

Tabelle 29: Datensatz E_{ring} zu Beispiel 53, siehe auch Beispiel 1 in Unterkapitel 2.1.

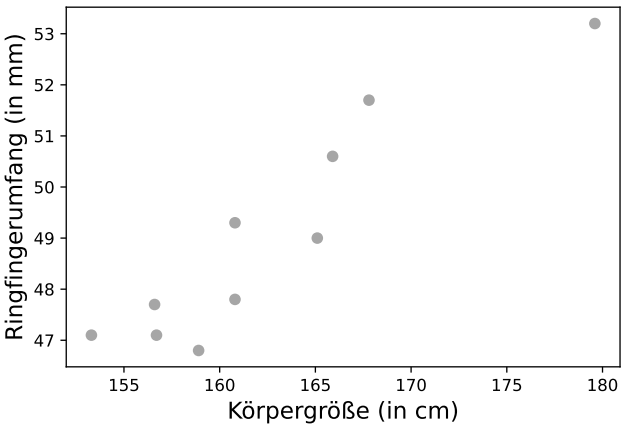


Abbildung 47: Datensatz E_{ring} zu Beispiel 53, siehe auch Beispiel 1 in Unterkapitel 2.1.

einem Datensatz E benötigt der Algorithmus noch die Spezifikation der Anzahl der Cluster k, die gelernt werden sollen. Der *naive K-Means-Algorithmus* (oder auch *Lloyds Algorithmus* genannt) ist eine heuristische Methode, um E so in k Cluster zu unterteilen, so dass die Summe der euklidischen Distanzen von jedem Datenpunkt zu dem Mittelpunkt des ihm zugewiesenen Clusters minimal ist. Der Algorithmus operiert dabei iterativ und startet von einer gegebenen Menge von Clustermittelpunkten (auch *Zentroiden* genannt) $\{m_1, \ldots, m_k\}$. Jedem Datenpunkt $x \in E$ wird derjenige Zentroid m_i $(i = 1, \ldots, k)$ zugewiesen, der ihm am nächsten ist (bzgl. der euklidischen Distanz). Anschließend aktualisieren wir m_i durch den Mittelwert aller m_i zugewiesenen Datenpunkte $(i = 1, \ldots, k)$. Diese beiden Schritte wiederholen wir, bis die Zentroiden konvergieren (dies ist bei Verwendung der euklidischen Distanz garantiert). Algorithmus 2 formali-

Algorithmus 2 Der naive K-Means-Algorithmus.

Eingabe: Datensatz E, Clusterzahl k
Ausgabe: $m_1, \ldots, m_k$ Zentroiden des finalen Clusterings

KMEANS(E,k)

1: Wähle initiale Zentroiden $m_1, \ldots, m_k$
2: **while** nicht konvergiert **do**
3: **for** $x \in E$ **do**
4: $cluster(x) = \arg\min_{i=1,\ldots,k} ||x - m_i||$
5: **for** $i = 1, \ldots, k$ **do**
6: $m_i = \frac{1}{|\{x|cluster(x)=i\}|} \sum_{x,cluster(x)=i} x$
7: **return** $m_1, \ldots, m_k$

siert die beschriebene Methode. Neben dem Parameter k kann sich die initiale Wahl der Zentroiden in Zeile 1 stark auf das Endresultat auswirken. Üblicherweise wählt man hier zufällig gleichverteilt k Elemente aus E als initiale Zentroiden aus und führt den Algorithmus wiederholt aus, um das bestmögliche Clustering zu erreichen. Wir werden uns mit diesem Aspekt noch etwas genauer in Abschnitt 3.1.3 beschäftigen.

Bevor wir den K-Means-Algorithmus an den Datensätzen aus den Beispielen 52 und 53 anwenden, schauen wir uns zunächst einen anderen künstlichen Datensatz an und führen den Algorithmus im Detail aus.

Beispiel 54. Wir betrachten den Datensatz $E_1 = \{x^{(1)}, \ldots, x^{(10)}\}$ aus Tabelle 30 und Abbildung 48. Nehmen wir an, wir rufen den Algorithmus KMEANS mit E_1 sowie $k = 2$ auf. Die folgenden Schritte werden dann ausgeführt:

1. Wir wählen initial $m_1 = x^{(6)} = (3.5, 3.5)^T$ und $m_2 = x^{(5)} = (4.1, 3.2)^T$.

Nr.	x_1	x_2
1	2	3
2	4	4.2
3	2.8	2.1
4	1.9	2
5	4.1	3.2
6	3.5	3.5
7	3.8	4.1
8	2	1.9
9	2.9	3.1
10	4	3.9

Tabelle 30: Datensatz E_1 zu Beispiel 54.

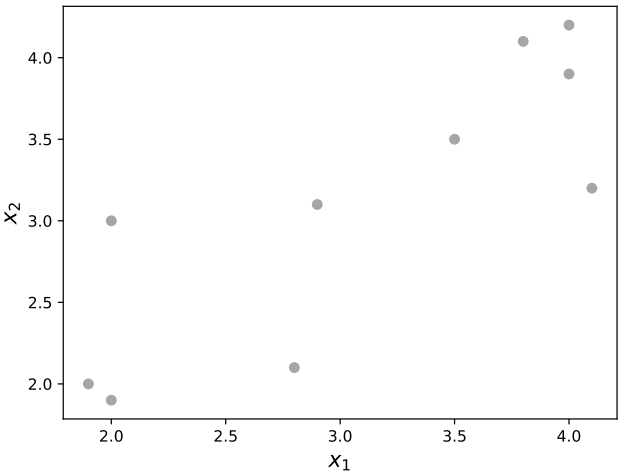


Abbildung 48: Datensatz E_1 zu Beispiel 54.

2. Wir berechnen für alle $x \in E_1$ die Clusterzugehörigkeit.

Beispielsweise gilt für $x^{(7)} = (3.8, 4.1)^T$:

$$\begin{aligned}
\|x^{(7)} - m_1\| &= \|(3.8, 4.1)^T - (3.5, 3.5)^T\| \\
&= \|(0.3, 0.6)^T\| = \sqrt{0.3^2 + 0.6^2} \approx 0.671 \\
\|x^{(7)} - m_2\| &= \|(3.8, 4.1)^T - (4.1, 3.2)^T\| \\
&= \|(-0.3, 0.9)^T\| = \sqrt{(-0.3)^2 + 0.9^2} \approx 0.949
\end{aligned}$$

und somit wird $x^{(7)}$ dem Cluster 1 bzw. dem Zentroiden m_1 zugewiesen. Insgesamt erhalten wir

$$\begin{aligned}
&cluster(x) = 1 \qquad \text{für } x \in \{x^{(1)}, x^{(2)}, x^{(3)}, x^{(4)}, x^{(6)}, x^{(7)}, \\
&\qquad\qquad\qquad\qquad x^{(8)}, x^{(9)}, x^{(10)}\} \\
&cluster(x) = 2 \qquad \text{für } x \in \{x^{(5)}\}
\end{aligned}$$

Abbildung 49 zeigt das Ergebnis der ersten Iteration des Algorithmus.

3. Wir berechnen die Zentroiden m_1 und m_2 neu:

$$\begin{aligned}
m_1 &:= \frac{1}{9}(x^{(1)} + x^{(2)} + x^{(3)} + x^{(4)} + x^{(6)} + x^{(7)} + \\
&\qquad\qquad x^{(8)} + x^{(9)} + x^{(10)}) \\
&\approx (2.989, 3.089)^T \\
m_2 &:= \frac{1}{1}(x^{(5)}) = x^{(5)} = (4.1, 3.2)^T
\end{aligned}$$

4. Wir berechnen für alle $x \in E_1$ wieder die Clusterzugehörigkeit. Insgesamt erhalten wir

$$\begin{aligned}
&cluster(x) = 1 \qquad \text{für } x \in \{x^{(1)}, x^{(3)}, x^{(4)}, x^{(6)}, x^{(8)}, x^{(9)}\} \\
&cluster(x) = 2 \qquad \text{für } x \in \{x^{(2)}, x^{(5)}, x^{(7)}, x^{(10)}\}
\end{aligned}$$

Abbildung 50 zeigt das Ergebnis der zweiten Iteration des Algorithmus.

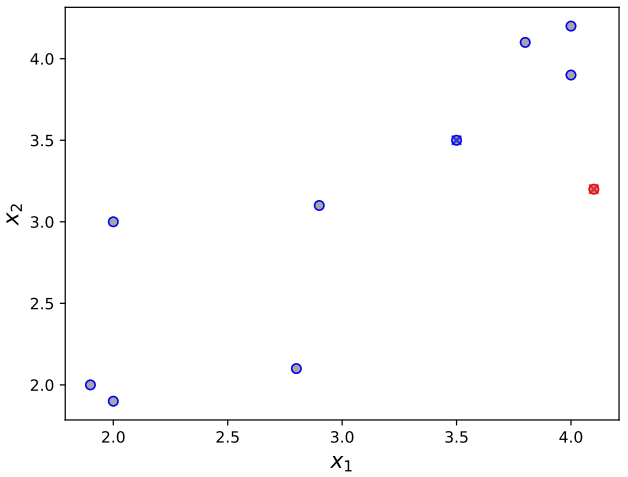


Abbildung 49: Erste Iteration von $KMEANS(E_1, 2)$ zu Beispiel 54. Zentroiden sind durch farbige Kreuze und die Zuordnung von Datenpunkten zu ihren zugehörigen Zentroiden ist durch farbige Kreise dargestellt.

5. Wir berechnen die Zentroiden m_1 und m_2 neu:

$$\begin{aligned} m_1 &:= \frac{1}{6}(x^{(1)} + x^{(3)} + x^{(4)} + x^{(6)} + x^{(8)} + x^{(9)}) \\ &= (2.517, 2.6)^T \\ m_2 &:= \frac{1}{4}(x^{(2)} + x^{(5)} + x^{(7)} + x^{(10)}) = (3.975, 3.85)^T \end{aligned}$$

6. Wir berechnen für alle $x \in E_1$ wieder die Clusterzugehörigkeit. Insgesamt erhalten wir

$$\begin{aligned} &cluster(x) = 1 \qquad \text{für } x \in \{x^{(1)}, x^{(3)}, x^{(4)}, x^{(8)}, x^{(9)}\} \\ &cluster(x) = 2 \qquad \text{für } x \in \{x^{(2)}, x^{(5)}, x^{(6)}, x^{(7)}, x^{(10)}\} \end{aligned}$$

Abbildung 51 zeigt das Ergebnis der dritten Iteration des Algorithmus.

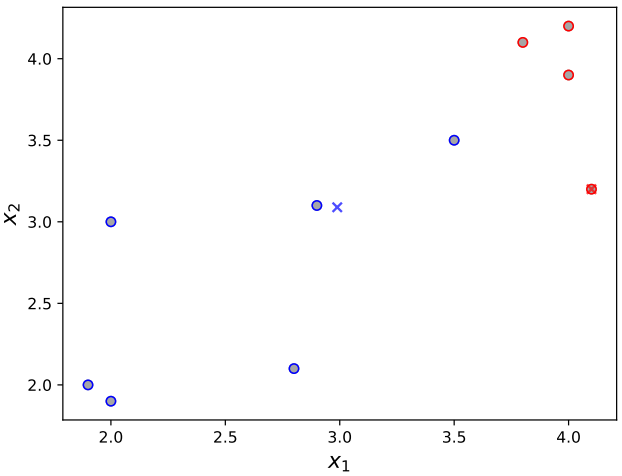


Abbildung 50: Zweite Iteration von $KMEANS(E_1, 2)$ zu Beispiel 54. Zentroiden sind durch farbige Kreuze und die Zuordnung von Datenpunkten zu ihren zugehörigen Zentroiden ist durch farbige Kreise dargestellt.

7. Wir berechnen die Zentroiden m_1 und m_2 neu:

$$m_1 := \frac{1}{6}(x^{(1)} + x^{(3)} + x^{(4)} + x^{(8)} + x^{(9)}) = (2.32, 2.42)^T$$
$$m_2 := \frac{1}{4}(x^{(2)} + x^{(5)} + x^{(6)} + x^{(7)} + x^{(10)}) = (3.88, 3.78)^T$$

8. Wir berechnen für alle $x \in E$ wieder die Clusterzugehörigkeit. Insgesamt erhalten wir

$$cluster(x) = 1 \qquad \text{für } x \in \{x^{(1)}, x^{(3)}, x^{(4)}, x^{(8)}, x^{(9)}\}$$
$$cluster(x) = 2 \qquad \text{für } x \in \{x^{(2)}, x^{(5)}, x^{(6)}, x^{(7)}, x^{(10)}\}$$

und damit die identische Clusterzuweisung wie im Schritt zuvor. Abbildung 52 zeigt das Ergebnis der vierten Iteration des Algorithmus.

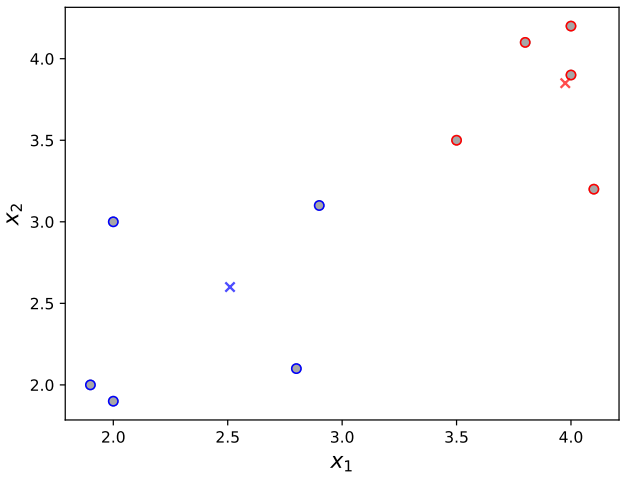


Abbildung 51: Dritte Iteration von $KMEANS(E_1, 2)$ zu Beispiel 54. Zentroiden sind durch farbige Kreuze und die Zuordnung von Datenpunkten zu ihren zugehörigen Zentroiden ist durch farbige Kreise dargestellt.

9. Eine Neuberechnung der Zentroiden erübrigt sich, da die Clusterzuordnung konvergiert ist.

Durch die Verwendung der euklidischen Distanz ist der naive K-Means-Algorithmus sehr anfällig für unterschiedlich skalierte Merkmale (siehe die Diskussion in Abschnitt 2.4.2), wie man am Datensatz E_{movies} aus Beispiel 52 sehen kann.

Beispiel 55. Wir führen Beispiel 52 fort. Eine Ausführung von $KMEANS(E_{\text{movies}}, 3)$ führt zu den Clustern wie in Abbildung 53 dargestellt. Dies entspricht nicht dem gewünschten Ergebnis. Insbesondere wurden alle Datenpunkte der beiden unteren Gruppen zu einem Cluster zusammengelegt. Das Problem liegt hier daran, dass das

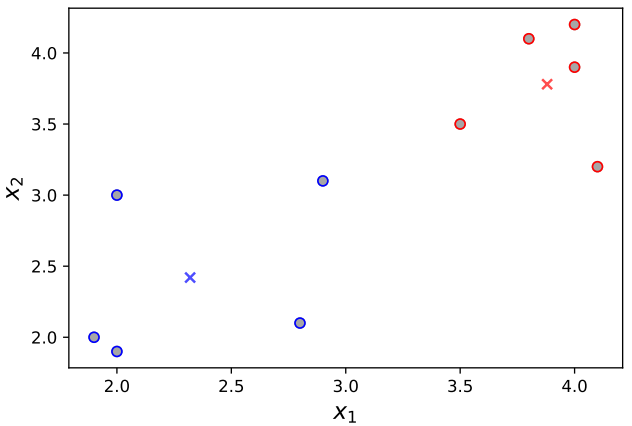


Abbildung 52: Vierte Iteration von $KMEANS(E_1, 2)$ zu Beispiel 54. Zentroiden sind durch farbige Kreuze und die Zuordnung von Datenpunkten zu ihren zugehörigen Zentroiden ist durch farbige Kreise dargestellt.

Merkmal *Kosten* auf einer größeren Skala (9–290 Mio. EUR) verteilt ist als das Merkmal *Länge* (80–130 Min.). Aus diesem Grund wirken sich Unterschiede im Merkmal *Länge* durch die Verwendung der euklidischen Distanz weniger signifikant aus als im Merkmal *Kosten*. Sei $\hat{E}_{\text{movies}}$ der z-transformierte Datensatz zu E_{movies}.[22] Eine Ausführung des Algorithmus $KMEANS(\hat{E}_{\text{movies}}, 3)$ führt zu den Clustern wie in Abbildung 54 dargestellt.

Beispiel 56. Wir führen Beispiel 53 fort. Abbildungen 55, 56 und 57 zeigen Clusterings für die Werte $k = 2, 3, 4$. Alle Clusterings erscheinen plausibel, gegeben der jeweiligen Anzahl an Clustern. Dennoch bleibt die Frage of-

[22] Siehe Definition 3 in Unterkapitel 2.4. Die z-Transformation für Datensätze über Datenpunkte ist definiert wie für Datensätze über Beispiele, die y-Komponente wird hier einfach ignoriert.

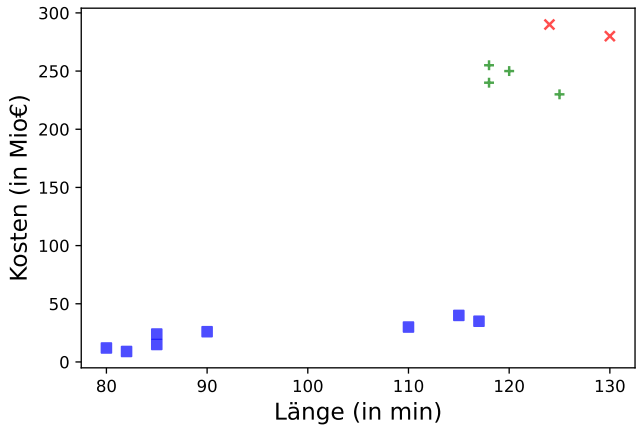


Abbildung 53: Clustering zu E_{movies} in Beispiel 55

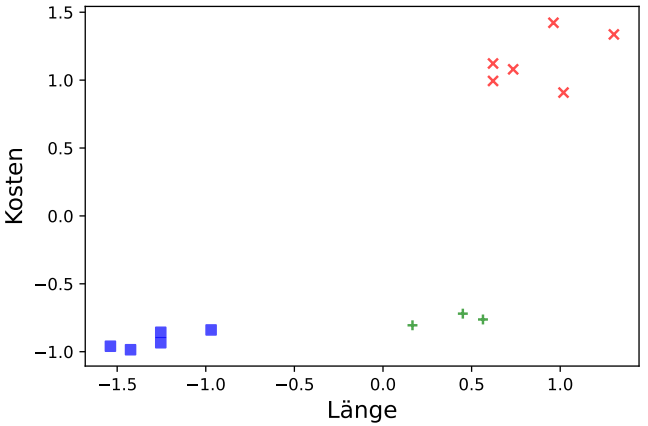


Abbildung 54: Clustering zu $\hat{E}_{\text{movies}}$ in Beispiel 55

fen, welches dieser Clusterings wir als optimal betrachten (und damit welche Anzahl von Ringgrößen wir stan-

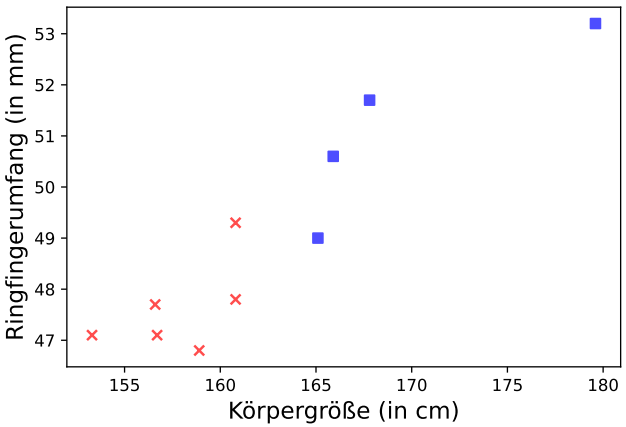


Abbildung 55: Clustering zu E_{ring} für $k = 2$ in Beispiel 56.

dardmäßig anbieten möchten). Diese Frage schauen wir uns in Abschnitt 3.1.2 genauer an.

3.1.2 Evaluation

Genau wie beim überwachten Lernen stellt sich beim unüberwachten Lernen die Frage, wie man die Qualität des gelernten Modells einschätzen und mit anderen gelernten Modellen vergleichen kann. Dazu gibt es beim unüberwachten Lernen, und speziell bei Clusteringproblemen, prinzipiell zwei Möglichkeiten: die externe und die interne Evaluation. Bei der externen Evaluation evaluiert man das Clustering auf einem zusätzlichen Testdatensatz, der allerdings vollständige Beispiele (also inklusive Klassenzugehörigkeit der Datenpunkte) enthalten muss. Für Probleme des unüberwachten Lernens ist das Erstellen eines solchen Datensatzes üblicherweise recht aufwändig, da dieser meist manuell annotiert wer-

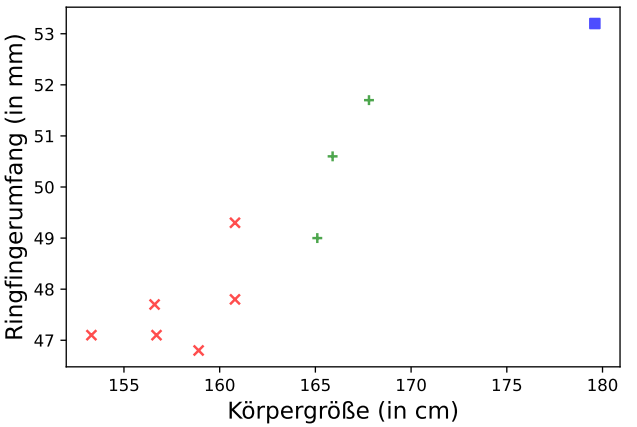


Abbildung 56: Clustering zu E_{ring} für $k = 3$ in Beispiel 56.

den muss. Steht allerdings ein solcher Testdatensatz zur Verfügung, können dieselben Metriken zur Evaluation eingesetzt werden, wie beim überwachten Lernen, also beispielsweise das Genauigkeitsmaß, Präzision, Sensitivität und das F1-Maß.

Bei der internen Evaluation eines Clusteringalgorithmus wird das eigentliche Optimierungsziel des Clusterings als Kriterium herangezogen. Beim K-Means-Algorithmus versuchen wir, Zentroiden $m_1, \ldots, m_k$ zu finden, sodass die (quadrierten) Distanzen aller Datenpunkte in E minimal zu den ihnen zugewiesenen Zentroiden sind. Formal wird dies durch das *Trägheitsmaß* (engl. *inertia*) modelliert.

Definition 23. Sei E ein Datensatz, $m_1, \ldots, m_k$ Zentroiden von k Clustern und *cluster* eine Funktion $\mathit{cluster} : E \to \{1, \ldots, k\}$, die jedem Datenpunkt $x \in E$ einen Zentroiden $m_{\mathit{cluster}(x)}$ zuweist. Die *Trägheit inertia* von *cluster* bezüglich

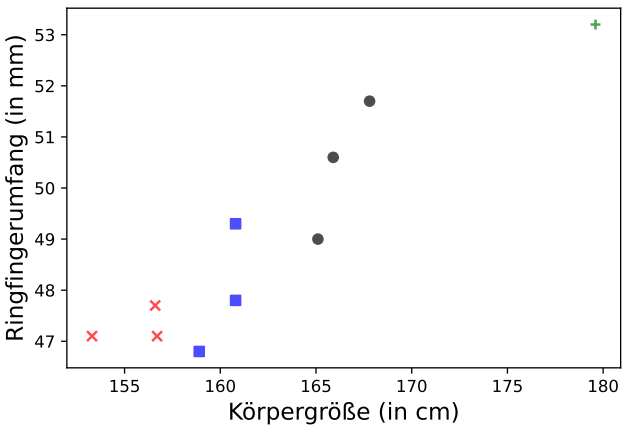


Abbildung 57: Clustering zu E_{ring} für $k = 4$ in Beispiel 56.

E und $m_1, \ldots, m_k$ ist definiert als

$$inertia(cluster, E, m_1, \ldots, m_k) = \sum_{x \in E} \|x - m_{cluster(x)}\|^2$$

Das direkte Lösen des zum Trägheitsmaß zugehörigen Optimierungsproblems, d. h., das Finden von *cluster* und $m_1, \ldots, m_k$, so dass $inertia(cluster, E, m_1, \ldots, m_k)$ minimal ist, ist rechnerisch sehr schwer (genauer: es ist ein NP-schweres Problem). Aus diesem Grund versucht der K-Means-Algorithmus auch nur eine Annäherung an das Optimum zu erreichen.

Beispiel 57. Wir führen Beispiel 54 fort. Für das finale Clustering von E_1 mit

$$\begin{aligned} m_1 &= (2.32, 2.42)^T \\ m_2 &= (3.88, 3.78)^T \\ cluster(x) &= 1 \qquad \text{für } x \in \{x^{(1)}, x^{(3)}, x^{(4)}, x^{(8)}, x^{(9)}\} \\ cluster(x) &= 2 \qquad \text{für } x \in \{x^{(2)}, x^{(5)}, x^{(6)}, x^{(7)}, x^{(10)}\} \end{aligned}$$

erhalten wir

$$
\begin{aligned}
& \mathit{inertia}(\mathit{cluster}, E_1, m_1, m_2) \\
= & \|x^{(1)} - m_1\|^2 + \|x^{(2)} - m_2\|^2 + \|x^{(3)} - m_1\|^2 + \|x^{(4)} - m_1\|^2 + \\
& \|x^{(5)} - m_2\|^2 + \|x^{(6)} - m_2\|^2 + \|x^{(7)} - m_2\|^2 + \|x^{(8)} - m_1\|^2 + \\
& \|x^{(9)} - m_1\|^2 + \|x^{(10)} - m_2\|^2 \\
\approx & 0.662^2 + 0.437^2 + 0.577^2 + 0.594^2 + 0.62^2 + 0.472^2 + 0.33^2 + \\
& 0.611^2 + 0.894^2 + 0.17^2 \\
\approx & 3.233
\end{aligned}
$$

Schauen wir uns zum Vergleich ein alternatives Clustering an, das definiert ist via

$$
\begin{aligned}
\mathit{cluster}(x) &= 1 \qquad \text{für } x \in \{x^{(1)}, x^{(3)}, x^{(4)}, x^{(8)}\} \\
\mathit{cluster}(x) &= 2 \qquad \text{für } x \in \{x^{(2)}, x^{(5)}, x^{(6)}, x^{(7)}, x^{(9)}, x^{(10)}\}
\end{aligned}
$$

und den zugehörigen Zentroiden

$$
\begin{aligned}
m_1 &= (2.175, 2.25)^T \\
m_2 &= (3.717, 3.667)^T
\end{aligned}
$$

Das obige Clustering ist in Abbildung 58 dargestellt. Der einzige Unterschied zum vorherigen Clustering ist, dass der Punkt $x^{(9)}$ dem zweiten Cluster zugewiesen wird. Für dieses Clustering erhalten wir

$$
\begin{aligned}
& \mathit{inertia}(\mathit{cluster}, E_1, m_1, m_2) \\
= & \|x^{(1)} - m_1\|^2 + \|x^{(2)} - m_2\|^2 + \|x^{(3)} - m_1\|^2 + \|x^{(4)} - m_1\|^2 + \\
& \|x^{(5)} - m_2\|^2 + \|x^{(6)} - m_2\|^2 + \|x^{(7)} - m_2\|^2 + \|x^{(8)} - m_1\|^2 + \\
& \|x^{(9)} - m_2\|^2 + \|x^{(10)} - m_2\|^2 \\
\approx & 0.77^2 + 0.603^2 + 0.643^2 + 0.372^2 + 0.604^2 + 0.274^2 + 0.441^2 + \\
& 0.391^2 + 1.055^2 + 0.367^2 \\
\approx & 3.543
\end{aligned}
$$

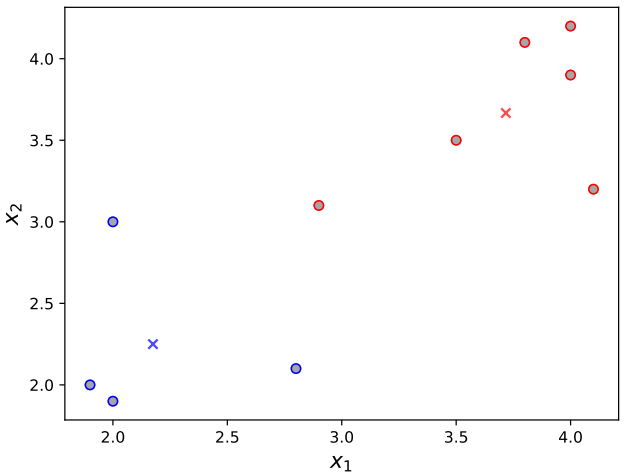


Abbildung 58: Alternatives Clustering zu E_1 in Beispiel 57.

Es ist zu beobachten, dass der Wert der Trägheit beim zweiten Clustering größer ist als beim ersten Clustering.

Der tatsächliche Wert der Trägheit ist nicht leicht zu interpretieren, da er natürlich stark von der Skalierung der Merkmale abhängt. Das heißt, vom Trägheitswert eines Clusterings alleine lässt sich keine Aussage über die Güte des Clusterings ableiten. Erst im Vergleich mit den Trägheitswerten anderer Clusterings können Güteunterschiede erkannt werden (wie im letzten Beispiel). Allerdings sind solche Vergleiche nur von Bedeutung, wenn beide Clusterings über die gleiche Zahl von Clustern verfügen. Denn je mehr Cluster ein Clustering besitzt, desto geringer ist üblicherweise die Trägheit. Der Extremfall tritt hier ein, wenn ein Clustering $k = m$ Cluster besitzt (d. h., ein Cluster pro Datenpunkt). In diesem

Fall ist der Trägheitswert gleich 0, das Clustering bringt aber keine richtigen Einblicke in die Struktur der Daten.

Falls die korrekte Anzahl an Clustern k für einen Datensatz E nicht bekannt ist (wie in Beispiel 53), so kann der Trägheitswert auch genutzt werden, um die geeignete Zahl der Cluster zu finden. Eine einfache Methode dazu ist die sogenannte *Ellenbogenmethode* (engl. *elbow method*). Dazu berechnet man die Trägheitswerte für eine Reihe verschiedener Clusterzahlen k und stellt diese Werte in einem Diagramm dar, siehe Abbildung 59. Die optimale

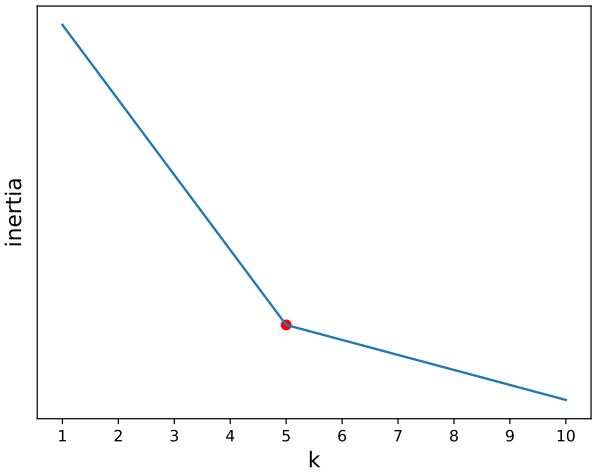


Abbildung 59: Typischer Verlauf der Trägheitswerte bei unterschiedlichen Clusterzahlen k. Die optimale Clusterzahl ist am Scheitelpunkt zu finden (in rot dargestellt).

Clusterzahl findet man dann bei dem k, an dessen Stelle die Kurve „abknickt“. Die Interpretation dazu ist, dass hier höhere Clusterzahlen weniger Information enthalten als die Clusterzahl bis zu diesem Punkt. Natürlich ist dies nur eine sehr grobe Heuristik und der Wert ist auch

nicht immer so eindeutig wie in der idealen Situation in Abbildung 59 dargestellt. An dieser Stelle wollen wir uns aber mit dieser Heuristik begnügen.

Beispiel 58. Wir führen Beispiel 56 fort. Die Trägheitswerte der drei Clusterings zu $k = 2,3,4$ berechnen sich zu $inertia_2$, $inertia_3$, $inertia_4$ mit

$$inertia_2 \approx 192.776$$
$$inertia_3 \approx 53.702$$
$$inertia_4 \approx 20.833$$

Diese drei Werte und Trägheitswerte zu weiteren mit K-Means berechneten Clusterings zu $k = 1$ und $k = 5$ sind graphisch in Abbildung 60 dargestellt. Die optimale Anzahl an Clustern scheint hier bei $k = 3$ zu liegen .

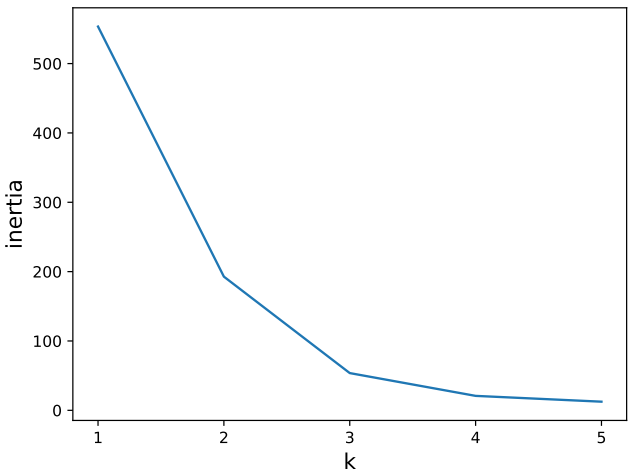


Abbildung 60: Verlauf der Trägheitswerte für $k = 1,2,3,4,5$ zu E_{ring} aus Beispiel 58.

3.1.3 K-Means-Initialisierung

Wie bereits im vorherigen Abschnitt diskutiert wurde, ist Algorithmus 2 ein heuristischer Algorithmus, der lokale Optima berechnet und dabei sogar beliebig weit vom globalen Optimum entfernt sein kann.

Beispiel 59. Betrachten Sie den Datensatz E_2, der in Abbildung 61 dargestellt ist und ignorieren wir für den Moment die Skalierungsproblematik aus Beispiel 55. Die optimale Aufteilung der vier Datenpunkte in zwei Cluster würde die beiden linken Datenpunkte zu einem Cluster zusammenführen und die beiden rechten Datenpunkte in einen zweiten Cluster (bei diesem Clustering erhalten wir einen Trägheitswert von 1). Angenommen wir würden als initiale Zentroiden die Punkte $m_1 = (6,1)^T$ und $m_2 = (6,2)^T$ wählen (also jeweils genau in der Mitte der linken und rechten Datenpunkte). Der Algorithmus *KMEANS* würde die beiden unteren Punkte dem Zentroiden m_1 und die beiden obigen Punkte dem Zentroiden m_2 zuweisen. Der Algorithmus terminiert damit auch schon, da sich die Koordinaten der Zentroiden nicht mehr verändern. Dasselbe Ergebnis würde man auch erzielen, falls $m_1 = (6,y)^T$ und $m_2 = (6,z)^T$ für beliebige $y \leq 1$ und $z \geq 2$ gesetzt werde. In beiden Fällen erhalten wir einen Trägheitswert von 64. Beachten Sie, dass wir diesen Wert beliebig verschlechtern können, indem wir beispielsweise die beiden rechten Punkte beliebig weiter nach rechts verschieben.

Ein anderes problematisches Verhalten tritt beispielsweise auf für $m_1 = (1,1)^T$ und $m_2 = (0,0)^T$. Hier sind alle Datenpunkte näher an m_1 als an m_2 und somit enthält der erste Cluster alle Datenpunkte und der zweite Cluster ist leer.

Während man das zweite in Beispiel 59 genannte Pro-

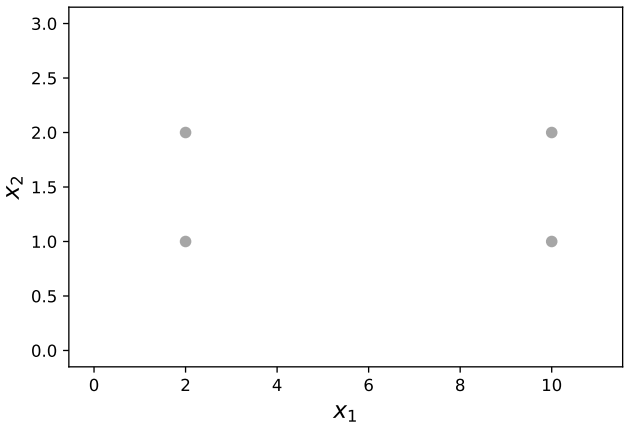


Abbildung 61: Datensatz E_2 zu Beispiel 59.

blem lösen kann, indem man stets als initiale Zentroiden Datenpunkte aus dem Datensatz auswählt (wie wir es bereits zuvor getan haben), so wird das erste Problem davon nicht notwendigerweise gelöst: die initialen Zentroiden $m_1 = (2,1)^T$ und $m_2 = (2,2)^T$ (d. h., die beiden linken Punkte) liefern dasselbe nicht-wünschenswerte Ergebnis.

Da der K-Means-Algorithmus im allgemeinen sehr schnell läuft, ist die naheliegende Lösung, den Algorithmus mehrmals mit zufällig verteilten Zentroiden laufen zu lassen und das Clustering auszuwählen, das am häufigsten berechnet wird (bzw. das Clustering mit minimaler *Trägheit*, siehe Abschnitt 3.1.2). Dies ist auch der in der Praxis gängige Ansatz. Eine weitere Möglichkeit besteht in einer geschickteren Wahl der initialen Zentroiden, wie es beispielsweise der K-Means++-Ansatz verfolgt. Für eine gegebene Anzahl k von Clustern und einen Datensatz E, besteht die Grundidee des K-Means++-Ansatzes darin, wie zuvor k Datenpunkte aus E als initiale Zen-

troiden zu wählen. Diese werden jedoch nicht zufällig gleichverteilt ausgewählt, sondern gewichtet nach der Distanz zu den bisher ausgewählten Zentroiden. Die so ausgewählten Zentroiden werden dann in Zeile 1 von Algorithmus 2 als initiale Definition der Zentroiden benutzt und der eigentliche K-Means-Algorithmus wird anschließend ausgeführt. Algorithmus 3 formalisiert diese Idee. Dabei wird der erste Zentroid m_1 gleichverteilt zufällig

Algorithmus 3 Der K-Means++-Algorithmus.

Eingabe: Datensatz E, Clusterzahl k
Ausgabe: $m_1, \ldots, m_k$ Zentroiden des finalen Clusterings

KMEANS++(E,k)
1: Wähle zufällig gleichverteilt $m_1 \in E$
2: **for** $i = 2, \ldots, k$ **do**
3: $E' := E \setminus \{m_1, \ldots, m_{i-1}\}$
4: Für alle $x \in E'$, $d(x) = \min\{\|x - m_1\|, \ldots, \|x - m_{i-1}\|\}$
5: Wähle $m_i \in E'$ zufällig mit Wahrscheinlichkeit $P(x) = \frac{d(x)}{\sum_{x' \in E'} d(x')}$ für alle $x \in E'$
6: **while** nicht konvergiert **do**
7: **for** $x \in E$ **do**
8: $cluster(x) = \arg\min_{i=1,\ldots,k} \|x - m_i\|$
9: **for** $i = 1, \ldots, k$ **do**
10: $m_i = \frac{1}{|\{x|cluster(x)=i\}|} \sum_{x, cluster(x)=i} x$
11: **return** $m_1, \ldots, m_k$

gewählt (Zeile 1). Für alle weiteren Zentroiden m_i, berechnen wir die Distanz zum nächsten schon gewählten Zentroiden (Zeile 4). Ein beliebiger, noch nicht gewählter Datenpunkt wird dann als nächster Zentroid mit Wahrscheinlichkeit proportional zu dieser Distanz gewählt (Zeile 5). Der Rest des Algorithmus ist dann identisch

zum naiven K-Means-Algorithmus (siehe Algorithmus 2).

Beispiel 60. Wir fahren mit Beispiel 59 fort. Angenommen im ersten Schritt des K-Means++-Algorithms wählen wir $m_1 = (2,2)^T$ (den linken oberen Punkt). Es ergibt sich für die übrigen Punkte:

$$\begin{aligned} d(2,1) &= 1 \\ d(10,1) &\approx 8.06 \\ d(10,2) &= 8 \end{aligned}$$

und damit

$$\begin{aligned} P(2,1) &\approx \frac{1}{17.06} \approx 0.059 \\ P(10,1) &\approx \frac{8}{17.06} \approx 0.469 \\ P(10,2) &\approx \frac{8.06}{17.06} \approx 0.472 \end{aligned}$$

Mit großer Wahrscheinlichkeit wird also $(10,1)^T$ oder $(10,2)^T$ als nächster initialer Zentroid ausgewählt. Wie man leicht sehen kann, tritt für beide Wahlen von initialen Zentroiden das Problem aus Beispiel 59 im weiteren Verlauf des Algorithmus nicht auf.

Im Gegensatz zur gleichverteilten Auswahl der initialen Zentroiden, erhalten wir beim K-Means++-Algorithmus eine Garantie über die *durchschnittliche Güte* des Algorithmus: der Trägheitswert des errechneten Clusterings ist maximal um einen Faktor $O(\log k)$ größer als der Trägheitswert des optimalen Clusterings.

3.2 Hierarchical Clustering

Wir beschäftigen uns in diesem Unterkapitel mit einer weiteren Familie zur Clusteranalyse, dem hierarchischen Clustering.

3.2.1 Clusteranzahl und hierarchisches Clustering

Eine besondere Herausforderung bei der Clusteranalyse ist die Bestimmung der „besten" Anzahl von Clustern für einen gegebenen Datensatz E. In Unterkapitel 3.1 haben wir uns bereits etwas mit diesem Thema beschäftigt und uns insbesondere eine heuristische Methode (die sog. *Ellenbogenmethode*) angeschaut, mit deren Hilfe man beim K-Means-Algorithmus eine gute Wahl für die Anzahl der Cluster treffen kann. Insbesondere bei dem Beispiel zur Marktsegmentierung sollte allerdings klar geworden sein, dass es keine allgemeingültige Definition der „korrekten" Anzahl an Clustern geben kann. Je nach Anwendung gibt es auch für denselben Datensatz verschiedene Clusterzahlen, die Sinn ergeben.

Beispiel 61. Wir betrachten den Datensatz $E_{\text{animals}} = \{x_1, \ldots, x_6\}$ aus Tabelle 31, siehe auch Abbildung 62, der eine Reihe von Tieren mit Merkmalen *Anzahl Beine* und *Giftigkeitsgrad* (auf einer imaginären Skala 0 =„nicht giftig", 1 =„giftig", 2 =„sehr giftig") enthält. Je nach Anwendungsfall sind hier mehrere Clusterings plausibel:

- Das Clustering $\{\{x_1\}, \{x_2\}, \{x_3\}, \{x_4\}, \{x_5\}, \{x_6\}\}$ bildet jede einzelne Tierart in einen eigenen Cluster ab.
- Das Clustering $\{\{x_1, x_2\}, \{x_3\}, \{x_4, x_5\}, \{x_6\}\}$ verbindet Skorpione und Spinnen zum Cluster „Spinnenartige" sowie Hausschweine und Schnabeltiere zu „Vierbeinige Säugetiere".

Nr.	Tier	Anzahl Beine	Giftigkeit
1	Nordafrikanischer Dickschwanzskorpion	8	2
2	Weißknie-Vogelspinne	8	1
3	Tropische Riesenameise	6	1
4	Hausschwein	4	0
5	Schnabeltier	4	1
6	Mensch	2	0

Tabelle 31: Datensatz E_{animal} zu Beispiel 61.

- Das Clustering $\{\{x_1, x_2, x_3\}, \{x_4, x_5, x_6\}\}$ teilt die Daten in „Gliederfüßer“ und „Säugetiere“ auf.

Neben dem finalen Clustering ist die im vorigen Beispiel illustrierte *Clusterhierarchie* für die Clusteranalyse ein wichtiges Analysewerkzeug. Methoden des *hierarchischen Clusterings* erzeugen solche Hierarchien und geben damit Einblick in die Zusammenhänge verschieden granularer Clusterings und helfen bei der Findung der optimalen Clusterzahl für die gegebene Anwendung. Formal ist die Ausgabe einer solchen hierarchischen Clusteringmethode gegeben durch ein *Dendrogramm*.

Definition 24. Sei E ein Datensatz. Ein *Dendrogramm* T über E ist ein gerichteter Baum $T = (V, K, r)$ mit Knoten V, Kanten K und Wurzel $r \in V$, sodass gilt:

- Für alle $v \in V$, $v \subseteq E$
- $r = E$

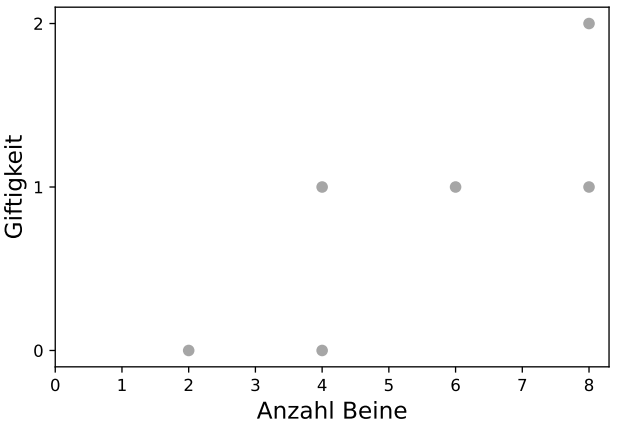


Abbildung 62: Datensatz E_{animal} zu Beispiel 61.

- Für alle $x \in E$ ist $\{x\}$ ein Blatt von T
- Ist $v \in V$ ein innerer Knoten und $v_1, \ldots, v_k$ sind seine Kinder (über Kanten in K), so gilt $v = v_1 \cup \ldots \cup v_k$.

Ein Dendrogramm bildet die Clusterhierarchie von E ab, jeder innere Knoten v mit Kindern $v_1, \ldots, v_k$ repräsentiert einen Cluster, der aus der Vereinigung seiner Untercluster $v_1, \ldots, v_k$ besteht. Die Blätter des Dendrogramms T enthalten das detaillierteste Clustering (jeder Datenpunkt hat seinen eigenen Cluster) und die Wurzel enthält das gröbste Clustering (alle Datenpunkte sind im gleichen Cluster). Üblicherweise annotiert man ein Dendrogramm mit den Schwellwerten, die nötig sind, um Cluster zu vereinigen (wir nennen dann das Dendrogramm *quantitativ*). Dies sind algorithmenabhängige Werte und werden genauer in den nächsten Abschnitten diskutiert. Jeder „Schnitt" durch T an einem bestimmten Schwellwert ergibt dann ein konkretes Clustering.

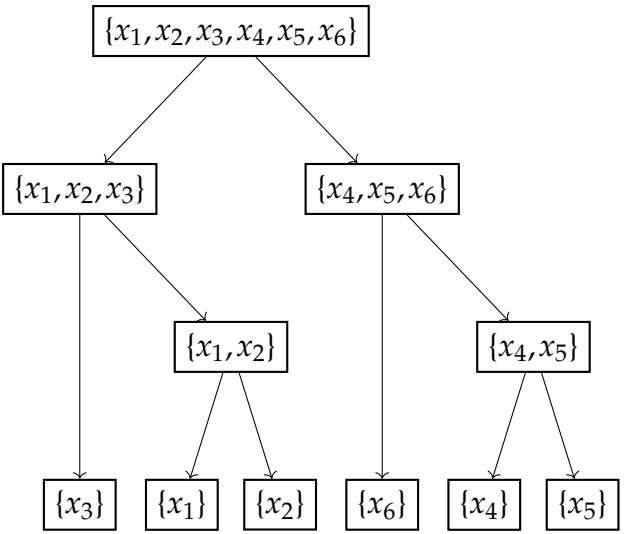


Abbildung 63: (Qualitatives) Dendrogramm T_{animals} aus Beispiel 62.

Beispiel 62. Wir führen Beispiel 61 fort. Ein plausibles Dendrogramm zu E_{animals} ist dargestellt in Abbildung 63. Das gröbste Clustering in der Wurzel besteht aus allen Datenpunkten, das nächst-feinere aus den beiden Clustern $\{x_1, x_2, x_3\}$ und $\{x_4, x_5, x_6\}$. Danach folgt das Clustering $\{\{x_3\}, \{x_1, x_2\}, \{x_6\}, \{x_4, x_5\}\}$ und schließlich mit $\{\{x_1\}, \{x_2\}, \{x_3\}, \{x_4\}, \{x_5\}, \{x_6\}\}$ das detaillierteste Clustering. Abbildung 64 zeigt ein *quantitatives* Dendrogramm. Die Knoten des Dendrogramms sind hier implizit als Schnittpunkte der Verbindungslinien dargestellt und die Elemente aus E sind auf der x-Achse verzeichnet. Die y-Achse zeigt die Schwellwerte der Distanzen von Knoten, an denen Cluster vereinigt (bzw. getrennt) werden (näheres dazu in den nächsten beiden Abschnitten). Schneidet man beispielsweise das Dendrogramm beim Distanzwert 2, so erhält man das Clustering $\{\{x_3\}, \{x_1, x_2\}, \{x_6\}, \{x_4, x_5\}\}$.

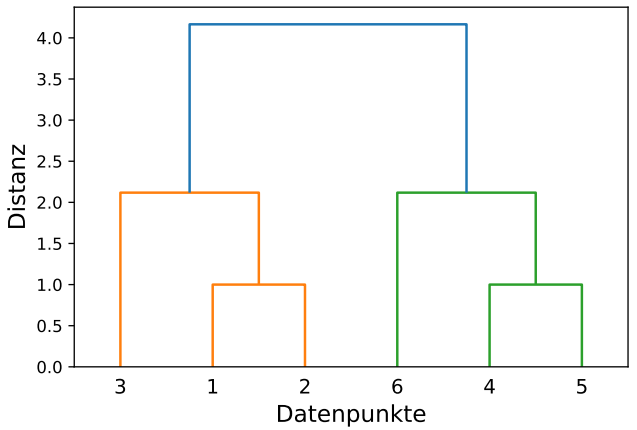


Abbildung 64: (Quantitatives) Dendrogramm D_{animals} aus Beispiel 62.

Quantitative Dendrogramme wie aus Abbildung 64 erlauben es (je nach Interpretation der Distanzfunktion) quantitativ abzuschätzen, wie plausibel Clusterings verschiedener Hierarchiestufen sind. Aus Abbildung 64 ist beispielsweise ersichtlich, dass es wenigstens zwei Cluster $\{x_1, x_2, x_3\}$ und $\{x_4, x_5, x_6\}$ geben sollte, da der Schwellwert zum Vereinigen dieser beiden Cluster (bei ca. 4.1) relativ groß ist im Vergleich zum Schwellwert zur Vereinigung der nächst-feineren Cluster (bei ca. 2.1).

Methoden des hierarchischen Clusterings lassen sich grundsätzlich in zwei Gruppen einteilen: *agglomerative* und *divisive* Verfahren. Agglomerative Verfahren (bzw. *bottom-up* Verfahren) beginnen die Konstruktion eines Dendrogramms von den Blättern an. Zunächst werden alle Datenpunkte individuell einem Cluster zugewiesen. Anschließend werden iterativ diejenigen Cluster miteinander vereinigt, die am „ähnlichsten“ zueinander sind.

Divisive Verfahren (bzw. *top-down* Verfahren) beginnen die Konstruktion eines Dendrogramms an der Wurzel. Zunächst werden alle Datenpunkte einem einzigen Cluster zugewiesen. Anschließend wird iterativ der aktuelle Knoten in zwei (oder mehr) Cluster aufgeteilt. Konkrete Clusteringverfahren dieser beiden Gruppen unterscheiden sich hauptsächlich durch die benutzte Ähnlichkeits- bzw. Distanzfunktion. Prototypische Algorithmen beider Gruppen schauen wir uns in Abschnitt 3.2.2 mit dem *Single-Link*-Clusteringverfahren und in Abschnitt 3.2.3 mit dem *DIANA*-Verfahren an.

3.2.2 Single-Link-Clustering

Single-Link-Clustering ist ein agglomeratives Clusteringverfahren, das initial alle Datenpunkte des Datensatzes E in einen separaten Cluster setzt und anschließend iterativ die beiden am *nächsten* gelegenen Cluster zu einem Cluster verbindet. Auf diese Weise entsteht *bottom-up* ein Dendrogramm. Sei D eine Funktion, die für zwei Mengen von Datenpunkten $E_1, E_2 \subseteq E$ die *Distanz* $D(E_1, E_2)$ von E_1 zu E_2 berechnet. Der allgemeine agglomerative Clusteringalgorithmus ist in Algorithmus 4 dargestellt. Die Menge X in Algorithmus 4 speichert zu jedem Zeitpunkt die Menge von Clustern, die noch vereinigt werden müssen. Am Ende des Algorithmus enthält X dann nur noch die Wurzel des konstruierten Baumes (siehe Zeile 10). Falls es mehrere Paare von Mengen mit minimalem Abstand in Zeile 7 gibt, wählen wir zufällig ein Paar aus.

Je nach Definition der Distanzfunktion D erhält man ein anderes konkretes Clusteringverfahren. Für das *Single-Link*-Clustering nutzen wir die Distanzfunktion D_{single}, die definiert ist durch

$$D_{\text{single}}(E_1, E_2) = \min_{x_1 \in E_1, x_2 \in E_2} \|x_1 - x_2\|$$

Algorithmus 4 Der allgemeine agglomerative Clusteringalgorithmus

Eingabe: Datensatz E, Distanzfunktion D
Ausgabe: Dendrogramm T

Agglo(E,D)
1: T := leeres Dendrogramm
2: $X := \emptyset$
3: **for** $x \in E$ **do**
4: Füge Knoten $\{x\}$ zu T hinzu
5: $X := X \cup \{\{x\}\}$
6: **while** $|X| \neq 1$ **do**
7: Finde $X_1, X_2 \in X$ mit $X_1 \neq X_2$ und $D(X_1, X_2)$ minimal
8: Erzeuge Knoten $X_3 = X_1 \cup X_2$ und setze X_1 und X_2 als Kinder von X_3
9: $X := (X \setminus \{X_1, X_2\}) \cup \{X_3\}$
10: Setze die Wurzel von T auf X', wobei $X = \{X'\}$
11: **return** T

Der Abstand zwischen zwei Mengen E_1 und E_2 ist also der minimale (euklidische) Abstand von beliebigen Datenpunkten aus E_1 und E_2. Wie schon zuvor, muss an dieser Stelle darauf geachtet werden, dass die Merkmale entsprechend skaliert sind, da ansonsten Unterschiede in den Ausprägungen mancher Merkmale stärker gewichtet werden als andere, siehe dazu Abschnitt 2.4.2. Algorithmus 4 geht davon aus, dass der Datensatz E entsprechend vorverarbeitet wurde.

Beispiel 63. Wir führen Beispiel 62 fort und führen Algorithmus 4 schrittweise durch:

1. Wir erzeugen zunächst Cluster für alle Datenpunk-

te aus E_{animals} und setzen

$$X := \{\{x_1\}, \{x_2\}, \{x_3\}, \{x_4\}, \{x_5\}, \{x_6\}\}$$

2. Wir berechnen $D_{\text{single}}(X_1, X_2)$ für alle $X_1, X_2 \in X$ mit $X_1 \neq X_2$ (wegen $D_{\text{single}}(X_1, X_2) = D_{\text{single}}(X_2, X_1)$ müssen wir nicht alle Kombinationen berechnen):

$$\begin{aligned}
D_{\text{single}}(\{x_1\}, \{x_2\}) &= \|x_1 - x_2\| = \sqrt{0^2 + 1^2} = 1 \\
D_{\text{single}}(\{x_1\}, \{x_3\}) &= \|x_1 - x_3\| = \sqrt{2^2 + 1^2} \approx 2.236 \\
D_{\text{single}}(\{x_1\}, \{x_4\}) &= \|x_1 - x_4\| = \sqrt{4^2 + 2^2} \approx 4.472 \\
D_{\text{single}}(\{x_1\}, \{x_5\}) &= \|x_1 - x_5\| = \sqrt{4^2 + 1^2} \approx 4.123 \\
D_{\text{single}}(\{x_1\}, \{x_6\}) &= \|x_1 - x_6\| = \sqrt{6^2 + 2^2} \approx 6.325 \\
D_{\text{single}}(\{x_2\}, \{x_3\}) &= \|x_2 - x_3\| = \sqrt{2^2 + 0^2} = 2 \\
D_{\text{single}}(\{x_2\}, \{x_4\}) &= \|x_2 - x_4\| = \sqrt{4^2 + 1^2} \approx 4.123 \\
D_{\text{single}}(\{x_2\}, \{x_5\}) &= \|x_2 - x_5\| = \sqrt{4^2 + 0^2} = 4 \\
D_{\text{single}}(\{x_2\}, \{x_6\}) &= \|x_2 - x_6\| = \sqrt{6^2 + 1^2} \approx 6.083 \\
D_{\text{single}}(\{x_3\}, \{x_4\}) &= \|x_3 - x_4\| = \sqrt{2^2 + 1^2} \approx 2.236 \\
D_{\text{single}}(\{x_3\}, \{x_5\}) &= \|x_3 - x_5\| = \sqrt{2^2 + 0^2} = 2 \\
D_{\text{single}}(\{x_3\}, \{x_6\}) &= \|x_3 - x_6\| = \sqrt{4^2 + 1^2} \approx 4.123 \\
D_{\text{single}}(\{x_4\}, \{x_5\}) &= \|x_4 - x_5\| = \sqrt{0^2 + 1^2} = 1 \\
D_{\text{single}}(\{x_4\}, \{x_6\}) &= \|x_4 - x_6\| = \sqrt{2^2 + 0^2} = 2 \\
D_{\text{single}}(\{x_5\}, \{x_6\}) &= \|x_5 - x_6\| = \sqrt{2^2 + 1^2} \approx 2.236
\end{aligned}$$

Es gilt, dass $D_{\text{single}}(\{x_1\}, \{x_2\}) = D_{\text{single}}(\{x_4\}, \{x_5\}) = 1$ minimal ist. Wir wählen also zufällig $\{x_1\}, \{x_2\}$ aus und aktualisieren X zu

$$X := \{\{x_1, x_2\}, \{x_3\}, \{x_4\}, \{x_5\}, \{x_6\}\}$$

3. Wir berechnen $D_{\text{single}}(X_1, X_2)$ für alle $X_1, X_2 \in X$ mit $X_1 \neq X_2$:

$$
\begin{aligned}
D_{\text{single}}(\{x_1, x_2\}, \{x_3\}) &= \min\{\|x_1 - x_3\|, \|x_2 - x_3\|\} = 2 \\
D_{\text{single}}(\{x_1, x_2\}, \{x_4\}) &= \min\{\|x_1 - x_4\|, \|x_2 - x_4\|\} \approx 4.123 \\
D_{\text{single}}(\{x_1, x_2\}, \{x_5\}) &= \min\{\|x_1 - x_5\|, \|x_2 - x_5\|\} = 4 \\
D_{\text{single}}(\{x_1, x_2\}, \{x_6\}) &= \min\{\|x_1 - x_6\|, \|x_2 - x_6\|\} \approx 6.083 \\
D_{\text{single}}(\{x_3\}, \{x_4\}) &= \|x_3 - x_4\| \approx 2.236 \\
D_{\text{single}}(\{x_3\}, \{x_5\}) &= \|x_3 - x_5\| = 2 \\
D_{\text{single}}(\{x_3\}, \{x_6\}) &= \|x_3 - x_6\| \approx 4.123 \\
D_{\text{single}}(\{x_4\}, \{x_5\}) &= \|x_4 - x_5\| = 1 \\
D_{\text{single}}(\{x_4\}, \{x_6\}) &= \|x_4 - x_6\| = 2 \\
D_{\text{single}}(\{x_5\}, \{x_6\}) &= \|x_5 - x_6\| \approx 2.236
\end{aligned}
$$

Es gilt, dass $D_{\text{single}}(\{x_4\}, \{x_5\}) = 1$ minimal ist. Wir aktualisieren X zu

$$X := \{\{x_1, x_2\}, \{x_3\}, \{x_4, x_5\}, \{x_6\}\}$$

4. Wir berechnen $D_{\text{single}}(X_1, X_2)$ für alle $X_1, X_2 \in X$ mit $X_1 \neq X_2$:

$$
\begin{aligned}
D_{\text{single}}(\{x_1, x_2\}, \{x_3\}) &= \min\{\|x_1 - x_3\|, \|x_2 - x_3\|\} = 2 \\
D_{\text{single}}(\{x_1, x_2\}, \{x_4, x_5\}) &= \min\{\|x_1 - x_4\|, \|x_2 - x_4\|, \\
&\qquad \|x_1 - x_5\|, \|x_2 - x_5\|\} \\
&= 4 \\
D_{\text{single}}(\{x_1, x_2\}, \{x_6\}) &= \min\{\|x_1 - x_6\|, \|x_2 - x_6\|\} \\
&\approx 6.083 \\
D_{\text{single}}(\{x_3\}, \{x_4, x_5\}) &= \min\{\|x_3 - x_4\|, \|x_3 - x_5\|\} = 2 \\
D_{\text{single}}(\{x_3\}, \{x_6\}) &= \|x_3 - x_6\| \approx 4.123 \\
D_{\text{single}}(\{x_4, x_5\}, \{x_6\}) &= \min\{\|x_4 - x_6\|, \|x_5 - x_6\|\} = 2
\end{aligned}
$$

Hier haben wir drei Paare mit Distanz 2, wir wählen zufällig $\{x_1, x_2\}$ und $\{x_3\}$ und aktualisieren X zu

$$X := \{\{x_1, x_2, x_3\}, \{x_4, x_5\}, \{x_6\}\}$$

5. Wir berechnen $D_{\text{single}}(X_1, X_2)$ für alle $X_1, X_2 \in X$ mit $X_1 \neq X_2$:

$$\begin{aligned} & D_{\text{single}}(\{x_1, x_2, x_3\}, \{x_4, x_5\}) \\ = \ & \min\{\|x_1 - x_4\|, \|x_2 - x_4\|, \|x_3 - x_4\|, \|x_1 - x_5\|, \\ & \quad \|x_2 - x_5\|, \|x_3 - x_5\|\} \\ = \ & 2 \end{aligned}$$

$$\begin{aligned} & D_{\text{single}}(\{x_1, x_2, x_3\}, \{x_6\}) \\ = \ & \min\{\|x_1 - x_6\|, \|x_2 - x_6\|, \|x_3 - x_6\|\} \\ \approx \ & 4.123 \end{aligned}$$

$$\begin{aligned} & D_{\text{single}}(\{x_4, x_5\}, \{x_6\}) \\ = \ & \min\{\|x_4 - x_6\|, \|x_5 - x_6\|\} = 2 \end{aligned}$$

Es gilt, dass

$$D_{\text{single}}(\{x_1, x_2, x_3\}, \{x_4, x_5\}) = D_{\text{single}}(\{x_4, x_5\}, \{x_6\}) = 2$$

minimal ist. Wir wählen also zufällig $\{x_4, x_5\}, \{x_6\}$ aus und aktualisieren X zu

$$X := \{\{x_1, x_2, x_3\}, \{x_4, x_5, x_6\}\}$$

6. Auch wenn diese Berechnung nicht mehr notwendig ist, da wir nur noch zwei Elemente in X haben, berechnen wir

$$\begin{aligned} & D_{\text{single}}(\{x_1, x_2, x_3\}, \{x_4, x_5, x_6\}) \\ = \ & \min\{\|x_1 - x_4\|, \|x_2 - x_4\|, \|x_3 - x_4\|, \|x_1 - x_5\|, \\ & \quad \|x_2 - x_5\|, \|x_3 - x_5\|, \|x_1 - x_6\|, \|x_2 - x_6\|, \|x_3 - x_6\|\} \\ = \ & 2 \end{aligned}$$

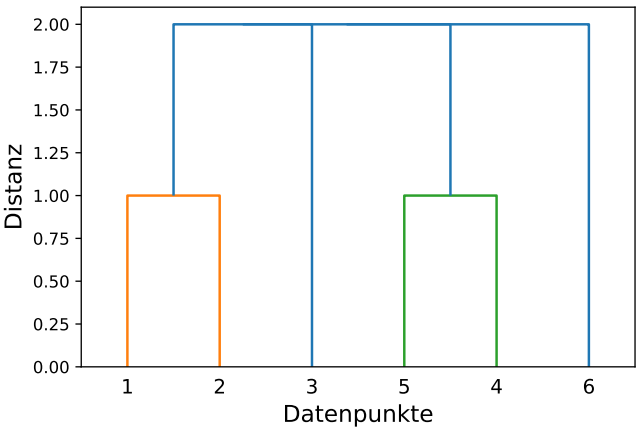


Abbildung 65: (Quantitatives) Dendrogramm D_{animals} aus Beispiel 63.

und wir aktualisieren X zu

$$X := \{\{x_1, x_2, x_3, x_4, x_5, x_6\}\}$$

Damit terminiert der Algorithmus und wir erhalten das qualitative Dendrogramm aus Abbildung 63. Als quantitatives Dendrogramm erhalten wir allerdings nicht das in Abbildung 64 dargestellte, sondern das in Abbildung 65. Die Distanzwerte zeigen nun an, bei welchem Wert die Cluster zusammengeführt wurden. Da die letzten 3 Clusterzusammenführungen alle beim Distanzwert 2 erfolgt sind, sind diese in Abbildung 65 auch nicht mehr unterscheidbar.

Wie man im vorigen Beispiel sieht, liefert das Single-Link-Clustering relativ „langgezogene“ Cluster und eignet sich daher gut für entsprechend strukturierte Daten.

Eine andere Distanzfunktion ist beispielsweise D_{avg} definiert durch

$$D_{\text{avg}}(E_1, E_2) = \frac{1}{|E_1||E_2|} \sum_{x_1 \in E_1, x_2 \in E_2} \|x_1 - x_2\|$$

Die Funktion D_{avg} misst den durchschnittlichen Abstand von Datenpunkten in E_1 zu Datenpunkten in E_2. Das Dendrogramm in Abbildung 64 erhält man, wenn man Algorithmus 4 auf E_{animals} mit D_{avg} ausführt.

3.2.3 Divisive Analysis Clustering

Wir schauen uns nun mit DIANA (engl. *Divise Analysis Clustering*) ein prototypisches divisives Verfahren zum Clustering an. Hier beginnt man die Konstruktion eines Dendrogramms an der Wurzel mit einem Cluster, der alle Datenpunkte eines gegebenen Datensatzes E enthält, und teilt iterativ solche Blattknoten auf, die noch mehr als einen Datenpunkt enthalten. Algorithmus 5 zeigt den DIANA-Algorithmus, der ebenso wie die agglomerative Version eine Distanzfunktion D zur Distanzberechnung von Mengen von Datenpunkten benötigt (wie etwa D_{single} oder D_{avg}). Die tatsächliche Entscheidung, wie ein Knoten X mit $|X| > 1$ in zwei Cluster aufgeteilt wird, erfolgt wie folgt (Zeilen 4–14). Zunächst wird der Datenpunkt $\hat{x}$ aus X gesucht, der am weitesten entfernt von allen anderen Datenpunkten in X ist (Zeile 4). Dann wird initial X aufgeteilt in einen Cluster X_1, der nur $\hat{x}$ enthält, und einen Cluster X_2, der alle übrigen Datenpunkte aus X enthält (Zeilen 5 und 6). Anschließend (Zeilen 7–13) wird der Datenpunkt $x^\dagger \in X_2$ gesucht, der von allen Punkten aus X_2 am ehesten noch zu X_1 gehören kann. Ist $x^\dagger$ darüberhinaus auch näher an X_1 als an X_2 (Zeile 8), so verschieben wir $x^\dagger$ von X_2 zu X_1 (Zeilen 10 und 11). Finden wir keinen Datenpunkt $x \in X_2$, der näher an X_1 als

Algorithmus 5 Der DIANA-Algorithmus

Eingabe: Datensatz E, Distanzfunktion D
Ausgabe: Dendrogramm T

DIANA(E,D)
1: T :=leeres Dendrogramm
2: Setze Wurzel von T auf E
3: **while** Es existiert Blattknoten X von T mit $|X| > 1$ **do**
4: $\quad \hat{x} := \arg\max_{x \in X} D(\{x\}, X \setminus \{x\})$
5: $\quad X_1 := \{\hat{x}\}$
6: $\quad X_2 := X \setminus \{\hat{x}\}$
7: $\quad$ **while** True **do**
8: $\quad\quad x^+ = \arg\max_{x \in X_2} \{D(\{x\}, X_2 \setminus \{x\}) - D(\{x\}, X_1)\}$
9: $\quad\quad$ **if** $\left(D(\{x^+\}, X_2 \setminus \{x^+\}) - D(\{x^+\}, X_1)\right) > 0$ **then**
10: $\quad\quad\quad X_1 := X \cup \{x^+\}$
11: $\quad\quad\quad X_2 := X \setminus \{x^+\}$
12: $\quad\quad$ **else**
13: $\quad\quad\quad$ break
14: $\quad$ Setze X_1 und X_2 als neue Kinder von X in T
15: **return** T

am Rest von X_2 liegt, so beenden wir die Konstruktion (Zeile 13) und setzen X_1 und X_2 als neue Kinder von X im Baum ein.

Beispiel 64. Wir führen Beispiel 62 fort und führen Algorithmus 5 schrittweise mit D_{avg} durch (beachten Sie, dass wir nur die erste Iteration des Algorithmus zur Verdeutlichung ausführen). Zur Vereinfachung der Darstellung berechnen wir alle paarweisen Distanzen (bzgl. der euklidischen Metrik) von Datenpunkten aus E_{animals} vor (siehe Beispiel 63 für die Rechnungen), die Ergebnisse sind in Tabelle 32 zu sehen.

1. Wir erzeugen zunächst die Wurzel $X_1 = \{x_1, x_2, x_3, x_4, x_5, x_6\}$ des Dendrogramms T.

$\|x - x'\|$	x_1	x_2	x_3	x_4	x_5	x_6
x_1	0	1	2.236	4.472	4.123	6.325
x_2		0	2	4.123	4	6.083
x_3			0	2.236	2	4.123
x_4				0	1	2
x_5					0	2.236
x_6						0

Tabelle 32: Paarweise Distanzen von Datenpunkten aus E_{animals}; beachten Sie, dass wir wegen $\|x - x'\| = \|x' - x\|$ nur die erste Variante angeben.

2. Da X_1 Blattknoten von T mit $|X_1| > 1$ ist, berechnen wir zunächst $\hat{x}$. Dazu bestimmen wir

$$\begin{aligned} & D_{\text{avg}}(\{x_1\}, \{x_2, x_3, x_4, x_5, x_6\}) \\ = & \frac{1 + 2.236 + 4.472 + 4.123 + 6.325}{5} \approx 3.631 \end{aligned}$$

$$\begin{aligned} & D_{\text{avg}}(\{x_2\}, \{x_1, x_3, x_4, x_5, x_6\}) \\ = & \frac{1 + 2 + 4.123 + 4 + 6.083}{5} \approx 3.441 \end{aligned}$$

$$\begin{aligned} & D_{\text{avg}}(\{x_3\}, \{x_1, x_2, x_4, x_5, x_6\}) \\ = & \frac{2.236 + 2 + 2.236 + 2 + 4.123}{5} \approx 2.519 \end{aligned}$$

$$\begin{aligned} & D_{\text{avg}}(\{x_4\}, \{x_1, x_2, x_3, x_5, x_6\}) \\ = & \frac{4.472 + 4.123 + 2.236 + 1 + 2}{5} \approx 2.766 \end{aligned}$$

$$
\begin{aligned}
&D_{\text{avg}}(\{x_5\}, \{x_1, x_2, x_3, x_4, x_6\}) \\
&= \frac{4.123 + 4 + 2 + 1 + 2.236}{5} \approx 2.672
\end{aligned}
$$

$$
\begin{aligned}
&D_{\text{avg}}(\{x_6\}, \{x_1, x_2, x_3, x_4, x_5\}) \\
&= \frac{6.325 + 6.083 + 4.123 + 2 + 2.236}{5} \approx 4.153
\end{aligned}
$$

Damit ist $\hat{x} = x_6$ und wir setzen $X_1 := \{x_6\}$ und $X_2 := \{x_1, x_2, x_3, x_4, x_5\}$.

3. Wir berechnen $x^\dagger$:

$$
\begin{aligned}
&D(\{x_1\}, \{x_2, x_3, x_4, x_5\}) - D(\{x_1\}, \{x_6\}) \\
&= \frac{1 + 2.236 + 4.472 + 4.123}{4} - 6.325 \\
&= 2.958 - 6.325 = -3.367
\end{aligned}
$$

$$
\begin{aligned}
&D(\{x_2\}, \{x_1, x_3, x_4, x_5\}) - D(\{x_2\}, \{x_6\}) \\
&= \frac{1 + 2 + 4.123 + 4}{4} - 6.083 \\
&= 2.781 - 6.083 = -3.302
\end{aligned}
$$

$$
\begin{aligned}
&D(\{x_3\}, \{x_1, x_2, x_4, x_5\}) - D(\{x_3\}, \{x_6\}) \\
&= \frac{2.236 + 2 + 2.236 + 2}{4} - 4.123 \\
&= 2.118 - 4.123 = -2.005
\end{aligned}
$$

$$
\begin{aligned}
&D(\{x_4\}, \{x_1, x_2, x_3, x_5\}) - D(\{x_4\}, \{x_6\}) \\
&= \frac{4.472 + 4.123 + 2.236 + 1}{4} - 2 \\
&= 2.958 - 2 = 0.958
\end{aligned}
$$

$$
\begin{aligned}
&D(\{x_5\},\{x_1,x_2,x_3,x_4\}) - D(\{x_5\},\{x_6\}) \\
&= \frac{4.123+4+2+1}{4} - 2.236 \\
&= 2.781 - 2.236 = 0.545
\end{aligned}
$$

Damit ist $x^\dagger = x_4$ und da

$$D(\{x_4\},\{x_1,x_2,x_3,x_5\}) - D(\{x_4\},\{x_6\}) > 0,$$

setzen wir $X_1 := \{x_4, x_6\}$ und $X_2 := \{x_1, x_2, x_3, x_5\}$ und fahren wir mit der Konstruktion fort.

4. Wir berechnen $x^\dagger$:

$$
\begin{aligned}
&D(\{x_1\},\{x_2,x_3,x_5\}) - D(\{x_1\},\{x_4,x_6\}) \\
&= \frac{1+2.236+4.123}{3} - \frac{4.472+6.325}{2} \\
&\approx 2.453 - 5.399 \approx -2.946 \\
&D(\{x_2\},\{x_1,x_3,x_5\}) - D(\{x_2\},\{x_4,x_6\}) \\
&= \frac{1+2+4}{3} - \frac{4.123+6.083}{2} \\
&\approx 2.333 - 5.103 \approx -2.77 \\
&D(\{x_3\},\{x_1,x_2,x_5\}) - D(\{x_3\},\{x_4,x_6\}) \\
&= \frac{2.236+2+2}{3} - \frac{2.236+4.123}{2} \\
&\approx 2.079 - 3.18 \approx -1.101 \\
&D(\{x_5\},\{x_1,x_2,x_3\}) - D(\{x_5\},\{x_4,x_6\}) \\
&\approx \frac{4.123+4+2}{3} - \frac{1+2.236}{2} \\
&\approx 3.374 - 1.618 \approx 1.756
\end{aligned}
$$

Damit ist $x^\dagger = x_5$ und da

$$D(\{x_5\},\{x_1,x_2,x_3\}) - D(\{x_5\},\{x_4,x_6\}) > 0,$$

setzen wir $X_1 := \{x_4,x_5,x_6\}$ und $X_2 := \{x_1,x_2,x_3\}$ und fahren wir mit der Konstruktion fort.

5. Wir berechnen $x^\dagger$:

$$\begin{aligned}
&D(\{x_1\},\{x_2,x_3\}) - D(\{x_1\},\{x_4,x_5,x_6\}) \\
&= \frac{1+2.236}{2} - \frac{4.472+4.123+6.325}{3} \\
&\approx 1.618 - 4.973 \approx -3.356
\end{aligned}$$

$$\begin{aligned}
&D(\{x_2\},\{x_1,x_3\}) - D(\{x_2\},\{x_4,x_5,x_6\}) \\
&= \frac{1+2}{2} - \frac{4.123+4+6.083}{3} \\
&\approx 1.5 - 4.735 \approx -3.235
\end{aligned}$$

$$\begin{aligned}
&D(\{x_3\},\{x_1,x_2\}) - D(\{x_3\},\{x_4,x_5,x_6\}) \\
&= \frac{2.236+2}{2} - \frac{2.236+2+4.123}{2} \\
&\approx 2.118 - 2.786 \approx -0.668
\end{aligned}$$

Damit ist $x^\dagger = x_3$, aber da

$$D(\{x_3\},\{x_1,x_2\}) - D(\{x_3\},\{x_4,x_5,x_6\}) \leq 0,$$

wird $x^\dagger$ nicht mehr X_1 hinzufügt und es bleibt bei $X_1 = \{x_4,x_5,x_6\}$ und $X_2 = \{x_1,x_2,x_3\}$.

Die Knoten $X_1 = \{x_4,x_5,x_6\}$ und $X_2 = \{x_1,x_2,x_3\}$ werden nun als Kinder von $X = \{x_1,x_2,x_3,x_4,\ x_5,x_6\}$ hinzugefügt und der Algorithmus würde iterativ gesondert mit X_1 und X_2 fortfahren. Aufgrund der langen Berechnungen führen wir dieses Verfahren allerdings nicht fort. Das

finale qualitative Dendrogramm ist identisch mit dem Dendrogramm in Abbildung 63 und das finale quantitative Dendrogramm ist identisch mit dem Dendrogramm in Abbildung 64.

Wie das vorherige Beispiel zeigt, ist die Ausführung des DIANA-Algorithms weitaus aufwändiger als die Berechnung mit einem agglomerativen Clusteringverfahren. In der Praxis werden deshalb nur sehr selten divisive Verfahren angewendet.

3.3 Assoziationsregeln

Wie bei anderen Problemen und Verfahren des unüberwachten Lernens, geht es beim Assoziationsregellernen darum, Strukturen in Daten zu erkennen. Beim Assoziationsregellernen geht es insbesondere darum, *häufig zusammen auftretende Mengen von Elementen* und *Assoziationen zwischen Elementen* zu finden.

3.3.1 Assoziationsregeln

Die Eingabedaten für das Assoziationsregellernen sind etwas anders strukturiert als bei den bisher betrachteten Problemen des unüberwachten und überwachten maschinellen Lernens, und ein Datensatz besteht (im einfachsten Fall) aus einer Menge von *Transaktionen*, die jeweils aus einer Menge von beliebigen Elementen bestehen. Das Standardbeispiel (siehe auch Beispiel 65 unten) dazu ist eine Menge von Einkäufen in einem Supermarkt: jeder Einkauf besteht aus der Menge der eingekauften Artikel. Ein Supermarkt kann daran interessiert sein, aus solchen erhobenen Daten zu ermitteln, welche Artikel oft zusammen eingekauft werden und ob das Kaufen eines gewissen Artikels den Kauf eines anderen Artikels impliziert. Solche Informationen sind aus Marketingperspektive von hohem Wert und können beispielsweise auch dazu genutzt werden, gewisse Artikel in die Nähe gewisser anderer Artikel zu positionieren, um Einkaufswege zu optimieren oder auch den Verkauf zu erhöhen.

Sei I eine Menge beliebiger Elemente. Ein Datensatz F ist dann eine Multimenge $F = \{t_1, \ldots, t_m\}$ (die *Transaktionen*) mit $t_i \subseteq I$, $i = 1, \ldots, m$.[23] Eine *(Assoziations-)Regel* hat

[23] Beachten Sie, dass F eine Multimenge ist, die gleichen Transaktionen können also mehrfach auftauchen.

Nr.	Milch	Käse	Butter	Brot	Kaffee	Zucker	Mehl
1	1	1	1	1	0	1	1
2	0	1	1	1	1	1	1
3	1	0	1	0	1	1	1
4	1	0	1	0	0	0	0
5	1	0	1	0	1	0	0
6	1	0	0	0	0	0	1
7	1	1	1	1	1	1	1
8	0	0	0	0	0	0	0
9	1	1	1	0	0	1	0
10	1	1	0	1	1	1	1

Tabelle 33: Die Datenbasis $F_{\text{supermarket}}$ aus Beispiel 65.

die Form $X \Rightarrow Y$ mit $X, Y \subseteq I$ und wird „Wenn X dann auch Y“ gelesen. Hier heißt X auch *Prämisse* und Y *Konklusion*.

Beispiel 65. Tabelle 33 zeigt den Datensatz $F_{\text{supermarket}}$, der die Einkäufe in einem Supermarkt (an einem bestimmten Tag oder einer bestimmten Stunde) darstellt. Wir haben hier

$$\begin{aligned} &I_{\text{supermarket}} \\ &= \{\text{Milch}, \text{Käse}, \text{Butter}, \text{Brot}, \text{Kaffee}, \text{Zucker}, \text{Mehl}\} \end{aligned}$$

Die Darstellung der Transaktionen in Tabelle 33 ist so zu lesen, dass eine 1 bei einem Element angibt, dass das Element in der entsprechenden Transaktion enthalten ist, und eine 0 bedeutet, dass das Element nicht vorhanden ist. Wir haben also

$$F_{\text{supermarket}} = \{t_1, \ldots, t_{10}\}$$

und beispielsweise

$$t_3 = \{\text{Milch}, \text{Butter}, \text{Kaffee}, \text{Zucker}, \text{Mehl}\}$$

Eine in $F_{\text{supermarket}}$ allgemeingültige Regel ist beispielsweise {Brot} $\Rightarrow$ {Käse}, also „Wenn Brot gekauft wurde, dann wurde auch Käse eingekauft“: wie man leicht in Tabelle 33 nachvollziehen kann, gilt, dass bei allen Transaktionen, bei denen Brot enthalten ist, auch Käse enthalten ist.

Das Ziel des Assoziationsregellernens ist es, Regeln wie {Brot} $\Rightarrow$ {Käse} aus dem obigen Beispiel automatisch aus dem Datensatz zu extrahieren. Üblicherweise sind wir allerdings nicht nur an solchen allgemeingültigen Regeln interessiert (also Regeln, die bei jeder einzelnen Transaktion erfüllt sind). Solche Regeln sind in der Praxis auch nicht zu erwarten. Vielmehr sind wir an Regeln interessiert, die *häufig* auftreten. Um diesen Aspekt zu formalisieren, gibt es eine Reihe von Evaluationsmetriken für Regeln. Die wichtigsten sind der *Support* und die *Konfidenz* (engl. *confidence*).

Definition 25. Seien $X, Y \subseteq I$. Der *Support* $support_F(X)$ von X in F ist definiert durch

$$support_F(X) = \frac{|\{t \in F \mid X \subseteq t\}|}{|F|}$$

Der *Support* $support_F(X \Rightarrow Y)$ einer Regel $X \Rightarrow Y$ in F ist definiert durch

$$support_F(X \Rightarrow Y) = support_F(X \cup Y)$$

Die Konfidenz $conf_F(X \Rightarrow Y)$ einer Regel $X \Rightarrow Y$ in F ist definiert durch

$$conf_F(X \Rightarrow Y) = \frac{support_F(X \cup Y)}{support_F(X)}$$

Der Support $support_F(X)$ einer Menge X ist also der Anteil der Transaktionen, bei denen X eine Teilmenge ist,

zu der Zahl aller Transaktionen. Weiterhin ist der Support $support_F(X \Rightarrow Y)$ einer Regel $X \Rightarrow Y$ der Anteil der Transaktionen, bei denen die Regel angewendet werden kann, zu der Zahl aller Transaktionen. Schließlich ist die Konfidenz $conf_F(X \Rightarrow Y)$ der Anteil der Transaktionen, bei denen die Regel angewendet werden kann, zu der Zahl der Transaktionen, bei denen die Prämisse der Regel vorhanden ist. Im Allgemeinen gibt der Support einen Hinweis darauf, wie *oft* eine Regel angewendet werden kann, während die Konfidenz angibt, wie *gut* die Regel den Zusammenhang zwischen Prämisse und Konklusion abbildet.

Beispiel 66. Wir fahren mit Beispiel 65 fort. Hier gilt beispielsweise

$$\begin{aligned} support_{F_{\text{supermarket}}}(\{\text{Zucker}, \text{Mehl}\}) &= \frac{5}{10} \\ support_{F_{\text{supermarket}}}(\{\text{Zucker}, \text{Mehl}, \text{Butter}\}) &= \frac{4}{10} \\ support_{F_{\text{supermarket}}}(\{\text{Brot}\}) &= \frac{4}{10} \\ support_{F_{\text{supermarket}}}(\{\text{Brot}, \text{Käse}\}) &= \frac{4}{10} \end{aligned}$$

und

$$\begin{aligned} & support_{F_{\text{supermarket}}}(\{\text{Zucker}, \text{Mehl}\} \Rightarrow \{\text{Butter}\}) \\ = & \frac{4}{10} \end{aligned}$$

$$\begin{aligned} & conf_{F_{\text{supermarket}}}(\{\text{Zucker}, \text{Mehl}\} \Rightarrow \{\text{Butter}\}) \\ = & \frac{support_{F_{\text{supermarket}}}(\{\text{Zucker}, \text{Mehl}, \text{Butter}\})}{support_{F_{\text{supermarket}}}(\{\text{Zucker}, \text{Mehl}\})} \\ = & \frac{4}{5} \end{aligned}$$

$$
\begin{aligned}
& support_{F_{\text{supermarket}}}(\{\text{Brot}\} \Rightarrow \{\text{Käse}\}) \\
= & \frac{4}{10}
\end{aligned}
$$

$$
\begin{aligned}
& conf_{F_{\text{supermarket}}}(\{\text{Brot}\} \Rightarrow \{\text{Käse}\}) \\
= & \frac{support_{F_{\text{supermarket}}}(\{\text{Brot}, \text{Käse}\})}{support_{F_{\text{supermarket}}}(\{\text{Brot}\})} \\
= & 1
\end{aligned}
$$

Beachten Sie, dass die Werte von $support_F$ und $conf_F$ stets zwischen 0 und 1 liegen. Beim Assoziationsregellernen ist es das Ziel, Regeln mit möglichst hohem Support und hoher Konfidenz zu lernen. Da die Kombinationsmöglichkeiten von Elementen aus I zur Bildung einer Regel sehr hoch sind, muss man bei der Suche nach diesen Regeln etwas geschickter vorgehen. Im Folgenden schauen wir uns mit dem Apriori-Algorithmus und dem FP-Growth-Algorithmus zwei Algorithmen zum Assoziationsregellernen an.

3.3.2 Der Apriori-Algorithmus

Der Apriori-Algorithmus ist der klassische Algorithmus zum Lernen von Assoziationsregeln aus Daten. Neben dem eigentlichen Datensatz bekommt der Algorithmus noch zwei Parameter. Der Parameter $minsupp \in [0,1]$ (*minimal support*) gibt an, welchen Wert $support_F(X \Rightarrow Y)$ eine gelernte Regel $X \Rightarrow Y$ mindestens haben muss, während der Parameter $minconf \in [0,1]$ (*minimal confidence*) angibt, welchen minimalen Wert $conf_F(X \Rightarrow Y)$ haben muss. Die Ausgabe des Algorithmus besteht dann aus allen Regeln, die diese beiden Bedingungen erfüllen. Durch Variation von *minsupp* und *minconf* kann die Anzahl gelernter Regeln gesteuert werden.

Der Apriori-Algorithmus verfährt in zwei Schritten. In einem ersten Schritt werden zunächst alle Mengen $Z \subseteq I$ bestimmt, die $support_F(Z) \geq minsupp$ erfüllen. Notwendigerweise muss dann für alle Regeln $X \Rightarrow Y$ mit $support_F(X \Rightarrow Y) \geq minsupp$ gelten, dass $X \cup Y = Z$ für solch ein Z gilt. Im zweiten Schritt werden dann solche Regeln $X \Rightarrow Y$ aus den Z bestimmt, für die $conf_F(X \Rightarrow Y) \geq minconf$.

Schauen wir uns zunächst den ersten Schritt des Apriori-Algorithmus an, die Bestimmung der *häufigen Mengen* (engl. *frequent item sets*), d. h., derjenigen Mengen X, für die $support_F(X) \geq minsupp$ gilt. Um eine stumpfe Auflistung aller Teilmengen und Überprüfung des Supports zu vermeiden, benutzt der Apriori-Algorithmus die folgende Einsicht.

Proposition 1. *Sei $X \subseteq I$.*

- *Wenn $support_F(X) \geq minsupp$, dann gilt für jedes $X' \subseteq X$ auch $support_F(X') \geq minsupp$.*
- *Wenn $support_F(X) < minsupp$, dann gilt für jedes $X' \supseteq X$ auch $support_F(X') < minsupp$.*

Mit anderen Worten, Teilmengen häufiger Mengen sind häufig und Obermengen nicht-häufiger Mengen sind nicht-häufig.

Der Apriori-Algorithmus berechnet die häufigen Mengen iterativ mit wachsender Kardinalität. Zunächst werden alle ein-elementigen Mengen betrachtet und deren Support berechnet. Nur diejenigen Mengen werden weiter berücksichtigt, deren Support größer oder gleich *minsupp* ist. Aus diesen Mengen werden dann Kandidaten für häufige zwei-elementige Mengen gebildet, indem je zwei verschiedene ein-elementige Mengen vereinigt werden (jede häufige zwei-elementige Menge muss wegen

Proposition 1 von dieser Form sein). Anschließend werden alle zwei-elementigen Mengen aussortiert, deren Support nicht größer oder gleich *minsupp* ist. Aus den häufigen zwei-elementigen Mengen werden dann Kandidaten für häufige drei-elementige Mengen gebildet, indem Paare von zwei-elementigen Mengen vereinigt werden, die genau ein Element gemeinsam haben. Entsteht dabei allerdings eine drei-elementige Menge, bei der nicht *alle* zwei-elementigen Teilmengen häufig sind, kann diese auch direkt aussortiert werden. Dann werden wieder alle drei-elementigen Mengen aussortiert, deren Support nicht größer oder gleich *minsupp* ist. Der Algorithmus fährt entsprechend mit der Berechnung häufiger vier-elementiger Mengen fort bis keine weiteren häufigen Mengen gefunden werden.

Algorithmus 6 formalisiert den obigen Ansatz zur Lösung des ersten Schrittes des Apriori-Algorithmus, der Berechnung häufiger Mengen FreqItems. Zeilen 1–4 berechnen alle häufigen ein-elementigen Mengen. Zeilen 6–11 berechnen iterativ alle häufigen i-elementigen Mengen. Dazu werden in Zeile 8 alle Paare $(i-1)$-elementiger Mengen betrachten und in Zeilen 9 und 10 solche Kandidaten aussortiert, die nicht häufig sind.

Nach der Berechnung aller häufigen Mengen werden im zweiten Schritt des Apriori-Algorithmus alle Regeln $X \Rightarrow Y$ berechnet, so dass $X \cup Y$ häufig ist und $conf_F(X \Rightarrow Y) \geq minconf$ gilt. Dazu macht sich der Apriori-Algorithmus die folgende Eigenschaft zunutze.

Proposition 2. *Seien $X, Y, Y' \subseteq I$ mit $Y' \subseteq Y \subseteq X$. Dann gilt*

$$conf_F((X \setminus Y') \Rightarrow Y') \geq conf_F((X \setminus Y) \Rightarrow Y)$$

Mit anderen Worten, die Konfidenz einer Regel $X \Rightarrow Y$ kann nicht geringer werden, wenn wir ein Element von Y nach X verschieben, d. h., die Prämisse der Regel weiter

Algorithmus 6 Algorithmus FreqItems zur Berechnung häufiger Mengen.

Eingabe: Datensatz F, $minsupp \in [0,1]$
Ausgabe: Menge K aller Mengen $X \subseteq I$ mit $support_F(X) \geq minsupp$

FreqItems(F,$minsupp$)

1: $K_1 = \emptyset$
2: **for** $x \in I$ **do**
3: **if** $support_F(\{x\}) \geq minsupp$ **then**
4: $K_1 := K_1 \cup \{\{x\}\}$
5: $i = 2$
6: **while** $K_{i-1} \neq \emptyset$ **do**
7: $K_i := \emptyset$
8: **for** $X_1, X_2 \in K_{i-1}$ mit $|X_1 \cup X_2| = i$ **do**
9: **if** es existiert kein $X' \subseteq X_1 \cup X_2$ mit $|X'| = i-1$ und $X' \notin K_{i-1}$ **then**
10: **if** $support_F(X_1 \cup X_2) \geq minsupp$ **then**
11: $K_i := K_i \cup \{X_1 \cup X_2\}$
12: $i := i + 1$
13: **return** $K_1 \cup \ldots \cup K_{i-1}$

spezialisieren. Diese Einsicht können wir nutzen, indem wir zunächst alle Regeln mit ein-elementiger Konklusion betrachten, die wir aus einer häufigen Menge X erzeugen können. Dann werden wie beim Algorithmus FreqItems (siehe Algorithmus 6) größere Kandidaten für Konklusionen generiert und die Regeln entsprechend auf ihre Konfidenz getestet.

Der vollständige Apriori-Algorithmus ist in Algorithmus 7 dargestellt.

Beispiel 67. Wir fahren mit Beispiel 66 fort und betrachten $minsupp = 0.4$ und $minconf = 0.9$. Wir führen den Apriori-Algorithmus schrittweise aus.

Algorithmus 7 Der Apriori-Algorithmus

Eingabe: Datensatz F, $minsupp, minconf \in [0,1]$
Ausgabe: Menge R aller Regeln $X \Rightarrow Y$ mit $support_F(X \Rightarrow Y) \geq minsupp$ und $conf_F(X \Rightarrow Y) \geq minconf$

Apriori(F,$minsupp$, $minconf$)
1: $K = FreqItems(F, minsupp)$
2: $R_0 := \emptyset$
3: **for** $X \in K$ **do**
4: $R_1 := \emptyset$
5: **for** $x \in X$ **do**
6: **if** $conf_F(X \setminus \{x\} \Rightarrow \{x\}) \geq minconf$ **then**
7: $R_1 := R_1 \cup \{X \setminus \{x\} \Rightarrow \{x\}\}$
8: $i = 2$
9: **while** $R_{i-1} \neq \emptyset$ **do**
10: $R_i := \emptyset$
11: **for** $X_1 \Rightarrow Y_1, X_2 \Rightarrow Y_2 \in R_{i-1}$ mit $|Y_1 \cup Y_2| = i$ **do**
12: **if** es existiert kein $Y' \subseteq Y_1 \cup Y_2$ mit $|Y'| = i-1$ und $X \setminus Y' \Rightarrow Y' \notin R_{i-1}$ **then**
13: **if** $conf_F(X_1 \cap X_2 \Rightarrow Y_1 \cup Y_2) \geq minconf$ **then**
14: $R_i := R_i \cup \{X_1 \cap X_2 \Rightarrow Y_1 \cup Y_2\}$
15: $i := i + 1$
16: $R := R \cup R_1 \cup \ldots \cup R_{i-1}$
17: **return** R

1. Wir berechnen zunächst die häufigen ein-elementigen Mengen von $F_{\text{supermarket}}$ und erhalten

$$support_{F_{\text{supermarket}}}(\{\text{Milch}\}) = 0.8$$
$$support_{F_{\text{supermarket}}}(\{\text{Käse}\}) = 0.5$$
$$support_{F_{\text{supermarket}}}(\{\text{Butter}\}) = 0.7$$
$$support_{F_{\text{supermarket}}}(\{\text{Brot}\}) = 0.4$$

$$support_{F_{\text{supermarket}}}(\{\text{Kaffee}\}) = 0.5$$
$$support_{F_{\text{supermarket}}}(\{\text{Zucker}\}) = 0.6$$
$$support_{F_{\text{supermarket}}}(\{\text{Mehl}\}) = 0.6$$

Alle ein-elementigen Mengen sind häufig bzgl. *minsupp* und damit folgt

$$K_1 = \{\{\text{Milch}\}, \{\text{Käse}\}, \{\text{Butter}\}, \{\text{Brot}\}, \{\text{Kaffee}\}, \{\text{Zucker}\}, \{\text{Mehl}\}\}$$

2. Wir berechnen die häufigen zwei-elementigen Mengen. Dazu kombinieren wir alle Paare von ein-elementigen Mengen aus K_1 und erhalten

$$support_{F_{\text{supermarket}}}(\{\text{Milch}, \text{Käse}\}) = 0.4$$
$$support_{F_{\text{supermarket}}}(\{\text{Milch}, \text{Butter}\}) = 0.6$$
$$support_{F_{\text{supermarket}}}(\{\text{Milch}, \text{Brot}\}) = 0.3$$
$$support_{F_{\text{supermarket}}}(\{\text{Milch}, \text{Kaffee}\}) = 0.4$$
$$support_{F_{\text{supermarket}}}(\{\text{Milch}, \text{Zucker}\}) = 0.5$$
$$support_{F_{\text{supermarket}}}(\{\text{Milch}, \text{Mehl}\}) = 0.5$$
$$support_{F_{\text{supermarket}}}(\{\text{Käse}, \text{Butter}\}) = 0.4$$
$$support_{F_{\text{supermarket}}}(\{\text{Käse}, \text{Brot}\}) = 0.4$$
$$support_{F_{\text{supermarket}}}(\{\text{Käse}, \text{Kaffee}\}) = 0.3$$
$$support_{F_{\text{supermarket}}}(\{\text{Käse}, \text{Zucker}\}) = 0.5$$
$$support_{F_{\text{supermarket}}}(\{\text{Käse}, \text{Mehl}\}) = 0.4$$
$$support_{F_{\text{supermarket}}}(\{\text{Butter}, \text{Brot}\}) = 0.3$$
$$support_{F_{\text{supermarket}}}(\{\text{Butter}, \text{Kaffee}\}) = 0.4$$

$$support_{F_{\text{supermarket}}}(\{\text{Butter}, \text{Zucker}\}) = 0.5$$
$$support_{F_{\text{supermarket}}}(\{\text{Butter}, \text{Mehl}\}) = 0.4$$
$$support_{F_{\text{supermarket}}}(\{\text{Brot}, \text{Kaffee}\}) = 0.3$$
$$support_{F_{\text{supermarket}}}(\{\text{Brot}, \text{Zucker}\}) = 0.4$$
$$support_{F_{\text{supermarket}}}(\{\text{Brot}, \text{Mehl}\}) = 0.4$$
$$support_{F_{\text{supermarket}}}(\{\text{Kaffee}, \text{Zucker}\}) = 0.4$$
$$support_{F_{\text{supermarket}}}(\{\text{Kaffee}, \text{Mehl}\}) = 0.4$$
$$support_{F_{\text{supermarket}}}(\{\text{Zucker}, \text{Mehl}\}) = 0.5$$

Beachten Sie, dass für alle zwei-elementigen Mengen oben die Bedingung von Zeile 9 aus Algorithmus 6 stets erfüllt ist. Wir sortieren noch die Mengen mit Support kleiner 0.4 aus und erhalten

$$\begin{aligned} K_2 = \{&\{\text{Milch}, \text{Käse}\}, \{\text{Milch}, \text{Butter}\}, \\ &\{\text{Milch}, \text{Kaffee}\}, \{\text{Milch}, \text{Zucker}\}, \{\text{Milch}, \text{Mehl}\}, \\ &\{\text{Käse}, \text{Butter}\}, \{\text{Käse}, \text{Brot}\}, \{\text{Käse}, \text{Zucker}\}, \\ &\{\text{Käse}, \text{Mehl}\}, \{\text{Butter}, \text{Kaffee}\}, \{\text{Butter}, \text{Zucker}\}, \\ &\{\text{Butter}, \text{Mehl}\}, \{\text{Brot}, \text{Zucker}\}, \{\text{Brot}, \text{Mehl}\}, \\ &\{\text{Kaffee}, \text{Zucker}\}, \{\text{Kaffee}, \text{Mehl}\}, \{\text{Zucker}, \text{Mehl}\}\} \end{aligned}$$

3. Wir berechnen die häufigen drei-elementigen Mengen. Dazu vereinigen wir je zwei zwei-elementige Mengen, die eine drei-elementige Menge erzeugen (d. h., wir vereinigen beispielsweise die Mengen {Milch, Käse} und {Butter, Kaffee} *nicht*). Wir erhalten zunächst die folgenden Kandidaten:

   $$\begin{aligned} &\{\text{Milch}, \text{Käse}, \text{Butter}\}, \{\text{Milch}, \text{Käse}, \text{Brot}\}, \\ &\{\text{Milch}, \text{Käse}, \text{Zucker}\}, \{\text{Milch}, \text{Käse}, \text{Mehl}\}, \end{aligned}$$

{Milch, Butter, Kaffee}, {Milch, Butter, Zucker},
{Milch, Butter, Mehl}, {Milch, Kaffee, Zucker},
{Milch, Kaffee, Mehl}, {Milch, Zucker, Mehl},
{Käse, Butter, Kaffee}, {Käse, Butter, Zucker},
{Käse, Butter, Mehl}, {Käse, Brot, Zucker},
{Käse, Brot, Mehl}, {Käse, Zucker, Mehl},
{Butter, Kaffee, Mehl}, {Butter, Zucker, Mehl},
{Brot, Zucker, Mehl}, {Kaffee, Zucker, Mehl},
{Milch, Käse, Kaffee}, {Käse, Butter, Brot},
{Käse, Zucker, Kaffee}, {Butter, Kaffee, Zucker},
{Butter, Zucker, Brot}, {Butter, Zucker, Kaffee},
{Butter, Mehl, Brot}, {Brot, Zucker, Kaffee},
{Brot, Kaffee, Mehl}

Nicht alle obigen Mengen erfüllen die Bedingung von Zeile 9 aus Algorithmus 6, beispielsweise gilt für die Menge {Milch, Käse, Brot}, dass die Teilmenge {Milch, Brot} *nicht* in K_2 enthalten ist. Für die übrigen Mengen berechnen wir

$$support_{F_{\text{supermarket}}}(\{\text{Milch}, \text{Käse}, \text{Butter}\}) = 0.3$$
$$support_{F_{\text{supermarket}}}(\{\text{Milch}, \text{Käse}, \text{Zucker}\}) = 0.4$$
$$support_{F_{\text{supermarket}}}(\{\text{Milch}, \text{Käse}, \text{Mehl}\}) = 0.3$$
$$support_{F_{\text{supermarket}}}(\{\text{Milch}, \text{Butter}, \text{Kaffee}\}) = 0.3$$
$$support_{F_{\text{supermarket}}}(\{\text{Milch}, \text{Butter}, \text{Zucker}\}) = 0.4$$
$$support_{F_{\text{supermarket}}}(\{\text{Milch}, \text{Butter}, \text{Mehl}\}) = 0.3$$
$$support_{F_{\text{supermarket}}}(\{\text{Milch}, \text{Kaffee}, \text{Zucker}\}) = 0.3$$
$$support_{F_{\text{supermarket}}}(\{\text{Milch}, \text{Kaffee}, \text{Mehl}\}) = 0.3$$
$$support_{F_{\text{supermarket}}}(\{\text{Milch}, \text{Zucker}, \text{Mehl}\}) = 0.4$$

$$support_{F_{\text{supermarket}}}(\{\text{Käse}, \text{Butter}, \text{Zucker}\}) = 0.4$$
$$support_{F_{\text{supermarket}}}(\{\text{Käse}, \text{Butter}, \text{Mehl}\}) = 0.3$$
$$support_{F_{\text{supermarket}}}(\{\text{Käse}, \text{Brot}, \text{Zucker}\}) = 0.4$$
$$support_{F_{\text{supermarket}}}(\{\text{Käse}, \text{Brot}, \text{Mehl}\}) = 0.4$$
$$support_{F_{\text{supermarket}}}(\{\text{Käse}, \text{Zucker}, \text{Mehl}\}) = 0.4$$
$$support_{F_{\text{supermarket}}}(\{\text{Butter}, \text{Kaffee}, \text{Mehl}\}) = 0.3$$
$$support_{F_{\text{supermarket}}}(\{\text{Butter}, \text{Zucker}, \text{Mehl}\}) = 0.4$$
$$support_{F_{\text{supermarket}}}(\{\text{Butter}, \text{Zucker}, \text{Kaffee}\}) = 0.3$$
$$support_{F_{\text{supermarket}}}(\{\text{Brot}, \text{Zucker}, \text{Mehl}\}) = 0.4$$
$$support_{F_{\text{supermarket}}}(\{\text{Kaffee}, \text{Zucker}, \text{Mehl}\}) = 0.4$$

und damit

$$\begin{aligned} K_3 = \{&\{\text{Milch}, \text{Käse}, \text{Zucker}\}, \{\text{Milch}, \text{Butter}, \text{Zucker}\}, \\ &\{\text{Milch}, \text{Zucker}, \text{Mehl}\}, \{\text{Käse}, \text{Butter}, \text{Zucker}\}, \\ &\{\text{Käse}, \text{Brot}, \text{Zucker}\}, \{\text{Käse}, \text{Brot}, \text{Mehl}\}, \\ &\{\text{Käse}, \text{Zucker}, \text{Mehl}\}, \{\text{Butter}, \text{Zucker}, \text{Mehl}\}, \\ &\{\text{Brot}, \text{Zucker}, \text{Mehl}\}, \{\text{Kaffee}, \text{Zucker}, \text{Mehl}\}\} \end{aligned}$$

4. Wir berechnen die häufigen vier-elementigen Mengen. Dazu vereinigen wir je zwei drei-elementige Mengen, die eine vier-elementige Menge erzeugen (d. h., wir vereinigen beispielsweise die Mengen {Milch, Käse, Zucker} und {Kaffee, Zucker, Mehl} *nicht*). Wir erhalten zunächst die folgenden Kandidaten:

   {Milch, Käse, Zucker, Mehl},
   {Milch, Käse, Zucker, Butter},
   {Milch, Butter, Zucker, Mehl},

{Milch, Käse, Zucker, Brot},
{Milch, Zucker, Mehl, Brot},
{Käse, Butter, Zucker, Mehl},
{Milch, Zucker, Mehl, Kaffee},
{Butter, Zucker, Mehl, Kaffee},
{Brot, Kaffee, Zucker, Mehl},
{Käse, Butter, Zucker, Brot},
{Käse, Brot, Mehl, Zucker},
{Käse, Zucker, Mehl, Kaffee},
{Butter, Zucker, Mehl, Brot}

Von den obigen Mengen erfüllt nur

$$\{\text{Käse}, \text{Brot}, \text{Mehl}, \text{Zucker}\}$$

die Bedingung von Zeile 9 aus Algorithmus 6 und wir erhalten

$$support_{F_{\text{supermarket}}}(\{\text{Käse}, \text{Brot}, \text{Mehl}, \text{Zucker}\}) = 0.4$$

und damit

$$K_4 = \{\{\text{Käse}, \text{Brot}, \text{Mehl}, \text{Zucker}\}\}$$

5. Da K_4 nur ein Element hat, ist $K_5 = \emptyset$ (wir können keine zwei Elemente aus K_4 vereinigen) und damit ist die Bestimmung der häufigsten Mengen abgeschlossen. Die Menge K aller häufigsten Mengen ist damit

$$\begin{aligned} K &= K_1 \cup K_2 \cup K_3 \cup K_4 \\ &= \{\{\text{Milch}\}, \{\text{Käse}\}, \{\text{Butter}\}, \{\text{Brot}\}, \{\text{Kaffee}\}, \\ &\quad \{\text{Zucker}\}, \{\text{Mehl}\}, \{\text{Milch}, \text{Käse}\}, \\ &\quad \{\text{Milch}, \text{Butter}\}, \{\text{Milch}, \text{Kaffee}\}, \end{aligned}$$

{Milch, Zucker}, {Milch, Mehl},
{Käse, Butter}, {Käse, Brot},
{Käse, Zucker}, {Käse, Mehl},
{Butter, Kaffee}, {Butter, Zucker},
{Butter, Mehl}, {Brot, Zucker},
{Brot, Mehl}, {Kaffee, Zucker},
{Kaffee, Mehl}, {Zucker, Mehl},
{Milch, Käse, Zucker},
{Milch, Butter, Zucker},
{Milch, Zucker, Mehl},
{Käse, Butter, Zucker},
{Käse, Brot, Zucker},
{Käse, Brot, Mehl},
{Käse, Zucker, Mehl},
{Butter, Zucker, Mehl},
{Brot, Zucker, Mehl},
{Kaffee, Zucker, Mehl},
{Käse, Brot, Mehl, Zucker}}

6. Für den zweiten Schritt des Apriori-Algorithmus betrachten wir exemplarisch nur die Menge $X = \{\text{Käse}, \text{Brot}, \text{Mehl}, \text{Zucker}\}$. Wir berechnen zunächst die Konfidenz aller Regeln mit einer ein-elementigen Konklusion, die man aus X bilden kann:

$$\begin{aligned} & conf_{F_{\text{supermarket}}}(\{\text{Brot}, \text{Mehl}, \text{Zucker}\} \Rightarrow \{\text{Käse}\}) \\ = & \frac{support_{F_{\text{supermarket}}}(\{\text{Brot}, \text{Mehl}, \text{Zucker}, \text{Käse}\})}{support_{F_{\text{supermarket}}}(\{\text{Brot}, \text{Mehl}, \text{Zucker}\})} \\ = & \frac{0.4}{0.4} = 1 \end{aligned}$$

$$
\begin{aligned}
& conf_{F_{\text{supermarket}}}(\{\text{Käse}, \text{Mehl}, \text{Zucker}\} \Rightarrow \{\text{Brot}\}) \\
= & \frac{support_{F_{\text{supermarket}}}(\{\text{Brot}, \text{Mehl}, \text{Zucker}, \text{Käse}\})}{support_{F_{\text{supermarket}}}(\{\text{Käse}, \text{Mehl}, \text{Zucker}\})} \\
= & \frac{0.4}{0.4} = 1
\end{aligned}
$$

$$
\begin{aligned}
& conf_{F_{\text{supermarket}}}(\{\text{Käse}, \text{Brot}, \text{Zucker}\} \Rightarrow \{\text{Mehl}\}) \\
= & \frac{support_{F_{\text{supermarket}}}(\{\text{Brot}, \text{Mehl}, \text{Zucker}, \text{Käse}\})}{support_{F_{\text{supermarket}}}(\{\text{Brot}, \text{Käse}, \text{Zucker}\})} \\
= & \frac{0.4}{0.4} = 1
\end{aligned}
$$

$$
\begin{aligned}
& conf_{F_{\text{supermarket}}}(\{\text{Käse}, \text{Brot}, \text{Mehl}\} \Rightarrow \{\text{Zucker}\}) \\
= & \frac{support_{F_{\text{supermarket}}}(\{\text{Brot}, \text{Mehl}, \text{Zucker}, \text{Käse}\})}{support_{F_{\text{supermarket}}}(\{\text{Brot}, \text{Mehl}, \text{Käse}\})} \\
= & \frac{0.4}{0.4} = 1
\end{aligned}
$$

Für alle vier Regeln ist die Konfidenz größer oder gleich $minconf = 0.9$, also gilt

$$
\begin{aligned}
R_1 = \{ & (\{\text{Brot}, \text{Mehl}, \text{Zucker}\} \Rightarrow \{\text{Käse}\}, \\
& \{\text{Käse}, \text{Mehl}, \text{Zucker}\} \Rightarrow \{\text{Brot}\}, \\
& \{\text{Käse}, \text{Brot}, \text{Zucker}\} \Rightarrow \{\text{Mehl}\}, \\
& \{\text{Käse}, \text{Brot}, \text{Mehl}\} \Rightarrow \{\text{Zucker}\}\}
\end{aligned}
$$

7. Wir bestimmen Regeln aus R_1 mit zwei-elementiger

Konklusion und bestimmen die Konfidenz:

$$
\begin{aligned}
& conf_{F_{\text{supermarket}}}(\{\text{Brot}, \text{Mehl}\} \Rightarrow \{\text{Käse}, \text{Zucker}\}) \\
= & \frac{support_{F_{\text{supermarket}}}(\{\text{Brot}, \text{Mehl}, \text{Zucker}, \text{Käse}\})}{support_{F_{\text{supermarket}}}(\{\text{Brot}, \text{Mehl}\})} \\
= & \frac{0.4}{0.4} = 1
\end{aligned}
$$

$$
\begin{aligned}
& conf_{F_{\text{supermarket}}}(\{\text{Brot}, \text{Käse}\} \Rightarrow \{\text{Mehl}, \text{Zucker}\}) \\
= & \frac{support_{F_{\text{supermarket}}}(\{\text{Brot}, \text{Mehl}, \text{Zucker}, \text{Käse}\})}{support_{F_{\text{supermarket}}}(\{\text{Brot}, \text{Käse}\})} \\
= & \frac{0.4}{0.4} = 1
\end{aligned}
$$

$$
\begin{aligned}
& conf_{F_{\text{supermarket}}}(\{\text{Brot}, \text{Zucker}\} \Rightarrow \{\text{Käse}, \text{Mehl}\}) \\
= & \frac{support_{F_{\text{supermarket}}}(\{\text{Brot}, \text{Mehl}, \text{Zucker}, \text{Käse}\})}{support_{F_{\text{supermarket}}}(\{\text{Brot}, \text{Zucker}\})} \\
= & \frac{0.4}{0.4} = 1
\end{aligned}
$$

$$
\begin{aligned}
& conf_{F_{\text{supermarket}}}(\{\text{Zucker}, \text{Mehl}\} \Rightarrow \{\text{Käse}, \text{Brot}\}) \\
= & \frac{support_{F_{\text{supermarket}}}(\{\text{Brot}, \text{Mehl}, \text{Zucker}, \text{Käse}\})}{support_{F_{\text{supermarket}}}(\{\text{Zucker}, \text{Mehl}\})} \\
= & \frac{0.4}{0.5} = 0.8
\end{aligned}
$$

$$
\begin{aligned}
& conf_{F_{\text{supermarket}}}(\{\text{Zucker}, \text{Käse}\} \Rightarrow \{\text{Mehl}, \text{Brot}\}) \\
= & \frac{support_{F_{\text{supermarket}}}(\{\text{Brot}, \text{Mehl}, \text{Zucker}, \text{Käse}\})}{support_{F_{\text{supermarket}}}(\{\text{Zucker}, \text{Käse}\})} \\
= & \frac{0.4}{0.5} = 0.8
\end{aligned}
$$

$$
\begin{aligned}
& conf_{F_{\text{supermarket}}}(\{\text{Käse}, \text{Mehl}\} \Rightarrow \{\text{Zucker}, \text{Brot}\}) \\
&= \frac{support_{F_{\text{supermarket}}}(\{\text{Brot}, \text{Mehl}, \text{Zucker}, \text{Käse}\})}{support_{F_{\text{supermarket}}}(\{\text{Mehl}, \text{Käse}\})} \\
&= \frac{0.4}{0.4} = 1
\end{aligned}
$$

Es folgt

$$
\begin{aligned}
R_2 = \{&\{\text{Brot}, \text{Mehl}\} \Rightarrow \{\text{Käse}, \text{Zucker}\}, \\
&\{\text{Brot}, \text{Käse}\} \Rightarrow \{\text{Mehl}, \text{Zucker}\}, \\
&\{\text{Brot}, \text{Zucker}\} \Rightarrow \{\text{Käse}, \text{Mehl}\}, \\
&\{\text{Käse}, \text{Mehl}\} \Rightarrow \{\text{Zucker}, \text{Brot}\}\}
\end{aligned}
$$

8. Wir bestimmen Regeln aus R_2 mit drei-elementiger Konklusion. Die Kandidaten dazu sind:

$$
\begin{aligned}
&\{\text{Brot}\} \Rightarrow \{\text{Käse}, \text{Zucker}, \text{Mehl}\} \\
&\{\text{Käse}\} \Rightarrow \{\text{Brot}, \text{Zucker}, \text{Mehl}\} \\
&\{\text{Mehl}\} \Rightarrow \{\text{Käse}, \text{Zucker}, \text{Brot}\}
\end{aligned}
$$

Nur die erste Regel oben erfüllt die Bedingung in Zeile 11 von Algorithmus 7. Wir berechnen ihren Konfidenz:

$$
\begin{aligned}
& conf_{F_{\text{supermarket}}}(\{\text{Brot}\} \Rightarrow \{\text{Käse}, \text{Zucker}, \text{Mehl}) \\
&= \frac{support_{F_{\text{supermarket}}}(\{\text{Brot}, \text{Mehl}, \text{Zucker}, \text{Käse}\})}{support_{F_{\text{supermarket}}}(\{\text{Brot}\})} \\
&= \frac{0.4}{0.4} = 1
\end{aligned}
$$

Es folgt

$$
R_3 = \{\{\text{Brot}\} \Rightarrow \{\text{Käse}, \text{Zucker}, \text{Mehl}\}\}
$$

9. Da R_3 nur ein Element hat, ist $R_4 = \emptyset$ und damit ist die Bestimmung der Regeln für X abgeschlossen.

Eine weiterführende Ausführung des Algorithmus findet insgesamt 29 Assoziationsregeln für $minsupp = 0.4$ und $minconf = 0.9$.

3.3.3 Der FP-Growth-Algorithmus

Wie in Beispiel 66 zu sehen war, generiert der Apriori-Algorithmus eine relativ große Menge an Kandidaten für häufige Mengen. Obwohl im Durchschnitt diese Anzahl immer noch signifikant geringer ist als die Menge aller möglichen Teilmengen, die ein naiver Algorithmus betrachten würde (beachten Sie, dass bei 7 Elementen die Anzahl aller Teilmengen $2^7 = 128$ ist), so ist sowohl die Anzahl der Kandidaten als auch der Aufwand, den man betreiben muss, um für alle diese Mengen den Support zu berechnen, recht hoch. Der FP-Growth-Algorithmus (FP steht hier für *frequent pattern*, d. h., „häufige Mengen“) ist eine Weiterentwicklung des Apriori-Algorithmus, der im Durchschnitt signifikant weniger Kandidaten berücksichtigt und deshalb signifikant schneller als der Apriori-Algorithmus ist. Im Folgenden schauen wir uns nur das Teilproblem der Extraktion häufiger Mengen an (das beim Apriori-Algorithmus durch den Algorithmus FreqItems, siehe Algorithmus 6, gelöst wird). Beachten Sie, dass die Bestimmung von Assoziationsregeln aus den häufigen Mengen analog zur Extraktion der häufigen Mengen funktioniert. Der folgende Algorithmus kann damit in ähnlicher Weise auch dazu verwendet werden.

Sei ein Datensatz $F = \{t_1, \ldots, t_m\}$ über den Elementen I und $minsupp \in [0,1]$ gegeben. Wir möchten wieder die *häufigen* Mengen aus I bestimmen, d. h., solche $X \subseteq I$ mit $support_F(X) \geq minsupp$. Im Folgenden ist es hilfreich neben

dem relativen minimalen Support *minsupp* auch den *absoluten minimalen Support* $minsupp_{\text{abs}}$ zu betrachten. Für einen konkreten Datensatz F ist der absolute minimale Support definiert durch $minsupp_{\text{abs}} = \lceil minsupp * |F| \rceil$, also die Anzahl an Transaktionen von F, bei denen eine Menge M enthalten sein muss, um als häufig zu gelten.

Der FP-Growth-Algorithmus geht zur Berechnung der häufigen Mengen in zwei Schritten vor. In einem ersten Schritt wird der Datensatz F zunächst in eine besondere Datenstruktur, den sogenannten FP-Baum, T_F, gelesen. Der FP-Baum enthält alle für das Extrahieren von häufigen Mengen relevanten Informationen aus F, ist allerdings in der Regel weitaus kompakter. In einem zweiten Schritt werden aus T_F die häufigen Mengen extrahiert, was aufgrund der intelligenten Struktur von T_F mit weniger Mehraufwand verbunden ist als beim Apriori-Algorithmus. Die generelle Struktur des FP-Baums ist wie folgt gegeben.

Definition 26. Sei I eine Menge von Elementen. Ein *FP-Baumknoten* N zu I ist eine Datenstruktur $N = (x, f, p)$ mit

1. $x \in I$,
2. $f \in \mathbb{N}$,
3. p ist ein FP-Baumknoten (der Elternknoten von N).

Ein *FP-Baum* T zu I ist eine Datenstruktur $T = (N_0, V)$ mit

1. N_0 ist ein FP-Baumknoten mit $N_0 = (null, 0, null)$ (die Wurzel des Baumes)
2. V ist eine Menge von FP-Baumknoten, so dass für alle $N \in V$ mit $N = (x, f, p)$ gilt: $p \in V \cup \{N_0\}$. Dabei gibt p den Elternknoten von N an. Jeder Knoten $N \in V$ verfügt über einen eindeutigen Pfad zur Wurzel über seinen Elternknoten.

Ein FP-Baum T_F wird zu einem Datensatz F so aufgebaut, dass die Knoten in T_F (bis auf die Wurzel) diejenigen Elemente $x \in I$ enthalten, so dass $\{x\}$ häufig in F ist (wobei solch ein x auch öfters in T_F vorkommen kann). Für einen Knoten $N = (x, f, p)$ ist x das jeweilige Element, f die absolute Zahl von Vorkommen von x (bzgl. des Teilpfades bis N) und p verweist auf den Elternknoten. Jeder Pfad von der Wurzel bis zu einem Blatt repräsentiert dabei eine Menge von Elementen, die schon als häufig in F erkannt wurde. Die genaue Interpretation eines FP-Baumes T_F wird klar, wenn wir uns den Algorithmus zur Konstruktion von T_F genauer anschauen.

Um aus einem Datensatz F einen FP-Baum T_F zu konstruieren, bestimmen wir in einem ersten Schritt zunächst alle häufigen ein-elementigen Mengen $\{x\}$ mit $x \in I$. Sei $I_f = (x_1, \ldots, x_k)$ eine geordnete Liste dieser Elemente, sodass $support_F(\{x_i\}) \geq support_F(\{x_j\})$ gdw. $i < j$ gilt (die Elemente in I_f sind also absteigend nach Support geordnet; bei gleichem Support-Wert zweier Elemente wird die Ordnung beliebig aber fix definiert). Für jede Transaktion $t \in F$ sei nun $t_f = (x_{t,1}, \ldots, x_{t,t_n})$ die nach I_f geordnete Liste der häufigen Elemente in t.

Beispiel 68. Wir betrachten wieder $F_{\text{supermarket}} = \{t_1, \ldots, t_{10}\}$ aus Beispiel 65, siehe auch Tabelle 33, und nehmen wieder $minsupp = 0.4$ an, d. h., $minsupp_{\text{abs}} = 4$. Wir bestimmen zunächst den Support aller ein-elementigen Mengen (siehe auch Beispiel 66):

$$support_{F_{\text{supermarket}}}(\{\text{Milch}\}) = 0.8$$
$$support_{F_{\text{supermarket}}}(\{\text{Käse}\}) = 0.5$$
$$support_{F_{\text{supermarket}}}(\{\text{Butter}\}) = 0.7$$
$$support_{F_{\text{supermarket}}}(\{\text{Brot}\}) = 0.4$$

$$support_{F_{\text{supermarket}}}(\{\text{Kaffee}\}) = 0.5$$
$$support_{F_{\text{supermarket}}}(\{\text{Zucker}\}) = 0.6$$
$$support_{F_{\text{supermarket}}}(\{\text{Mehl}\}) = 0.6$$

Damit sind alle ein-elementigen Mengen häufig. Aufgrund der Support-Werte definieren wir die Ordnung

$$I_f = (\text{Milch}, \text{Butter}, \text{Zucker}, \text{Mehl}, \text{Kaffee}, \text{Käse}, \text{Brot})$$

Damit gilt für die Transaktionen aus Tabelle 33:

$$t_f^1 = (\text{Milch}, \text{Butter}, \text{Zucker}, \text{Mehl}, \text{Käse}, \text{Brot})$$
$$t_f^2 = (\text{Butter}, \text{Zucker}, \text{Mehl}, \text{Kaffee}, \text{Käse}, \text{Brot})$$
$$t_f^3 = (\text{Milch}, \text{Butter}, \text{Zucker}, \text{Mehl}, \text{Kaffee})$$
$$t_f^4 = (\text{Milch}, \text{Butter})$$
$$t_f^5 = (\text{Milch}, \text{Butter}, \text{Kaffee})$$
$$t_f^6 = (\text{Milch}, \text{Mehl})$$
$$t_f^7 = (\text{Milch}, \text{Butter}, \text{Zucker}, \text{Mehl}, \text{Kaffee}, \text{Käse}, \text{Brot})$$
$$t_f^8 = ()$$
$$t_f^9 = (\text{Milch}, \text{Zucker}, \text{Mehl}, \text{Kaffee}, \text{Käse}, \text{Brot})$$
$$t_f^{10} = (\text{Milch}, \text{Zucker}, \text{Mehl}, \text{Kaffee}, \text{Käse}, \text{Brot})$$

Die geordneten Listen der häufigen Elemente werden nun iterativ in einen initial leeren (d. h., nur aus der Wurzel N_0 bestehenden) FP-Baum $T = (N_0, V)$ eingefügt. Für ein $t_f = (x_1, \ldots, x_k)$ wird dabei zunächst überprüft, ob für ein Kind $N = (x, f, p)$ der Wurzel $x = x_1$ gilt. Im negativen Fall, wird ein neuer Knoten $N = (x_1, 1, N_0)$ erstellt und V hinzugefügt. Sei nun also $N = (x, f, p)$ das Kind der Wurzel mit $x = x_1$. Dann wird der Wert f von N um 1 erhöht

Algorithmus 8 Der Algorithmus FPTree zur Konstruktion eines FP-Baumes

Eingabe: Datensatz F, $minsupp_{\text{abs}} \in \mathbb{N}$
Ausgabe: FP-Baum für F

FPTree(F,$minsupp_{\text{abs}}$)
1: $T := (N_0, V)$
2: $N_0 := (null, 0, null)$
3: $V := \emptyset$
4: Bestimme I_f (eine geordnete Liste häufiger Elemente)
5: **for** $t \in F$ **do**
6: $S := t_f = (x_1, \ldots, x_k)$
7: $N := N_0$
8: **while** $S \neq ()$ **do**
9: **if** N hat kein Kind N' für x_1 **then**
10: Sei $N' = (x_1, 0, N)$ neuer Knoten
11: $V := V \cup \{N'\}$
12: Sei $N' = (x_1, f, N)$
13: $f := f + 1$
14: $N := N'$
15: $S := (x_2, \ldots, x_k)$
16: **return** T

und die Prozedur wird rekursiv am Knoten N für die verkürzte Sequenz $(x_2, \ldots, x_k)$ fortgeführt. Algorithmus 8 formalisiert diese Idee.

Beispiel 69. Wir führen Beispiel 68 fort, wo wir bereits die Ordnung I_f erstellt haben. Wir fügen nun die ersten drei Transaktionen der Reihe nach in einen leeren FP-Baum (siehe Abbildung 66a) ein:

1. Für die erste Transaktion gilt

$$t_f^1 = (\text{Milch}, \text{Butter}, \text{Zucker}, \text{Mehl}, \text{Käse}, \text{Brot})$$

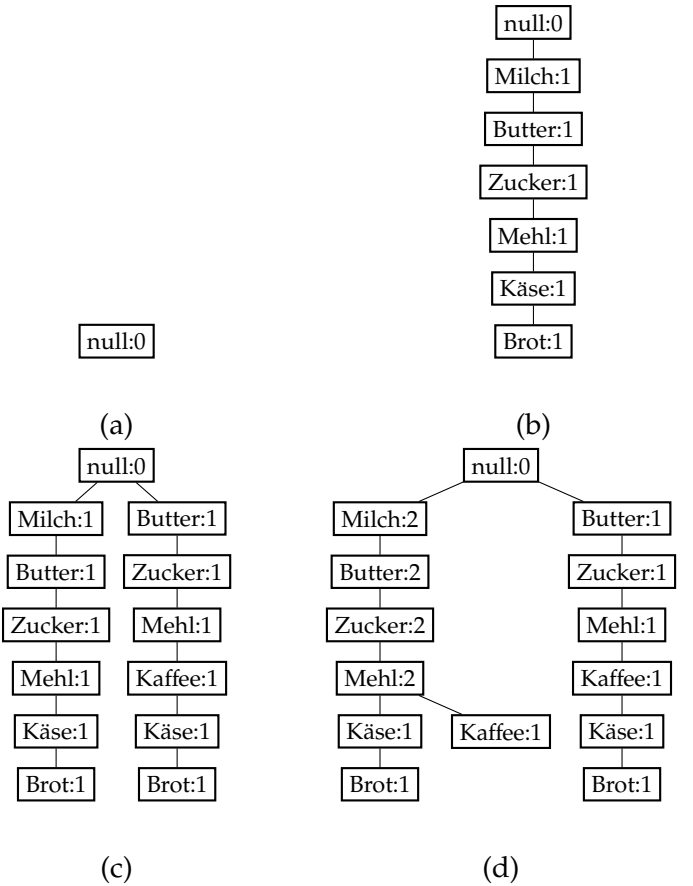


Abbildung 66: Die ersten vier Schritte in der Konstruktion des FP-Baumes aus Beispiel 69; jeder Knoten zeigt das Element und die Frequenz (f) durch Doppelpunkt getrennt an.

Da die Wurzel von T_F keinerlei Kinder besitzt, erzeugen wir zunächst ein neues Kind für „Milch“. Wir fahren fort, indem wir beim gerade erstellten Knoten den Rest (Butter, Zucker, Mehl, Käse, Brot) der Sequenz einfügen. Wir erhalten den Baum, der in Abbildung 66b dargestellt ist.

2. Für die zweite Transaktion gilt

$$t_f^2 = (\text{Butter}, \text{Zucker}, \text{Mehl}, \text{Kaffee}, \text{Käse}, \text{Brot})$$

   Da die Wurzel von T_F kein Kind für „Butter“ besitzt, erzeugen wir zunächst ein neues Kind für „Butter“, fügen es entsprechend ein und fahren mit der Einfügung fort. Wir erhalten den Baum, der in Abbildung 66c dargestellt ist.

3. Für die dritte Transaktion gilt

$$t_f^3 = (\text{Milch}, \text{Butter}, \text{Zucker}, \text{Mehl}, \text{Kaffee})$$

   Für die ersten drei Elemente von t_f^3 können wir einen bereits existierenden Pfad in T_F abschreiten, dabei wird nur stets die Frequenz des Knotens (Komponente f) erhöht. Nach dem Knoten Mehl:2 fügen wir dann noch einen neuen Knoten für „Kaffee“ ein. Wir erhalten den Baum, der in Abbildung 66d dargestellt ist.

Der fertige FP-Baum nach Einfügung aller 10 Transaktionen ist in Abbildung 67 dargestellt.

Ein nach Algorithmus 8 konstruierter FP-Baum enthält kompakt alle Information über das Auftreten häufiger Elemente und mit welchen anderen *häufigeren* Elementen diese häufig auftreten. Diese Information ist in den Pfaden von den jeweiligen Elementen bis zur Wurzel enthalten. Schauen wir uns den FP-Baum in Abbildung 67 noch einmal an und betrachten das Element „Mehl“. Insgesamt gibt es vier Knoten in dem Baum für das Element „Mehl“ und die zugehörigen Pfade (ohne die Wurzel)

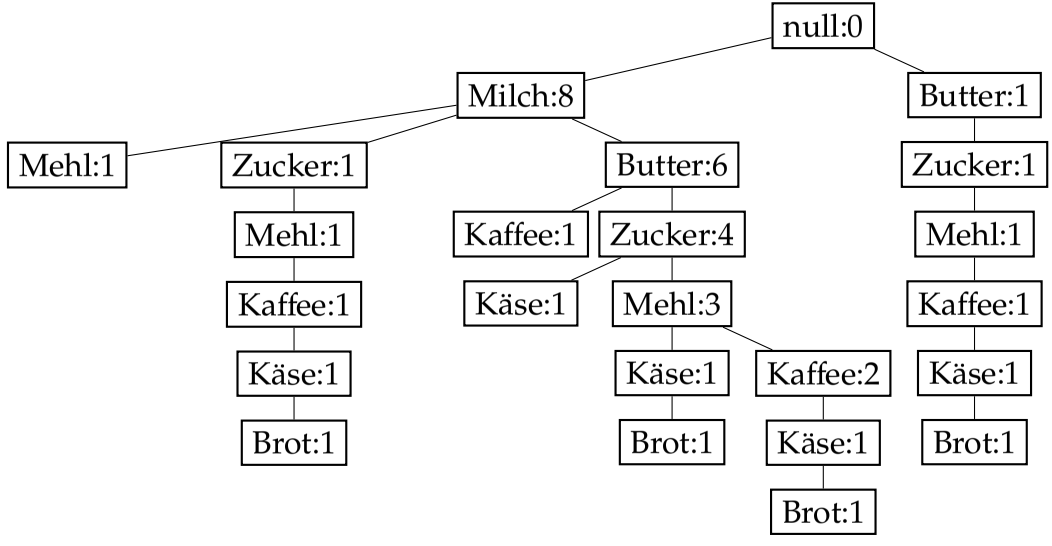


Abbildung 67: Der vollständige FP-Baum aus Beispiel 69.

sind

$$P_1 = (\text{Mehl} : 1, \text{Milch} : 8)$$
$$P_2 = (\text{Mehl} : 1, \text{Zucker} : 1, \text{Milch} : 8)$$
$$P_3 = (\text{Mehl} : 3, \text{Zucker} : 4, \text{Butter} : 6, \text{Milch} : 8)$$
$$P_4 = (\text{Mehl} : 1, \text{Zucker} : 1, \text{Butter} : 1)$$

Aus diesen Pfaden kann man ablesen, dass „Mehl" 1-mal in Kombination mit „Milch" auftaucht (beachten Sie, dass für diese Beobachtung nur die Frequenz des betrachteten Elements von Relevanz ist, nicht die Frequenz der anderen Knoten), 1-mal mit „Zucker" und „Milch", 3-mal mit „Zucker", „Butter" und „Milch" und 1-mal mit „Zucker" und „Butter". Betrachten wir nun den folgenden Datensatz $F'_{\text{supermarket}}$:

$$\begin{aligned} F'_{\text{supermarket}} = \{&\{\text{Milch}\},\\ &\{\text{Zucker}, \text{Milch}\},\\ &\{\text{Zucker}, \text{Butter}, \text{Milch}\},\\ &\{\text{Zucker}, \text{Butter}, \text{Milch}\},\\ &\{\text{Zucker}, \text{Butter}, \text{Milch}\},\\ &\{\text{Zucker}, \text{Butter}\}\} \end{aligned}$$

$F'_{\text{supermarket}}$ enthält genau die in $F_{\text{supermarket}}$ enthaltenen Informationen zu häufigen Elementen, die mit „Mehl" zusammen vorkommen und *vor* „Mehl" in I_f einsortiert sind. Insbesondere gilt:

> $M \subseteq \{\text{Milch}, \text{Butter}, \text{Zucker}, \text{Mehl}\}$ ist eine häufige Menge in $F_{\text{supermarket}}$ mit „Mehl" $\in M$ (bzgl. $\text{minsupp}_{\text{abs}}$) gdw. $M \setminus \{\text{Mehl}\}$ eine häufige Menge in $F'_{\text{supermarket}}$ ist (bzgl. $\text{minsupp}_{\text{abs}}$).

Die obige Einsicht erlaubt es, zur Bestimmung aller häufigen Mengen $M \subseteq \{\text{Milch}, \text{Butter}, \text{Zucker}, \text{Mehl}\}$ mit „Mehl" $\in$

M einfach die häufigen Mengen von $F'_{\text{supermarket}}$ zu bestimmen und „Mehl“ hinzuzufügen. Hierzu kann man dann den (noch zu konkretisierenden) FP-Growth-Algorithmus rekursiv aufrufen. Den Datensatz $F'_{\text{supermarket}}$ nennt man auch den zu „Mehl“ *konditionierten* Datensatz und dieser ist allgemein wie folgt definiert.

Definition 27. Sei F ein Datensatz, $x \in I$ und $T_F = (N_0, V, h)$ der mit Algorithmus 8 konstruierte FP-Baum zu F. Eine *x-Transaktion* $\hat{t}$ in T_F ist eine Menge $\hat{t} = \{x_1, \ldots, x_k\}$, sodass es Knoten $n_x, n_{x_1}, \ldots, n_{x_k}$ für die Elemente $x, x_1, \ldots, x_k$ gibt, die einen Pfad von $n_x, n_{x_1}, \ldots, n_{x_k}, N_0$ bilden. Bei $n_x = (x, f, p)$ heißt $\text{mult}(\hat{t}) = f$ die *Multiplizität* von $\hat{t}$.

Die zu x *konditionierte* Datenbasis $F|x$ ist genau die Menge von Transaktionen, bei der jede x-Transaktion $\hat{t}$ genau $mult(\hat{t})$-mal vorkommt.

Allgemein gilt dann das folgende Theorem (das wir hier nicht beweisen).

Theorem 1. *Sei $x \in J$ und $J' = \{x' \mid x' \text{ kommt vor } x \text{ in } I_f\}$. $M \cup \{x\}$ mit $M \subseteq J'$ ist eine häufige Menge in F (bzgl. $minsupp_{abs}$) gdw. M eine häufige Menge in $F|x$ ist (bzgl. $minsupp_{abs}$).*

Will man nun alle häufigen Mengen zu F bestimmen und es ist $I_f = (x_1, \ldots, x_k)$, so gilt zunächst trivialerweise, dass $\{x_1\}, \ldots, \{x_k\}$ häufige Mengen sind. Anschließend bestimmt man die häufigen Mengen von $F|x_k$. Da x_k als letztes in I_f vorkommt, sind diese Mengen identisch mit den häufigen Mengen von F, die x_k enthalten. Anschließend[24] bestimmt man alle häufigen Mengen von $F|x_{k-1}$. Da x_k hinter x_{k-1} in I_f ist, enthalten alle diese Mengen x_k *nicht*, es wird also keine häufige Menge mehrfach berechnet. Insgesamt gilt die folgende Aussage (sei $fp(F)$ die Menge der häufigen Mengen von F).

[24] Beachten Sie, dass die tatsächliche Reihenfolge der Berechnung nicht relevant ist, solange dies für alle Elemente $x_1, \ldots, x_k$ geschieht.

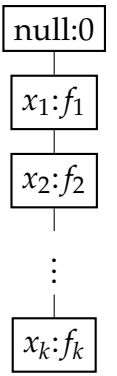


Abbildung 68: Trivialer Fall eines FP-Baums mit nur einem Pfad.

Theorem 2. *Sei* $I_f = (x_1, \ldots, x_k)$*, dann gilt*

$$fp(F) = \{x_1\} \cup \ldots \cup \{x_k\} \cup \bigcup_{i=1}^{k} \{\{x_i\} \cup M \mid M \in fp(F|x_i)\}$$

Wie bereits oben erwähnt, kann man die häufigen Mengen der konditionierten Datensätze rekursiv wieder mit dem FP-Growth-Algorithmus berechnen. Offen ist an dieser Stelle nur noch, wann diese Rekursion endet. Der Basisfall ist erreicht, wenn für einen Datensatz F der FP-Baum T_F nur noch aus einem einzigen Pfad besteht. In diesem Fall ist es einfach einzusehen, dass die häufigen Mengen von F genau alle nicht-leeren Teilmengen der in T_F vorkommenden Elemente sind. Sei dazu F ein Datensatz und T_F wie in Abbildung 68 dargestellt. Nach Konstruktion gilt $f_1 \geq \ldots \geq f_k \geq minsupp_{\text{abs}}$. Sei nun $M \subseteq \{x_1, \ldots, x_k\}$ eine beliebige nicht-leere Menge und j der größte Index mit $x_j \in M$. Dann kommt die Menge M genau f_j-mal in den Transaktionen von F vor, ist also häufig.

Algorithmus 9 formalisiert die obigen Ideen. Anders

Algorithmus 9 Der FP-Growth-Algorithmus.

Eingabe: Datensatz F, $minsupp_{\text{abs}} \in \mathbb{N}$
Ausgabe: fp(F)
FPGrowth(F, $minsupp_{\text{abs}}$)
1: $T := FPTree(F, minsupp_{\text{abs}})$
2: $K := \emptyset$
3: **if** T besteht aus einem einzigen Pfad **then**
4: $K :=$ Menge aller nicht-leeren Teilmengen von Elementen in T
5: **else**
6: **for** Element x in T **do**
7: $K' :=$ FPGrowth($F|x$, $minsupp_{\text{abs}}$)
8: $K := K \cup \{x\} \cup \{\{x\} \cup M \mid M \in K'\}$
9: **return** K

als bei Algorithmus 6 gehen wir davon aus, dass dem Algorithmus $minsupp_{\text{abs}}$ statt $minsupp$ übergeben wird.

Beispiel 70. Wir führen Beispiel 69 fort und führen Algorithmus 9 schrittweise durch. Wir nehmen weiterhin an, dass $minsupp_{\text{abs}} = 4$.

1. Den FP-Baum $T_{F_{\text{supermarket}}}$ des Datensatzes $F_{\text{supermarket}}$ haben wir bereits in Beispiel 69 berechnet und dieser ist in Abbildung 67 dargestellt.

2. Da $T_{F_{\text{supermarket}}}$ nicht aus nur einem Pfad besteht, führen wir Zeilen 7 und 8 für alle Elemente aus. Wir tun dies nun exemplarisch für „Brot“.

3. $T_{F_{\text{supermarket}}}$ enthält genau vier Pfade von der Wurzel mit einem Endknoten zu „Brot“ und alle abgeleiteten „Brot“-Transaktionen haben Multiplizität 1. Der

Datensatz $F_{\text{supermarket}}|\text{Brot}$ berechnet sich daher zu

$$\begin{aligned} & F_{\text{supermarket}}|\text{Brot} \\ = \{ & \{\text{Milch}, \text{Zucker}, \text{Mehl}, \text{Kaffee}, \text{Käse}\}, \\ & \{\text{Milch}, \text{Butter}, \text{Zucker}, \text{Mehl}, \text{Käse}\}, \\ & \{\text{Milch}, \text{Butter}, \text{Zucker}, \text{Mehl}, \text{Kaffee}, \text{Käse}\}, \\ & \{\text{Butter}, \text{Zucker}, \text{Mehl}, \text{Kaffee}, \text{Käse}\}\} \end{aligned}$$

Wir rufen rekursiv

$$FPGrowth(F_{\text{supermarket}}|\text{Brot}, minsupp_{\text{abs}})$$

auf.

4. Wir berechnen den FP-Baum zu $F_{\text{supermarket}}|\text{Brot}$, dessen häufige ein-elementige Mengen gegeben sind durch

   $$\{\{\text{Zucker}\}, \{\text{Mehl}\}, \{\text{Käse}\}\}$$

   jeweils mit absolutem Support 4. Wir legen die Ordnung $I_f = (\text{Zucker}, \text{Mehl}, \text{Käse})$ fest.

5. Der aus $F_{\text{supermarket}}|\text{Brot}$ bzgl. I_f konstruierte FP-Baum ist in Abbildung 69 dargestellt.

6. Da der FP-Baum nur einen Pfad hat, bestimmen sich die häufigen Mengen von $F_{\text{supermarket}}|\text{Brot}$ zu

   $$\begin{aligned} & fp(F_{\text{supermarket}}|\text{Brot}) \\ = \{ & \{\text{Zucker}\}, \{\text{Mehl}\}, \{\text{Käse}\}, \{\text{Zucker}, \text{Mehl}\}, \\ & \{\text{Zucker}, \text{Käse}\}, \{\text{Mehl}, \text{Käse}\}, \{\text{Zucker}, \text{Mehl}, \text{Käse}\}\} \end{aligned}$$

   und damit sind die zu $F_{\text{supermarket}}$ häufigen Mengen, die „Brot“ enthalten bestimmt zu

   $$\begin{aligned} & \{\text{Brot}\}, \{\text{Brot}, \text{Zucker}\}, \{\text{Brot}, \text{Mehl}\}, \{\text{Brot}, \text{Käse}\}, \\ & \{\text{Brot}, \text{Zucker}, \text{Mehl}\}, \{\text{Brot}, \text{Zucker}, \text{Käse}\}, \\ & \{\text{Brot}, \text{Mehl}, \text{Käse}\}, \{\text{Brot}, \text{Zucker}, \text{Mehl}, \text{Käse}\} \end{aligned}$$

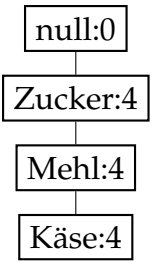


Abbildung 69: Der aus $F_{\text{supermarket}}|\text{Brot}$ bzgl. $I_f = (\text{Zucker}, \text{Mehl}, \text{Käse})$ konstruierte FP-Baum aus Beispiel 70.

Vergleiche hierzu die Liste aller häufigen Mengen von Seite 199, Schritt 5.

3.4 Anomalieerkennung

Bei der Anomalieerkennung (engl. *anomaly detection*) geht es um die Erkennung von abnormalen Datenpunkten oder Ausreißern. Gegeben eine Menge von Datenpunkten $E = \{x^{(1)}, \ldots, x^{(m)}\}$ und ein neuer Datenpunkt x, besteht die Frage der Anomalieerkennung darin, zu entscheiden, ob x *ähnlich* genug zu den Datenpunkten in E ist oder eher *abnormal*. Es gibt dazu mehrere Möglichkeiten, wie man dieses Problem mit Methoden des maschinellen Lernens angehen kann. Falls wir nicht nur Daten zu *normalen* Datenpunkten haben, sondern auch zu *abnormalen* Datenpunkten, dann kann man das Problem als überwachtes Lernproblem ansehen. Wir haben solch ein Problem prinzipiell schon in Abschnitt 2.2.3, Beispiel 14, und Abschnitt 2.3.2, Beispiel 5, betrachtet, wo es um die Erkennung von Schrauben als „ok“ und "nicht ok“ ging. Üblicherweise verfügt man bei dieser Art von Problemen allerdings eher über eine große Menge an *normalen* Datenpunkten und nur relativ wenig (oder keine) *abnormale* Datenpunkte. Aus diesem Grund benutzt man für Anomalieerkennung üblicherweise unüberwachte Lernverfahren, bei denen nur die Menge E an *normalen* Datenpunkten gegeben ist.

Beispiel 71. Wir betrachten ein ähnliches Problem wie in Beispiel 14, Abschnitt 2.2.3. Tabelle 71 zeigt den Datensatz E_{screws}, der die Merkmale „Gewicht“ und „Länge“ von 10 Schrauben enthält. Diese 10 Schrauben werden als *normal* angesehen.

Abbildung 70 visualisiert den Datensatz E_{screws}.

Wir schauen uns in diesem Unterkapitel eine einfache Methode der Anomalieerkennung an, nämlich die Dichteabschätzung einer Normalverteilung. Dieser Ansatz hat zwei Anknüpfungspunkte zu Unterkapitel 2.5

Nr.	Gewicht (in g)	Länge (in mm)
1	11	27
2	10	28
3	10.3	27.5
4	11	28
5	10.9	27.1
6	10.9	28.1
7	10.2	28.2
8	10.7	27.6
9	10.9	27.2
10	10.1	27

Tabelle 34: Datensatz E_{screws} zu Beispiel 71.

zur Bayes-Klassifikation. Zum einen ist die Dichteabschätzung eine Maximum-Likelihood-Methode, da das gelernte Modell auf den Grundsätzen der Maximum-Likelihood-Hypothese aufgebaut wird (vergleiche hierzu die Modellbildung unten mit der Herleitung in Abschnitt 2.5.1). Zum anderen löst der Ansatz der Dichteabschätzung das in Abschnitt 2.5.3 angesprochene Problem der Bayes-Klassifikation mit kontinuierlichen Merkmalen, da die dort gesuchte Dichte $p(x_i \mid c, D)$ damit bestimmt werden kann.

Um das Problem der Anomalieerkennung zu lösen, bilden wir ein wahrscheinlichkeitstheoretisches Modell p^E auf den Datenpunkten. Die zugrundeliegende Intuition ist, dass $p^E(x)$ für einen beliebigen Datenpunkt x angibt, wie ähnlich x zu den Datenpunkten in E ist. Die tatsächliche Klassifikation eines Datenpunktes x als *normal* oder *abnormal* geschieht dann durch einen Schwellenwert $\epsilon \in [0,1]$, d. h., wir klassifizieren einen Datenpunkt x als *abnormal* gdw. $p^E(x) < \epsilon$ gilt. Wir machen zwei zentrale

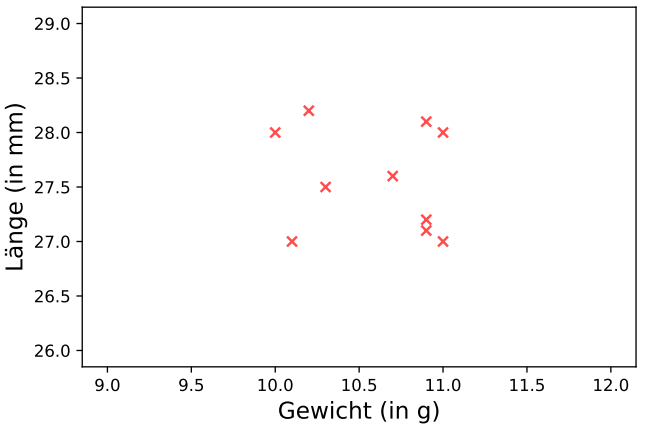


Abbildung 70: Datensatz E_{screws} zu Beispiel 71.

Annahmen für die Bestimmung von p^E:

1. Die Merkmale von $x = (x_1, \ldots, x_n)$ sind *unabhängig voneinander*, d. h., wir können $p^E(x)$ schreiben als

$$p^E(x) = p_1^E(x_1) \cdot \ldots \cdot p_n^E(x_n)$$

   mit Wahrscheinlichkeitsdichten $p_1^E, \ldots, p_n^E$ für die einzelnen Merkmale $x_1, \ldots, x_n$.

2. Die Merkmalsausprägungen eines jeden Merkmals x_i, $i = 1 \ldots, n$ sind *normalverteilt*, d. h., es gilt

$$p_i^E(x_i) = \frac{1}{\sqrt{2\pi\sigma_i^2}} e^{-\frac{(x_i - \mu_i)^2}{2\sigma_i^2}}$$

   für „passende“ $\mu_i, \sigma_i^2 \in \mathbb{R}$.

Die beiden obigen Annahmen sind für einen gegebenen Datensatz E eines bestimmten Problems nicht immer korrekt. Ähnlich wie bei der naiven Bayes-Klassifikation liefert das resultierende Modell jedoch auch bei Nichtzutreffen der Annahmen gute Ergebnisse. Mit den obigen Annahmen besteht damit das Lernproblem daraus, passende Werte für $\mu_1, \ldots, \mu_n$ und $\sigma_1^2, \ldots, \sigma_n^2$ zu finden. Wir benutzen dazu die Maximum-Likelihood-Methode und nehmen an, dass μ_i genau die *empirischen* Mittelwerte und σ_i^2 genau die *empirischen* Varianzen aus dem Datensatz E sind, d. h., wir setzen

$$\mu_i = \frac{1}{m} \sum_{j=1}^{m} x_i^{(j)}$$

$$\sigma_i^2 = \frac{1}{m} \sum_{j=1}^{m} (x_i^{(j)} - \mu_i)^2$$

für alle $i = 1, \ldots, n$. Mit anderen Worten, jedes μ_i ist genau der Durchschnitt der Datenpunkte in E und jedes σ_i^2 ist genau der Durchschnitt der quadratischen Abweichungen der Datenpunkte aus E zu μ_i.

Beispiel 72. Wir führen Beispiel 71 fort. Wir berechnen

$$\begin{aligned} &\mu_{\text{Gewicht}} \\ &= \frac{11 + 10 + 10.3 + 11 + 10.9 + 10.9 + 10.2 + 10.7 + 10.9 + 10.1}{10} \\ &= 10.6 \end{aligned}$$

$$\begin{aligned} &\mu_{\text{Länge}} \\ &= \frac{27 + 28 + 27.5 + 28 + 27.1 + 28.1 + 28.2 + 27.6 + 27.2 + 27}{10} \\ &= 27.57 \end{aligned}$$

und

$$\begin{aligned}\sigma^2_{\text{Gewicht}} = \frac{1}{10}((11-\mu_{\text{Gewicht}})^2 + (10-\mu_{\text{Gewicht}})^2 + \\ (10.3-\mu_{\text{Gewicht}})^2 + (11-\mu_{\text{Gewicht}})^2 + \\ (10.9-\mu_{\text{Gewicht}})^2 + (10.9-\mu_{\text{Gewicht}})^2 + \\ (10.2-\mu_{\text{Gewicht}})^2 + (10.7-\mu_{\text{Gewicht}})^2 + \\ (10.9-\mu_{\text{Gewicht}})^2 + (10.1-\mu_{\text{Gewicht}})^2) \\ = 0.146\end{aligned}$$

$$\begin{aligned}\sigma^2_{\text{Länge}} = \frac{1}{10}((27-\mu_{\text{Länge}})^2 + (28-\mu_{\text{Länge}})^2 + \\ (27.5-\mu_{\text{Länge}})^2 + (28-\mu_{\text{Länge}})^2 + \\ (27.1-\mu_{\text{Länge}})^2 + (28.1-\mu_{\text{Länge}})^2 + \\ (28.2-\mu_{\text{Länge}})^2 + (27.6-\mu_{\text{Länge}})^2 + \\ (27.2-\mu_{\text{Länge}})^2 + (27-\mu_{\text{Länge}})^2 \\ = 0.2061\end{aligned}$$

Beispielsweise gilt dann für einen neuen Datenpunkt $x = (x_1, x_2)^T = (10.8, 27.5)^T$

$$\begin{aligned}p^{E_{\text{screws}}}(x) &= p^{E_{\text{screws}}}_{\text{Gewicht}}(x_1) p^{E_{\text{screws}}}_{\text{Länge}}(x_2) \\ &= \frac{1}{\sqrt{2\pi\sigma^2_{\text{Gewicht}}}} e^{-\frac{(x_1-\mu_{\text{Gewicht}})^2}{2\sigma^2_{\text{Gewicht}}}} \frac{1}{\sqrt{2\pi\sigma^2_{\text{Länge}}}} e^{-\frac{(x_2-\mu_{\text{Länge}})^2}{2\sigma^2_{\text{Länge}}}} \\ &\approx 0.791\end{aligned}$$

und für einen Datenpunkt $x' = (x'_1, x'_2)^T = (9.8, 26.9)^T$

$$p^{E_{\text{screws}}}(x') \approx 0.034$$

Würden wir also beispielsweise den Schwellwert ϵ auf $\epsilon = 0.1$ setzen, so würde x als *normal* und x' als *abnormal*

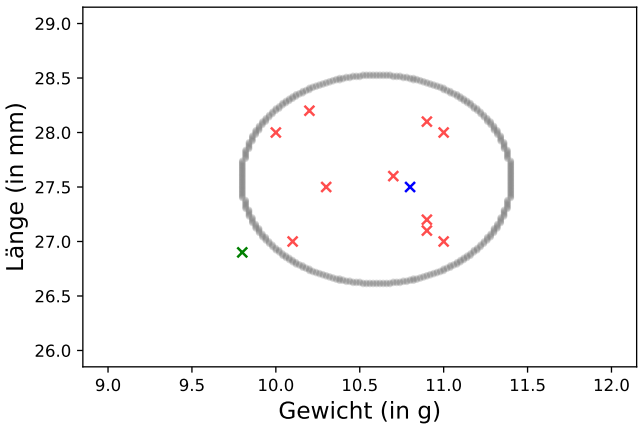


Abbildung 71: Datensatz E_{screws} zu Beispiel 72, die beiden Testpunkte x (in blau) und x' (in grün) und die Entscheidungsgrenze für $\epsilon = 0.1$.

klassifiziert werden. Abbildung 71 visualisiert den Datensatz E_{screws},die beiden Testpunkte x und x' und die Entscheidungsgrenze für $\epsilon = 0.1$.

3.5 Hauptkomponentenanalyse

In den vorangegangenen Unterkapiteln haben wir bereits das eine oder andere Mal den Sachverhalt von korrelierenden Merkmalen angesprochen. Insbesondere haben wir an einigen Stellen Unabhängigkeitsannahmen zu Merkmalen getroffen, um eine besondere Umformung einer Gleichung zu rechtfertigen (zentral war dies beispielsweise bei der naiven Bayes-Klassifikation der Fall). In diesen Fällen ist ein Nichtzutreffen dieser Annahme normalerweise ohne schwerwiegende Auswirkungen und die entsprechenden Lernmethoden können damit robust umgehen. In der Praxis treten Merkmale mit korrelierenden Merkmalsausprägungen allerdings recht häufig auf und können sich mitunter negativ auf den Lernprozess auswirken. Schauen wir uns dazu ein triviales Beispiel an, nämlich die *Dopplung* eines Merkmals. Nehmen wir einen Datensatz an, bei dem sich die einzelnen Beispiele auf Merkmale von Personen beziehen und nehmen wir an, dass wir zwei Merkmale „Größe (in cm)“ und „Größe (in mm)“ haben, also einfach die Größe der Person in zwei verschiedenen Einheiten. Sind die Daten korrekt, so ist jede Merkmalsausprägung des zweiten Merkmals einfach 10-mal die Merkmalsausprägung des ersten Merkmals. Das zweite Merkmal bringt also keine weitere Information. Haben wir beide Merkmale in unserem Datensatz, so können einige Lernmethoden eine unbeabsichtigte Gewichtung in das gelernte Modell einbringen. Da das Merkmal „Größe“ doppelt vorkommt, hat es damit auch einen doppelten Einfluss auf das Lernziel. Dieser Effekt ist aber normalerweise nicht erwünscht. Weiterhin können wir durch das Weglassen eines dieser Merkmale unseren Datensatz kompakter darstellen, ohne Information zu verlieren. Gerade bei großen Datensätzen und komplexen Lernalgorithmen kann solch eine Platzerspar-

nis erheblich zur Verringerung der Laufzeit beitragen. Die Dopplung eines Merkmals im Datensatz ist natürlich leicht zu erkennen und zu lösen. Ähnliche Effekte wie bei der Dopplung eines Merkmals treten allerdings auch schon bei Merkmalen auf, die nicht perfekt korrelieren, wie beispielsweise bei „Größe“ und „Gewicht“ einer Person. Ebenso kann es vorkommen, dass ein Merkmal als eine (ungefähre) Kombination von mehr als einem anderen Merkmal dargestellt werden kann. Auch in solchen Fällen kann es sinnvoll sein, die Daten mit weniger Merkmalen zu beschreiben, um damit unerwünschte Gewichtungen beim Lernen zu vermeiden und die Daten kompakter darzustellen. Allerdings ist es nicht immer einfach zu erkennen, wie sich Merkmale zueinander verhalten.

3.5.1 Grundlagen

Die *Hauptkomponentenanalyse* (engl. *Principal Component Analysis*, PCA) ist die bekannteste und am häufigsten genutzte Methode, um die die oben genannten Probleme zu lösen. PCA ist eine Methode zur *Dimensionsreduktion*, die einen Datensatz über Beispielen oder Datenpunkten im $\mathbb{R}^n$ auf einen Datensatz über Beispielen oder Datenpunkten im $\mathbb{R}^{n'}$ mit $n' < n$ in einer Form abbildet, sodass möglichst wenig Information verloren geht (diesen Aspekt werden wir in Abschnitt 3.5.4 präzisieren). Sie kann prinzipiell als Vorverarbeitungsschritt für das überwachte oder unüberwachte Lernen eingesetzt werden, oder auch einfach nur um einen Datensatz zu komprimieren. PCA selbst kann auch als eine Methode des unüberwachten Lernens verstanden werden (deswegen diskutieren wir sie auch an dieser Stelle), da sie nach Mustern im Datensatz sucht. Die allgemeine Methodik der PCA ist mathematisch recht komplex und wir werden uns zunächst mit der Intuition der Methode bei der Re-

duktion eines Datensatzes vom $\mathbb{R}^2$ zum $\mathbb{R}^1$ beschäftigen und erst in Abschnitt 3.5.3 die allgemeine Methode einführen.

Beispiel 73. Wir betrachten den Datensatz D_{ships} (siehe Tabelle 35 und Abbildung 72), der einige Schiffe anhand der Merkmale „Länge (in m)“ und „Verdrängung (in t)“, sowie deren Klassifikation in „Frachtschiff“ und „Passagierschiff“ auflistet. Anhand der Visualisierung der Da-

Nr.	Länge (in m)	Verdrängung (in t)	Typ
1	313	71485	Frachtschiff
2	269	53752	Passagierschiff
3	245	42124	Passagierschiff
4	332	97875	Frachtschiff
5	312	75364	Frachtschiff
6	211	32157	Passagierschiff

Tabelle 35: Datensatz D_{ships} aus Beispiel 73.

ten in Abbildung 72 ist klar zu erkennen, dass die beiden Merkmale „Länge (in m)“ und „Verdrängung (in t)“ stark korrelieren.

Um PCA anwenden zu können, ist es zunächst notwendig, den Datensatz zu standardisieren, sodass die Merkmalsausprägungen aller Merkmale den Mittelwert 0 haben. Dies geschieht am einfachsten mit der z-Transformation (siehe auch Abschnitt 2.4.3).[25] Beachten Sie,

[25] Es sind hier allerdings auch andere Standardisierungsmethoden anwendbar. Insbesondere ist es nicht zwingend notwendig, dass die Standardisierung die *Varianz* aller Merkmale auf 1 setzt (wie es die z-Transformation macht).

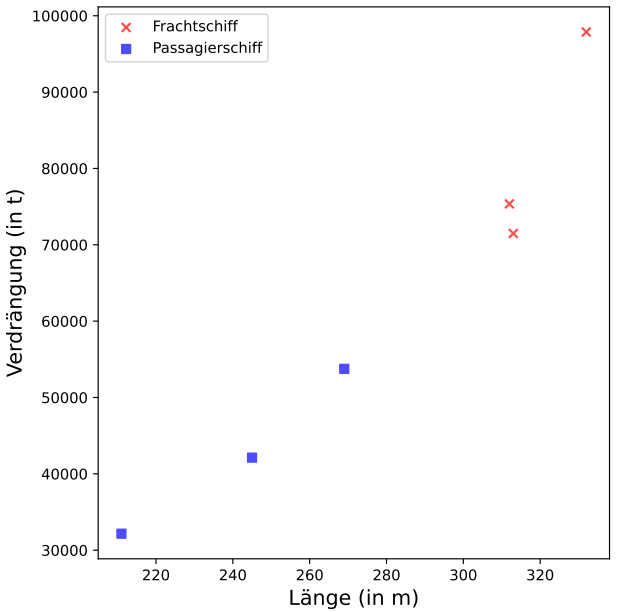


Abbildung 72: Datensatz D_{ships} aus Beispiel 73.

dass (anders als bei vorherigen Methoden) diese Standardisierung nicht nur hilfreich, sondern auch notwendig ist.

Beispiel 74. Wir führen Beispiel 73 fort. Der z-transformierte Datensatz $\hat{D}_{\text{ships}}$ (auf 3 Nachkommastellen gerundet) zu Datensatz D_{ships} ist in Tabelle 36 und Abbildung 73 dargestellt.

Falls PCA auf ein Problem des überwachten Lernens angewendet werden soll (wie im obigen Beispiel), so können wir die Klassen der Beispiele ignorieren, da die Dimensionsreduktion einzig auf den Datenpunkten realisiert wird. Im Folgenden gehen wir also davon aus, dass wir einen Datensatz $E = \{x^{(1)}, \ldots, x^{(m)}\}$ mit $x^{(i)} \in \mathbb{R}^2$,

Nr.	Länge	Verdrängung	Typ
1	0.766	0.425	Frachtschiff
2	-0.266	-0.38	Passagierschiff
3	-0.828	-0.908	Passagierschiff
4	1.211	1.624	Frachtschiff
5	0.742	0.601	Frachtschiff
6	-1.625	-1.361	Passagierschiff

Tabelle 36: Datensatz $\hat{D}_{\text{ships}}$ aus Beispiel 74.

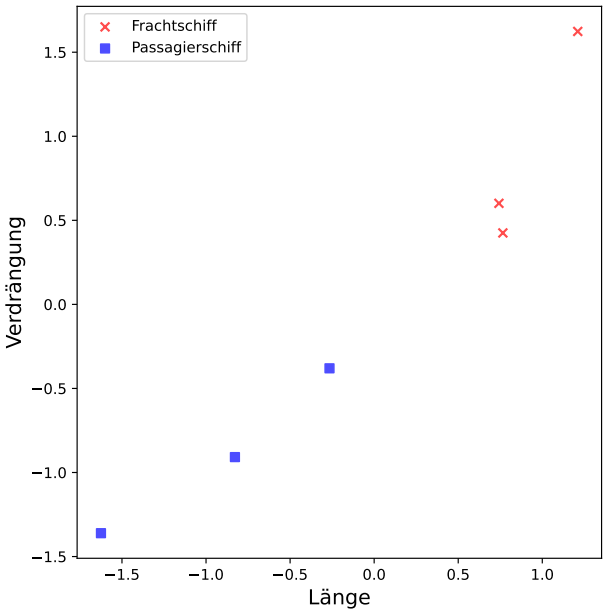


Abbildung 73: Datensatz $\hat{D}_{\text{ships}}$ aus Beispiel 74.

$i = 1, \ldots, m$ als Eingabe erhalten (im obigen Beispiel ignorieren wir dazu einfach die Spalte „Typ“). Für die Aufgabe, einen Datensatz vom $\mathbb{R}^2$ auf den $\mathbb{R}^1$ zu reduzieren,

versucht PCA nun einen 1-dimensionalen Unterraum (eine Gerade) von $\mathbb{R}^2$ zu finden, sodass die (quadrierten) Distanzen aller Punkte von E zu ihren jeweiligen Projektionen auf diesen Unterraum minimal sind.[26] Formal löst PCA in diesem Szenario das folgende Optimierungsproblem:

$$v^* = \arg \min_{v \in \mathbb{R}^2, \|v\|=1} \sum_{x \in E} \|x - (x^T v)v\|^2 \tag{21}$$

Der Vektor v^* charakterisiert dann den gesuchten 1-dimensionalen Unterrraum als alle Vielfachen von v^*, d. h., die gesuchte Gerade besteht aus allen Punkten $w \in \mathbb{R}^2$ mit $w = xv^*$ mit $x \in \mathbb{R}$. Der Ausdruck $\|x - (x^T v)v\|$ ist hierbei der Abstand des Punktes x zu der von v aufgespannten Geraden (d. h., der kleinste Abstand von x zu irgendeinem Punkt auf dieser Geraden). Die zusätzliche Bedingung $\|v\| = 1$ fordert, dass v genau Länge 1 hat. Diese Bedingung ist keine signifikante Einschränkung, sondern dient nur der Normierung. Beachten Sie, dass das Problem (21) im allgemeinen nicht eindeutig lösbar ist[27], in diesem Fall sei v^* eine beliebige minimale Lösung. Der Vektor v^* heißt dann (erste) *Hauptkomponente* von E.

Beispiel 75. Wir führen Beispiel 74 fort. Abbildung 74 zeigt die (in diesem Fall eindeutig bestimmte) Gerade, die durch den Nullpunkt geht und minimalen Abstand zu allen Punkten aus $\hat{D}_{\text{ships}}$ hat. Insbesondere ist hier

$$v^*_{\text{ships}} \approx (0.707, 0.707)^T$$

[26] Diese Gerade geht dabei notwendigerweise (da es sich um einen Unterraum handelt) durch den Nullpunkt. Aus diesem Grund ist auch die Standardisierung der Daten, insbesondere die Sicherstellung, dass der Mittelwert bei 0 liegt, notwendig.

[27] Insbesondere gilt, dass für eine Lösung v^* auch $-v^*$ eine Lösung ist, da diese Vektoren dieselbe Gerade charakterisieren. Es kann aber auch vorkommen, dass es verschiedene Geraden gibt, die das Problem (21) lösen.

eine Lösung des Optimierungsproblems (21), also eine erste Hauptkomponente von $\hat{D}_{\text{ships}}$. Abbildung 74 zeigt weiterhin auch die Projektionen der Punkte aus $\hat{D}_{\text{ships}}$ auf diese Gerade an (diese sind allerdings ohne Klassen aufgeführt).

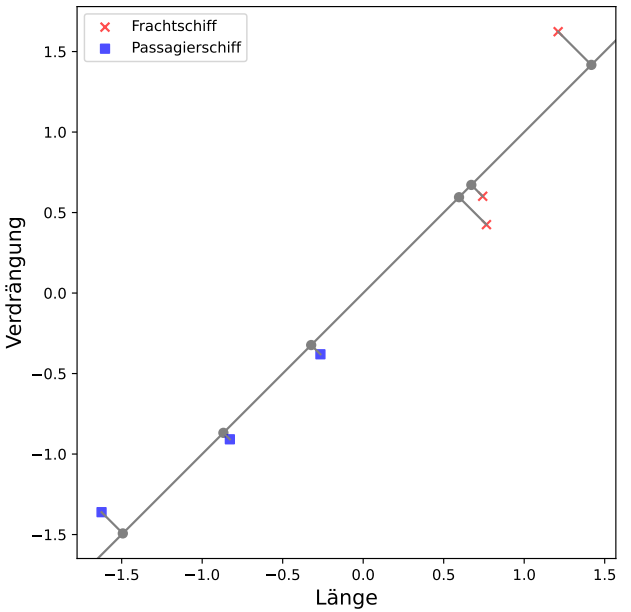


Abbildung 74: Datensatz $\hat{D}_{\text{ships}}$ und die zu $\hat{D}_{\text{ships}}$ nächste Gerade durch den Nullpunkt aus Beispiel 75.

Ist v^* bestimmt, so können wir nun den Datensatz E komprimieren, indem wir für jeden Datenpunkt $x \in E$ anstatt der beiden Koordinaten $x = (x_1, x_2)^T \in \mathbb{R}^2$ nur die Koordinate $z \in \mathbb{R}$ speichern, sodass zv^* die Projektion von x auf die durch v^* aufgespannte Gerade ist. Wir erhalten damit einen komprimierten Datensatz $E^{\text{compressed}}$.

Beispiel 76. Wir führen Beispiel 75 fort. Abbildung 74 zeigt bereits die Projektionen der Datenpunkte aus $\hat{D}_{\text{ships}}$ auf die durch v^*_{ships} aufgespannte Gerade. Die Projektionen $\hat{x}^{(1)}, \ldots, \hat{x}^{(6)}$ zu den Datenpunkten $x^{(1)}, \ldots, x^{(6)}$ sind dabei gegeben durch

$$\begin{aligned}
\hat{x}^{(1)} &\approx (0.595, 0.595)^T \approx 0.841 v^*_{\text{ships}} \\
\hat{x}^{(2)} &\approx (-0.323, -0.323)^T \approx -0.457 v^*_{\text{ships}} \\
\hat{x}^{(3)} &\approx (-0.868, -0.868)^T \approx -1.228 v^*_{\text{ships}} \\
\hat{x}^{(4)} &\approx (1.417, 1.417)^T \approx 2.004 v^*_{\text{ships}} \\
\hat{x}^{(5)} &\approx (0.672, 0.672)^T \approx 0.95 v^*_{\text{ships}} \\
\hat{x}^{(6)} &\approx (-1.493, -1.493)^T \approx -2.112 v^*_{\text{ships}}
\end{aligned}$$

Nun können wir die Punkte $x^{(1)}, \ldots, x^{(6)} \in \mathbb{R}^2$ komprimiert darstellen durch Punkte $z^{(1)}, \ldots, z^{(6)} \in \mathbb{R}^1$ mit

$$\begin{aligned}
z^{(1)} &= (0.841) \\
z^{(2)} &= (-0.457) \\
z^{(3)} &= (-1.228) \\
z^{(4)} &= (2.004) \\
z^{(5)} &= (0.95) \\
z^{(6)} &= (-2.112)
\end{aligned}$$

Abbildung 75 zeigt den komprimierten Datensatz

$$\hat{D}^{\text{compressed}}_{\text{ships}} = \{(z^{(1)}, y^{(1)}), \ldots, (z^{(6)}, y^{(6)})\}.$$

Nun können wir mit $\hat{D}^{\text{compressed}}_{\text{ships}}$ die ursprüngliche Aufgabe angehen, d. h., beispielsweise logistische Regression anwenden, um einen Klassifikator zu erlernen.

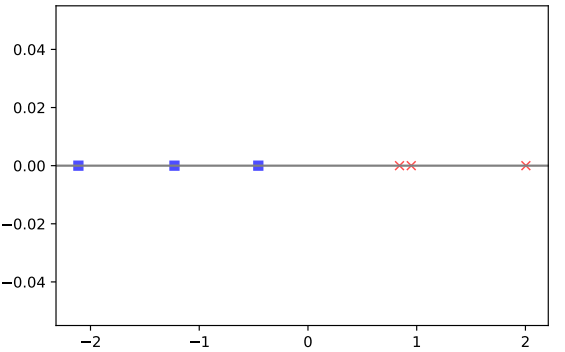


Abbildung 75: Datensatz $\hat{D}^{\text{compressed}}_{\text{ships}}$ aus Beispiel 76.

Wie im vorigen Beispiel bereits angedeutet, kann nun die Ausgabe der PCA, d. h., der komprimierte Datensatz $E^{\text{compressed}}$, für die eigentliche Lernaufgabe verwendet werden. Das daraus entstehende Modell (also beispielsweise ein Klassifikator *clf*) ist damit allerdings nur auf 1-dimensionalen Datenpunkten definiert (also $clf : \mathbb{R}^1 \to C$, wobei C die Menge der Klassen ist). Um Datenpunkte $x \in \mathbb{R}^2$ des ursprünglichen Problems mit *clf* zu klassifizieren, müssen wir für diese die entsprechende Projektion auf der von v^* aufgespannten Gerade und damit deren Koordinate bzgl. v^* bestimmen. Diese ist gegeben $z = x^T v^*$ und damit können wir via $clf(x^T v^*)$ auch Datenpunkte $x \in \mathbb{R}^2$ des ursprünglichen Problems klassifizieren.

3.5.2 Hauptkomponentenanalyse und lineare Regression

Abbildung 74 weist Ähnlichkeiten mit der Methode der linearen Regression (siehe Unterkapitel 2.1) auf und man

mag versucht sein, die Hauptkomponentenanalyse mit der linearen Regression gleichzusetzen. Gegeben eine Menge von Datenpunkten im $\mathbb{R}^2$ (oder präziser, eine Menge von Beispielen (x, y), sodass $x, y \in \mathbb{R}$ und x ist das einzige Merkmal und y der Funktionswert), versucht man bei der linearen Regression eine lineare Funktion *regr* (=Gerade) zu finden, sodass die Summe der (quadrierten) Abweichungen der vorhergesagten Funktionswerte $regr(x)$ von y minimal ist. Einfach ausgedrückt heißt dies, dass die lineare Regression die Summe der (quadrierten) „vertikalen Abstände" zu der gesuchten Gerade minimiert. PCA hingegeben minimiert die tatsächlichen (quadrierten) Abstände zu der Geraden, d. h., die kürzeste Verbindung zur Geraden (beachten Sie, dass diese Anschauung nur im 2-dimensionalen gültig ist). Die unterschiedlichen Optimierungsprobleme von PCA und linearer Regression sind in Abbildung 76 veranschaulicht. Für typische 2-dimensionale Probleme sind die Lösungen dieser beiden Optimierungsprobleme nicht identisch, oft aber auch recht ähnlich. Dennoch handelt es sich bei PCA um eine grundsätzlich andere Methode (was spätestens auch im nächsten Abschnitt 3.5.3 deutlich werden sollte) für einen anderen Zweck, als die lineare Regression. Während es bei der linearen Regression um *Funktionsapproximation* geht, geht es bei PCA um *Dimensionsreduktion*.

3.5.3 Die allgemeine Hauptkomponentenanalyse

Im Folgenden schauen wir uns die allgemeine Hauptkomponentenanalyse an, d. h., das Problem, einen Datensatz $E = \{x^{(1)}, \ldots, x^{(m)}\}$ mit $x^{(i)} \in \mathbb{R}^n$ auf einen Datensatz $E^{\text{compressed}} = \{z^{(1)}, \ldots, z^{(m)}\}$ mit $z^{(i)} \in \mathbb{R}^{n'}$ zu komprimieren, für $i = 1, \ldots, m$ und $n' < n$. Wie zuvor gehen wir davon aus, dass der Datensatz *E standardisiert* ist, d. h., dass der Mittelwert aller Merkmalsausprägungen 0 ist. In diesem

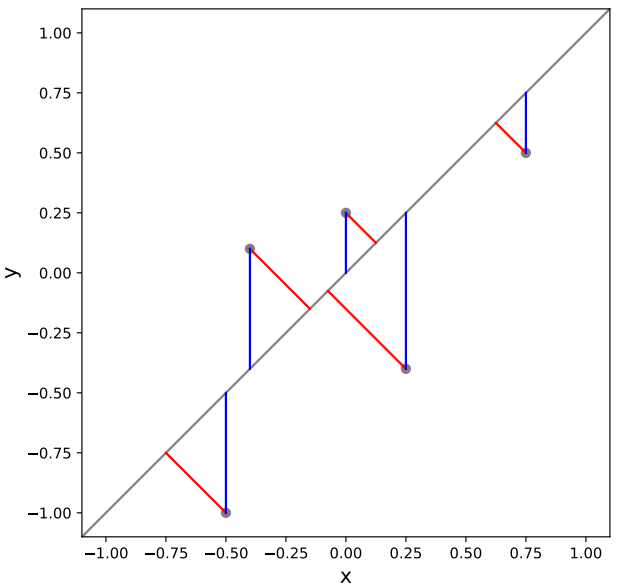


Abbildung 76: Unterschied der Optimierungsprobleme von PCA und linearer Regression; die Zielfunktion von PCA minimiert die Summe der quadrierten roten Abstände, während die Zielfunktion der linearen Regression die Summe der quadrierten blauen Abstände minimiert.

allgemeinen Fall suchen wir also einen n'-dimensionalen Unterraum von $\mathbb{R}^n$, sodass die Summe der (quadrierten) Abstände von jedem Punkt $x \in E$ zu seiner Projektion $\hat{x}$ auf diesem Unterraum minimal ist. Einen n'-dimensionalen Unterraum kann man charakterisieren durch n' paarweise orthogonale Einheitsvektoren $v_1^*, \ldots, v_{n'}^* \in \mathbb{R}^n$ (diese bilden dann eine *orthonormale Basis*, kurz ONB, dieses Unterraums). Das zugehörige Optimierungsproblem ist damit

gegeben durch

$$\begin{aligned}&(v_1^*,\ldots,v_{n'}^*)\\ &= \arg\min_{v_1,\ldots,v_{n'} \text{ ist ONB}} \sum_{x\in E} \|x-(x^Tv_1)v_1-\ldots-(x^Tv_{n'})v_{n'}\|^2\end{aligned} \tag{22}$$

wobei „$v_1,\ldots,v_{n'}$ ist ONB" heißt, dass $\|v_1\| = \ldots = \|v_{n'}\| = 1$ und $v_i^T v_j = 0$ für alle $i,j = 1,\ldots,n'$ mit $i \neq j$. Wie man leicht einsehen sollte, ist das Problem (21) ein Spezialfall des Problems (22). Ist $(v_1^*,\ldots,v_{n'}^*)$ eine Lösung von (22), so ist der gesuchte n'-dimensionale Unterraum durch die Menge aller *Linearkombinationen* von $v_1^*,\ldots,v_{n'}^*$ gegeben. Die neuen Koordinaten eines Punktes $x \in \mathbb{R}^n$ sind dann definiert durch $z \in \mathbb{R}^{n'}$ mit

$$z = (x^Tv_1^*,\ldots,x^Tv_{n'}^*)$$

Die Vektoren $v_1^*,\ldots,v_{n'}^*$ heißen dann die ersten n' Hauptkomponenten von E.[28] Eine Berechnung von $(v_1^*,\ldots,v_{n'}^*)$ durch explizite Lösung des Optimierungsproblems (22) ist recht aufwändig und in der Praxis in dieser Form unüblich. Es existiert allerdings eine elegante algebraische Charakterisierung der Hauptkomponenten durch die Eigenvektoren der *Kovarianzmatrix*.

Definition 28. Sei $E = \{x^{(1)},\ldots,x^{(m)}\}$ ein standardisierter Datensatz. Die *Kovarianzmatrix* $Cov(E) \in \mathbb{R}^{n\times n}$ von E ist definiert durch

$$Cov(E) = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}(x^{(i)})^T$$

[28] Die genaue Reihenfolge dieser Vektoren in der Hauptkomponentenanalyse, d. h., welcher dieser Vektoren die 1./2./... Hauptkomponente ist, wird durch die Größe des zugehörigen Eigenwertes bzw. absteigend durch den Grad der Varianz bestimmt, siehe dazu weiter unten und Abschnitt 3.5.4.

Die Kovarianzmatrix $Cov(E)$ ist stets symmetrisch und positiv definit[29], d. h., es existieren genau n reale Eigenwerte $\lambda_1, \ldots, \lambda_n$ (potentiell treten Eigenwerte mehrfach auf) von $Cov(E)$ mit

$$\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_n$$

Seien nun $w_1, \ldots, w_n$ passende Eigenvektoren mit $\|w_1\| = \ldots = \|w_n\| = 1$, d. h.,

$$Cov(E)w_i = \lambda_i w_i$$

für alle $i = 1, \ldots, n$. Dann lässt sich zeigen, dass genau diese Eigenvektoren die Hauptkomponenten von E sind:

Theorem 3. *$(w_1, \ldots, w_{n'})$ ist für jedes $n' < n$ Lösung von (22).*

Das obige Theorem (das wir hier nicht beweisen werden) charakterisiert somit die Hauptkomponenten von E als die (zu den zugehörigen Eigenwerten absteigend sortierten) Eigenvektoren von $Cov(E)$.

3.5.4 Evaluation

Betrachten wir noch einmal Abbildung 74 aus Beispiel 75. Es sollte hier offensichtlich sein, dass wir bei der Projektion der Datenpunkte auf die erste Hauptkomponente keine „relevanten" Informationen verlieren, d. h., für die anschließende Lernaufgabe sind alle notwendigen Informationen für die Klassifikation von Datenpunkten auf dem projizierten Unterraum noch vorhanden (beispielsweise sind, bei Verwendung von SVMs, die beiden Klassen sowohl im Ursprungsraum als auch im projiziertem Raum linear separierbar). Die Frage, die man sich bei der

[29] Dies gilt nur, wenn E den $\mathbb{R}^n$ aufspannt. Dies kann allerdings für alle praktischen Datensätze E angenommen werden.

Verwendung von PCA stellen sollte, ist, wieweit man die Dimensionalität des Ursprungsraumes reduzieren kann, ohne für die eigentliche Lernaufgabe relevante Informationen zu verlieren (oder ob man überhaupt die Dimensionalität reduzieren sollte). Die direkteste Methode, diese Frage zu beantworten, ist es, die Evaluationsmaße der eigentlichen Lernaufgabe für verschiedene Varianten der Dimensionsreduktion zu Rate zu ziehen. Ist beispielsweise die eigentliche Lernaufgabe die Klassifikation, so können wir (beispielsweise) das F1-Maß für Modelle verschiedener Dimensionalität berechnen und anschließend die Dimensionalität wählen, bei der das F1-Maß maximal bzw. noch „ausreichend hoch" ist. Dieser Ansatz ist jedoch mit einem erheblichen Aufwand verbunden, da wir zum einen Datensätze verschiedener Dimensionalität generieren müssen und zum anderen für jeden dieser Datensätze ein Modell lernen und evaluieren müssen.

Die übliche Methode, um zu bestimmen, auf welche Dimension n' wir einen Datensatz der Dimensionalität n ($n > n'$) komprimieren sollten, ist es, den Verlust an *Varianz* bei einer Dimensionsreduktion zu messen und anschließend die Dimensionalität wählen, bei der der Verlust an Varianz „nicht zu hoch" ist (hier sparen wir uns also das erneute Lernen eines Modells). Intuitiv beschreibt die Varianz eines Datensatzes E, wie weit „verstreut" die Datenpunkte E im Raum liegen. Beispielsweise sehen wir in Abbildung 74, dass die paarweisen Abstände der Punkte zueinander nicht signifikant kleiner unter der Projektion werden, der Verlust an Varianz ist hier also sehr gering. Für einen standardisierten Datensatz $E = \{x^{(1)}, \ldots, x^{(m)}\}$ ist die Varianz $var(E)$ definiert als

$$var(E) = \frac{1}{m} \sum_{i=1}^{m} ||x^{(i)}||^2$$

und ist $E^{\text{compressed}}$ eine komprimierte Variante von E, so ist

$$retVar(E^{\text{compressed}}, E) = \frac{var(E^{\text{compressed}})}{var(E)}$$

der Anteil der Varianz von E, der in $E^{\text{compressed}}$ noch erhalten bleibt (beachten Sie, dass stets $var(E^{\text{compressed}}) \leq var(E)$ gilt; *retVar* steht hier für *retained variance*). Üblicherweise wählt man n' so klein wie möglich, sodass

$$retVar(E^{\text{compressed}}, E) \geq \alpha$$

für einen Schwellwert $\alpha \in [0,1]$ gilt. Typische Werte für α liegen dabei im Bereich $[0.9, 0.99]$.

Beispiel 77. Wir setzen Beispiel 76 fort. Die Varianz des Datensatzes $\hat{D}_{\text{ships}}$ (siehe Tabelle 36), $var(\hat{D}_{\text{ships}})$, berechnet sich zu

$$\begin{aligned} var(\hat{D}_{\text{ships}}) = \frac{1}{6}(&||(0.766, 0.425)^T||^2 + ||(-0.266, -0.38)^T||^2 + \\ &||(-0.828, -0.908)^T||^2 + ||(1.211, 1.624)^T||^2 + \\ &||(0.742, 0.601)^T||^2 + ||(-1.625, -1.361)^T||^2) \\ \approx\ & 2 \end{aligned}$$

Die Varianz $var(\hat{D}^{\text{compressed}}_{\text{ships}})$ des Datensatzes $\hat{D}^{\text{compressed}}_{\text{ships}}$ (siehe Beispiel 76) berechnet sich zu

$$\begin{aligned} var(\hat{D}^{\text{compressed}}_{\text{ships}}) = \frac{1}{6}(&||(0.841)||^2 + ||(-0.457)||^2 + ||(-1.228)||^2 + \\ &||(2.004)||^2 + ||(0.95)||^2 + ||(-2.112)||^2) \\ \approx\ & 1.967 \end{aligned}$$

und somit ist

$$retVar(\hat{D}^{\text{compressed}}_{\text{ships}}, \hat{D}_{\text{ships}}) = \frac{1.967}{2} \approx 0.984$$

die bei der Komprimierung erhaltene Varianz (es werden also ungefähr 98.4% der Varianz erhalten). Dieser Wert verdeutlicht, dass der Verlust an Varianz hier prinzipiell ohne Bedeutung ist.

4 Reinforcement Learning

Überblick über dieses Kapitel

Das *Reinforcement Learning*[30] ist, neben dem überwachten und dem unüberwachten Lernen, die dritte Kategorie des maschinellen Lernens und unterscheidet sich von den beiden genannten insbesondere durch die Situierung des Lernalgorithmus. Beim *Reinforcement Learning* betrachten wir einen in einer Umgebung eingebetteten Agenten, der durch Aktionen die Umgebung ändern kann und durch erfolgreiches Handeln in der Umgebung belohnt wird. Ein anschauliches Beispiel für diese Situation ist ein *Staubsaugerroboter*, der sich in der Wohnung (=Umgebung) bewegen, den aktuellen Raum saugen, oder seine Batterie aufladen kann (falls er gerade an der Ladestation ist). Der Roboter soll selbst lernen, wie er am besten die Wohnung sauber hält und gleichzeitig sicherstellt, dass seine Batterie stets ausreichend geladen ist. Je mehr Staub der Roboter an einem Tag gesaugt hat, desto erfolgreicher schätzen wir ihn ein, dies entspricht dann der (immateriellen) *Belohnung* des Roboters. Das Ziel eines *Reinforcement Learning*-Algorithmus ist es, eine *Strategie* zur Reinigung der Wohnung zu ermitteln, sodass die Belohnung maximiert wird. Dabei unterscheidet man grundsätzlich zwischen *Offline*- und *Online*-Lernverfahren. Bei Offline-Lernverfahren (mit denen wir uns in Unterkapitel 4.1 beschäftigen werden) hat der Lernalgorithmus ein korrektes *Modell* der Umgebung, der Auswirkungen von Aktionen und den Belohnungen. Bei Online-Lernverfahren (mit denen wir uns in Unterkapiteln 4.2 und 4.3 beschäf-

[30] Wir verzichten hier auf die Nutzung des nicht gebräuchlichen deutschen Terms „Bestärkendes Lernen".

tigen werden) hat der Lernalgorithmus keinerlei Vorwissen. Um in diesen Situationen eine optimale Strategie zu ermitteln, führen *Reinforcement Learning*-Algorithmen verschiedene Aktionen in der (simulierten) Umgebung aus und lernen dabei iterativ, welche Strategien in der Umgebung erfolgversprechend sind.

Wir werden uns in diesem Kapitel zunächst mit Grundlagen zu Markov-Entscheidungsprozessen (Unterkapitel 4.1) auseinandersetzen. In Unterkapitel 4.2 schauen wir uns Methoden für das passive *Reinforcement Learning* an, d. h. Methoden, die ein Umgebungsmodell und die Güte von Aktionen lernen. In Unterkapitel 4.3 schauen wir uns Methoden für das aktive *Reinforcement Learning* an, d. h., Methoden, die eine optimale Strategie für das Handeln des Agenten in der Umgebung lernen.

Bibliographische Anmerkungen

Eine umfassende Einführung in das *Reinforcement Learning* bietet das Buch [14]. Weiterhin ist der Coursera-Kurs „Unsupervised Learning, Recommenders, Reinforcement Learning“ von Andrew Ng [12] sehr zu empfehlen. Die Darstellung hier ist angelehnt an [13], insbesondere Kapitel 17 (für Unterkapitel 4.1) und Kapitel 22 (für Unterkapitel 4.2 und 4.3). Eine alternative Darstellung bietet auch [8, Kapitel 13]. Für *Deep Reinforcement Learning* (das wir hier nicht diskutieren) siehe auch [1, Kapitel 9].

4.1 Markov Entscheidungsprozesse

In diesem Unterkapitel beschäftigen wir uns mit einer grundlegenden Komponente für das *Reinforcement Learning*, nämlich der Modellierung der Dynamik der Umgebung, in der ein Agent eingebettet ist.

4.1.1 Modellierung einer Umgebung

Für alle Varianten des *Reinforcement Learning* werden *Markov-Entscheidungsprozesse* als Mittel zur Formalisierung dynamischer Prozesse benutzt.

Definition 29. Ein *Markov-Entscheidungsprozess* (engl. *Markov decision process*, MDP) D ist ein Tupel $D = (S, A, P, R, s^0, S^t)$ mit folgenden Eigenschaften:

1. S ist eine Menge von Zuständen (der *Zustandsraum*)
2. A ist eine Menge von Aktionen (der *Aktionsraum*)
3. $P : (S \setminus S^t) \times A \times S \to [0,1]$ ist eine Funktion (die *Transitionswahrscheinlichkeitsfunktion*) mit $\sum_{s' \in S} P(s,a,s') = 1$ für alle $s \in (S \setminus S^t)$, $a \in A$.
4. $R : (S \setminus S^t) \times A \times S \to \mathbb{R}$ ist eine beliebige reellwertige Funktion (die *Belohnungsfunktion* engl. *reward function*).
5. $s^0 \in S$ ist der *Startzustand*.
6. $S^t \subseteq S$ ist die Menge der *Zielzustände*.

Ist $D = (S, A, P, R, s^0, S^t)$ ein MDP, so repräsentiert S die Menge aller möglichen Zustände, in denen sich der Agent befinden kann (im Beispiel eines Staubsaugerroboters die verschiedenen Räume der Wohnung). In jedem Zustand $s \in (S \setminus S^t)$ kann der Agent eine der Aktionen in

A ausführen (beispielsweise „saugen“, „laden“ oder sich in einen Nachbarraum bewegen). In allgemeinen MDPs ist das Resultat einer Aktion nicht immer deterministisch bestimmt (beispielsweise kann das Laden des Roboters fehlschlagen, wenn die Batterie eine Fehlfunktion hat) und $P(s,a,s')$, für $s \in (S \setminus S^t), s' \in S, a \in A$, ist dann die Wahrscheinlichkeit, in den Zustand s' zu wechseln, wenn im Zustand s die Aktion a ausgeführt wird. Die Funktion R beschreibt dann die Belohnung, die der Agent erfährt: Falls eine Ausführung der Aktion a in Zustand s zum Zustand s' geführt hat, so erhält der Agent $R(s,a,s')$ als Belohnung. Dabei kann $R(s,a,s')$ mitunter auch negativ sein und damit eine *Bestrafung* des Agenten realisieren. Der Agent startet im Zustand s^0 und *D terminiert*, sobald der Agent einen Zustand aus S^t erreicht.

Beispiel 78. Wir schauen uns eine stark vereinfachte Instanz des Problems des Staubsaugerroboters an. Wir gehen davon aus, dass unsere Wohnung aus genau zwei Räumen r_1 und r_2 besteht und jeder Raum kann entweder verschmutzt sein oder sauber. Damit ist unser Zustandsraum definiert durch

$$S_{\text{vc}} = \{s_1^{1,1}, s_2^{1,1}, s_1^{0,1}, s_2^{0,1}, s_1^{1,0}, s_2^{1,0}, s_1^{0,0}, s_2^{0,0}, s^t\}$$

wobei der Zustand $s_i^{k,l}$ die Situation repräsentiert, dass der Staubsaugerroboter in Raum r_i ist, dass Raum r_1 bei $k = 0$ sauber und bei $k = 1$ verschmutzt ist und dass der Raum r_2 bei $l = 0$ sauber und bei $l = 1$ verschmutzt ist. Der Zustand s^t ist weiterhin der Zielzustand ($S^t_{\text{vc}} = \{s^t\}$) und $s_1^{1,1}$ ist der Startzustand. In jedem Zustand (ausser s^t) stehen unserem Roboter drei Aktionen zur Verfügung:

1. *move*: Gehe in den anderen Raum.

2. *clean*: Reinige den aktuellen Raum.

3. *charge*: Lade die Batterie.

Es gilt also $A_{\text{vc}} = \{move, clean, charge\}$. Wir betrachten nun die Zustandsübergänge (modelliert durch P_{vc}). Es befindet sich nur in Raum r_1 eine Ladestation und nach Ausführung der Aktion *charge* in r_1 endet der aktuelle „Tag" und wir wechseln zum finalen Zustand s^t. Weiterhin gehen wir davon aus, dass die Aktion *move* stets zu 90 % erfolgreich ist (d. h., mit Wahrscheinlichkeit 0.1 verbleibt der Roboter im aktuellen Raum) und die Aktion *clean* ist stets zu 80 % erfolgreich (d. h., ein schmutziger Raum bleibt mit Wahrscheinlichkeit 0.2 schmutzig; ein sauberer Raum bleibt sauber). Weiterhin vergeben wir die folgenden Belohnungen (R_{vc}):

1. Reinigt der Roboter erfolgreich einen zuvor schmutzigen Raum, so erhält er 10 Punkte.

2. Reinigt der Roboter einen sauberen Raum, so erhält er -2 Punkte (da er unnötig Energie verbraucht hat).

3. Versucht der Roboter in Raum r_2 zu laden, so erhält er -5 Punkte (er hat bei dem Versuch seine Batterie beschädigt).

4. Lädt sich der Roboter in r_1 auf, ohne vorher alle Räume gereinigt zu haben, so erhält er -7 Punkte.

5. Für jede Bewegung mittels *move* (unabhängig davon, ob erfolgreich oder nicht) erhält der Roboter -1 Punkt (der Roboter soll möglichst effizient die Wohnung reinigen.

Für alle weiteren Situationen beträgt die Belohnung 0 Punkte.

Abbildung 77 zeigt ein *Zustandsübergangsdiagramm* zu $D_{\text{vc}} = (S_{\text{vc}}, A_{\text{vc}}, P_{\text{vc}}, R_{\text{vc}}, s_1^{1,1}, S_{\text{vc}}^t = \{s^t\})$, das die obige Beschreibung formalisiert.

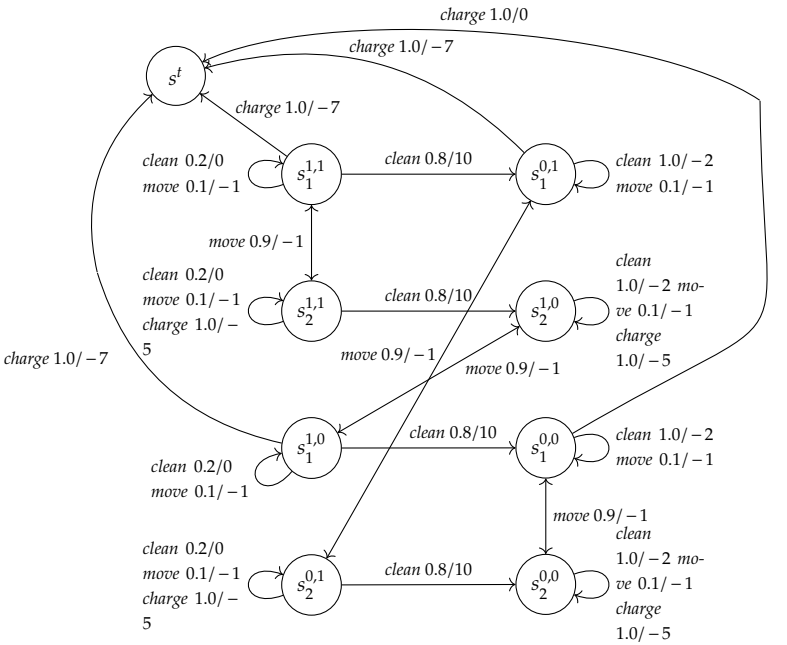


Abbildung 77: Zustandsübergangsdiagramm zu $D_{\text{vc}} = (S_{\text{vc}}, A_{\text{vc}}, P_{\text{vc}}, R_{\text{vc}}, s_1^{1,1}, S_{\text{vc}}^t = \{s^t\})$ aus Beispiel 78. Eine Kantenbeschriftung ap/r von einem Knoten s zu einem Knoten s' kodiert $P_{\text{vc}}(s,a,s') = p$ und $R_{\text{vc}}(s,a,s') = r$. Nichtgezeichnete Zustandsübergänge (z. B., von $s_1^{1,1}$ zu $s_2^{1,1}$ bei Ausführung der Aktion *clean*) haben Wahrscheinlichkeit 0.

Um den Agenten in einem MDP zu steuern, besitzt dieser eine *Strategie* (engl. *policy*), die (im einfachsten Fall) für jeden Zustand die auszuwählende Aktion bestimmt.

Definition 30. Sei $D = (S, A, P, R, s^0, S^t)$ ein MDP. Eine *Strategie* π für D ist eine Funktion $\pi : S \setminus S^t \to A$.

Eine Strategie π nach obiger Definition ist eine sog. *konstante* Strategie. Gegebenenfalls kann es sinnvoll sein,

probabilistische Strategien zu betrachten, d. h., Strategien, die zu jedem Zustand eine Wahrscheinlichkeitsverteilung über die auszuwählende Aktion vorhalten. Wir beschäftigen uns hier allerdings zunächst nur mit konstanten Strategien.

Beispiel 79. Wir führen Beispiel 78 fort. Eine mögliche Strategie für den Staubsaugerroboter ist gegeben durch π_{vc} mit

$$\pi_{\text{vc}}(s_1^{1,1}) = \pi_{\text{vc}}(s_1^{1,0}) = \pi_{\text{vc}}(s_2^{1,1}) = \pi_{\text{vc}}(s_2^{0,1}) = \mathit{clean}$$
$$\pi_{\text{vc}}(s_1^{0,1}) = \pi_{\text{vc}}(s_2^{1,0}) = \pi_{\text{vc}}(s_2^{0,0}) = \mathit{move}$$
$$\pi_{\text{vc}}(s_1^{0,0}) = \mathit{charge}$$

Mit anderen Worten, ist der Roboter in einem verschmutzten Raum, so wird stets die Aktion *clean* gewählt. Ist der aktuelle Raum sauber und der andere verschmutzt, so wird in den anderen Raum gewechselt. Sind beide Räume sauber, so wird gegebenfalls zunächst in Raum r_1 gewechselt und dort dann aufgeladen.

Die Aufgabe eines Offline-Lernverfahrens besteht darin, zu einem gegebenen fixen MDP D eine *optimale* Strategie π^* zu erlernen. Optimalität ist hierbei definiert durch die maximale akkumulierte erwartete Belohnung, die der Agent durch Nutzung der Strategie erhält. Um dies vernünftig definieren zu können, benötigen wir noch einige weitere Konzepte.

Definition 31. Sei $D = (S, A, P, R, s^0, S^t)$ ein MDP. Eine *Episode* e in D ist eine (potentiell unendliche) Folge

$$e = (s_0, a_1, s_1, a_2, s_2, \ldots)$$

mit $s_0, s_1, s_2 \ldots \in S \setminus S^t$ und $a_1, a_2, \ldots \in A$.[31] Die Wahrscheinlichkeit $P(e)$ ist definiert durch

$$P(e) = \prod_{i>0} P(s_{i-1}, a_i, s_i)$$

Eine Episode e repräsentiert einen „Lauf" durch den MDP D, wenn der Agent in s_0 nacheinander die Aktionen $a_1, a_2, \ldots$ ausführt und dabei jeweils in den Zuständen $s_1, s_2, \ldots$ landet. Eine Episode e heißt *initial*, wenn $s_0 = s^0$ gilt. Eine Episode e heißt *terminierend*, wenn $e = (s_0, a_1, s_1, a_2, s_2, \ldots, a_n, s_n)$ und $s_n \in S^t$ gilt. Der unendliche Fall tritt beispielsweise für den Roboter aus unserem Staubsaugerbeispiel auf, wenn dieser permanent nur die *move*-Aktion ausführt oder wenn eine *move*-Aktion permanent fehlschlägt (wobei für solch eine Episode die Wahrscheinlichkeit gegen 0 geht).

Jede Episode akkumuliert Belohnungen über die Zeit und die Summe aller Belohnungen nennen wir *Nutzen* der Episode (engl. *utility* oder auch *return*).

Definition 32. Sei $D = (S, A, P, R, s^0, S^t)$ ein MDP und $e = (s_0, a_1, s_1, a_2, s_2, \ldots)$ eine Episode in D. Für $\gamma \in [0,1]$ heißt $U_D^\gamma(e)$ definiert via

$$U_D^\gamma(e) = \sum_{i>0} \gamma^{i-1} R(s_{i-1}, a_i, s_i)$$

der *mit γ diskontierte Nutzen* von e in D.

Den Parameter γ in der obigen Definition nennt man auch *Discountfaktor* und wägt den Nutzen früher Belohnungen gegen den Nutzen späterer Belohnungen ab. Bei kleinen Werten von γ werden Strategien bevorzugt, die

[31] Ist $(s_0, a_1, s_1, a_2, s_2, \ldots, a_n, s_n)$ endlich, so darf s_n auch in S^t sein.

schnell hohe Belohnungen erhalten, während bei größeren Werten „nahe 1“ spätere Belohnungen auch wichtig werden. Üblicherweise wählt man für γ einen Wert echt kleiner als 1, der aber immer noch recht nahe an 1 ist (beispielsweise 0.9 oder 0.99). Für Werte echt kleiner als 1 ist es auch gewährleistet, dass $U_D^\gamma(e)$ stets endlich ist, selbst bei unendlich langen Episoden mit positiver Wahrscheinlichkeit.

Beispiel 80. Wir führen Beispiel 79 fort. Betrachten wir die folgende (initiale und terminierende) Episode e_1, definiert durch

$$e_1 = (s_1^{1,1}, clean, s_1^{0,1}, move, s_2^{0,1}, clean, s_2^{0,0}, move, s_1^{0,0}, charge, s^t)$$

Der Roboter reinigt also zunächst Raum r_1, wechselt dann in Raum r_2, reinigt diesen, wechselt wieder in Raum r_1 und lädt sich dann auf. Alle Aktionen sind hier also erfolgreich. Wir erhalten als Wahrscheinlichkeit P_{e_1} hier

$$\begin{aligned} P(e_1) &= P(s_1^{1,1}, clean, s_1^{0,1}) P(s_1^{0,1}, move, s_2^{0,1}) P(s_2^{0,1}, clean, s_2^{0,0}) \\ &\quad P(s_2^{0,0}, move, s_1^{0,0}) P(s_1^{0,0}, charge, s^t) \\ &= 0.8 \cdot 0.9 \cdot 0.8 \cdot 0.9 \cdot 1.0 \\ &= 0.5184 \end{aligned}$$

und mit $\gamma = 0.9$ als Nutzen

$$\begin{aligned} U_D^\gamma(e_1) &= R(s_1^{1,1}, clean, s_1^{0,1}) + \gamma R(s_1^{0,1}, move, s_2^{0,1}) + \\ &\quad \gamma^2 P(s_2^{0,1}, clean, s_2^{0,0}) + \gamma^3 R(s_2^{0,0}, move, s_1^{0,0}) + \\ &\quad \gamma^4 R(s_1^{0,0}, charge, s^t) \\ &= 10 + 0.9 \cdot (-1) + 0.9^2 \cdot 10 + 0.9^3 \cdot (-1) + 0.9^4 \cdot 0 \\ &= 10 - 0.9 + 8.1 - 0.729 \\ &= 16.471 \end{aligned}$$

Schauen wir uns eine weitere Episode e_2 mit

$$e_2 = (s_1^{1,1}, clean, s_1^{1,1}, clean, s_1^{0,1}, charge, s^t)$$

an. Hier versucht der Roboter zunächst vergeblich, Raum r_1 zu reinigen. Nachdem es beim zweiten Versuch geklappt hat, geht er direkt an die Ladestation. Wir erhalten

$$\begin{aligned} P(e_2) &= P(s_1^{1,1}, clean, s_1^{1,1}) P(s_1^{1,1}, clean, s_1^{0,1}) P(s_1^{0,1}, charge, s^t) \\ &= 0.2 \cdot 0.8 \cdot 1 \\ &= 0.16 \end{aligned}$$

und

$$\begin{aligned} U_D^\gamma(e_2) &= R(s_1^{1,1}, clean, s_1^{1,1}) + \gamma R(s_1^{1,1}, clean, s_1^{0,1}) + \\ &\qquad \gamma^2 R(s_1^{0,1}, charge, s^t) \\ &= 0 + 0.9 \cdot 10 + 0.9^2(-7) \\ &= 9 - 5.67 \\ &= 3.33 \end{aligned}$$

Nun können wir den Nutzen einer Strategie π als den erwarteten Nutzen aller durch π generierten Episoden definieren. Eine Episode $e = (s_0, a_1, s_1, a_2, s_2, \ldots)$ wird generiert aus π, geschrieben $\pi \sim e$, wenn $\pi(s_{i-1}) = a_i$ für alle i gilt. Ist e zusätzlich initial, so schreiben wir $\pi \sim_0 e$.

Definition 33. Sei $D = (S, A, P, R, s^0, S^t)$ ein MDP, $\gamma \in [0,1]$ und $\pi : S \setminus S^t \to A$ eine Strategie für D. Der Nutzen $U_D^\gamma(\pi)$ von π in D ist definiert durch

$$\begin{aligned} U_D^\gamma(\pi) &= \mathbb{E}_{\pi \sim_0 e}\left(U_D^\gamma(e)\right) \\ &= \sum_{\pi \sim_0 e} P(e) U_D^\gamma(e) \end{aligned}$$

Mit anderen Worten, der Nutzen $U_D^\gamma(\pi)$ von π in D ist der durchschnittliche Nutzen aller aus π generierten initialen Episoden, gewichtet nach deren Wahrscheinlichkeit.

Beispiel 81. Wir führen Beispiel 80 fort und betrachten die Strategie π_{vc} aus Beispiel 79. Beachten Sie, dass jede von π_{vc} generierte initiale Episode die folgende Struktur e^{n_1,n_2,n_3,n_4} für alle $n_1, n_2, n_3, n_4 \in \mathbb{N}^+$ mit

$$\begin{aligned} e^{n_1,n_2,n_3,n_4} = (&\underbrace{s_1^{1,1}, \mathit{clean}, \ldots, s_1^{1,1}, \mathit{clean}}_{n_1\text{-mal}}, \underbrace{s_1^{0,1}, \mathit{move}, \ldots, s_1^{0,1}, \mathit{move}}_{n_2\text{-mal}}, \\ &\underbrace{s_2^{0,1}, \mathit{clean}, \ldots, s_2^{0,1}, \mathit{clean}}_{n_3\text{-mal}}, \underbrace{s_2^{0,0}, \mathit{move}, \ldots, s_2^{0,0}, \mathit{move}}_{n_4\text{-mal}}, \\ &s_1^{0,0}, \mathit{charge}, s^t) \end{aligned}$$

hat. Die einzelnen Episoden unterscheiden sich also darin, wie oft das erste *clean*, das erste *move*, das zweite *clean* und das zweite *move* fehlschlagen. Es gilt

$$\begin{aligned} P(e^{n_1,n_2,n_3,n_4}) &= 0.2^{n_1-1} \cdot 0.8 \cdot 0.1^{n_2-1} \cdot 0.9 \cdot 0.2^{n_3-1} \cdot 0.8 \cdot \\ &\quad 0.1^{n_4-1} \cdot 0.9 \cdot 1.0 \\ &= 0.5184 \cdot 0.2^{n_1+n_3-2} \cdot 0.1^{n_2+n_4-2} \end{aligned}$$

und weiterhin

$$\begin{aligned} &U_D^\gamma(e^{n_1,n_2,n_3,n_4}) \\ &= \gamma^{n_1-1}10 + \gamma^{n_1+n_2+n_3-1}10 - \sum_{i=n_1}^{n_1+n_2-1} \gamma^i - \sum_{i=n_1+n_2+n_3}^{n_1+n_2+n_3+n_4-1} \gamma^i \end{aligned}$$

Damit folgt

$$
\begin{aligned}
U_D^{\gamma}(\pi_{\text{vc}}) &= \sum_{n_1,n_2,n_3,n_4 \in \mathbb{N}^+} P(e^{n_1,n_2,n_3,n_4}) U_D^{\gamma}(e^{n_1,n_2,n_3,n_4}) \\
&= \sum_{n_1,n_2,n_3,n_4 \in \mathbb{N}^+} \Bigg[0.5184 \cdot 0.2^{n_1+n_3-2} \cdot 0.1^{n_2+n_4-2} \\
&\qquad \left(\gamma^{n_1-1} 10 + \gamma^{n_1+n_2+n_3-1} 10 - \right. \\
&\qquad \left. \sum_{i=n_1}^{n_1+n_2-1} \gamma^i - \sum_{i=n_1+n_2+n_3}^{n_1+n_2+n_3+n_4-1} \gamma^i \right) \Bigg]
\end{aligned}
$$

Mit numerischen Methoden erhalten wir $U_D^{\gamma}(\pi_{\text{vc}}) \approx 15.53$.

Ein Strategie π^* ist nun *optimal*, wenn sie den Nutzen maximiert:

$$\pi^* = \arg\max_{\pi} U_D^{\gamma}(\pi)$$

Eine naive Methode, eine optimale Strategie zu bestimmen, besteht darin, für alle möglichen Strategien ihren Nutzen zu berechnen und eine Strategie mit maximalem Nutzen auszuwählen. Betrachten wir nur konstante Strategien, so ist deren Anzahl $|A|^{|S \setminus S^t|}$ und dies macht natürlich diese naive Methode nicht praktikabel. Im Folgenden werden wir uns zwei effektivere Verfahren zur Ermittlung einer optimalen Strategie π^*, für den Fall, dass der MDP D bekannt ist, anschauen.

4.1.2 Iterative Entwicklung der Zustandsnutzen

Das Verfahren der iterativen Entwicklung der Zustandsnutzen (engl. *value iteration*, VI) benutzt eine Charakterisierung von optimalen Strategien durch den *Nutzen von*

Zuständen. Wir haben zuvor schon den Nutzen von Episoden und Strategien definiert. Der Nutzen eines Zustands s bzgl. einer Strategie π ist definiert als der erwartete Nutzen aller in s startenden Episoden.

Definition 34. Sei $D = (S, A, P, R, s^0, S^t)$ ein MDP, $\gamma \in [0,1]$, $s \in S$, und $\pi : S \setminus S^t \to A$ eine Strategie für D. Der *Nutzen* $U_D^\gamma(s \mid \pi)$ von s bzgl. π in D ist definiert durch

$$\begin{aligned} U_D^\gamma(s \mid \pi) &= \mathbb{E}_{\pi \sim e=(s,a_1,s_1,\ldots)}\left(U_D^\gamma(e)\right) \\ &= \sum_{\pi \sim e=(s,a_1,s_1,\ldots)} P(e) U_D^\gamma(e) \end{aligned}$$

Ist π^* optimal, so schreiben wir statt $U_D^\gamma(s \mid \pi^*)$ nur $U_D^\gamma(s)$ (dies ist dann der optimale erwartete Nutzen von s).

Beispiel 82. Wir führen Beispiel 81 fort. Wir können nach den Überlegungen dort leicht sehen, dass

$$U_D^\gamma(s_1^{1,1} \mid \pi_{\text{vc}}) = U_D^\gamma(\pi_{\text{vc}}) \approx 15.529$$

gilt. Schauen wir uns den Zustand $s_2^{0,0}$ an (der Agent ist in Raum r_2 und beide Räume sind sauber). Jede von π_{vc} in $s_2^{0,0}$ startende Episode e^m (für $m \in \mathbb{N}$) hat die Form

$$e^m = (\underbrace{s_2^{0,0}, \mathit{move}, s_1^{0,0}}_{m\text{-mal}}, \mathit{charge}, s^t)$$

und damit die Wahrscheinlichkeit

$$P(e^m) = 0.9 \cdot 0.1^{m-1} \cdot 1$$

und den Nutzen

$$U_D^\gamma(e^m) = -\sum_{i=1}^{m} 0.9^{i-1}$$

Es folgt

$$U_D^{\gamma}(s_2^{0,0} \mid \pi_{\text{vc}}) = -\sum_{m=1}^{\infty}\left(0.9 \cdot 0.1^{m-1}\sum_{i=1}^{m} 0.9^{i-1}\right)$$
$$\approx -1.099$$

Ist $U_D^{\gamma}(s)$ für alle s gegeben, so kann eine optimale Strategie π^* wieder einfach durch

$$\pi^*(s) = \arg\max_{a \in A} \sum_{s' \in S} P(s,a,s')\left[R(s,a,s') + \gamma U_D^{\gamma}(s')\right] \qquad (23)$$

bestimmt werden. Mit anderen Worten, π^* wählt in jedem Zustand s die Aktion aus, die im Erwartungsfall in einen Zustand mit hohem Nutzen führt. Der VI-Ansatz versucht nun die Werte $U_D^{\gamma}(s)$ iterativ zu bestimmen, um daraus eine optimale Strategie ableiten zu können. Zentral dafür ist der folgende rekursive Zusammenhang zwischen den Zustandswerten, der aus der obigen Charakterisierung der optimalen Strategie einfach abgeleitet werden kann:

$$U_D^{\gamma}(s) = \max_{a \in A} \sum_{s' \in S} P(s,a,s')\left[R(s,a,s') + \gamma U_D^{\gamma}(s')\right] \qquad (24)$$

Mit anderen Worten, der Nutzen von s ist gleich der Summe des erwarteten Nutzens der direkten Belohnung und des erwarteten Nutzens des Folgezustands, gegeben, dass der Agent die beste Aktion ausführt. Gleichungen der Form (24) nennen wir *Bellmann*-Gleichungen und sie bilden einen wichtigen Bestandteil für die Entwicklung von Algorithmen. Die Bestimmung der Werte $U_D^{\gamma}(s)$ für alle $s \in S$ (und damit die Bestimmung einer optimalen Strategie) kann nun als Gleichungssystem mit $|S|$ Gleichungen der Form (24) und $|S|$ Unbekannten (den Werten $U_D^{\gamma}(s)$ für alle $s \in S$) dargestellt werden. Dieses Gleichungssystem ist allerdings *nicht-linear* (aufgrund des

Algorithmus 10 Der VI-Algorithmus (*value iteration*)

Eingabe: MDP $D = (S, A, P, R, s^0, S^t)$, $\gamma \in [0,1]$, $N \in \mathbb{N}$
Ausgabe: Optimale Strategie π^* für D

VI(D,γ,N)
1: $u_0(s) := 0$ für alle $s \in S$
2: $i := 0$
3: **repeat**
4: $\quad u_{i+1}(s) := \max_{a \in A} \sum_{s' \in S} P(s,a,s') \left[R(s,a,s') + \gamma u_i(s')\right]$ (für alle $s \in S$)
5: $\quad i++$
6: **until** $i > N$
7: $\pi^*(s) = \arg\max_{a \in A} \sum_{s' \in S} P(s,a,s') \left[R(s,a,s') + \gamma u_N(s')\right]$ (für alle $s \in S \setminus S^t$)
8: **return** π^*

max-Operators) und ermöglicht keine direkte analytische Lösung. Durch die Nutzung iterativer Techniken kann die Lösung jedoch beliebig genau approximiert werden. Dabei kann (24) folgendermaßen als Update-Regel benutzt werden. Definiere Startnutzen $u_0(s) := 0$ und setze für $i \geq 0$

$$u_{i+1}(s) := \max_{a \in A} \sum_{s' \in S} P(s,a,s') \left[R(s,a,s') + \gamma u_i(s')\right] \qquad (25)$$

Die Regel (25) nennt man auch *Bellmann-Update* und es kann gezeigt werden, dass die Folge der u_i gegen die Nutzen der Zustände konvergiert.

Theorem 4. *Für alle $s \in S$, $\lim_{i \to \infty} u_i(s) = U_D^\gamma(s)$.*

Der VI-Algorithmus ist in Algorithmus 10 abgebildet. Dieser Algorithmus aktualisiert iterativ die Zustandsnutzenwerte und leitet anschließend eine Strategie ab. Da es sich um ein approximatives Verfahren handelt, ist es

natürlich nicht gewährleistet, dass die berechnete Strategie tatsächlich optimal ist. Mit dem zusätzlichen Parameter N wird bei dem Algorithmus die Anzahl der Iterationen (und damit die Genauigkeit des Ergebnisses) gesteuert.[32]

Beispiel 83. Wir führen Beispiel 82 fort. Wir setzen initial

$$\begin{aligned} u_0(s_1^{1,1}) = u_0(s_2^{1,1}) = u_0(s_1^{1,0}) = u_0(s_2^{1,0}) = u_0(s_1^{0,1}) = u_0(s_2^{0,1}) \\ = u_0(s_1^{0,0}) = u_0(s_2^{0,0}) = u_0(s^t) = 0 \end{aligned}$$

und berechnen für $\gamma = 0.9$ exemplarisch (beachten Sie, dass wir nur Zustandsübergänge mit positiver Wahrscheinlichkeit in den Summen aufzählen)

$$\begin{aligned} u_1(s_1^{1,1}) = \max\{ & \\ & P(s_1^{1,1}, move, s_1^{1,1})\left[R(s_1^{1,1}, move, s_1^{1,1}) + \gamma u_0(s_1^{1,1})\right] + \\ & \quad P(s_1^{1,1}, move, s_2^{1,1})\left[R(s_1^{1,1}, move, s_2^{1,1}) + \gamma u_0(s_2^{1,1})\right], \\ & P(s_1^{1,1}, clean, s_1^{1,1})\left[R(s_1^{1,1}, clean, s_1^{1,1}) + \gamma u_0(s_1^{1,1})\right] + \\ & \quad P(s_1^{1,1}, clean, s_1^{0,1})\left[R(s_1^{1,1}, clean, s_1^{0,1}) + \gamma u_0(s_1^{0,1})\right], \\ & P(s_1^{1,1}, charge, s^t)\left[R(s_1^{1,1}, charge, s^t) + \gamma u_0(s^t)\right] \quad \} \\ = \max\{ & \\ & 0.1\,[-1 + 0.9 \cdot 0] + 0.9\,[-1 + 0.9 \cdot 0], \\ & 0.2\,[0 + 0.9 \cdot 0] + 0.8\,[10 + 0.9 \cdot 0], \\ & 1\,[-7 + 0.9 \cdot 0]\} \\ = \max\{-1, 8, -7\} = 8 & \end{aligned}$$

[32] Alternativ kann man den Algorithmus so abändern, dass bei Unterschreitung einer Fehlergrenze bei der Aktualisierung der Zustandswerte der Algorithmus abgebrochen wird.

4.1.3 Iterative Strategieentwicklung

Der VI-Ansatz aus dem vorherigen Abschnitt bestimmt die Strategie π^* erst im letzten Schritt des Algorithmus, vgl. Algorithmus 10. Dabei kann es vorkommen, dass die Nutzenwerte u_i tatsächlich schon zu genau berechnet wurden und die Strategie prinzipiell früher schon als optimale Strategie erkannt werden kann. Wir schauen uns nun einen weiteren Algorithmus, die sogenannte *iterative Strategieentwicklung* (engl. *policy iteration*, PI) an, die zusätzlich zur Berechnung der Nutzen der Zustände direkt auch die Strategie berechnet. Der Algorithmus verfährt dabei iterativ in zwei Schritten, der *Strategieevaluation* (engl. *policy evaluation*) und der *Strategieverbesserung* (engl. *policy improvement*). Ausgehend von einer beliebigen Strategie π_0 werden zunächst die Nutzen der Zustände bzgl. dieser Strategie ausgewertet. Anschließend werden diese neuen Nutzenwerte benutzt, um eine neue Strategie zu berechnen.

Für die Berechnung der Nutzenwerte der Zustände bzgl. einer Strategie π (siehe Definition 34) können wir in ähnlicher Weise wie in Gleichung (24) einen rekursiven Zusammenhang erfassen:

$$U_D^\gamma(s \mid \pi) = \sum_{s' \in S} P(s, \pi(s), s') \Big[R(s, \pi(s), s') + \gamma U_D^\gamma(s' \mid \pi) \Big] \tag{26}$$

Mit anderen Worten, der Nutzen eines Zustands bzgl. einer Strategie ist die Summe über alle direkten Belohnungen der möglichen Nachfolgezustände, die durch einmalige Anwendung der Strategie erreicht werden können, und der diskontierte Nutzen, wenn die Strategie in den Nachfolgezuständen fortgeführt wird (jeweils gewichtet nach der Wahrscheinlichkeit der jeweiligen Zustände bei Ausführung der Aktion $\pi(s)$). Die Bellmann-Gleichung

(30) beschreibt wieder ein Gleichungssystem mit $|S|$ Gleichungen und $|S|$ Unbekannten. Anders als (24) ist das durch (30) beschriebene Gleichungssystem *linear*, kann also mit Methoden der linearen Programmierung effizient gelöst werden. Bei sehr großen Zustandsräumen skaliert die exakte Berechnung von (30) allerdings wenig gut. Hier können wir auch auf eine iterative Berechnung zurückgreifen. Definiere Startnutzen $u_0(s,\pi) := 0$ und setze für $i \geq 0$

$$u_{i+1}(s,\pi) := \sum_{s' \in S} P(s,\pi(s),s') \left[R(s,\pi(s),s') + \gamma u_i(s',\pi) \right] \quad (27)$$

Es kann auch wieder gezeigt werden, dass die Folge der u_i gegen die Nutzen der Zustände konvergiert.

Theorem 5. *Für alle $s \in S$ und eine beliebige Strategie π, $\lim_{i\to\infty} u_i(s,\pi) = U_D^\gamma(s \mid \pi)$.*

Haben wir für eine Strategie π_i die Werte $U_D^\gamma(s \mid \pi_i)$ nach obigem Schema berechnet, können wir nun wie bei (33) eine neue Strategie π_{i+1} durch

$$\pi_{i+1}(s) = \arg\max_{a \in A} \sum_{s' \in S} P(s,a,s') \left[R(s,a,s') + \gamma U_D^\gamma(s' \mid \pi_i) \right] \quad (28)$$

bestimmen, für alle $a \in S \setminus S^t$. Hervorzuheben ist hier, dass die Bestimmung der Strategie π' unter Verwendung der Nutzenwerte der Zustände bzgl. einer anderen Strategie (π) erfolgt. Allerdings gilt auch hier, dass diese iterative Berechnung schlussendlich immer zur optimalen Strategie führt.

Theorem 6. *Sei π_0 eine beliebige Strategie und π_i für $i > 0$ wie in (28) definiert. Dann ist*

$$\lim_{i\to\infty} \pi_i = \pi^*$$

wohldefiniert und optimal.

Algorithmus 11 Der PI-Algorithmus (*policy iteration*)

Eingabe: MDP $D = (S, A, P, R, s^0, S^t)$, $\gamma \in [0,1]$
Ausgabe: Optimale Strategie π^* für D

PI(D,γ)
1: Setze π_0 beliebig
2: $i = 0$
3: **repeat**
4: Für alle $s \in S \setminus S^t$, bestimme $U_D^\gamma(s \mid \pi_i)$ (bspw. via (27))
5: Für alle $s \in S \setminus S^t$, $\pi_{i+1}(s) = \arg\max_{a \in A} \sum_{s' \in S} P(s,a,s')$ $[R(s,a,s') + \gamma U_D^\gamma(s' \mid \pi_i)]$
6: $i{+}{+}$
7: **until** $\pi_i = \pi_{i-1}$
8: **return** π_i

Algorithmus 11 formalisiert die obigen Ideen. Im Gegensatz zum VI-Algorithmus können wir hier leicht die Terminierung des Algorithmus bestimmen. Sobald die neu berechnete Strategie π_i gleich der zuvor berechneten Strategie π_{i-1} ist, haben wir einen Fixpunkt der Gleichungen (30) und (28) erreicht und können den Algorithmus abbrechen.

Beispiel 84. Wir führen Beispiel 83 fort und setzen die initiale Strategie π_0 via

$$\begin{aligned}\pi_0(s_1^{1,1}) = \pi_0(s_2^{1,1}) = \pi_0(s_1^{1,0}) = \pi_0(s_2^{1,0}) = \pi_0(s_1^{0,1}) = \pi_0(s_2^{0,1}) \\ = \pi_0(s_1^{0,0}) = \pi_0(s_2^{0,0}) = \mathit{move}\end{aligned}$$

Die Nutzenwerte $U_D^\gamma(s \mid \pi_0)$ für $s \in S \setminus S^t$ und $\gamma = 0.9$ berechnen sich zu

$$\begin{aligned}U_D^\gamma(s_1^{1,1} \mid \pi_0) = U_D^\gamma(s_2^{1,1} \mid \pi_0) = U_D^\gamma(s_1^{1,0} \mid \pi_0) = U_D^\gamma(s_2^{1,0} \mid \pi_0) \\ = U_D^\gamma(s_1^{0,1} \mid \pi_0) = U_D^\gamma(s_2^{0,1} \mid \pi_0) = U_D^\gamma(s_1^{0,0} \mid \pi_0) \\ = U_D^\gamma(s_2^{0,0} \mid \pi_0) \approx -10\end{aligned}$$

Nach (28) ergibt sich dadurch für die neue Strategie π_1:

$$\pi_1(s_2^{0,0}) = \pi_1(s_2^{1,0}) = \pi_1(s_1^{0,1}) = \mathit{move}$$
$$\pi_1(s_1^{0,0}) = \mathit{charge}$$
$$\pi_1(s_1^{1,1}) = \pi_1(s_1^{1,0}) = \pi_1(s_2^{1,1}) = \pi_1(s_2^{0,1}) = \mathit{clean}$$

Die neuen Nutzenwerte $U_D^\gamma(s \mid \pi_1)$ für $s \in S \setminus S^t$ berechnen sich zu

$$U_D^\gamma(s_1^{1,1} \mid \pi_1) \approx 13.513$$
$$U_D^\gamma(s_2^{1,1} \mid \pi_1) \approx 16.416$$
$$U_D^\gamma(s_1^{1,0} \mid \pi_1) \approx 9.756$$
$$U_D^\gamma(s_2^{1,0} \mid \pi_1) \approx 7.585$$
$$U_D^\gamma(s_1^{0,1} \mid \pi_1) \approx 6.726$$
$$U_D^\gamma(s_2^{0,1} \mid \pi_1) \approx 8.791$$
$$U_D^\gamma(s_1^{0,0} \mid \pi_1) \approx 0.0$$
$$U_D^\gamma(s_2^{0,0} \mid \pi_1) \approx -1.099$$

Nach (28) ergibt sich dadurch für die neue Strategie π_2:

$$\pi_2(s_2^{0,0}) = \pi_2(s_2^{1,0}) = \pi_2(s_1^{0,1}) = \mathit{move}$$
$$\pi_2(s_1^{0,0}) = \mathit{charge}$$
$$\pi_2(s_1^{1,1}) = \pi_2(s_1^{1,0}) = \pi_2(s_2^{1,1}) = \pi_2(s_2^{0,1}) = \mathit{clean}$$

π_2 wird in der nächsten Iteration nicht mehr verbessert und stellt damit die optimale Strategie dar.

4.2 Passives Reinforcement Learning

Die im vorherigen Unterkapitel diskutierten Methoden lernen eine optimale Strategie π^*, gegeben, dass ein korrektes (wenn auch probabilistisches) Modell der Umgebung in Form eines Markov-Entscheidungsprozesses zur Verfügung steht. Die Verfügbarkeit eines solchen Modells ist jedoch in den meisten realistischen Anwendungsszenarien nicht gegeben: ein Staubsaugerroboter hat üblicherweise keinen Lageplan der zu reinigenden Wohnung (zumindest zu Beginn seiner Existenz) und ein Schachspieler kein Modell der gegnerischen Strategie. Neben der Nichtverfügbarkeit eines Modells der Umgebung haben die Methoden des vorherigen Unterkapitels einen weiteren Nachteil, nämlich deren hohe Lernzeiten. Schach besitzt beispielsweise ungefähr 10^{45} Zustände (=verschiedene Stellungen auf dem Spielbrett) und eine *einzige* Iteration des SI-Algorithmus müsste bereits die Nutzenwerte aller dieser Zustände bzgl. einer initialen Strategie approximieren. Im Folgenden werden wir uns allerdings ausschließlich auf den erstgenannten Aspekt beschränken und Ansätze diskutieren, die eine optimale Strategie erlernen, ohne ein Modell der Umgebung zu nutzen. Dabei werden wir uns im aktuellen Unterkapitel auf Methoden des *passiven Reinforcement Learnings* beschränken, d. h., Methoden, die bereits eine festgelegte Strategie annehmen und ausschließlich die Nutzenwerte der Zustände erlernen (die anschließend dann auch zur Verbesserung der Strategie genutzt werden können). Im nächsten Unterkapitel schauen wir uns *aktives Reinforcement Learning* an, bei dem während des Lernens der Nutzenwerte auch direkt die optimale Strategie erlernt wird.

4.2.1 Probeläufe und Zustandsnutzen

Sei π eine Strategie. Das Ziel der folgenden Methoden ist die Bestimmung des Nutzens $U_D^\gamma(s \mid \pi)$ aller Zustände $s \in S \setminus S^t$ bzgl. π eines *unbekannten* Markov-Entscheidungsprozesses $D = (S, A, P, R, s^0, S^t)$ (und für einen gegebenen Discountfaktor γ). Nach Definition 6 aus Unterkapitel 4.1 ist $U_D^\gamma(s \mid \pi)$ definiert als

$$U_D^\gamma(s \mid \pi) = \sum_{\pi \sim e = (s, a_1, s_1, \ldots)} P(e) \sum_i \gamma^{i-1} R(s_{i-1}, a_i, s_i) \qquad (29)$$

Die Aufgabe des *passiven Reinforcement Learnings* entspricht also der Strategieevaluation aus Abschnitt 4.1.3, mit dem Unterschied, dass die Effekte von Aktionen (also insbesondere die Transitionswahrscheinlichkeiten P) zunächst nicht bekannt sind. Um $U_D^\gamma(s \mid \pi)$ zu lernen, führt der Agent eine Anzahl von „Probeläufen" in der Umgebung aus und protokolliert sowohl die besuchten Zustände als auch die erhaltenen Belohnungen.

Definition 35. Sei $\pi : S \to A$ eine Strategie bzgl. einer Menge von Zuständen S und einer Menge von Aktionen A. Eine *Beobachtung* o ist ein Tupel $o = (s, a, s', r)$ mit $s, s' \in S$, $a \in A$, $r \in \mathbb{R}$. Eine Beobachtung $o = (s, a, s', r)$ ist π-induziert, falls $a = \pi(s)$. Ein *Probelauf* O bzgl. π ist eine (endliche) Sequenz $O = (o_1, \ldots, o_n)$ mit π-induzierten Beobachtungen $o_i = (s_i, a_i, s_i', r_i)$ mit $s_i' = s_{i+1}$, für $i = 1, \ldots, n-1$. Für $\gamma \in [0, 1]$ definiere

$$rew_\gamma(O) = \sum_{i=1}^{n} \gamma^{i-1} r_i$$

Eine π-induzierte Beobachtung $o = (s, a, s', r)$ sagt, dass im Zustand s, nach Ausführung der Aktion $a = \pi(s)$ der Agent in den Zustand s' gewechselt ist und eine Belohnung r erfahren hat. Ein Probelauf O ist dann eine

Reihe von Beobachtungen unter Ausführung der Strategie π. Beachten Sie, dass ein Probelauf eine direkte Entsprechung zu einer (initialen) *Episode* hat, ein Probelauf enthält nur noch die zusätzlichen Informationen zu Belohnungen. Der Wert $rew_\gamma(O)$ ist dann die gesamte erhaltene Belohnung bzgl. des Discountfaktors γ. Die Lernaufgabe besteht nun darin, aus einer Reihe von Probeläufen $O_1, \ldots, O_m$, die Werte $U_D^\gamma(s \mid \pi)$ für alle $s \in S \setminus S^t$ zu approximieren.

Beispiel 85. Wir betrachten wieder das Beispiel des Staubsaugerroboters aus Unterkapitel 4.1. Insbesondere ist die Menge der Zustände S_{vc} gegeben durch

$$S_{\text{vc}} = \{s_1^{1,1}, s_2^{1,1}, s_1^{0,1}, s_2^{0,1}, s_1^{1,0}, s_2^{1,0}, s_1^{0,0}, s_2^{0,0}, s^t\}$$

und die Menge der Aktionen A_{vc} ist

$$A_{\text{vc}} = \{move, clean, charge\}$$

Wir betrachten weiterhin die Strategie π_{vc} mit

$$\begin{aligned}
&\pi_{\text{vc}}(s_1^{1,1}) = \pi_{\text{vc}}(s_1^{1,0}) = \pi_{\text{vc}}(s_2^{1,1}) = \pi_{\text{vc}}(s_2^{0,1}) = clean \\
&\pi_{\text{vc}}(s_1^{0,1}) = \pi_{\text{vc}}(s_2^{1,0}) = \pi_{\text{vc}}(s_2^{0,0}) = move \\
&\pi_{\text{vc}}(s_1^{0,0}) = charge
\end{aligned}$$

und die folgenden drei Probeläufe

$$\begin{aligned}
O_1 = ((&s_1^{1,1}, clean, s_1^{0,1}, 10), (s_1^{0,1}, move, s_2^{0,1}, -1), \\
&(s_2^{0,1}, clean, s_2^{0,0}, 10), (s_2^{0,0}, move, s_1^{0,0}, -1), \\
&(s_1^{0,0}, charge, s^t, 0)) \\
O_2 = ((&s_1^{1,1}, clean, s_1^{1,1}, 0), (s_1^{1,1}, clean, s_1^{0,1}, 10), \\
&(s_1^{0,1}, move, s_1^{0,1}, -1), (s_1^{0,1}, move, s_2^{0,1}, -1), \\
&(s_2^{0,1}, clean, s_2^{0,0}, 10), (s_2^{0,0}, move, s_1^{0,0}, -1), \\
&(s_1^{0,0}, charge, s^t, 0))
\end{aligned}$$

$$
\begin{aligned}
O_3 = ((&s_1^{1,1}, \mathit{clean}, s_1^{0,1}, 10), (s_1^{0,1}, \mathit{move}, s_1^{0,1}, -1),\\
&(s_1^{0,1}, \mathit{move}, s_1^{0,1}, -1), (s_1^{0,1}, \mathit{move}, s_2^{0,1}, -1),\\
&(s_2^{0,1}, \mathit{clean}, s_2^{0,0}, 10), (s_2^{0,0}, \mathit{move}, s_1^{0,0}, -1),\\
&(s_1^{0,0}, \mathit{charge}, s^t, 0))
\end{aligned}
$$

Probelauf O_1 entspricht dem erwarteten Ablauf (der Roboter reinigt zunächst Raum r_1, bewegt sich anschließend in Raum r_2, reinigt diesen, kehrt zurück in Raum r_1, und lädt sich auf). Die Probeläufe O_2 und O_3 enthalten dagegen einige fehlgeschlagene Aktionen.

Nach der Definition von $U_D^\gamma(s \mid \pi)$, siehe Gleichung (29), ist der Nutzen eines Zustands (bzgl. einer Strategie π) das gewichtete Mittel aller Episoden, die in diesem Zustand starten. Taucht ein Zustand s in einem Probelauf O auf (außer an letzter Stelle), so enthält O als Teilsequenz eine Episode, die in s startet (eventuell auch mehrere, falls s mehrmals in O besucht wurde). $U_D^\gamma(s \mid \pi)$ kann dann durch den Durchschnitt der Nutzen dieser beobachteten Teilsequenzen approximiert werden.

Definition 36. Sei O ein Probelauf mit $O = (o_1, \ldots, o_n)$, $o_i = (s_i, a_i, s_i', r_i)$ (für $i = 1, \ldots, n$) und $s \in S \setminus S^t$ ein Zustand. Definiere die Menge O_s durch

$$O_s = \{(o_k, \ldots, o_n) \mid s_k = s\}$$

Für eine (Multi-)Menge $\boldsymbol{O} = \{O^1, \ldots, O^m\}$ an Probeläufen definiere die Multimenge

$$\boldsymbol{O}_s = O_s^1 \cup \ldots \cup O_s^m$$

und

$$U_{\boldsymbol{O}}^\gamma(s \mid \pi) = \frac{\sum_{O \in \boldsymbol{O}_s} \mathit{rew}_\gamma(O)}{|\boldsymbol{O}_s|}$$

Beispiel 86. Wir führen Beispiel 85 fort und betrachten den Zustand $s_1^{0,1}$. Dieser Zustand kommt (als erste Komponente einer Beobachtung) einmal in O_1, zweimal in O_2 und dreimal in O_3 vor. Insgesamt gibt es also 6 (Teil-)Probeläufe die in $s_1^{0,1}$ starten:

$$\begin{aligned}
O_{s_1^{0,1}} = \{&((s_1^{0,1}, move, s_2^{0,1}, -1), (s_2^{0,1}, clean, s_2^{0,0}, 10),\\
&\quad (s_2^{0,0}, move, s_1^{0,0}, -1), (s_1^{0,0}, charge, s^t, 0)),\\
&((s_1^{0,1}, move, s_1^{0,1}, -1), (s_1^{0,1}, move, s_2^{0,1}, -1),\\
&\quad (s_2^{0,1}, clean, s_2^{0,0}, 10), (s_2^{0,0}, move, s_1^{0,0}, -1),\\
&\quad (s_1^{0,0}, charge, s^t, 0)),\\
&((s_1^{0,1}, move, s_2^{0,1}, -1), (s_2^{0,1}, clean, s_2^{0,0}, 10),\\
&\quad (s_2^{0,0}, move, s_1^{0,0}, -1), (s_1^{0,0}, charge, s^t, 0)),\\
&((s_1^{0,1}, move, s_1^{0,1}, -1), (s_1^{0,1}, move, s_1^{0,1}, -1),\\
&\quad (s_1^{0,1}, move, s_2^{0,1}, -1), (s_2^{0,1}, clean, s_2^{0,0}, 10),\\
&\quad (s_2^{0,0}, move, s_1^{0,0}, -1), (s_1^{0,0}, charge, s^t, 0)),\\
&((s_1^{0,1}, move, s_1^{0,1}, -1), (s_1^{0,1}, move, s_2^{0,1}, -1),\\
&\quad (s_2^{0,1}, clean, s_2^{0,0}, 10), (s_2^{0,0}, move, s_1^{0,0}, -1),\\
&\quad (s_1^{0,0}, charge, s^t, 0))\\
&((s_1^{0,1}, move, s_2^{0,1}, -1), (s_2^{0,1}, clean, s_2^{0,0}, 10),\\
&\quad (s_2^{0,0}, move, s_1^{0,0}, -1), (s_1^{0,0}, charge, s^t, 0))\}
\end{aligned}$$

Wir erhalten weiterhin (für $\gamma = 0.9$)

$$\begin{aligned}
&U_O^{\gamma}(s_1^{0,1} \mid \pi_{\text{vc}})\\
&= \frac{1}{6}(3(-1 + 0.9 \cdot 10 + 0.9^2(-1) + 0.9^3 \cdot 0) +\\
&\quad 2(-1 + 0.9(-1) + 0.9^2 \cdot 10 + 0.9^3(-1) + 0.9^4 \cdot 0) +
\end{aligned}$$

$$(-1 + 0.9(-1) + 0.9^2(-1) + 0.9^3 \cdot 10 + 0.9^4(-1) + 0.9^5 \cdot 0))$$
$$\approx \frac{1}{6}(3 \cdot 7.19 + 2 \cdot 5.471 + 3.924) \approx 6.073$$

Für eine genügend große Anzahl an Probeläufen in $\mathcal{O}$ kann gezeigt werden, dass die Werte $U^{\gamma}_{\mathcal{O}}(s \mid \pi)$ gegen die tatsächlichen Werte $U^{\gamma}_{D}(s \mid \pi)$ des generierenden Markov-Entscheidungsprozesses D konvergieren. Der obige Ansatz hat allerdings zwei grundlegende Nachteile. Zum einen können natürlich nur die Werte $U^{\gamma}_{\mathcal{O}}(s \mid \pi)$ berechnet werden, für die s in einem Probelauf vorkommt. Für ungesehene Zustände s können also keine Werte $U^{\gamma}_{\mathcal{O}}(s \mid \pi)$ approximiert werden. Zum anderen nutzt die Approximation über $U^{\gamma}_{\mathcal{O}}(s \mid \pi)$ nicht die Zusammenhänge zwischen den Zuständen aus. Der Nutzen eines Zustands s ist direkt abhängig von den Nutzen seiner direkten Nachfolgezustände und die Definition von $U^{\gamma}_{\mathcal{O}}(s \mid \pi)$ als Durchschnitt der Nutzen aller beobachteten Probeläufe berechnet diese Nutzen für jeden Zustand s unabhängig von allen anderen Zuständen. In der Praxis bedeutet dies, dass eine Berechnung von $U^{\gamma}_{D}(s \mid \pi)$ via $U^{\gamma}_{\mathcal{O}}(s \mid \pi)$ üblicherweise nur sehr langsam konvergiert. Während der erste Nachteil inhärent für alle Methoden des passiven Reinforcement Learnings ist (da wir ja eine feste Strategie annehmen und dadurch üblicherweise nicht alle möglichen Zustände sehen werden), können wir durch alternative Methoden den zweiten Nachteil umgehen. Solche Methoden schauen wir uns in den nächsten beiden Abschnitten an.

4.2.2 Adaptive dynamische Programmierung

Bei der adaptiven dynamischen Programmierung machen wir uns den rekursiven Zusammenhang zwischen den Nutzen der Zustände bzgl. der Strategie π zunutze.

Dieser rekursive Zusammenhang ist durch die *Bellmann*-Gleichung (4) aus Abschnitt 4.1.2 gegeben, die wir hier noch einmal wiederholen:

$$U_D^\gamma(s \mid \pi) = \sum_{s' \in S} P(s, \pi(s), s') \Big[R(s, \pi(s), s') + \gamma U_D^\gamma(s' \mid \pi) \Big] \tag{30}$$

Aus einer gegebenen Menge von Probeläufen $\boldsymbol{O} = \{O^1, \ldots, O^m\}$ können wir dazu zunächst die Werte $P(s, \pi(s), s')$ und $R(s, \pi(s), s')$ für alle beobachteten Zustände s und s' approximieren. Anschließend können wir das durch Gleichung (30) aufgestellte Gleichungssystem nach den unbekannten Werten $U_D^\gamma(s \mid \pi)$, beispielsweise mit den in Abschnitt 4.1.2 genannten Methoden, lösen.

Um aus einer Menge von Probeläufen $\boldsymbol{O} = \{O^1, \ldots, O^m\}$ die Wahrscheinlichkeiten $P(s, \pi(s), s')$ zu approximieren, zählen wir einfach, wie oft wir nach Ausführen der Aktion $\pi(s)$ in Zustand s den Zustand s' beobachten und nehmen die relative Häufigkeit.

Definition 37. Sei $\boldsymbol{O} = \{O^1, \ldots, O^m\}$ eine Menge von Probeläufen (bzgl. π) mit $O^j = (o_1^j, \ldots, o_{n_j}^j)$, $o_i^j = (s_i^j, \pi(s_i^j), (s')_i^j, r_i^j)$ (für $j = 1, \ldots, m$ und $i = 1, \ldots, n_j$ für passende n_j). Definiere

$$\tilde{P}_{\boldsymbol{O}}(s, \pi(s), s') = \frac{|\{o_i^j \mid s_i^j = s, (s')_i^j = s'\}|}{|\{o_i^j \mid s_i^j = s\}|}$$

Falls $|\{o_i^j \mid s_i^j = s\}| = 0$, definiere $\tilde{P}(s, \pi(s), s') = 0$.

Um $R(s, \pi(s), s')$ zu bestimmen, schauen wir in $\boldsymbol{O} = \{O^1, \ldots, O^m\}$, ob wir einen Zustandswechsel von s nach s' beobachtet haben und nehmen die beobachtete Belohnung.[33]

[33] Beachten Sie, dass wir stets fixe Belohnungen für einen Zustands-

Definition 38. Sei $O = \{O^1, \ldots, O^m\}$ eine Menge von Probeläufen (bzgl. π) mit $O^j = (o_1^j, \ldots, o_{n_j}^j)$, $o_i^j = (s_i^j, \pi(s_i^j), (s')_i^j, r_i^j)$ (für $j = 1, \ldots, m$ und $i = 1, \ldots, n_j$ für passende n_j). Definiere

$$\tilde{R}_O(s, \pi(s), s') = \begin{cases} r_i^j & \text{für beliebige } i, j \text{ mit } s_i^j = s \\ & \text{und } (s')_i^j = s' \\ 0 & \text{sonst} \end{cases}$$

Nun können wir das durch

$$\tilde{U}_O^\gamma(s \mid \pi) = \sum_{s' \in S} \tilde{P}_O(s, \pi(s), s') \left[\tilde{R}_O(s, \pi(s), s') + \gamma \tilde{U}_O^\gamma(s' \mid \pi) \right] \tag{31}$$

definierte Gleichungssystem mit den Methoden aus Abschnitt 4.1.3 lösen. Es kann weiterhin gezeigt werden, dass bei einer „genügend großen" Anzahl an Probeläufen die Werte $\tilde{U}_O^\gamma(s \mid \pi)$ gegen die tatsächlichen Werte $U_D^\gamma(s \mid \pi)$ des generierenden Markov-Entscheidungsprozesses konvergieren.

Beispiel 87. Wir führen Beispiel 86 fort. Hier war $O = \{O_1, O_2, O_3\}$ gegeben durch (siehe auch Beispiel 85):

$$\begin{aligned} O_1 = ((&s_1^{1,1}, \mathit{clean}, s_1^{0,1}, 10), (s_1^{0,1}, \mathit{move}, s_2^{0,1}, -1), \\ &(s_2^{0,1}, \mathit{clean}, s_2^{0,0}, 10), (s_2^{0,0}, \mathit{move}, s_1^{0,0}, -1), \\ &(s_1^{0,0}, \mathit{charge}, s^t, 0)) \end{aligned}$$

wechsel annehmen: falls ein Zustandswechsel von s nach s' mehrmals beobachtet wurde, so müssen alle erhaltenen Belohnungen gleich sein.

$$
\begin{aligned}
O_2 = ((&s_1^{1,1}, \mathit{clean}, s_1^{1,1}, 0), (s_1^{1,1}, \mathit{clean}, s_1^{0,1}, 10),\\
&(s_1^{0,1}, \mathit{move}, s_1^{0,1}, -1), (s_1^{0,1}, \mathit{move}, s_2^{0,1}, -1),\\
&(s_2^{0,1}, \mathit{clean}, s_2^{0,0}, 10), (s_2^{0,0}, \mathit{move}, s_1^{0,0}, -1),\\
&(s_1^{0,0}, \mathit{charge}, s^t, 0))\\
O_3 = ((&s_1^{1,1}, \mathit{clean}, s_1^{0,1}, 10), (s_1^{0,1}, \mathit{move}, s_1^{0,1}, -1),\\
&(s_1^{0,1}, \mathit{move}, s_1^{0,1}, -1), (s_1^{0,1}, \mathit{move}, s_2^{0,1}, -1),\\
&(s_2^{0,1}, \mathit{clean}, s_2^{0,0}, 10), (s_2^{0,0}, \mathit{move}, s_1^{0,0}, -1),\\
&s_1^{0,0}, \mathit{charge}, s^t, 0))
\end{aligned}
$$

In den obigen Probeläufen gibt es ingesamt vier Beobachtungen, bei denen der Agent in Zustand $s_1^{1,1}$ die Aktion $\pi_{\text{vc}} = \text{clean}$ durchführt. Bei drei von diesen vier Beobachtungen landet der Agent in Zustand $s_1^{0,1}$ und bei einer Beobachtung wieder in Zustand $s_1^{1,1}$. Wir erhalten also

$$
\begin{aligned}
\tilde{P}_O(s_1^{1,1}, \pi(s_1^{1,1}), s_1^{0,1}) &= \frac{3}{4}\\
\tilde{P}_O(s_1^{1,1}, \pi(s_1^{1,1}), s_1^{1,1}) &= \frac{1}{4}
\end{aligned}
$$

und $\tilde{P}_O(s_1^{1,1}, \pi(s_1^{1,1}), s') = 0$ für alle übrigen Zustände s'. Weiterhin erhalten wir

$$
\begin{aligned}
\tilde{R}_O(s_1^{1,1}, \pi(s_1^{1,1}), s_1^{0,1}) &= 10\\
\tilde{R}_O(s_1^{1,1}, \pi(s_1^{1,1}), s_1^{1,1}) &= 0
\end{aligned}
$$

Die Berechnung der übrigen Werte für $\tilde{P}_O$ und $\tilde{R}_O$ geschieht analog.

4.2.3 *Temporal Difference Learning*

Ein Nachteil der bisherigen Methoden zum passiven *Reinforcement Learning* ist, dass zunächst die in den Probeläufen enthaltenen Informationen extrahiert werden

und anschließend die Nutzen *aller* Zustände berechnet werden. Bei großen Zustandsräumen kann dieses Vorgehen zu sehr langen Laufzeiten führen und nicht mehr praktikabel sein. Weiterhin ist es für typische Anwendungsfälle auch nicht notwendig, den Nutzen aller Zustände (und damit die optimalen Aktionen für diese Zustände) zu bestimmen, da viele Zustände bei Ausführung einer optimalen Strategie niemals erreicht werden (beispielsweise sind beim Schach Zustände, bei denen der Gegner nur noch drei Spielfiguren hat, praktisch ausgeschlossen). Das *Temporal Difference Learning* (TD) ist eine allgemeine Methode beim *Reinforcement Learning*, die die Nutzen der besuchten Zustände *während* der Probeläufe in der Umgebung laufend aktualisiert. Viele in der Praxis gebräuchliche Algorithmen zum *Reinforcement Learning* basieren auf dieser Methode und wir werden uns die einfachste Instanz davon zur Bestimmung der Nutzenwerte $U_D^\gamma(s \mid \pi)$ bzgl. einer festen Strategie π im Folgenden anschauen.

Ausgehend von einer beliebigen Initialisierung der Nutzenwerte $u^\gamma(s \mid \pi)$ für alle $s \in S \setminus S^t$ (beispielsweise mit 0), ist die Grundidee von TD, dass bei jeder neuen Beobachtung $o = (s, \pi(s), s', r)$ der Wert $u^\gamma(s \mid \pi)$ so angepasst wird, dass er mit o kompatibel ist. Schauen wir uns dazu ein kleines Beispiel an. Angenommen wir haben für zwei Zustände s_1 und s_2 bereits Nutzenwerte $u^\gamma(s_1 \mid \pi) = 7$ und $u^\gamma(s_2 \mid \pi) = 2$ approximiert und wir erhalten eine neue Beobachtung $o = (s_1, \pi(s_1), s_2, 3)$. Falls der generierende Markov-Entscheidungsprozess *deterministisch* wäre, d. h., dass beim Ausführen der Aktion $\pi(s_1)$ in s_1 immer in den Zustand s_2 gewechselt wird (also $P(s_1, \pi(s_1), s_2) = 1$), so müsste nach Gleichung (30) die folgende Gleichung gelten:

$$u^\gamma(s_1 \mid \pi) = 3 + \gamma u^\gamma(s_2 \mid \pi)$$

Für $\gamma = 0.9$ evaluiert hier die rechte Seite der obigen Gleichung zu $3 + 0.9 \cdot 2 = 4.8$, ein Wert, der etwas kleiner als unsere aktuelle Schätzung $u^\gamma(s_1 \mid \pi) = 7$ ist. Da der Wert $u^\gamma(s_2 \mid \pi)$ auch eine Schätzung ist, sollten wir natürlich nicht $u^\gamma(s_1 \mid \pi)$ auf den Wert 4.8 setzen, aber die Beobachtung o zeigt, dass unsere aktuelle Schätzung $u^\gamma(s_1 \mid \pi) = 7$ vermutlich zu hoch ist und etwas „nach unten korrigiert" werden sollte, beispielsweise auf $u^\gamma(s_1 \mid \pi) = 6$. Die konkrete *Updateregel* des TD bei einer neuen Beobachtung $o = (s, \pi(s), s', r)$ ist gegeben durch

$$u^\gamma(s \mid \pi) := u^\gamma(s \mid \pi) + \alpha(r + \gamma u^\gamma(s' \mid \pi) - u^\gamma(s \mid \pi)) \qquad (32)$$

wobei $\alpha \in [0,1]$ der *Lernparameter* ist.

Die Regel (32) vergleicht den gerade beobachteten Nutzen $(r + \gamma u^\gamma(s' \mid \pi))$ mit dem bisher geschätzten Nutzen $u^\gamma(s \mid \pi))$ und aktualisiert ihn entsprechend in die richtige Richtung. Hierbei ist zu beobachten, dass diese Regel kein Modell der Umgebung, d. h., die Übergangswahrscheinlichkeiten zwischen den Zuständen, in die Berechnung einbezieht. Dies geschieht bei TD implizit, da bei vielen aufeinanderfolgenden Aktualisierungen von $u^\gamma(s \mid \pi)$ mittels (32) die Zielzustände s' entsprechend ihrer Verteilung unterschiedlich oft vorkommen und deswegen auch unterschiedlich stark einbezogen werden. Üblicherweise ist der Lernparameter in (32) nicht konstant, sondern wird mit steigender Anzahl von Beobachtungen geringer[34]. Ist dies der Fall, so kann wieder gezeigt werden, dass die Werte $u^\gamma(s \mid \pi)$ gegen die tatsächlichen Werte $U_D^\gamma(s \mid \pi)$ des generierenden Markov-Entscheidungsprozesses konvergieren.

[34] Beispielsweise $\alpha = 1/n$ wobei n die Anzahl der bisher gemachten Beobachtungen ist.

Beispiel 88. Wir führen Beispiel 87 fort. Initial setzen wir die geschätzten Nutzen aller Zustände auf 0:

$$u^\gamma(s_1^{1,1} \mid \pi_{\text{vc}}) = u^\gamma(s_2^{1,1} \mid \pi_{\text{vc}}) = \ldots = u^\gamma(s_2^{0,0} \mid \pi_{\text{vc}}) = 0$$

Betrachten wir nun die erste Beobachtung $(s_1^{1,1}, \mathit{clean}, s_1^{0,1}, 10)$ (dies entspricht der ersten Beobachtung aus dem Probelauf O_1 aus Beispiel 85). Nach (32) erhalten wir (für $\gamma = 0.9$ und $\alpha = 0.5$)

$$\begin{aligned} & u^\gamma(s_1^{1,1} \mid \pi_{\text{vc}}) \\ := \; & u^\gamma(s_1^{1,1} \mid \pi_{\text{vc}}) + \alpha(10 + \gamma u^\gamma(s_1^{0,1} \mid \pi_{\text{vc}}) - u^\gamma(s_1^{1,1} \mid \pi_{\text{vc}})) \\ = \; & 0 + 0.5(10 + 0.9 \cdot 0 - 0) = 5 \end{aligned}$$

Der Wert $u^\gamma(s_1^{1,1} \mid \pi_{\text{vc}})$ wird also von der ersten Schätzung 0 auf die neue Schätzung 5 aktualisiert.

4.3 Aktives Reinforcement Learning

Nachdem wir uns in den Unterkapiteln 4.1 und 4.2 mit grundlegenden Fragestellungen und Teilprobleme des *Reinforcement Learnings* beschäftigt haben, kommen wir nun zu der eigentlichen Herausforderung, nämlich des Lernens der optimalen Strategie in einer nicht-deterministischen und unbekannten Umgebung.

4.3.1 Exploration vs. Exploitation

Für die optimale Strategie π^* gilt (siehe auch Gleichung (1) aus Unterkapitel 4.1):

$$\pi^*(s) = \arg\max_{a \in A} \sum_{s' \in S} P(s,a,s') \Big[R(s,a,s') + \gamma U_D^{\gamma}(s') \Big] \qquad (33)$$

Mit anderen Worten, gegeben wir haben korrekte und vollständige Informationen zu den Nutzen aller Zustände (U_D^{γ}), zu den Belohnungen in den Zuständen (R) und den Zustandsübergangswahrscheinlichkeiten (P), so sollten wir in jedem Zustand die Aktion auswählen, die den erwarteten Nutzen maximiert. Nun haben wir uns in Unterkapitel 4.2 mit Methoden beschäftigt, die zu einer gegebenen Strategie π die Nutzen der Zustände $U_D^{\gamma}(s \mid \pi)$ berechnen und wir wissen, dass für die optimale Strategie $U_D^{\gamma}(s) = U_D^{\gamma}(s \mid \pi^*)$ gilt. Weiterhin haben wir bei dem Ansatz der adaptiven dynamischen Programmierung (siehe Abschnitt 4.2.1) auch die Werte $P(s,a,s')$ und $R(s,a,s')$ approximieren können. Auf den ersten Blick wirkt es, dass wir unter der Abschätzung von $U_D^{\gamma}(s)$ durch $U_D^{\gamma}(s \mid \pi)$ alle nötigen Komponenten haben, um durch Gleichung (33) die optimale Strategie zu ermitteln. Allerdings gibt es hierbei zwei grundlegende Probleme:

1. Mit der adaptiven dynamischen Programmierung können wir nur die Werte $P(s,a,s')$ und $R(s,a,s')$ für

$a = \pi(s)$ approximieren. Da wir eine feste Strategie angenommen haben, hat unser Agent keine Erfahrung, was andere Aktionen im Zustand s bewirken.

2. Eine Abschätzung von $U_D^\gamma(s)$ durch $U_D^\gamma(s \mid \pi)$ für eine nicht-optimale Strategie π kann sehr fehlgeleitet sein. Stellen Sie sich einen Roboter vor, der den schnellsten Weg aus einem Hochhaus finden soll. Ist π beispielsweise die Strategie, die vorschreibt in das jeweilige nächsthöhere Stockwerk zu gehen und im obersten Stockwerk den Fahrstuhl nach unten zu nehmen und dann das Gebäude zu verlassen, so hat der Zustand, bei dem sich der Agent im Erdgeschoss befindet, bzgl. π einen sehr kleinen Nutzen (schließlich ist es für π noch ein weiter Weg bis zum Ausgang). Für eine optimale Strategie π^* hätte dieser Zustand allerdings einen sehr hohen Nutzen, da wir direkt zum Ausgang gehen könnten.

Ein wichtiger Aspekt des aktiven *Reinforcement Learnings*, der auch die beiden obigen Punkte adressiert, ist die *Exploration*. Im Gegensatz zum passiven *Reinforcement Learnings* ist es wichtig, in einem Zustand verschiedenste Aktionen auszuprobieren, um zu erlernen, welche Aktion tatsächlich optimal ist. Sobald der Agent gelernt hat, welche Aktionen zu welchen Zuständen führen, kann dieses Wissen ausgenutzt werden (engl. *exploitation*), um gewinnbringend in der Umgebung zu agieren. Das *exploration vs. exploitation*-Dilemma des *Reinforcement Learnings* beschreibt hierbei, wie diese beiden Aspekte gegeneinander abgewogen werden müssen, d. h., wann kann ein Agent entscheiden, dass er genug ausprobiert hat und die bisher beste Strategie tatsächlich optimal ist. Eine anschauliche Darstellung des *exploration vs. exploitation*-Dilemmas ist durch die Betrachtung des k-armigen Banditenproblems gegeben.

Beispiel 89. Ein einarmiger Bandit ist ein Spielautomat, bei dem nach Einwerfen einer Münze (z. B. im Wert von 1 EUR) ein Gewinn, beispielsweise gleichverteilt im Intervall $[a, b]$, $a, b \in \mathbb{R}$, $a < b$ ausgegeben wird. Bei einem k-armigen Banditen wählt man nach Einwerfen der Münze einen von k Armen, die Gewinne in unterschiedlichen Intervallen ausgeben. Betrachten wir den Fall $k = 10$ und nehmen wir an, dass wir eine große Menge an Geld zur Verfügung haben, beispielsweise 1000 EUR. In diesem Fall könnten wir beispielsweise die ersten 100 Versuche nutzen, um jeden Arm jeweils 10 Mal zu betätigen und somit plausible Abschätzungen der jeweiligen Intervalle $[a_i, b_i]$, für $i = 1, \ldots, 10$, zu erhalten. Anschließend können wir die verbleibenden 900 EUR nutzen[35], um ausschließlich den Arm zu betätigen, der am gewinnbringendsten erscheint (beispielsweisen den Arm i, bei dem a_i maximal ist). Die allgemeine Frage beim k-Banditenproblem ist also, wieviel Geld wir investieren, um ein möglichst genaues Modell vom Banditen zu erhalten (*exploration*), und wieviel Geld wir investieren, um das gelernte Wissen zur Gewinnmaximierung zu nutzen (*exploitation*).

Das k-Banditenproblem ist aus Sicht des *Reinforcement Learnings* recht einfach zu modellieren: es gibt einen Startzustand und k Aktionen, die aus dem Startzustand gewählt werden können. Nach Ausführen einer Aktion enden wir direkt in einem Zielzustand und erhalten den Gewinn[36]. Es gibt also auch nur k verschiedene Strategien für dieses Problem. In realen Anwendungsszenarien ist die Anzahl der möglichen Strategien allerdings zu groß

[35] Wir ignorieren hier der Einfachheit halber die Gewinne aus den ersten 100 Versuchen.

[36] Um das vorherige Beispiel in einem endlichen Markov-Entscheidungsprozess zu modellieren, müssten wir noch annehmen, dass jeder Arm nur eine endliche Auswahl an Gewinnen erzeugt

und es ist nicht möglich, zunächst alle möglichen Strategien ausreichend auszuprobieren (wie im obigen Beispiel angedeutet) und dann die beste Strategie auszuwählen.

4.3.2 ϵ-greedy Learning

Die meisten Methoden des aktiven *Reinforcement Learnings* benutzen eine *Meta-Strategie* um das *exploration vs. exploitation*-Dilemma zu adressieren. Algorithmen dieser Art starten mit einen initialen Strategie π und nutzen diese zunächst, um die Nutzenwerte der Zustände, die mit dieser Strategie erreicht werden, zu erlernen (wie in Unterkapitel 4.2 diskutiert). Je nach Meta-Strategie wird allerdings gelegentlich von der Strategie π abgewichen und in einem Zustand s eine andere Aktion $a \neq \pi(s)$ gewählt, um neue Teile des Zustandsraums zu erkunden und die Nutzenwerte der Zustände entsprechend zu aktualisieren. Zeichnet sich in den Nutzenwerten ab, dass in einem Zustand s tatsächlich eine andere Aktion als $\pi(s)$ besser ist, so wird π entsprechend geändert. Die einfachste Instanz einer solchen Meta-Strategie ist die *ϵ-greedy*-Strategie. Hierbei ist $\epsilon \in [0,1]$ ein Parameter, der angibt, wie häufig die *exploration* der *exploitation* vorgezogen wird. Mit Wahrscheinlichkeit ϵ wird in einem Zustand s eine zufällige Aktion ausgewählt und mit Wahrscheinlichkeit $1-\epsilon$ wird $\pi(s)$ ausgeführt. Durch Nutzung einer solchen Meta-Strategie können wir beide oben genannten Probleme 1 und 2 lösen:

1. Führen wir hinreichend oft zufällige Aktionen in einem Zustand s aus, so können wir sowohl $P(s,a,s')$ als auch $R(s,a,s')$ für $a \neq \pi(s)$ approximieren.

2. Ist die Strategie π nicht optimal, so können dennoch durch eine zufällig gute Auswahl von Aktionen die

tatsächlichen Nutzen von Zuständen erkannt werden. Für das Beispiel des Hochhauses würden wir irgendwann zufällig im Erdgeschoss direkt die Aktion des Verlassens des Gebäudes ausführen und könnten damit direkt den Nutzen des Zustands im Erdgeschoss zu sein, erhöhen.

Eine Kombination der ϵ-greedy-Strategie mit der adaptiven dynamischen Programmierung (siehe Abschnitt 4.2.1) ist in Algorithmus 12 formalisiert. Der Algorithmus implementiert einen einzigen Schritt eines Agenten in einer Umgebung und erhält als Eingabe die aktuelle Beobachtung o (falls der Agent im Startzustand ist, setzen wir $o = \mathit{null}$) und gibt die als nächstes auszuführende Aktion aus. Der Algorithmus besitzt eine Reihe von statischen Datenstrukturen, die über die verschiedenen Aufrufe des Algorithmus erhalten bleiben:

- Die Strategie π ist die aktuell beste Strategie des Agenten.
- Der MDP D ist das Modell der Umgebung, das der Agent mit jedem Aufruf besser erlernt.
- Die Funktion u speichert die Nutzen der Zustände (bzgl. π).
- Die Funktionen n_1 und n_2 speichern die Häufigkeiten, wie oft ein einem Zustand s die Aktion a ausgeführt wurde ($n_1(s,a)$) bzw. wie häufig beim Ausführen der Aktion a in s der Zustand s' beobachtet wurde ($n_2(s,a,s')$).

In den Zeilen 1–5 von Algorithmus 12 werden die Informationen aus der aktuellen Beobachtung in das Modell D der Umgebung integriert. Insbesondere werden die Häufigkeiten der beobachteten Zustände (Zeilen 2 und 3)

Algorithmus 12 Adaptive dynamische Programmierung mit ϵ-greedy.

Eingabe: Beobachtung $o = (s,a,s',r)$
($o = null$ falls aktueller Zustand s^0)
Ausgabe: Aktion a
Statisch: Strategie π (initial beliebig)
MDP $D = (S,A,P,R,s^0,S^t)$
(initial $P(s,a,s') = 0$ und $R(s,a,s') = 0$
für alle $s,s' \in S$, $a \in A$)
$u : S \setminus S^t \to \mathbb{R}$
(initial $u(s) = 0$ für alle $s \in S \setminus S^t$)
$n_1 : S \times A \to \mathbb{N}$
(initial $n_1(s,a) = 0$ für alle $s \in S$, $a \in A$)
$n_2 : S \times A \times S \to \mathbb{N}$
(initial $n_2(s,a,s') = 0$ für alle $s,s' \in S$, $a \in A$)

ADPe(o)
1: **if** $o \neq null$ **then**
2: $n_1(s,a) := n_1(s,a) + 1$
3: $n_2(s,a,s') := n_2(s,a,s') + 1$
4: $P(s,a,s'') := n_2(s,a,s'')/n_1(s,a)$ für alle $s'' \in S$
5: $R(s,a,s') := r$
6: $u := \mathrm{VAL}(\pi,u,D)$
7: $\pi := \mathrm{POL}(u,D)$
8: Falls $s' \in S^t$, **return** *null*
9: Mit W'keit ϵ, **return** zufällige Aktion $a \in A$
10: Mit W'keit $1-\epsilon$, **return** $\pi(s')$ (oder $\pi(s^0)$ falls $o = null$)

dazu genutzt, die Zustandsübergangswahrscheinlichkeiten zu aktualisieren (Zeile 4). Weiterhin wird die Belohnung des Zustandsübergangs gespeichert (Zeile 5). In Zeile 6 werden die Nutzen der Zustände bzgl. der aktuellen Strategie aktualisiert. Dies geschieht analog zu den Ausführungen in Abschnitt 4.2.1 unter Nutzung der

Methoden aus Abschnitt 4.1.2 (wir kürzen dies in Algorithmus 12 einfach durch Aufruf der Unterfunktion VAL ab). In Zeile 7 wird die aktuelle Strategie angepasst (die Unterfunktion POL benutzt dazu direkt die Charakterisierung aus Gleichung (33)). Falls der aktuelle Zustand ein Zielzustand ist, gibt der Algorithmus *null* als Aktion zurück. In diesem Fall muss der Agent in der Umgebung „neu gestartet" werden, wobei die statischen Datenstrukturen nicht neu initialisiert werden. Ansonsten wird mit Wahrscheinlichkeit ϵ eine beliebige Aktion zurückgegeben oder mit Wahrscheinlichkeit $1-\epsilon$ die Aktion $\pi(s')$. Es kann wieder gezeigt werden, dass bei hinreichend vielen Aufrufen von Algorithmus 12 die Strategie π gegen die optimale Strategie konvergiert.

Beispiel 90. Wir betrachten wieder das Beispiel des Staubsaugerroboters aus den vorherigen Unterkapiteln. Insbesondere ist die Menge der Zustände S_{vc} gegeben durch

$$S_{\text{vc}} = \{s_1^{1,1}, s_2^{1,1}, s_1^{0,1}, s_2^{0,1}, s_1^{1,0}, s_2^{1,0}, s_1^{0,0}, s_2^{0,0}, s^t\}$$

wobei $s_1^{1,1}$ der Startzustand und s^t der einzige Zielzustand ist. Die Menge der Aktionen A_{vc} ist gegeben durch

$$A_{\text{vc}} = \{move, clean, charge\}$$

Nehmen wir weiterhin an, die initiale Strategie π des Roboters ist gegeben durch

$$\begin{aligned}\pi(s_1^{1,1}) = \pi(s_2^{1,1}) = \pi(s_1^{0,1}) = \pi(s_2^{0,1}) = \pi(s_1^{1,0}) \\ = \pi(s_2^{1,0}) = \pi(s_1^{0,0}) = \pi(s_2^{0,0}) = move\end{aligned}$$

Wir führen Algorithmus 12 exemplarisch zweimal aus:

1. Beim ersten Aufruf befindet sich der Agent im Startzustand $s_1^{1,1}$. Zeilen 1–5 werden hierbei nicht ausgeführt. Da noch keine Belohnungen beobachtet

wurden, gilt initial $R(s,a,s') = 0$ für alle $s,s' \in S, a \in A$. Es folgt, dass in Zeile 6, $u(s) = 0$ für alle $s \in S$ errechnet wird. In Zeile 7 ergibt sich, dass alle Strategien gleich gut/schlecht sind, wir ändern die aktuelle Strategie also nicht ab. Da wir uns nicht in einem Zielzustand befinden und wir annehmen, dass der Zufallstest in Zeilen 9/10 zugunsten von Zeile 10 ausgefallen ist, geben wir $\pi(s_1^{1,1}) = \mathit{move}$ zurück.

2. Beim zweiten Aufruf erhält der Algorithmus die Beobachtung

$$o = (s_1^{1,1}, \mathit{move}, s_2^{1,1}, -1)$$

Mit anderen Worten, der Roboter ist erfolgreich in Raum r_2 gewechselt. Wir aktualisieren in den Zeilen 2–5 die Werte unserer Datenstrukturen wie folgt:

$$\begin{aligned}
n_1(s_1^{1,1}, \mathit{move}) &:= n_1(s_1^{1,1}, \mathit{move}) + 1 = 0 + 1 = 1 \\
n_2(s_1^{1,1}, \mathit{move}, s_2^{1,1}) &:= n_2(s_1^{1,1}, \mathit{move}, s_2^{1,1}) + 1 = 0 + 1 = 1 \\
P(s_1^{1,1}, \mathit{move}, s_2^{1,1}) &:= 1 \\
P(s_1^{1,1}, \mathit{move}, s'') &:= 0 \text{ für alle } s'' \neq s_2^{1,1}) \\
R(s_1^{1,1}, \mathit{move}, s_2^{1,1}) &:= -1
\end{aligned}$$

In Zeile 6 ergibt sich

$$u(s) := 0 \qquad \text{für alle } s$$

Insbesondere ändert sich also (zunächst) nicht der Wert von $u(s_1^{1,1})$ und bleibt 0, da er in Zeile 6 über das Maximum aller Aktionen berechnet wird und nur der Nutzen bei Ausführen der Aktion *move* auf -1 gesetzt wurde.

In Zeile 7 ergibt sich, dass für den Zustand $s_1^{1,1}$ jede Aktion außer *move* einen maximal erwarteten Nutzen von 0 bringt. Wir setzen zufällig $\pi(s_1^{1,1}) = \mathit{clean}$, für alle anderen Zustände behalten wir die Strategie. In Zeilen 9/10 wählen wir eine zufällige Aktion *charge* anstelle von $\pi(s_2^{1,1})$.

Ein Nachteil von Algorithmus 12 ist, dass in Zeile 6 die Nutzen aller Zustände (potentiell) neu berechnet werden und anschließend die Strategie aktualisiert wird. Dies ist begründet in der Nutzung der adaptiven dynamischen Programmierung, bei der wir die gleiche Kritik schon in Abschnitt 4.2.2 benutzt haben, um den Ansatz des *Temporal Difference Learning* (TD) zu motivieren. In gleicher Weise können wir allerdings auch den TD-Ansatz für das aktive *Reinforcement Learning* erweitern. Dies werden wir im nächsten Abschnitt tun.

4.3.3 Q-Learning

Q-Learning ist ein TD-Ansatz für das aktive *Reinforcement Learning*, der den meisten modernen Ansätzen zum *Reinforcement Learning* unterliegt. Im Gegensatz zu den bisherigen Ansätzen, lernt das Q-Learning nicht die Nutzen (U) der Zustände direkt, sondern die sogenannte Q-Funktion. Hier ist Q^γ die Funktion $Q^\gamma : A \times (S \setminus S^t) \to \mathbb{R}$, die einem Zustand $s \in S \setminus S^t$ und einer Aktion a den erwarteten maximalen Nutzen zuweist, den man bei Verfolgung der optimalen Strategie nach Ausführen der Aktion a in s erhält. Der Zusammenhang zwischen Q^γ und den Nutzen der Zustände ist damit gegeben durch

$$U_D^\gamma(s) = \max_{a \in A} Q^\gamma(a, s)$$

für alle Zustände $s \in S \setminus S^t$. Sind die wahren Q^γ-Werte eines jeden Zustands gegeben, so kann die optimale Stra-

tegie π^* leicht bestimmt werden durch

$$\pi^*(s) = \arg\max_{a \in A} Q^\gamma(a,s) \tag{34}$$

Der Vorteil der Nutzung der Q^γ-Funktion anstelle der Nutzenwerte ist, dass wir zur Bestimmung der optimalen Strategie kein Modell der Umgebung (insbesondere der Übergangswahrscheinlichkeiten P) erlernen müssen, vgl. Gleichungen (33) und (34). Aus diesem Grund nennt man das Q-Learning auch einen *modellfreien* Ansatz (engl. *model free method*).

Um die Q^γ-Funktion zu lernen, verfolgen wir den TD-Ansatz und aktualisieren bei jeder Beobachtung $o = (s,a,s',r)$ unsere aktuelle Schätzung $q^\gamma(a,s)$. Dies geschieht in analoger Weise zum normalen TD-Ansatz (siehe Abschnitt 4.2.2, Gleichung (2)) durch eine *Updateregel*, die für das Q-Learning definiert ist durch

$$q^\gamma(a,s) := q^\gamma(a,s) + \alpha(r + \gamma \max_{a' \in A} q^\gamma(a',s') - q^\gamma(a,s))$$

wobei $\alpha \in [0,1]$ der *Lernparameter* ist. Hierbei ist

$$r + \gamma \max_{a' \in A} q^\gamma(a',s')$$

die Schätzung des Q-Wertes aus der gerade getätigten Beobachtung, von dem die aktuelle Schätzung $q^\gamma(a,s)$ abgezogen wird und das Ergebnis wieder genutzt wird, um $q^\gamma(a,s)$ in die entsprechende Richtung zu korrigieren. In der Praxis ist α nicht konstant, sondern wird mit steigender Anzahl von Beobachtungen geringer.

Um das *exploration vs. exploitation*-Dilemma mit Q-Learning zu lösen, muss der oben beschriebene Ansatz wieder mit eine Meta-Strategie zur Exploration kombiniert werden. Eine Kombination der ϵ-greedy-Strategie mit Q-Learning ist in Algorithmus 13 formalisiert. Un-

Algorithmus 13 Q-Learning mit ϵ-greedy.

Eingabe:	Beobachtung $o = (s, a, s', r)$
	($o = null$ falls aktueller Zustand s^0)
Ausgabe:	Aktion a
Statisch:	Strategie π (initial beliebig)
	$q : (A \times S \setminus S^t) \to \mathbb{R}$ (initial $q(a,s) = 0$
	für alle $s \in S \setminus S^t$, $a \in A$)

Qe(o)

1: **if** $o \neq null$ **then**
2: $\quad q^\gamma(a,s) := q^\gamma(a,s) + \alpha(r + \gamma \max_{a' \in A} q^\gamma(a',s') - q^\gamma(a,s))$
3: $\quad \pi(s) := \arg\max_{a \in A} q(a,s)$
4: Falls $s' \in S^t$, **return** $null$
5: Mit W'keit ϵ, **return** zufällige Aktion $a \in A$
6: Mit W'keit $1 - \epsilon$, **return** $\pi(s')$ (oder $\pi(s^0)$ falls $o = null$)

ter wenigen formalen Annahmen kann wieder gezeigt werden, dass bei hinreichend vielen Aufrufen von Algorithmus 13 die Strategie π gegen die optimale Strategie konvergiert. Wie man im direkten Vergleich von Algorithmus 12 und Algorithmus 13 sehen kann, ist Q-Learning weitaus simpler und benötigt weniger Speicherplatz. Ein Nachteil des Q-Learnings gegenüber des Ansatzes der adaptiven dynamischen Programmierung ist jedoch, dass ersteres weitaus langsamer gegen die optimale Strategie konvergiert.

Beispiel 91. Wir führen Beispiel 90 fort. Wir nehmen dabei an, dass $\gamma = 0.9$, $\alpha = 0.1$ und die initiale Strategie π des Roboters gegeben ist durch

$$\begin{aligned}\pi(s_1^{1,1}) = \pi(s_2^{1,1}) = \pi(s_1^{0,1}) = \pi(s_2^{0,1}) = \pi(s_1^{1,0}) \\ = \pi(s_2^{1,0}) = \pi(s_1^{0,0}) = \pi(s_2^{0,0}) = \mathit{move}\end{aligned}$$

Wir führen Algorithmus 13 exemplarisch zweimal aus:

1. Beim ersten Aufruf befindet sich der Agent im Startzustand $s_1^{1,1}$. Zeilen 1–3 werden hierbei nicht ausgeführt. Da wir uns nicht in einem Zielzustand befinden und wir annehmen, dass der Zufallstest in den Zeilen 5/6 zugunsten von Zeile 6 ausgefallen ist, geben wir $\pi(s_1^{1,1}) = \mathit{move}$ zurück.

2. Beim zweiten Aufruf erhält der Algorithmus die Beobachtung

$$o = (s_1^{1,1}, \mathit{move}, s_2^{1,1}, -1)$$

Mit anderen Worten, der Roboter ist erfolgreich in Raum r_2 gewechselt. Wir aktualisieren in den Zeilen 2 und 3 die Werte $q^\gamma(\mathit{move}, s_1^{1,1})$ und $\pi(s_1^{1,1})$ wie folgt:

$$\begin{aligned} q^\gamma(\mathit{move}, s_1^{1,1}) &:= q^\gamma(\mathit{move}, s_1^{1,1}) + \alpha(r + \\ &\qquad \gamma \max\{q^\gamma(\mathit{move}, s_2^{1,1}), q^\gamma(\mathit{clean}, s_2^{1,1}), \\ &\qquad q^\gamma(\mathit{charge}, s_2^{1,1})\} - q^\gamma(\mathit{move}, s_1^{1,1})) \\ &= 0 + 0.1(-1 + 0.9 \cdot 0 - 0) = -0.1 \\ \pi(s_1^{1,1}) &= \mathit{clean} \end{aligned}$$

Beachten Sie, dass für Bestimmung von $\pi(s_1^{1,1})$ oben gilt

$$\begin{aligned} q^\gamma(\mathit{move}, s_1^{1,1}) &= -0.1 \\ q^\gamma(\mathit{clean}, s_1^{1,1}) &= 0 \\ q^\gamma(\mathit{charge}, s_1^{1,1}) &= 0 \end{aligned}$$

Die Wahl $\pi(s_1^{1,1}) = \mathit{clean}$ wurde also zufällig aus den beiden maximalen Werten für *clean* und *charge* getroffen. In den Zeilen 9/10 wählen wir eine zufällige Aktion *charge* anstelle von $\pi(s_2^{1,1})$.

5 Deep Learning

Überblick über dieses Kapitel

Deep Learning bezeichnet das maschinelle Lernen mit tiefen neuronalen Netzwerken. In jüngster Zeit haben sich diese Netzwerke als das bevorzugte Modell für eine Reihe von Problemen etabliert, sowohl für das überwachte Lernen, das unüberwachte Lernen, als auch das *Reinforcement Learning*. Auch wenn die ersten künstlichen neuronalen Netzwerke schon in den 1940/50er Jahren entwickelt wurden, begann die eigentliche *Deep Learning Revolution* erst in 2012. Insbesondere bedingt durch neue Hardware-Möglichkeiten, Parallelverarbeitung, und die Verfügbarkeit von großen Mengen an Trainingsdaten, hat sich das Potential künstlicher neuronaler Netzwerke gezeigt. Heutzutage lösen neuronale Netzwerke immer mehr die klassischen Modelle (aus den Kapiteln 2-4) für praxisrelevante Aufgaben ab, insbesondere wenn die „Erklärbarkeit" der Vorhersagen eine untergeordnete Rolle spielt.

Wir werden uns in diesem Kapitel zunächst mit (künstlichen) neuronalen Netzwerken in ihrer Allgemeinheit beschäftigen (Unterkapitel 5.1). In Unterkapitel 5.2 diskutieren wir *Convolutional Neural Networks*, eine besondere Architektur für neuronale Netzwerke, die sich insbesondere für die Bildverarbeitung eignet. In Unterkapitel 5.3 schauen wir uns *Recurrent Neural Networks* an, die sich insbesondere für die Analyse von geschriebener oder gesprochener Sprache eignen. In Unterkapitel 5.4 schauen wir uns abschließend einige weitere Modelle für das Lernen von Repräsentationen an.

Bibliographische Anmerkungen

Das Buch [5] bietet einen breiten Einstieg in das *Deep Learning*. Weiterhin enthalten auch die zuvor genannten Coursera-Kurse „Supervised Machine Learning: Regression and Classification“ [11] und „Unsupervised Learning, Recommenders, Reinforcement Learning“ [12] von Andrew Ng Inhalte zu *Deep Learning*. Eine weitere allgemeine Quelle ist auch [1]. Einen kurzen Überblick bietet [13, Kapitel 21].

5.1 Künstliche Neuronale Netze

Ein *künstliches neuronales Netzwerk* (engl. *artificial neural network*, ANN) ist ein sehr allgemeines Modell für diverse Aufgaben des maschinellen Lernens. Es ist in seiner Funktionsweise angelehnt an biologische neuronale Netzwerke und besteht aus einer Menge von *(künstlichen) Neuronen*, die jeweils eine einfache mathematische Operation ausführen und in einer Netzwerkstruktur angeordnet sind. Jedes Neuron nimmt dazu eine Menge von Eingaben (reelle Zahlen) entgegen, berechnet mithilfe einer Reihe von Gewichten eine Funktion auf diesen Eingaben und gibt das Ergebnis anschließend an nachfolgende Neuronen weiter bzw. als Gesamtergebnis des Netzwerks aus. Auch wenn das Grundprinzip dieser Verarbeitung von biologischen neuronalen Netzwerken inspiriert ist, so ist die moderne Forschung zu ANNs etwas losgelöst von der Forschung in Biologie und Kognitionswissenschaften und wir sehen hier ANNs auch ausschließlich als eine sehr flexible Datenstruktur zur Repräsentation von Modellen für das maschinelle Lernen. Wir werden daher im Folgenden auch öfters nur den Begriff *neuronales Netzwerk* verwenden, da wir uns ausschließlich mit der künstlichen Variante beschäftigen werden. ANNs können für überwachtes und unüberwachtes Lernen, sowie für das Reinforcement Learning benutzt werden, wir werden uns aber zunächst hauptsächlich mit der Anwendung für das überwachte Lernen beschäftigen.

5.1.1 Neuronen

Die Grundeinheit eines ANNs ist das *Neuron*, das in seiner allgemeinsten Form wie folgt definiert ist.

Definition 39. Sei $n \in \mathbb{N}$. Ein *Neuron* r ist ein Tupel $r =$

(w, act) mit $w \in \mathbb{R}^{n+1}$ und $act : \mathbb{R} \to \mathbb{R}$.

Der Parameter n ist die Länge der Eingabe für das Neuron $r = (w, act)$. Der Vektor $w = (w_0, w_1, \ldots, w_n)^T \in \mathbb{R}^{n+1}$ sind die Parameter von r und act ist die *Aktivierungsfunktion*. Informell betrachtet steuert die Aktivierungsfunktion, wann die Eingabe eines Neurons ausreicht, dieses zu aktivieren, d. h., einen relativ „hohen" Wert als Ausgabe zu produzieren. Ein einfaches Beispiel für eine Aktivierungsfunktion ist die Schwellwertfunktion h^{thresh}, definiert durch

$$h^{\text{thresh}}(x) = \begin{cases} 1 & \text{falls } x > 0 \\ 0 & \text{sonst} \end{cases}$$

In der Praxis ist h^{thresh} keine sehr gute Aktivierungsfunktion, aber an dieser Stelle genügt es, sich unter dem Begriff der Aktivierungsfunktion die Schwellwertfunktion vorzustellen. Wir werden uns mit dem Thema der Aktivierungsfunktionen genauer in Abschnitt 5.1.3 beschäftigen. Ist $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$ die Eingabe zu r, so berechnet r den *Aktivierungswert* a_r definiert durch

$$z_r = w_0 + w_1 x_1 + \ldots + w_n x_n$$
$$a_r = act(z_r)$$

Der Zwischenwert z_r heisst auch *linearer Anteil* von r. Gilt $r = (w, act)$, so schreiben wir auch $r_{w,act}(x) = act(w_0 + w_1 x_1 + \ldots + w_n x_n)$ für die gesamte durch r dargestellte Funktion. Eine schematische Darstellung eines Neurons ist in Abbildung 78 zu finden. Um die Notation zu vereinfachen, gehen wir üblicherweise davon aus, dass jedes Neuron neben der Eingabe $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$ noch einen zusätzlichen konstanten Eingabewert $x_0 = 1$ enthält. Ist $x = (x_0, x_1, \ldots, x_n) \in \mathbb{R}^{n+1}$ (mit $x_0 = 1$), so gilt einfach

$$z_r = w^T x = w_0 x_0 + w_1 x_1 + \ldots + w_n x_n$$
$$a_r = act(z_r)$$

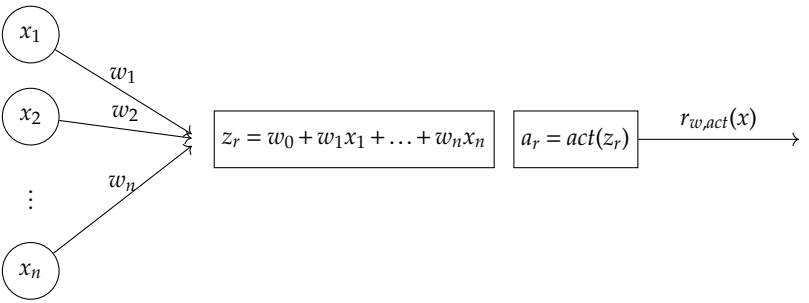


Abbildung 78: Schematische Darstellung eines (künstlichen) Neurons $r = (w, act)$.

Die zusätzliche Eingabe x_0 nennen wir *Bias-Eingabe* und eine schematische Darstellung eines um eine Bias-Eingabe erweiterten Neurons ist in Abbildung 79 gezeigt. Beide Darstellungen von Neuronen sind in der Literatur üblich, die letzte Darstellung macht allerdings einige Berechnungen und Beschreibungen von Algorithmen eleganter und wird normalerweise bevorzugt.

Ein einzelnes Neuron $r = (w, act)$ kann bereits als einfaches Modell für das maschinelle Lernen benutzt werden. Sei $D = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ ein Trainingsdatensatz für ein überwachtes Lernproblem (ob für Klassifikation oder Regression ist zunächst unerheblich). Sei weiterhin L eine *Kostenfunktion* für das Lernproblem, d. h., $L(D, f)$ gibt an, wie gut eine Funktion $f : \mathbb{R}^n \to \mathbb{R}$ die Daten in D repräsentiert. Wir sind daran interessiert, Parameter $\hat{w}$ zu finden, sodass $L(D, r_{\hat{w},act})$ minimal ist. Dann verallgemeinert die durch ein Neuron r dargestellte Funktion $r_{w,act}$ bereits einige uns bekannte Modelle des maschinellen Lernens:

1. Ist $act = h^{\mathrm{id}}$ die *Identitätsfunktion* mit $h^{\mathrm{id}}(x) = x$ für

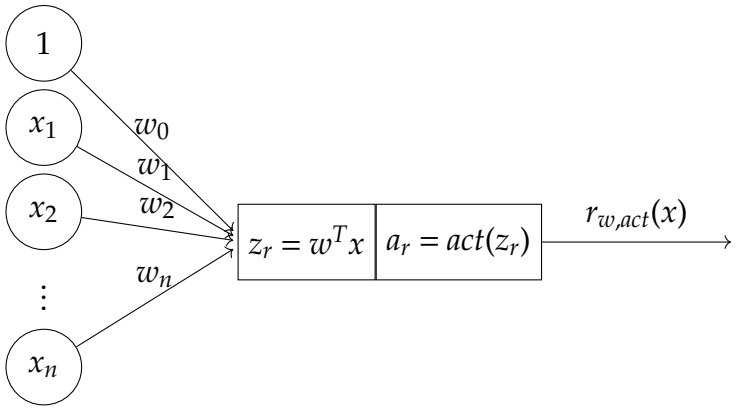


Abbildung 79: Schematische Darstellung eines (künstlichen) Neurons $r = (w, act)$ mit zusätzlicher Bias-Eingabe.

alle $x \in \mathbb{R}$ und ist $L = L^{\text{qF}}$ der *quadratische Fehler*

$$L^{\text{qF}}(D, f) = \sum_{i=1}^{m} (f(x^{(i)}) - y^{(i)})^2$$

so ist $r_{\hat{w},act}$ äquivalent zu der optimal angepassten linearen Funktion (mit Parametern $w_0, \ldots, w_n$) für die lineare Regression (siehe Unterkapitel 2.1).

2. Ist $act = h^{\text{logit}}$ die *Sigmoid-Funktion*[37]

$$h^{\text{logit}}(x) = \frac{1}{1 + e^{-x}}$$

[37] Beachten Sie, dass die folgende Definition der Sigmoid-Funktion, im Gegensatz zu der Definition aus Unterkapitel 2.2, nur den nicht-linearen Anteil darstellt.

und $L = L^{\text{logit}}$ die logistische Kostenfunktion

$$L^{\text{logit}}(D, f) = -\sum_{i=1}^{m} y^{(i)} \log f(x^{(i)}) + (1 - y^{(i)}) \log(1 - f(x^{(i)}))$$

so ist $r_{\hat{w},act}$ äquivalent zu dem optimalen Modell für die logistische Regression aus Unterkapitel 2.2.

In ähnlicher Weise kann auch eine *Support Vector Machine* als einzelnes Neuron dargestellt werden, wir überlassen es hier der Leserin/dem Leser, die konkrete Aktivierungs- und Kostenfunktion aufzustellen.

Der Umstand, dass bereits ein einzelnes Neuron eine Reihe klassischer Modelle des maschinellen Lernens verallgemeinert, machen neuronale Netzwerke zu einer attraktiven Wahl für komplexere Lernaufgaben. Durch Aneinanderreihung mehrerer Neuronen, wird die Ausdrucksstärke signifikant erhöht. Im nächsten Abschnitt werden wir uns deshalb mit der allgemeinen Architektur von neuronalen Netzwerken beschäftigen.

5.1.2 Neuronale Architekturen

Wie bereits zuvor gesehen, kann ein einzelnes Neuron bereits eine Reihe von klassischen Modellen des maschinellen Lernens darstellen. Eine einfache Architektur eines neuronalen Netzwerks mit einem einzelnen Neuron wurde bereits in der 1940/50er Jahren unter dem Namen *Perzeptron* entwickelt. Das Perzeptron benutzt als Aktivierungsfunktion die schon angesprochene Schwellwertfunktion h^{thresh}. In den ursprünglichen Arbeiten zum Perzeptron wurde die Kostenfunktion nicht explizit beschrieben (der Lernalgorithmus wurde dort direkt in prozeduraler Form angegeben), wir nehmen hier der Einfachheit halber wieder den quadratischen Fehler L^{qF}. Wir

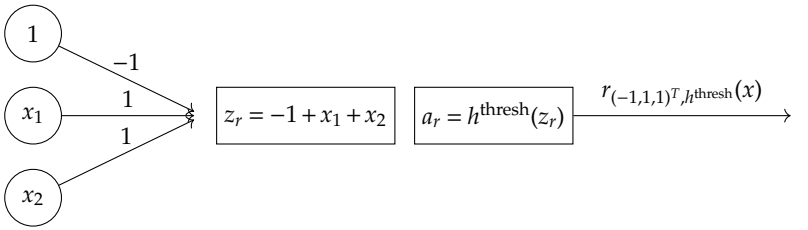


Abbildung 80: Perzeptron zur Modellierung der AND-Funktion.

betrachten bei dieser vereinfachten Form des Perzeptrons also Funktionen der Form $r_{w,h^{\text{thresh}}}$ und suchen Parameter w, so dass $L^{\text{qF}}(D, r_{w,h^{\text{thresh}}})$ bzgl. eines Trainingsdatensatzes E minimal ist. Das Problem mit dem Perzeptron, das zur Folge hatte, dass die Forschung zu neuronalen Netzen lange Zeit in Vergessenheit geraten ist, ist die Unzulänglichkeit dieses Modells für die Darstellung der relativ simplen XOR-Funktion.

Beispiel 92. Wir betrachten zunächst das Problem der Repräsentation der AND-Funktion, d. h., der Bool'schen Funktion $\text{AND} : \{0,1\} \times \{0,1\} \to \{0,1\}$ definiert durch

$$\text{AND}(0,0) = \text{AND}(0,1) = \text{AND}(1,0) = 0$$
$$\text{AND}(1,1) = 1$$

Die AND-Funktion kann leicht mit einem Perzeptron implementiert werden, insbesondere gilt

$$\text{AND} = r_{(-1,1,1)^T, h^{\text{thresh}}},$$

siehe Abbildung 80. In ähnlicher Weise kann die OR-Funktion durch $r_{(0,1,1)^T, h^{\text{thresh}}}$ und die NEG-Funktion durch $r_{(1,-1)^T, h^{\text{thresh}}}$ dargestellt werden. Betrachten wir nun die

XOR-Funktion definiert durch

$$\begin{aligned}\mathrm{XOR}(0,0) &= \mathrm{XOR}(1,1) = 0\\ \mathrm{XOR}(0,1) &= \mathrm{XOR}(1,0) = 1\end{aligned}$$

Die XOR-Funktion kann nicht durch ein Perzeptron (oder jedes andere einzelne Neuron) dargestellt werden. Die Leserin und der Leser ist eingeladen, sich von diesem Umstand selbst zu überzeugen.

Da einzelne Perzeptronen die Bool'schen Operatoren AND, OR und NEG repräsentieren können und es leicht zu erkennen ist, dass

$$\mathrm{XOR}(x_1, x_2) = \mathrm{AND}(\mathrm{OR}(x_1, x_2), \mathrm{NEG}(\mathrm{AND}(x_1, x_2))$$

gilt, liegt es nahe, komplexe Funktionen wie XOR durch Aneinanderreihung von Neuronen der Basisfunktionen zu realisieren. Mit den obigen Erkenntnissen folgt auch direkt, dass

$$\begin{aligned}\mathrm{XOR}(x_1, x_2) = r_{(-1,1,1)^T, h^{\mathrm{thresh}}}(&r_{(0,1,1)^T, h^{\mathrm{thresh}}}(x_1, x_2),\\ &r_{(1,-1)^T, h^{\mathrm{thresh}}}(r_{(-1,1,1)^T, h^{\mathrm{thresh}}}(x_1, x_2))\end{aligned}$$

gilt, eine entsprechende schematische Darstellung des neuronalen Netzwerks ist in Abbildung 81 zu finden.

Üblicherweise arrangiert man ein neuronales Netzwerk in *Schichten*, wobei jede Schicht als Eingabe die Ausgaben der vorangegangen Schichten (oder der Eingabe) enthält und die Ausgabe in die nächste Schicht (oder die Ausgabe) weiterleitet.[38] Ein solches Netzwerk nennt man *Feedforward*-Netzwerk und dessen allgemeine Struktur für eine Regressions- oder binäre Klassifikationsauf-

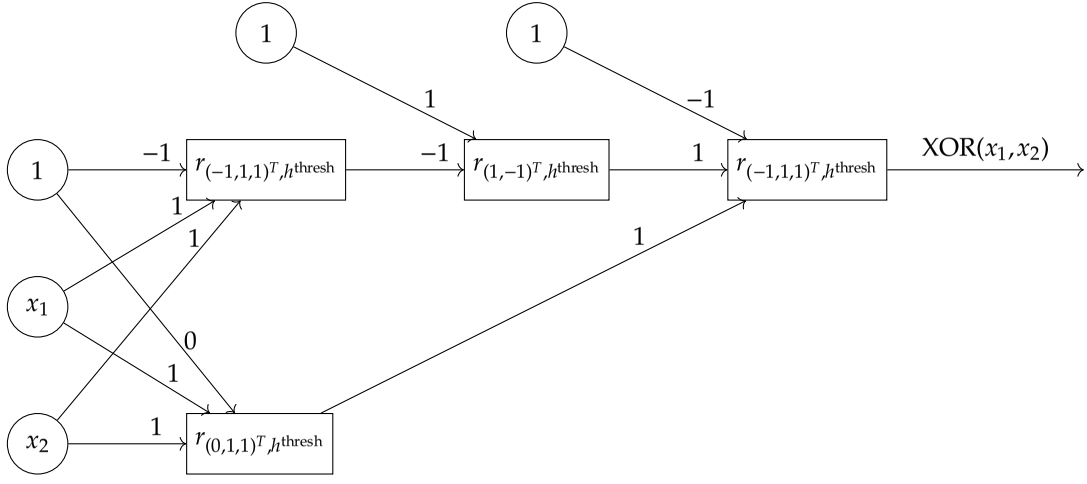


Abbildung 81: Geschichtete Perzeptronen zur Modellierung von XOR.

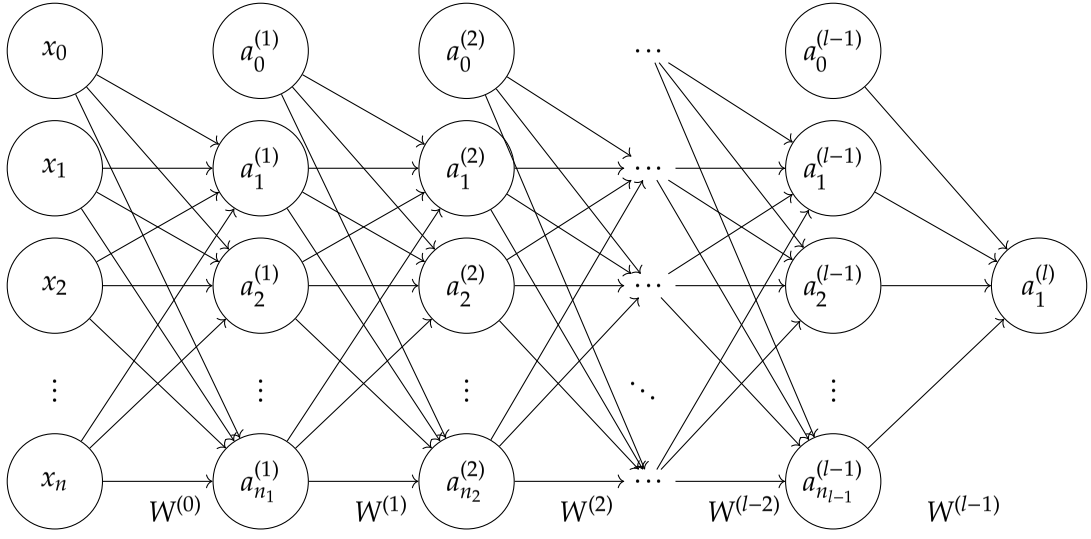


Abbildung 82: Allgemeine Struktur eines *Feedforward*-Netzwerkes mit l Schichten. Die Neuronen $x_0, a_0^{(1)}, \ldots, a_0^{(l-1)}$ haben dabei den konstanten Wert 1. Dieses Netzwerk besitzt genau ein Ausgabeneuron in der letzten Schicht und ist deshalb für Regression und binäre Klassifikation anwendbar.

gabe ist in Abbildung 82 dargestellt. Das dort dargestellte Netzwerk besteht aus $l \in \mathbb{N}$ Schichten (die *Eingabeschicht*, bestehend aus dem Eingabevektor $(x_0, x_1, \ldots, x_n)$, inklusive dem Bias-Neuron $x_0 = 1$, wird üblicherweise bei dieser Zählweise nicht hinzugenommen). Die Schicht mit dem Index l heißt Ausgabeschicht und die Schichten mit den Indizes $1, \ldots, l-1$ heißen *versteckte Schichten*. Jede Schicht kann potentiell eine verschiedene Anzahl an Neuronen enthalten, in Abbildung 82 enthält die Eingabeschicht $n_0 = n+1$ Eingabeneuronen (inklusive dem Bias-Neuron), die versteckte Schicht $k = 1, \ldots, l-1$ enthält n_k Neuronen und die Ausgabeschicht ein Neuron. Zwischen jedem Paar $k-1, k$ von Schichten (jetzt für $k = 1, \ldots, l$) gibt es eine Parametermatrix $W^{(k-1)} \in \mathbb{R}^{n_k \times n_{k-1}}$, die die Parameter der einzelnen Neuronen der k-ten Schicht beinhaltet. Abbildung 83 zeigt dabei die einzelnen Gewichtsbezeichner zwischen zwei beliebigen Schichten $k-1$ und k. Allgemein bezeichnet $W^{(x)}_{y,z}$ das Gewicht des z-ten Neurons der x-ten Schichten in der Berechnung des Wertes im y-ten Neuron der $x+1$-ten Schicht. Genauer, ist $x = (x_0, x_1, \ldots, x_n)$ die Eingabe des Netzwerkes (mit $x_0 = 1$), so berechnen sich die Aktivierungswerte der ersten versteckten Schicht durch

$$a^{(1)}_i = \mathrm{act}\left(W^{(0)}_{i,0} x_0 + W^{(0)}_{i,1} x_1 + \ldots + W^{(0)}_{i,n} x_n\right)$$

für $i = 1, \ldots, n_1$. Für $a^{(1)} = (a^{(1)}_1, \ldots, a^{(1)}_{n_1}) \in \mathbb{R}^{n_1}$ schreiben wir dies verkürzt als

$$a^{(1)} = \mathrm{act}\left(W^{(0)} x\right)$$

[38] Das in Abbildung 81 dargestellte Netzwerk kann leicht in diese Form gebracht werden, indem die Ausgabe des Neurons unten links zunächst in eine weiteres Neuron geleitet wird, das den Wert unverändert an das letzte Neuron oben rechts weiterleitet.

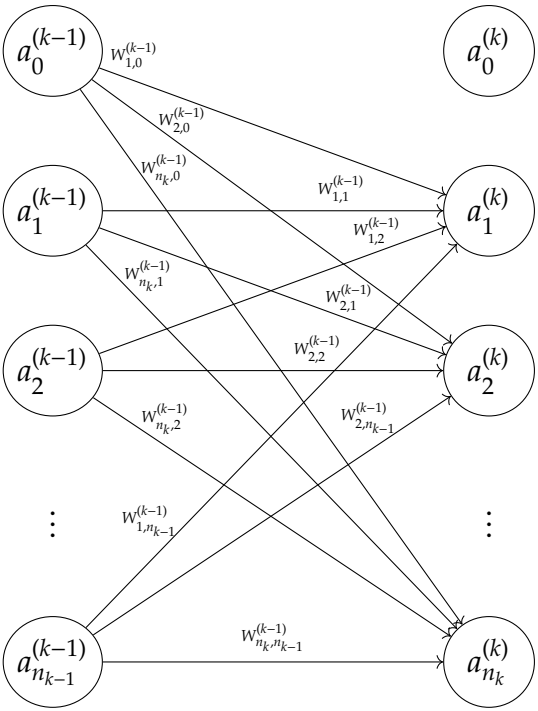


Abbildung 83: Gewichtsbezeichnungen zwischen Schicht $k-1$ und Schicht k.

wobei die Anwendung der Aktivierungsfunktion act hierbei komponentenweise zu verstehen ist. Allgemein berechnen sich die Aktivierungswerte the k-ten Schicht aus den schon berechneten Werten der $k-1$-ten Schicht durch

$$a^{(k)} = \mathrm{act}\left(W^{(k-1)} a^{(k-1)}\right)$$

für $k = 1, \ldots, l$ (beachten Sie auch, dass die Bias-Neuronen stets auf den rechten Seiten zusätzlich hinzugefügt werden müssen). Sei $\mathbb{W} = \{W^{(0)}, \ldots, W^{(l-1)}\}$ die geordnete Menge aller Gewichtsmatrizen der einzelnen Schichten. Die insgesamt durch das Netzwerk in Abbildung 82 darge-

stellte Funktion $\phi_{\mathbb{W}}(x)$ (also der Wert $a_1^{(l)}$ aus dem „Ausgabevektor“ $a^{(l)}) = (a_1^{(l)})$ (bei Eingabe von x) kann also geschrieben werden als

$$\phi_{\mathbb{W}}(x) = \mathrm{act}\Big(W^{(l-1)}\mathrm{act}\Big(W^{(l-2)}\ldots\mathrm{act}\Big(W^{(1)}\mathrm{act}\Big(W^{(0)}x\Big)\Big)\Big)\Big) \tag{35}$$

In der obigen Darstellung haben wir implizit angenommen, dass die Aktivierungsfunktion act für alle Neuronen des Netzwerks identisch ist. Dies ist nicht notwendigerweise der Fall. Üblicherweise haben Neuronen der inneren Schichten eine identische Aktivierungsfunktion, die Aktivierungsfunktion der Ausgabeschicht ist aber anwendungsabhängig und üblicherweise nicht identisch mit denen der versteckten Schichten. Unabhängig von diesem Aspekt sieht man Gleichung (35) bereits deutlich an, dass durch ANNs sehr komplexe Funktionen modelliert werden können. Tatsächlich kann gezeigt werden, dass bereits ein ANN mit nur einer versteckten Schicht jede beliebige Funktion beliebig genau approximieren kann (unter der zusätzlichen Bedingung, dass die verwendeten Aktivierungsfunktionen nicht-linear sind). Solche ANNs sind in der Praxis aber von geringem Interesse, da dort die versteckte Schicht sehr viele Neuronen besitzen müsste. Besitzt ein ANN wenigstens zwei versteckte Schichten, so spricht man bereits von einem *tiefen Netzwerk* (engl. *deep network*), das die Grundlage für alle modernen Techniken des *Deep Learnings* bildet. Mit einem tiefen Netzwerk kann eine komplexe Lernaufgabe auf Teilaufgaben heruntergebrochen werden und die einzelnen Schichten eines Netzwerks können diese verschiedenen Teilaufaufgaben lösen und deren Ergebnisse können in späteren Schichten miteinander verknüpft werden. Dies kann anschaulich am Beispiel eines *Convolutional Neural Networks* zur Bildklassifikation

verdeutlicht werden. Dazu werden die Pixel eines Bildes als reelle Zahlen repräsentiert (z. B. als einzelne Zahl, die den Grauwert darstellt, oder als 3 Zahlen für Kanäle rot/blau/grün) und als Eingabe in ein Netzwerk gespeist. Die erste Schicht des Netzwerks könnte dann Kanten im Bild erkennen (als große Unterschiede in der Helligkeit benachbarter Pixel), die nächste Schicht könnten Ecken erkennen (als zwei orthogonal zueinander liegender Kanten) und spätere Schichten geometrische Formen bis hin zu komplexen Konzepten wie Räder, Flügel, etc.. Diese Informationen können dann genutzt werden, um beispielsweise Autos und Flugzeuge in Bildern zu erkennen. Wir werden uns mit *Convolutional Neural Networks* im Detail in Unterkapitel 5.2 beschäftigen, aber die Intuition, dass eine komplexe Lernaufgabe durch viele versteckte Schichten in Teilaufgaben aufgeteilt wird, liegt allen tiefen Netzwerken zugrunde. Der große Vorteil des *Deep Learnings*, also das Erlernen der Gewichte eines tiefen Netzwerks, ist dabei auch, dass die Auswahl, welche „Zwischenkonzepte" die Schichten repräsentieren, vom Lernalgorithmus selbst entschieden wird, und nicht vom Benutzer vorgegeben werden muss. Tiefe Netzwerke können direkt auf den Rohdaten trainiert werden und eine manuelle Definition der Merkmale (engl. *feature engineering*) entfällt hier. Dies macht tiefe Netzwerke zu sehr breit anwendbaren Modellen des maschinellen Lernens.

Der zuletzt erwähnte Aspekt von tiefen Netzwerken, d. h., die Fähigkeit, beim Lernen automatisch „Zwischenkonzepte" zu erlernen, bringt auch direkt die größte Herausforderung mit sich, nämlich den Aufwand des Lernens. Sei $D = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ ein Trainingsdatensatz für ein binäres Klassifikationsproblem (es gilt also $y^{(1)}, \ldots, y^{(m)} \in \{0, 1\}$). Um ein tiefes Netzwerk für die binäre

Klassifikation zu trainieren, benutzt man üblicherweise[39] die schon bekannte logistische Kostenfunktion L^{logit}

$$L^{\text{logit}}(D, f) = -\sum_{i=1}^{m} y^{(i)} \log f(x^{(i)}) + (1 - y^{(i)}) \log(1 - f(x^{(i)}))$$

Ist unser Netzwerk wie in Abbildung 82 aufgebaut, so ist die repräsentierte Funktion wie in Gleichung (35) und die Lernaufgabe ist damit das Lösen des folgenden Optimierungsproblems:

$$\hat{\mathbb{W}} = \arg\min_{\mathbb{W}} L^{\text{logit}}(D, \phi_{\mathbb{W}}) \tag{36}$$

Numerische Methoden zur Lösung des obigen Optimierungsproblems, wie etwa *Gradient Descent* (siehe Unterkapitel 1.2), benötigen die Berechnung und Auswertung diverser Ableitungen der Zielfunktion $L^{\text{logit}}(D, \phi_{\mathbb{W}})$, dessen zentraler Bestandteil $\phi_{\mathbb{W}}$ ist. Wie Gleichung (35) vermuten lässt, ist die Berechnung und Auswertung von Ableitungen von $\phi_{\mathbb{W}}$ bzgl. eines Parameters $W_{x,y}^{(k)}$ eine sehr aufwändige Angelegenheit und selbst für relativ kleine Netzwerke in direkter Form nicht praktikabel. Dieses Hindernis wird allerdings durch den *Backpropagation*-Algorithmus gelöst, den wir uns etwas genauer in Abschnitt 5.1.4 anschauen werden. Ein weiteres Problem zur Lösung von (36) ist gegeben durch die sehr hohe Anzahl an Parametern und die sehr große Menge an Trainingsdaten, die nötig ist, diese Parameter ausreichend gut zu erlernen. Erst durch Hardwareentwicklungen Anfang der 2010er konnten tiefe Netzwerke ausreichend gut gelernt werden, um in der Praxis gute Ergebnisse zu liefern. Wir

[39] Bei neuronalen Netzwerken ist Regularisierung (siehe Abschnitt 2.1.4) auch sehr wichtig, wir ignorieren diesen Aspekt aber zunächst.

werden uns in Abschnitt 5.1.5 mit ein paar wenigen Herausforderungen in diesem Kontext beschäftigen.

Das in Abbildung 82 dargestellte *Feedforward*-Netzwerk besitzt genau ein Ausgabeneuron und ist damit sowohl für die Regression als auch die binäre Klassifikation geeignet. Wird das Netzwerk für die Regression verwendet, so muss die Zielmenge der Aktivierungsfunktion des Ausgabeneurons auch die gesamten reellen Zahlen erfassen (wie beispielsweise die Identitätsfunktion). Wird das Netzwerk für die binäre Klassifikation verwendet, so muss die Aktivierungsfunktion entweder einen Bool'schen Wert ausgeben (wie die Schwellwertfunktion) oder alternativ eine Wahrscheinlichkeit für die Zugehörigkeit zur ersten Klasse (wie die Sigmoid-Funktion). Um neuronale Netzwerke für die Mehrklassenklassifikation für k Klassen zu verwenden, muss die Ausgabeschicht entsprechend k Neuronen enthalten.[40] Die Ausgabeschicht wird dann so trainiert, dass Sie eine Wahrscheinlichkeitsverteilung über die Klassen angibt, d. h., die Ausgabewerte aller Neuronen sind nicht-negativ und summieren sich zu 1.

5.1.3 Aktivierungsfunktionen

Aktivierungsfunktionen sind ein zentraler Bestandteil der Architektur neuronaler Netzwerke und die Wahl der richtigen Aktivierungsfunktion kann sowohl den Lernaufwand als auch die Qualität des gelernten Netzwerks stark beeinflussen. Wir haben bisher schon die folgenden Ak-

[40] Alternative Repräsentationen, wie etwa die Nutzung eines einzelnen Ausgabeneurons, das k verschiedene Werte ausgeben kann, sind in der Praxis unüblich.

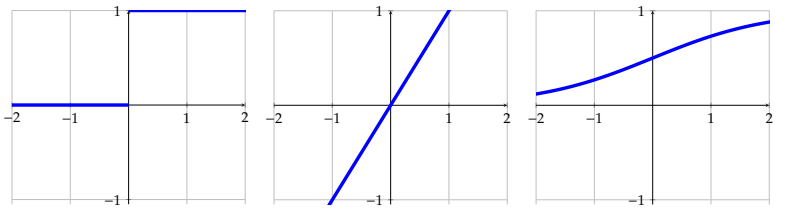
tivierungsfunktionen kennengelernt:

$$h^{\text{thresh}}(x) = \begin{cases} 1 & \text{falls } x > 0 \\ 0 & \text{sonst} \end{cases}$$
$$h^{\text{id}}(x) = x$$
$$h^{\text{logit}}(x) = \frac{1}{1+e^{-x}}$$

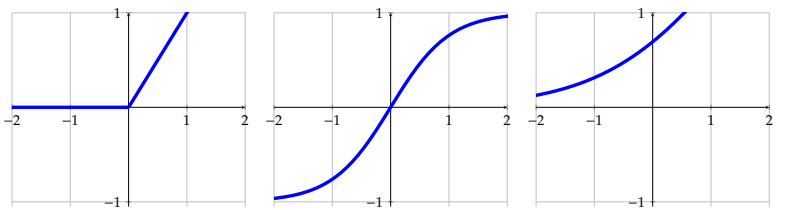
Die Funktion h^{thresh} ist die *Schwellwertfunktion*, h^{id} die *Identitätsfunktion* und h^{logit} die *Sigmoid-Funktion* oder auch *logistische Funktion*. Weitere in der Praxis gebräuchliche Aktivierungsfunktionen sind

$$h^{\text{relu}}(x) = \max\{0, x\}$$
$$h^{\text{tanh}}(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$
$$h^{\text{splus}}(x) = \ln(1 + e^{x})$$

Die Funktion h^{relu} heißt *Rectified Linear Unit* (*ReLU*, dt. *Gleichrichter*), h^{tanh} ist die *Hyperbeltangens-Funktion* und h^{splus} heißt *Softplus*. Die Kurvenverläufe dieser Aktivierungsfunktionen sind in Abbildung 84 dargestellt. Die Funktion h^{id} findet kaum Anwendung in neuronalen Netzen, da sie die Ausdrucksstärke des Netzes stark einschränkt. Besitzen alle Neuronen als Aktivierungsfunktion die Identitätsfunktion, so kann gezeigt werden, dass das gesamte Netzwerk eine lineare Funktion repräsentiert und damit nicht ausdruckstärker ist als ein einzelnes Neuron mit der Identitätsfunktion als Aktivierungsfunktion. Ein wichtiger Aspekt der Aktivierungsfunktion ist ihre Ableitung. Beim Lernen von neuronalen Netzen (siehe nächster Abschnitt) müssen lokale Ableitungen im Netz gebildet werden, um mithilfe von Methoden wie *Gradient Descent* Änderungen der Gewichte



(a) h^{thresh} (b) h^{id} (c) h^{logit}



(d) h^{relu} (e) h^{tanh} (f) h^{splus}

Abbildung 84: Kurvenverläufe der Aktivierungsfunktionen h^{thresh}, h^{id}, h^{logit}, h^{relu}, h^{tanh} und h^{splus}.

zu berechnen. Ist die Ableitung der Aktivierungsfunktion wohldefiniert und informativ, so vereinfacht dies den Lernprozess. Beispielsweise ist die Ableitung der Schwellwertfunktion h^{thresh} überall 0 (bis auf den Punkt $x = 0$, wo die Ableitung nicht definiert ist). Somit kann die Lernrichtung bei Verwendung der Schwellwertfunktion nicht bestimmt werden. Die Schwellwertfunktion wird deshalb auch kaum in praktischen Anwendungen genutzt. Die übrigen Funktionen finden alle Anwendung in der Praxis, wobei die Funktion h^{relu} aufgrund ihrer mathematischen Einfachheit und leicht berechenbarer Ableitung aktuell am häufigsten verwendet wird (obwohl auch ihre Ableitung bei $x = 0$ undefiniert ist). Je nach

konkretem Anwendungsfall wählt man für die Ausgabeschicht allerdings andere Aktivierungsfunktionen. Soll das Netz beispielsweise eine Regressionsaufgabe lösen, so ist diesem Fall die Verwendung der Identitätsfunktion tatsächlich passend, da sie als Zielmenge die gesamten reellen Zahlen darstellen kann (und solange die versteckten Schichten nichtlineare Aktivierungsfunktionen nutzen, mindert die einmalige Nutzung der Identitätsfunktion auch nicht die Ausdrucksstärke). Für eine binäre Klassifikationsaufgabe bietet sich die Verwendung der Funktion h^{logit} an.

Bei der Mehrklassenklassifikation wird in der Ausgabeschicht üblicherweise zusätzlich die *Softmax*-Funktion angewendet, damit alle Werte der Ausgabeschicht sich zu 1 aufsummieren. Seien $a_1^{(l-1)}, \ldots, a_k^{(l-1)}$ die Ausgaben der vorherigen Schicht. Diese repräsentieren üblicherweise die „Stärke" der Klassifikation der einzelnen k Klassen. Die Ausgaben der Ausgabeschicht berechnen sich dann zu

$$a_j^{(l)} = \frac{e^{a_j^{(l-1)}}}{\sum_{i=1}^{k} e^{a_i^{(l-1)}}}$$

für alle $j = 1, \ldots, k$. Es sollte leicht zu sehen sein, dass $a_1^{(l)} + \ldots + a_k^{(l)} = 1$ gilt.

5.1.4 Backpropagation

Wir nehmen die in Abbildung 82 dargestellte Architektur eines ANNs an und betrachten ein binäres Klassifikationsproblem. Sei $D = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ ein entsprechender Datensatz mit $y^{(1)}, \ldots, y^{(m)} \in \{0, 1\}$. Wie zuvor bietet sich für dieses Lernproblem die logistische Kosten-

funktion L^{logit} definiert durch

$$L^{\text{logit}}(D, f) = -\sum_{i=1}^{m} y^{(i)} \log f(x^{(i)}) + (1 - y^{(i)}) \log(1 - f(x^{(i)}))$$

an. Definieren wir den logistischen Einzelkostenwert via

$$\text{cost}_{L^{\text{logit}}}(y, y') = -y \log y' - (1 - y) \log(1 - y')$$

so können wir $L^{\text{logit}}(D, f)$ einfacher schreiben als

$$L^{\text{logit}}(D, f) = \sum_{i=1}^{m} \text{cost}_{L^{\text{logit}}}(y^{(i)}, f(x^{(i)}))$$

Beachten Sie allerdings, dass für andere Architekturen und andere Probleme auch andere Kostenfunktionen genutzt werden können, der obige Zusammenhang zwischen der (gesamten) Kostenfunktion und der Einzelkostenwerte ist für die meisten interessanten Kostenfunktionen aber derselbe, d. h.

$$L(D, f) = \sum_{i=1}^{m} \text{cost}(y^{(i)}, f(x^{(i)}))$$

Im Folgenden bezeichne L eine beliebige Kostenfunktion und cost_L den entsprechenden Einzelkostenwert. Gegeben L und D, ist es das Ziel des Lernprozesses, die Gewichte $\hat{\mathbb{W}}$ so zu bestimmen, dass unser ANN, also die Funktion $\phi_{\mathbb{W}}$, optimal an den Datensatz angepasst ist, d. h., im Allgemeinen suchen wir

$$\hat{\mathbb{W}} = \arg\min_{\mathbb{W}} L(D, \phi_{\mathbb{W}}) \tag{37}$$

Um das obige Optimierungsproblem zu lösen, können wir prinzipiell wieder Optimierungsmethoden wie *Gradient Descent* (siehe Unterkapitel 1.2) anwenden. Erinnern

wir uns kurz an das Grundprinzip von *Gradient Descent*: in jeder Iteration verändern wir die Gewichte $\mathbb{W}$ einen kleinen Schritt in Richtung des negativen Gradienten, um so schrittweise ein (lokales) Minimum zu erreichen. Genauer, sei $\mathbb{W}_0$ eine zufällige[41] Initialisierung der Gewichtsmatrizen und γ die Lernrate (die wir hier als konstant annehmen), so berechnen sich die neuen Gewichtsmatrizen via[42]

$$\mathbb{W}_{t+1} := \mathbb{W}_t - \gamma \nabla_{\mathbb{W}_t} L(D, \phi_{\mathbb{W}_t}) \tag{38}$$

Essentiell für die Anwendung der obigen Regel ist also die Bestimmung des Gradienten $\nabla_{\mathbb{W}_t} L(D, \phi_{\mathbb{W}_t})$, d. h., die Bestimmung der partiellen Ableitungen

$$\frac{\partial L(D, \phi_{\mathbb{W}})}{\partial W_{i,j}^{(k)}} \tag{39}$$

für alle Gewichte $W_{i,j}^{(k)}$ aus $\mathbb{W}$ (wir verzichten nun auf eine explizite Indizierung der Menge $\mathbb{W}$). $\nabla_{\mathbb{W}} L(D, \phi_{\mathbb{W}})$ ist dann ein Vektor bestehend aus all diesen partiellen Ableitungen. Wie zuvor schon erwähnt, ist eine direkte Bestimmung dieser partiellen Ableitungen nicht praktikabel.

Der *Backpropagation*-Algorithmus ist ein Algorithmus, der die partiellen Ableitungen (39) in geschickter Weise berechnet. Er benutzt dazu Aspekte der dynamischen

[41] Ein zufällige Initialisierung der Gewichte anstatt einer Initialisierung ausschließlich mit 0 ist beim Lernen mit Backpropagation sehr wichtig. Wären alle Gewichte initial 0, so werden zwangsweise alle Neuronen einer Schicht stets identische Gewichte auf ihren ausgehenden Kanten haben.

[42] In der folgenden Schreibweise interpretieren wir $\mathbb{W}$ nicht als geordnete Menge der Gewichtsmatrizen, sondern als einen langen Vektor, der alle Gewichte der Gewichtsmatrizen nacheinander auflistet.

Programmierung und nutzt die Kettenregel der Differentialrechnung aus. Sei $(x, y) \in D$ ein Beispiel. Zunächst werden in einem Vorwärtspropagationsschritt der Funktionswert $\phi_{\mathbb{W}}(x)$ und die linearen Anteile und Aktivierungswerte aller Neuronen im ANN berechnet. Im eigentlichen Rückwärtspropagationsschritt (engl. *back propagation*) wird der am Ausgabeneuron beobachtete Fehler in der Vorhersage (d. h. der Unterschied von y zum vorhergesagten Wert $\phi_{\mathbb{W}}(x)$ von der Ausgabeschicht zurückgegeben und damit iterativ Fehler an den Neuronen der versteckten Schichten berechnet. Diese Fehler an Neuronen (über alle Beispiele aus D) können anschließend benutzt werden, um die einzelnen partiellen Ableitungen (39) zu berechnen. Im folgenden stellen wir diesen Algorithmus im Detail vor.

Wir nehmen an, dass alle Neuronen dieselbe Aktivierungsfunction act benutzen (dies vereinfacht die Darstellung; in der Praxis müssten dann gegebenenfalls an den entsprechenden Stellen andere Aktivierungsfunktionen eingesetzt werden). Für $D = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ erhalten wir zunächst direkt aus (39)

$$\begin{aligned} &\frac{\partial L(D, \phi_{\mathbb{W}_i})}{\partial W_{i,j}^{(k)}} \\ &= \frac{\partial \text{cost}(y^{(1)}, \phi_{\mathbb{W}}(x^{(1)}))}{\partial W_{i,j}^{(k)}} + \ldots + \frac{\partial \text{cost}(y^{(m)}, \phi_{\mathbb{W}}(x^{(m)}))}{\partial W_{i,j}^{(k)}} \end{aligned} \tag{40}$$

Es genügt also, sich auf die Bestimmung der Ableitung

$$\frac{\partial \text{cost}(y, \phi_{\mathbb{W}}(x))}{\partial W_{i,j}^{(k)}} \tag{41}$$

für ein einzelnes Beispiel $(x, y) \in D$ zu konzentrieren. Nachdem wir diese Ableitungen für alle Beispiele in D berech-

net haben, bilden wir einfach die Summe der Werte, um (39) zu berechnen.

Sei also $(x, y) \in D$. Für die Vorwärtspropagation berechnen wir zunächst die Vektoren $z^{(1)}$ (die linearen Anteile) und $a^{(1)}$ (die Aktivierungswerte) der ersten Schicht via

$$z^{(1)} = W^{(0)}x \qquad\qquad a^{(1)} = \text{act}(z^{(1)})$$

Wir propagieren die Werte durch das gesamte Netzwerk bis zur Ausgabeschicht, d. h., wir berechnen

$$z^{(i)} = W^{(i-1)}a^{(i-1)} \qquad\qquad a^{(i)} = \text{act}(z^{(i)})$$

für alle $i = 2\ldots, l$ (beachten Sie, dass wir bei der obigen Verwendung von $a^{(i-1)}$ immer noch die Bias-Eingabe mit dem Wert 1 als zusätzliche erste Komponente einfügen müssen). Für unser Netzwerk aus Abbildung 82 besteht die Ausgabeschicht auch nur aus einem Neuron $a^{(l)} = (a_1^{(l)})$.

Um nun die partiellen Ableitungen (41) zu bestimmen, nutzen wir den Umstand, dass die Werte $z^{(i)}$ Ergebnisse von Teilfunktionen der Gesamtfunktion ϕ_W darstellen und wir die Kettenregel der Differentialrechnung nutzen können. In ihrer einfachsten Form besagt die Kettenregel, dass die Ableitung einer Komposition von Funktionen identisch ist mit der Multiplikation der Ableitungen der Einzelfunktionen, also $(f(g(x)))' = f'(g(x))g'(x)$ oder äquivalent

$$\frac{df}{dx} = \frac{df}{dg}\frac{dg}{dx}$$

Angenommen, jede Schicht unseres ANNs bestünde aus einem einzelnen Neuron (also insbesondere $z^{(i)} = (z_1^{(i)})^T$ für alle $i = 1, \ldots, l$), dann entspricht dies einer mehrfachen

Funktionskomposition wie oben und wir könnten für die Ableitung eines beliebigen Gewichts $W_{1,1}^{(k)}$ schreiben

$$\frac{\partial \text{cost}(y, \phi_{\mathbb{W}}(x))}{\partial W_{1,1}^{(k)}} = \frac{\partial \text{cost}(y, \phi_{\mathbb{W}}(x))}{\partial z_1^{(l)}} \frac{\partial z_1^{(l)}}{\partial z_1^{(l-1)}} \cdots \frac{\partial z_1^{(k+2)}}{\partial z_1^{(k+1)}} \frac{\partial z_1^{(k+1)}}{\partial W_{1,1}^{(k)}}$$

Für den allgemeinen Fall mehrerer Neuronen pro Schicht, müssen wir die mehrdimensionale Kettenregel benutzen. Dabei müssen wir die Summe der Ableitungen aller möglichen Pfade vom betrachteten Neuron bis zum Ausgabeneuron betrachten. Für ein beliebiges Gewicht $W_{i,j}^{(k)}$ ergibt sich damit (beachten Sie, dass das Gewicht $W_{i,j}^{(k)}$ zu der Kante gehört, die auf das Neuron mit dem linearen Anteil $z_i^{(k+1)}$ zeigt)

$$\begin{aligned}
&\frac{\partial \text{cost}(y, \phi_{\mathbb{W}}(x))}{\partial W_{i,j}^{(k)}} \\
&= \frac{\partial \text{cost}(y, \phi_{\mathbb{W}}(x))}{\partial z_i^{(k+1)}} \frac{\partial z_i^{(k+1)}}{\partial W_{i,j}^{(k)}} \\
&= \left[\frac{\partial \text{cost}(y, \phi_{\mathbb{W}}(x))}{\partial z_1^{(l)}} \sum_{i_{l-1}=1}^{n_{l-1}} \cdots \right. \\
&\qquad \left. \cdots \sum_{i_{k+2}=1}^{n_{k+2}} \frac{\partial z_1^{(l)}}{\partial z_{i_{l-1}}^{(l-1)}} \frac{\partial z_{i_{l-1}}^{(l-1)}}{\partial z_{i_{l-2}}^{(l-2)}} \cdot \ldots \cdot \frac{\partial z_{i_{k+2}}^{(k+2)}}{\partial z_i^{(k+1)}} \right] \frac{\partial z_i^{(k+1)}}{\partial W_{i,j}^{(k)}}
\end{aligned} \tag{42}$$

Beachten Sie, dass die Summen im oberen Ausdruck über alle möglichen Kombinationen von Neuronen zwischen den Schichten $k+2$ bis $l-1$ laufen. Eine Berechnung der Ableitung zu $W_{i,j}^{(k)}$ mit der obigen Gleichung ist ebenso unpraktikabel wie die direkte Berechnung, da die

Zahl dieser möglichen Kombinationen exponentiell groß ist. Die Berechnung dieser verschiedenen Teilableitungen wird allerdings auch bei der Berechnung der Ableitungen anderer Gewichte benötigt, und so kann durch einen Ansatz der dynamischen Programmierung die Reihenfolge der Berechnungen systematisiert und so überflüssige Rechnungen vermieden werden. Dazu definieren wir

$$\begin{aligned}
&\delta(z_i^{(k+1)}) \\
&= \frac{\partial \text{cost}(y, \phi_{\mathbb{W}}(x))}{\partial z_i^{(k+1)}} \\
&= \frac{\partial \text{cost}(y, \phi_{\mathbb{W}}(x))}{\partial z_1^{(l)}} \left[\sum_{i_{l-1}=1}^{n_{l-1}} \cdots \sum_{i_{k+2}=1}^{n_{k+2}} \frac{\partial z_1^{(l)}}{\partial z_{i_{l-1}}^{(l-1)}} \frac{\partial z_{i_{l-1}}^{(l-1)}}{\partial z_{i_{l-2}}^{(l-2)}} \cdot \ldots \cdot \frac{\partial z_{i_{k+2}}^{(k+2)}}{\partial z_i^{(k+1)}} \right]
\end{aligned}$$

für $k = 0, \ldots, l-1$ und $i = 1, \ldots, n_k$. Die Werte $\delta(z_i^{(k)})$ können nun rekursiv absteigend von $\delta(z_1^{(l)})$ an einfach auf den bisher berechneten Werten berechnet werden. Es gilt dazu zunächst

$$\delta(z_1^{(l)}) = \frac{\partial \text{cost}(y, \phi_{\mathbb{W}}(x))}{\partial z_1^{(l)}} \tag{43}$$

Gegeben eine konkrete Kostenfunktion L, ist der Ausdruck (43) einfach auszuwerten (siehe Beispiel 94 unten). Für die Neuronen der versteckten Schichten lässt sich folgender rekursiver Zusammenhang feststellen (sei $k = 1, \ldots, l-1$ und $j = 1, \ldots, n_k$):

$$\delta(z_j^{(k)}) = \frac{\partial \text{cost}(y, \phi_{\mathbb{W}}(x))}{\partial z_j^{(k)}} = \sum_{i=1}^{n_{k+1}} \underbrace{\frac{\partial \text{cost}(y, \phi_{\mathbb{W}}(x))}{\partial z_i^{(k+1)}}}_{\text{(I)}} \underbrace{\frac{\partial z_i^{(k+1)}}{\partial z_j^{(k)}}}_{\text{(II)}} \tag{44}$$

Die obige Gleichung wendet wieder die Kettenregel an: wir summieren hier über alle Nachfolgeneuronen $z_i^{(k+1)}$ des betrachteten Neurons $z_j^{(k)}$ und multiplizieren deren lokale Ableitungen (I) mit der „inneren" Ableitung (II) bzgl. $z_j^{(k)}$. Der Ausdruck (I) ist identisch mit $\delta(z_i^{(k+1)})$ und wurde nach dem Prinzip der dynamischen Programmierung bereits berechnet. Der Ausdruck (II) kann direkt berechnet werden:

$$\begin{aligned}
&\frac{\partial z_i^{(k+1)}}{\partial z_j^{(k)}} \\
&= \frac{\partial \left(W_{i,0}^{(k)} \text{act}(z_0^{(k)}) + \ldots + W_{i,j}^{(k)} \text{act}(z_j^{(k)}) + \ldots + W_{i,n_k}^{(k)} \text{act}(z_{n_k}^{(k)}) \right)}{\partial z_j^{(k)}} \\
&= W_{i,j}^{(k)} \text{act}'(z_j^{(k)})
\end{aligned}$$

Wir erhalten also

$$\begin{aligned}
\delta(z_j^{(k)}) &= \sum_{i=1}^{n_{k+1}} \delta(z_i^{(k+1)}) W_{i,j}^{(k)} \text{act}'(z_j^{(k)}) \\
&= \text{act}'(z_j^{(k)}) \sum_{i=1}^{n_{k+1}} \delta(z_i^{(k+1)}) W_{i,j}^{(k)} \qquad (45)
\end{aligned}$$

Eine letzte Vereinfachung der Gleichung (42) ergibt die finale Berechnungsvorschrift für die partiellen Ableitungen bzgl. der Gewichte

$$\begin{aligned}
&\frac{\partial \text{cost}(y, \phi_{\mathbb{W}}(x))}{\partial W_{i,j}^{(k)}} \\
&= \frac{\partial \text{cost}(y, \phi_{\mathbb{W}}(x))}{\partial z_i^{(k+1)}} \frac{\partial z_i^{(k+1)}}{\partial W_{i,j}^{(k)}}
\end{aligned}$$

$$
\begin{aligned}
&= \delta(z_i^{(k+1)}) \frac{\partial z_i^{(k+1)}}{\partial W_{i,j}^{(k)}} \\
&= \delta(z_i^{(k+1)}) \frac{\partial \left(W_{i,0}^{(k)} a_0^{(k)} + \ldots + W_{i,j}^{(k)} a_j^{(k)} + \ldots + W_{i,n_k}^{(k)} a_{n_k}^{(k)} \right)}{\partial W_{i,j}^{(k)}} \\
&= \delta(z_i^{(k+1)}) a_j^{(k)} \qquad (46)
\end{aligned}
$$

Wir schauen uns nun die obigen Berechnungen einmal an einem konkreten Beispiel an.

Beispiel 93. Wir betrachten das Netzwerk in Abbildung 85. Es gilt

$$
\begin{aligned}
W^{(0)} &= \begin{pmatrix} W_{1,0}^{(0)} & W_{1,1}^{(0)} & W_{1,2}^{(0)} \\ W_{2,0}^{(0)} & W_{2,1}^{(0)} & W_{2,2}^{(0)} \end{pmatrix} \\
W^{(1)} &= \begin{pmatrix} W_{1,0}^{(1)} & W_{1,1}^{(1)} & W_{1,2}^{(1)} \\ W_{2,0}^{(1)} & W_{2,1}^{(1)} & W_{2,2}^{(1)} \end{pmatrix} \\
W^{(2)} &= \begin{pmatrix} W_{1,0}^{(2)} & W_{1,1}^{(2)} & W_{1,2}^{(2)} \end{pmatrix}
\end{aligned}
$$

und $\mathbb{W} = \{W^{(0)}, W^{(1)}, W^{(2)}\}$.[43] Wir nehmen folgende initiale Gewichte an:

$$
\begin{aligned}
W^{(0)} &= \begin{pmatrix} 2 & -1 & -1 \\ 0 & 1 & -1 \end{pmatrix} \\
W^{(1)} &= \begin{pmatrix} 3 & -2 & -1 \\ 2 & -1 & 1 \end{pmatrix} \\
W^{(2)} &= \begin{pmatrix} 1 & 1 & -3 \end{pmatrix}
\end{aligned}
$$

Weiterhin sei die Aktivierungsfunktion für alle Neuronen des Netzes die Sigmoid-Funktion h^{logit} und die Kostenfunktion sei L^{logit}.

[43] Für die Anwendung im Sinne von Gleichung (38) schreiben wir $\mathbb{W}$ auch als Vektor $\mathbb{W} = (W_{1,0}^{(0)}, W_{1,1}^{(0)}, W_{1,2}^{(0)}, W_{2,0}^{(0)}, \ldots, W_{1,2}^{(2)})^T$

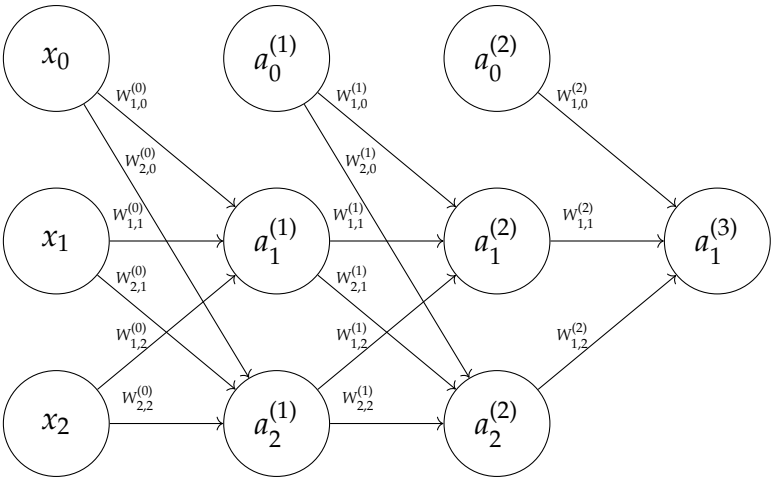


Abbildung 85: Feedforward-Netzwerk aus Beispiel 94.

Sei ein Beispiel (x, y) gegeben durch

$$x = (x_0, x_1, x_2)^T = (1, 2, -1)^T \qquad y = 1$$

Für die Vorwärtspropagation berechnen wir zunächst den linearen Anteil $z^{(1)}$ und den Aktivierungswert $a^{(1)}$ der ersten Schicht

$$z^{(1)} = W^{(0)} x = \begin{pmatrix} 2 & -1 & -1 \\ 0 & 1 & -1 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \\ -1 \end{pmatrix} = \begin{pmatrix} 1 \\ 3 \end{pmatrix}$$

$$a^{(1)} = h^{\text{logit}}(z^{(1)}) = \begin{pmatrix} h^{\text{logit}}(1) \\ h^{\text{logit}}(3) \end{pmatrix} \approx \begin{pmatrix} 0.731 \\ 0.953 \end{pmatrix}$$

Wir ergänzen nun zunächst bei $a^{(1)}$ den Bias und schreiben $a^{(1)} = (1, 0.731, 0.953)^T$. Nun evaluieren wir die nächste

Schicht:

$$z^{(2)} = W^{(1)}a^{(1)} = \begin{pmatrix} 3 & -2 & -1 \\ 2 & -1 & 1 \end{pmatrix}\begin{pmatrix} 1 \\ 0.731 \\ 0.953 \end{pmatrix} = \begin{pmatrix} 0.585 \\ 2.222 \end{pmatrix}$$

$$a^{(2)} = h^{\text{logit}}(z^{(2)}) = \begin{pmatrix} h^{\text{logit}}(0.585) \\ h^{\text{logit}}(2.222) \end{pmatrix} \approx \begin{pmatrix} 0.642 \\ 0.902 \end{pmatrix}$$

Wir ergänzen zunächst bei $a^{(2)}$ den Bias und schreiben $a^{(2)} = (1, 0.642, 0.902)^T$. Wir evaluieren nun die Ausgabeschicht

$$z^{(3)} = W^{(2)}a^{(2)} = \begin{pmatrix} 1 & 1 & -3 \end{pmatrix}\begin{pmatrix} 1 \\ 0.642 \\ 0.902 \end{pmatrix} = \begin{pmatrix} -1.064 \end{pmatrix}$$

$$a^{(3)} = h^{\text{logit}}(z^{(3)}) = \begin{pmatrix} h^{\text{logit}}(-1.064) \end{pmatrix} \approx \begin{pmatrix} 0.257 \end{pmatrix}$$

Wir erhalten für unser Beispiel (x, y) und unser aktuelles Netz damit folgenden Einzelkostenwert:

$$\text{cost}_{L^{\text{logit}}}(y, a_1^{(3)}) = \text{cost}_{L^{\text{logit}}}(1, 0.257) \approx 1.359$$

Wir berechnen nun die Fehlerwerte δ an den einzelnen Neuronen und fangen mit der Ausgabeschicht an. Nach (43) gilt

$$\begin{aligned} \delta(z_1^{(3)}) &= \frac{\partial \text{cost}_{L^{\text{logit}}}(y, \phi_{\mathbb{W}}(x))}{\partial z_1^{(3)}} \\ &= \frac{\partial \text{cost}_{L^{\text{logit}}}(y, a_1^{(3)})}{\partial z_1^{(3)}} \\ &= \frac{\partial(-y \log h^{\text{logit}}(z_1^{(3)}) - (1-y)\log(1 - h^{\text{logit}}(z_1^{(3)})))}{\partial z_1^{(3)}} \end{aligned}$$

Beachten Sie, dass die Ableitung der Sigmoid-Funktion gegeben ist durch

$$(h^{\text{logit}})'(x) = h^{\text{logit}}(x)(1 - h^{\text{logit}}(x))$$

Es folgt

$$\begin{aligned}
\delta(z_1^{(3)}) &= -y\frac{1}{h^{\text{logit}}(z_1^{(3)})}(h^{\text{logit}})'(z_1^{(3)}) - \\
&\qquad (1-y)\frac{1}{1-h^{\text{logit}}(z_1^{(3)})}(-(h^{\text{logit}})'(z_1^{(3)})) \\
&= -y\frac{h^{\text{logit}}(z_1^{(3)})(1-h^{\text{logit}}(z_1^{(3)}))}{h^{\text{logit}}(z_1^{(3)})} - \\
&\qquad (1-y)\frac{-h^{\text{logit}}(z_1^{(3)})(1-h^{\text{logit}}(z_1^{(3)}))}{1-h^{\text{logit}}(z_1^{(3)})} \\
&= -y(1-h^{\text{logit}}(z_1^{(3)})) + (1-y)h^{\text{logit}}(z_1^{(3)}) \\
&= -y + yh^{\text{logit}}(z_1^{(3)}) + h^{\text{logit}}(z_1^{(3)}) - yh^{\text{logit}}(z_1^{(3)}) \\
&= h^{\text{logit}}(z_1^{(3)}) - y \\
&= a_1^{(3)} - y \\
&= 0.257 - 1 = -0.743
\end{aligned}$$

Beachten Sie, dass die obige Ableitung von $\delta(z_1^{(3)})$ nicht spezifisch für unser Beispiel ist. Für jedes ANN, dass als Kostenfunktion L^{logit} und Aktivierungsfunktion h^{logit} in der Ausgabeschicht benutzt, gilt $\delta(z_i^{(l)}) = a_i^{(l)} - y_i$ für jedes Neuron $z_i^{(l)}$ der Ausgabeschicht.

Wir berechnen nun die Fehlerwerte der letzten versteckten Schicht. Nach (45) gilt (beachten Sie, dass diese Neuronen nur jeweils ein Nachfolgeneuron haben, die

Summe ist deshalb entartet):

$$\begin{aligned}\delta(z_1^{(2)}) &= (h^{\text{logit}})'(z_1^{(2)})\delta(z_1^{(3)})W_{1,1}^{(2)} \\ &= h^{\text{logit}}(z_1^{(2)})(1-h^{\text{logit}}(z_1^{(2)}))\delta(z_1^{(3)})W_{1,1}^{(2)} \\ &= a_1^{(2)}(1-a_1^{(2)})\delta(z_1^{(3)})W_{1,1}^{(2)} \\ &= 0.642(1-0.642)\cdot(-0.743)\cdot 1 \\ &\approx -0.171\end{aligned}$$

und analog

$$\begin{aligned}\delta(z_2^{(2)}) &= a_2^{(2)}(1-a_2^{(2)})\delta(z_1^{(3)})W_{1,2}^{(2)} \\ &= 0.902(1-0.902)\cdot(-0.743)\cdot(-3) \\ &\approx 0.197\end{aligned}$$

Für die erste versteckte Schicht erhalten wir nun

$$\begin{aligned}\delta(z_1^{(1)}) &= (h^{\text{logit}})'(z_1^{(1)})\left[\delta(z_1^{(2)})W_{1,1}^{(1)}+\delta(z_2^{(2)})W_{2,1}^{(1)}\right] \\ &= a_1^{(1)}(1-a_1^{(1)})\left[\delta(z_1^{(2)})W_{1,1}^{(1)}+\delta(z_2^{(2)})W_{2,1}^{(1)}\right] \\ &= 0.731(1-0.731)\left[-0.171\cdot(-2)+0.197\cdot(-1)\right] \\ &\approx 0.029\end{aligned}$$

$$\begin{aligned}\delta(z_2^{(1)}) &= a_2^{(1)}(1-a_2^{(1)})\left[\delta(z_1^{(2)})W_{1,2}^{(1)}+\delta(z_2^{(2)})W_{2,2}^{(1)}\right] \\ &= 0.953(1-0.953)\left[-0.171(-1)+0.197\cdot 1\right] \\ &\approx 0.017\end{aligned}$$

Schließlich können wir nach (46) die Ableitungen der einzelnen Gewichte berechnen, beispielsweise

$$\begin{aligned}\frac{\partial \text{cost}_{L^{\text{logit}}}(y,\phi_{\mathbb{W}}(x))}{\partial W_{2,1}^{(1)}} &= \delta(z_2^{(2)})a_1^{(1)} \\ &= 0.197\cdot 0.731 \\ &\approx 0.144\end{aligned}$$

Beachten Sie, dass für die Gewichte der Eingabeschicht der Vektor $a^{(0)}$ der Eingabe x entspricht, also beispielsweise

$$\begin{aligned}\frac{\partial \text{cost}_{L^{\text{logit}}}(y, \phi_{\mathbb{W}}(x))}{\partial W^{(0)}_{1,2}} &= \delta(z^{(1)}_1) a^{(0)}_2 \\ &= \delta(z^{(1)}_1) x_2 \\ &\approx -0.029\end{aligned}$$

Ist (x, y) unser einziges Beispiel (also $D = \{(x, y)\}$), so gilt $L^{\text{logit}}(D, \phi_{\mathbb{W}}) = \text{cost}_{L^{\text{logit}}}(y, \phi_{\mathbb{W}}(x))$ und wir könnten nun die Gewichte anhand von (38) aktualisieren. Wäre beispielsweise $\gamma = 0.1$ unsere Lernrate, so berechnen wir den neuen Wert von Gewicht $W^{(0)}_{1,2}$ zu

$$\begin{aligned}(W^{(0)}_{1,2})^{\text{neu}} &:= W^{(0)}_{1,2} - \gamma \frac{\partial L^{\text{logit}}(D, \phi_{\mathbb{W}})}{\partial W^{(0)}_{1,2}} \\ &= W^{(0)}_{1,2} - \gamma \frac{\partial \text{cost}_{L^{\text{logit}}}(y, \phi_{\mathbb{W}}(x))}{\partial W^{(0)}_{1,2}} \\ &= -1 - 0.1 \cdot (-0.029) \approx -0.997\end{aligned}$$

Üblicherweise besitzt der Datensatz D mehr als nur ein Beispiel und wir können mithilfe von (40) die einzelnen partiellen Ableitungen bzgl. der einzelnen Beispiele aufsummieren, bevor wir einen einzelnen Schritt beim *Gradient Descent* unternehmen. Algorithmus 14 zeigt den gesamten *Backpropagation*-Algorithmus zur Bestimmung der partiellen Ableitungen (39), die dort mit $D^{(k)}_{i,j}$ bezeichnet werden.

5.1.5 Praktische Probleme beim Lernen

Der *Backpropagation*-Algorithmus, so wie er in Algorithmus 14 dargestellt ist, wird in ähnlicher Form in allen

Algorithmus 14 Der *Backpropagation*-Algorithmus für ANNs der Form aus Abbildung 82.

Eingabe: Datensatz $D = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$
Gewichtsmatrix W
Aktivierungsfunktion act

Ausgabe: Partielle Ableitungen $D_{i,j}^{(k)}$
für $k = 1, \ldots, l$, $i = 1, \ldots, n_k$, $j = 1, \ldots, n_{k-1}$

BackProp(D)

$D_{i,j}^{(k)} := 0$ für alle i, j, k
for $(x, y) \in D$ **do**
  % *Vorwärtspropagation*
  $z^{(1)} := W^{(0)} x$
  $a^{(1)} := \text{act}(z^{(1)})$
  for $i = 2, \ldots, l$ **do**
    $z^{(i)} := W^{(i-1)} a^{(i-1)}$
    $a^{(i)} := \text{act}(z^{(i)})$
  % *Rückwärtspropagation*
  $\delta(z_i^l) = 1/(\partial z_i^{(l)}) \partial \text{cost}(y, \phi_{\mathbb{W}}(x))$ für $i = 1, \ldots, n_l$
  $D_{i,j}^{(l-1)} := D_{i,j}^{(l-1)} + \delta(z_i^{(l)}) a_j^{(l-1)}$ für $i = 1, \ldots, n_l$, $j = 0, \ldots, n_{l-1}$
  for $k = l-1, ..., 1$ **do**
    $\delta(z_j^{(k)}) = \text{act}'(z_j^{(k)}) \sum_{i=1}^{n_{k+1}} \delta(z_i^{(k+1)}) W_{i,j}^{(k)}$ für $j = 1, \ldots, n_k$
    $D_{i,j}^{(k-1)} := D_{i,j}^{(k-1)} + \delta(z_i^{(k)}) a_j^{(k-1)}$ für $i = 1, \ldots, n_k$, $j = 0, \ldots, n_{k-1}$

modernen Ansätzen des Deep Learnings verwendet, mitunter aber mit einigen Modifikationen, die das Lernen auf großen Datenmengen und in sehr tiefen Netzwerken performanter machen. Zwei dieser Aspekte und allgemeine Herausforderungen beim Deep Learning wollen wir im Rest dieses Abschnitts diskutieren.

Beim klassischen *Gradient Descent*-Ansatz, siehe Gleichung (38), wird für eine einzelne Iteration der Gradient $\nabla_{\mathbb{W}_i} L(D, \phi_{\mathbb{W}_i})$ bezüglich des gesamten Datensatzes D bestimmt (mithilfe von Algorithmus 14). Bei sehr großen

Datensätzen kann diese Berechnung sehr lange dauern. Der Ansatz des *Stochastic Gradient Descent* (SGD) ist es, für jedes Update (38) nur ein zufällig gewähltes Beispiel $(x, y) \in D$ heranzuziehen und Algorithmus 14 nur für den Datensatz $\{(x, y)\}$ auszuführen (effektiv also nur die partiellen Ableitungen (41) zu benutzen). Jede Iteration von SGD kann damit signifikant schneller durchgeführt werden. Der Nachteil von SGD ist, dass das nächste lokale Minimum nicht sehr zielstrebig angesteuert wird und die Gesamtkosten des Datensatzes stark während der Durchführung fluktuieren können. Nichtsdestotrotz ist SGD aus Anwendungssicht praktikabler und führt gerade bei größeren Datensätzen zu signifikant besseren Endresultaten. Ein weiterer Nachteil von SGD ist allerdings die fehlende Parallelisierbarkeit. Der Mini-batch Gradient Descent-Ansatz (MBGD) adressiert genau diesen Aspekt und kann als Kompromiss zwischen Gradient Descent und SGD angesehen werden. In jeder Iteration von MBGD wird eine kleine Teilmenge (engl. *Batches*) $D' \subseteq D$ (in Größenordnungen von 1–100 Beispielen) ausgewählt und Algorithmus 14 mit D' ausgeführt. Bei diesem Ansatz wird zum einen der Fluktuationsgrad von SGD etwas gesenkt (da der Gradient über mehrere Beispiele gebildet wird). Zum anderen kann die äußerste for-Schleife in Algorithmus 14 parallelisiert werden. Aus diesem Grund bietet es sich an, die *Batch-Size*, d. h., die Größe $|D'|$, entsprechend der Anzahl zur Verfügung stehender Prozessoren zu wählen (bzw. entsprechend der Kapazität der zur Verfügung stehenden Parallelhardware wie GPUs).

Wir haben bereits in Abschnitt 5.1.3 angemerkt, dass die Verwendung der Schwellwertfunktion h^{thresh} als Aktivierungsfunktion nicht zu empfehlen ist, insbesondere aufgrund der Tatsache, dass $(h^{\text{thresh}})'(x) = 0$ für alle $x \in \mathbb{R} \setminus \{0\}$ gilt (und für $x = 0$ nicht definiert ist). Da die Ab-

leitung der Aktivierungsfunktion ein multiplikativer Bestandteil der Gradientenberechnung ist, siehe (45), werden alle partiellen Ableitung der versteckten Schichten stets 0 sein. Dies führt bei Verwendung von Gradient Descent (oder SGD oder MBDG) dazu, dass die Gewichte nicht geändert werden und kein Lernen erfolgt. Andere Aktivierungsfunktionen können allerdings auch zum „Problem des verschwindenden Gradienten" (engl. *vanishing gradient problem*) führen. Die Ableitung der Sigmoid-Funktion hat beispielsweise schon für $x \geq 3$ oder $x \leq -3$ einen Wert von unter 0.1. Durch die Komposition mehrerer Aktivierungsfunktionen in den einzelnen Schichten kann es hier leicht dazu kommen, dass durch Multiplikation mehrerer kleiner Ableitungswerte die Fehlerwerte in frühen Schichten des Netzwerks verschwindend gering werden (und evtl. aus hardwaretechnischen Gründen zu 0 abgerundet werden). Ein ähnliches Problem (das *exploding gradient problem*) kann auftreten, wenn große Ableitungswerte miteinander multipliziert werden. Gerade aus diesem Grund ist die Verwendung der Aktivierungsfunktion h^{relu} in der Praxis sehr üblich geworden, da für $x > 0$ die Ableitung von h^{ReLU} stets 1 ist und damit weder einen verschwindenden noch einen explodierenden Gradienten erzeugen kann. Ein Nachteil von h^{ReLU} ist, dass bei $x < 0$ die Ableitung stets 0 ist. Eine Abhilfe schaffen hier Aktivierungsfunktionen wie h^{splus}, die vom Kurvenverlauf sehr ähnlich zu h^{ReLU} sind, aber dennoch überall eine positive Ableitung haben.

5.2 Convolutional Neural Networks

Ein *Convolutional Neural Network* (dt. faltendes neuronales Netzwerk, CNN) ist eine spezielle Form eines *Feedforward*-Netzwerks, wie wir es in Unterkapitel 5.1 kennengelernt haben. Es findet Anwendung für Daten, die in einer Gitterstruktur angeordnet sind bzw. bei denen einzelne Merkmale in einer räumlichen oder zeitlichen Beziehung zueinander stehen. Das einfachste Beispiel dazu sind Bilddaten: ein (Graustufen-)Bild kann als Matrix interpretiert werden, bei dem die einzelnen Zellen den Graustufenwert kodieren. Farbbilder können als dreidimensionales Array interpretiert werden, bei dem drei einzelne zweidimensionale Matrizen, die jeweils den Farbwert des Rot-/Grün- und Blaukanals kodieren, zusammengefasst werden. Ein anderes Beispiel sind Zeitreihen, wie beispielsweise Folgen von Signalmessungen oder auch Sätze natürlicher Sprache. Wir werden uns hier allerdings ausschließlich der Anwendung in der Bildanalyse widmen. Eine typische Aufgabenstellung ist hier die Klassifikation eines auf einem Bild dargestellten Objekts, z. B. die Erkennung, ob ein Bild eines Tieres einen Hund oder eine Katze darstellt. Für dieses Problem kann man natürlich auch ein „normales" *Feedforward*-Netzwerk benutzen, üblicherweise wird man damit aufgrund von Überanpassung aber keine zufriedenstellenden Ergebnisse erreichen. Die Architektur eines CNNs integriert eine Reihe von zusätzlichen Annahmen, die die Generalisierbarkeit der Ergebnisse stark erhöhen. Wir werden uns formaler mit diesen Annahmen in Abschnitt 5.2.4 beschäftigen und werden im Folgenden zunächst nur informell einen Aspekt der Bildanalyse diskutieren, um die zentrale Methode von CNNs, die *Faltungsoperation*, zu motivieren.

Um ein Bild eines Tieres entweder als Hund oder Kat-

ze zu klassifizieren, kann eine Teilaufgabe beispielsweise daraus bestehen, zu erkennen, ob das Tier Schnurrhaare besitzt oder nicht (falls es welche besitzt, ist die Klassifikation als Katze wohl wahrscheinlicher). Nun können Schnurrhaare auf einem Bild an einer beliebigen Stelle zu finden sein, das Tier könnte beispielsweise stehen oder liegen, nach links oder nach rechts schauen. Weiterhin können die Schnurrhaare in jedem beliebigen Winkel auf dem Bild liegen. Um die Erkennung von Schnurrhaaren und weiteren Merkmalen mit einem „normalen" *Feedforward*-Netzwerk zu realisieren, benötigt es sehr viele Parameter, die an verschiedenen Stellen des Bildes ähnlich trainiert werden, um potentiell überall gewisse Merkmale erkennen zu können. Weiterhin sind üblicherweise für die Erkennung vieler Merkmale nur Pixel in der unmittelbaren Umgebung wichtig, d. h., für die Erkennung von Schnurrhaaren im unteren linken Bereich sind die Pixel im oberen rechten Bereich nicht von Relevanz. Die entsprechenden Schichten des Netzwerks müssen also nicht voll vernetzt sein. Zentral für die Implementierung der genannten Aspekte ist die *Faltungsoperation* (engl. *convolution operation*), die wir uns in ihrer Grundform in Abschnitt 5.2.1 anschauen. In Abschnitt 5.2.2 werden wir uns weitergehenden Aspekten zur Faltung widmen, in Abschnitt 5.2.3 die zweite wichtige Operation von CNNs, die *Pooling*-Operation, betrachten und in Abschnitt 5.2.4 die Gesamtarchitektur eines CNNs diskutieren.

5.2.1 Faltung

Die (kontinuierliche) Faltungsoperation $*$ ist im einfachsten Fall eine mathematische Operation, die zwei reellwertige Funktionen $i, k : \mathbb{R} \to \mathbb{R}$ als Eingabe nimmt und

eine reellwertige Funktion als Ausgabe liefert, d. h.,

$$* : (\mathbb{R} \to \mathbb{R}) \times (\mathbb{R} \to \mathbb{R}) \to (\mathbb{R} \to \mathbb{R})$$

Konkret ist $*$ definiert als

$$(i * k)(x) = \int_{-\infty}^{\infty} i(y)k(x-y)dy$$

Die Funktion i heißt dabei *Eingabefunktion* und k heißt *Kernelfunktion*[44] (oder auch Filterfunktion). Die Ausgabe $i * k$ heißt (im CNN-Kontext) auch *Feature Map* von i bzgl. k.

Beispiel 94. Eine einfache Anwendung der Faltungsoperation ist das *Denoising*, d. h., das Entfernen von Rauschen aus einem Signal. Angenommen, i ist eine Funktion, die uns zu jedem Zeitpunkt $x \in \mathbb{R}$ die gemessene Höhe eines Flugzeugs angibt. Üblicherweise sind unsere Messinstrumente nicht perfekt und das gemessene Signal ist verzerrt, siehe Abbildung 86(a). Sei nun k die Funktion

$$k(x) = \begin{cases} 1 - \frac{1}{2}x & \text{für } 0 \leq x \leq 2 \\ 0 & \text{sonst} \end{cases}$$

Der Funktionsverlauf von k ist in Abbildung 86(b) dargestellt. Die Funktion k kann hier als Wahrscheinlichkeitsdichte interpretiert werden (vergewissern Sie sich bitte, dass das Integral über k tatsächlich 1 ist), die dafür sorgt, dass in der Funktion $i * k$ die Funktionswerte von i über die letzten Punkte gemittelt werden, d. h., $(i * k)(x)$ ist der Mittelwert der Werte $[i(x-2), i(x)]$ bzgl. der Gewichtung k. Hier gibt k eine stärkere Gewichtung der Punkte nahe x und eine absteigende Gewichtung Richtung $x-2$. Abbildung 86(c) skizziert die Funktion $i * k$, die damit eine entrauschte Version der Funktion i darstellt.

[44] Nicht zu verwechseln mit Kernelfunktionen im Zusammenhang mit *Support Vector Machines*, siehe Unterkapitel 2.3.

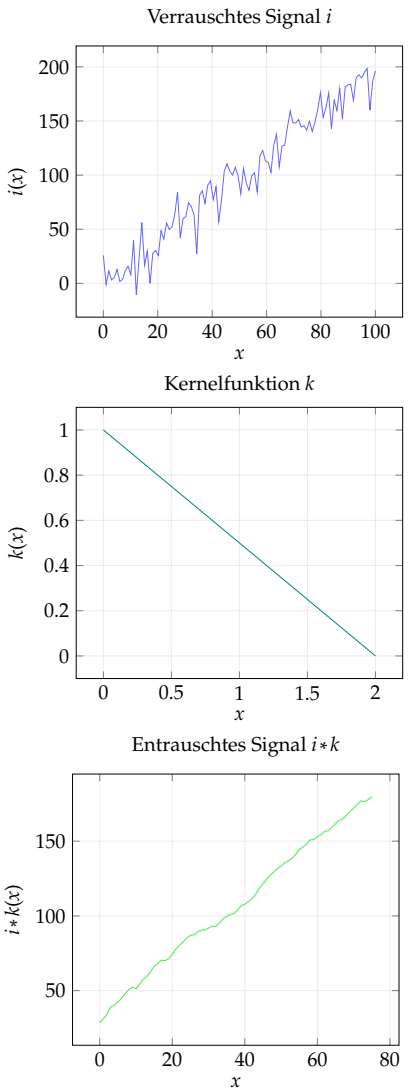


Abbildung 86: Entfernen von Rauschen in einem Signal mittels Faltung: (Links) Originalsignal, (Mitte) Kernelfunktion, (Rechts) entrauschtes Signal.

In unserem Kontext ist die *diskrete* Faltungsoperation relevanter als die obige kontinuierliche Faltungsoperation. Gegeben zwei auf den ganzen Zahlen definierten Funktionen $I, K : \mathbb{Z} \to \mathbb{R}$ ist die Faltung $I * K$ definiert durch

$$(I * K)(n) = \sum_{m=-\infty}^{\infty} I(m)K(n-m)$$

Da wir in unserem Anwendungsfall der Bildanalyse mit zweidimensionalen Daten arbeiten, ist hier insbesondere die Verallgemeinerung auf zweiwertige Funktionen von Relevanz. Sind Funktionen $I, K : \mathbb{Z} \times \mathbb{Z} \to \mathbb{R}$ gegeben, so ist $I * K$ definiert durch

$$(I * K)(n_1, n_2) = \sum_{m_1=-\infty}^{\infty} \sum_{m_2=-\infty}^{\infty} I(m_1, m_2)K(n_1 - m_1, n_2 - m_2) \tag{47}$$

Beispiel 95. Eine konkrete Anwendung von (47) in der Bildverarbeitung ist der Glättungsfilter (und speziell der *Gauß*-Filter), eine Methode zum Weichzeichnen bzw. Entfernen von Rauschen in einem Bild. Ähnlich wie in Beispiel 94 ist die Kernelfunktion so definiert, dass der neue Wert an einem Pixel ein Mittelwert seiner Umgebung ist. Eine konkrete Instanz einer solchen Kernelfunktion K ist beispielsweise definiert durch

$$K(i,j) = \begin{cases} \frac{1}{2} & \text{bei } i = j = 0 \\ \frac{1}{16} & \text{bei } i, j \in \{-1, 0, 1\} \text{ und nicht } i = j = 0 \\ 0 & \text{sonst} \end{cases}$$

für alle $i, j \in \mathbb{N}$. Diese Kernelfunktion bildet den Helligkeitswert eines Pixels auf einen Mittelwert ab, wobei der aktuelle Wert des Pixels (an Position $(0,0)$) mit $1/2$ gewichtet wird und die Werte aller direkt umliegenden Pixel (horizontal, vertikal und diagonal) mit jeweils $1/16$

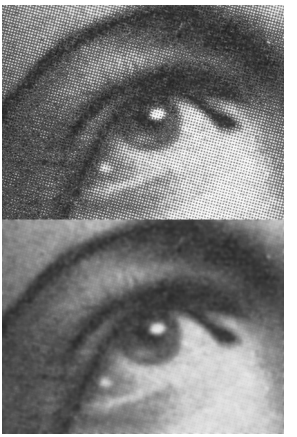


Abbildung 87: Anwendung eines Gauß-Filters auf ein Graustufenbild (oben Original, unten Resultat); Bildquelle: `https://commons.wikimedia.org/wiki/File:Halftone,_Gaussian_Blur.jpg`

gewichtet werden. Ein konkretes Beispiel für die Anwendung eines Gauß-Filters, d. h. eines Glättungsfilters, bei dem die Werte in K den Werten einer Gauß-Verteilung entsprechen, ist in Abbildung 87 zu sehen.

Da die Faltungsoperation üblicherweise eine „lokale“ Operation ist (in dem Sinne, dass eine Merkmalsbestimmung eines Pixels nur seine Umgebung, etwa alle maximal 5 Schritte entfernten Pixel, miteinbezieht), wird die Funktion $K(i, j)$ nur für einige wenige Indizes i, j ungleich Null sein. Deshalb beschreibt man K, genau wie die Eingabefunktion I, üblicherweise durch eine Matrix und die Faltungsoperation kann dann als Matrixoperation interpretiert werden. Diese Interpretation ist in Abbildung 88 veranschaulicht. Seien M_I, M_K und M_{I*K} die Matrixrepräsentationen der Eingabefunktion, Kernelfunktion und der resultierenden Feature Map. Bei der

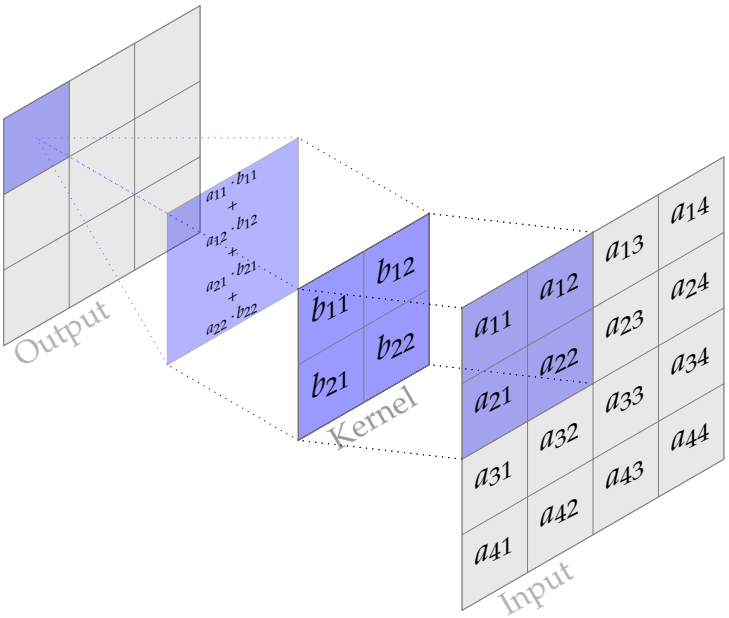


Abbildung 88: Die Faltungsoperation als Operation auf Matrixdarstellungen der Eingabefunktion und der Kernelfunktion (ohne Padding, siehe dazu Abschnitt 5.2.2).

Repräsentation als Matrixoperation gibt es einige Freiheiten, wie mit den Randpixeln verfahren werden soll. In Abbildung 88 ist die Ausgabe beschränkt auf solche Pixel, bei denen M_K vollständig auf Elemente in M_I angewendet werden kann (anschaulich gesprochen: die Matrix M_K darf nicht aus M_I „herausragen“). Ist $M_I \in \mathbb{R}^{n \times m}$ und $M_K \in \mathbb{R}^{n' \times m'}$, so gilt in diesem Fall $M_{I*K} \in \mathbb{R}^{(n-n'+1)\times(m-m'+1)}$ (üblicherweise ist auch n' signifikant kleiner als n und m' ist signifikant kleiner als m). Diesen Ansatz nennen wir *Faltung ohne Padding* oder auch *Faltung mit validem Padding*. Wir schauen uns Alternativen dazu in Abschnitt 5.2.2 an.

Beispiel 96. Wir führen Beispiel 95 fort. Die Matrixdarstellung $M_K \in \mathbb{R}^{3\times 3}$ von K ist gegeben durch

$$M_K = \begin{pmatrix} \frac{1}{16} & \frac{1}{16} & \frac{1}{16} \\ \frac{1}{16} & \frac{1}{2} & \frac{1}{16} \\ \frac{1}{16} & \frac{1}{16} & \frac{1}{16} \end{pmatrix}$$

Sei weiterhin die Eingabefunktion I (d. h., das Eingabebild) charakterisiert durch eine Matrix $M_I \in \mathbb{R}^{6\times 6}$ mit

$$M_I = \begin{pmatrix} 12 & 24 & 28 & 105 & 250 & 251 \\ 54 & 43 & 43 & 42 & 221 & 241 \\ 67 & 50 & 89 & 92 & 210 & 211 \\ 105 & 156 & 178 & 115 & 201 & 187 \\ 19 & 78 & 125 & 108 & 52 & 188 \\ 112 & 101 & 154 & 205 & 198 & 192 \end{pmatrix}$$

Die *Feature Map* $J = I * K$ kann dann durch eine Matrix $M_J \in \mathbb{R}^{4\times 4}$ dargestellt werden (wir benutzen kein Padding). Insbesondere gilt für den Eintrag $(M_J)_{1,1}$, dass sich dieser aus der elementweisen Matrixmultiplikation von K mit der 3×3-Untermatrix von M_I, die nur aus den ersten drei Zeilen und den ersten drei Spalten besteht, zusammensetzt. Mit anderen Worten, es gilt

$$\begin{aligned}(M_J)_{1,1} &= \frac{1}{16}12 + \frac{1}{16}24 + \frac{1}{16}28 + \frac{1}{16}54 + \frac{1}{2}43 + \\ &\quad \frac{1}{16}43 + \frac{1}{16}67 + \frac{1}{16}50 + \frac{1}{16}89 \\ &\approx 44\end{aligned}$$

Insgesamt berechnet sich die Matrix M_J zu

$$M_J \approx \begin{pmatrix} 44 & 51 & 86 & 198 \\ 71 & 89 & 115 & 187 \\ 122 & 140 & 123 & 173 \\ 98 & 131 & 131 & 113 \end{pmatrix}$$

Der Glättungsfilter aus den vorherigen Beispielen ist für das Problem der Klassifikation von Bildinhalten weniger wichtig. Kernelfunktionen können allerdings so definiert werden, dass sie beispielsweise die Präsenz von Kanten oder Ecken, und damit auch komplexere geometrische Formen, erkennen können. Die konkreten Parameter der Kernelfunktion (d. h., die Matrixeinträge) werden dabei während des Lernens gesetzt. Eine Faltungsschicht in einem CNN besteht üblicherweise aus einer Menge verschiedener Kernelfunktionen, die gleichzeitig auf das gesamte Bild angewendet werden und damit in der Lage sind, verschiedene Merkmale zu erkennen, siehe Abbildung 89. Hierbei wird auch direkt ein Vorteil bei der Verwendung der Faltungsoperation in neuronalen Netzwerken deutlich, nämlich die geringere Anzahl an zu lernenden Parametern. Besteht beispielsweise das Eingangsbild aus 100×100 Pixeln (in Graustufen), also 10000 Eingangsneuronen, und wenden wir in der ersten Schicht 10 verschiedene Kernelfunktionen der Größe 5×5 an, so besteht die zweite Schicht aus 10 *Feature Maps*, deren Größe jeweils 96×96 ist. Damit haben wir insgesamt $96 \cdot 96 \cdot 10 = 92160$ Neuronen in der zweiten Schicht. Bei Verwendung eines vollvernetzen neuronalen Netzwerks gäbe es $10000 \cdot 92160 = 921{,}600{,}000$ Kanten, und damit genau so viele zu lernende Parameter zwischen den beiden Schichten. Bei 10 verschiedenen Kernelfunktionen der Größe 5×5 sind dies im CNN allerdings nur $5 \cdot 5 \cdot 10 = 250$ verschiedene zu lernende Parameter. Wir werden die genaue Einbettung von Faltungsschichten in die Architektur von CNNs in Abschnitt 5.2.4 weiter diskutieren. Zunächst beschäftigen wir uns noch mit zwei weiteren Aspekten der Faltung selbst, nämlich dem *Padding* und dem *Stride*.

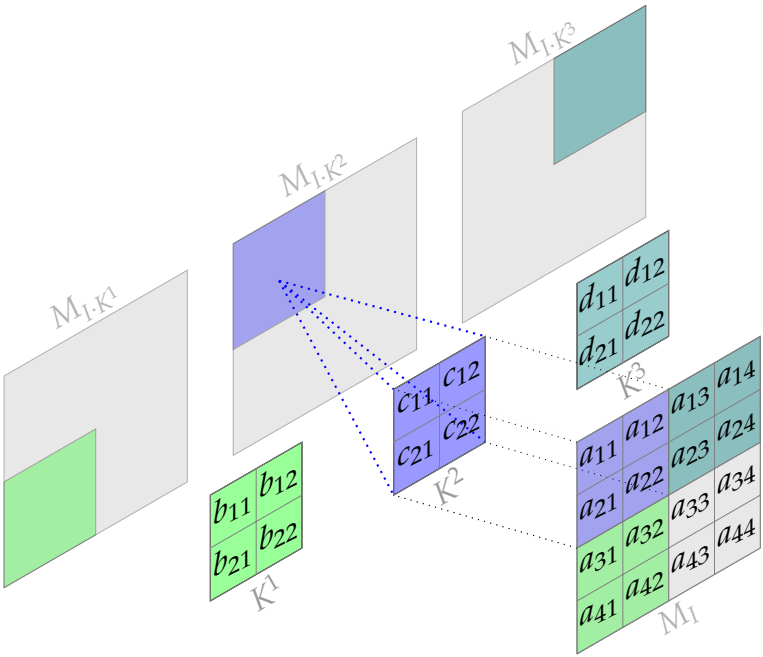


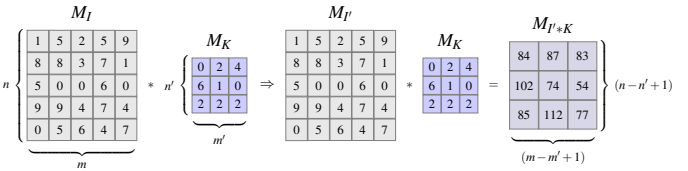
Abbildung 89: Gleichzeitige Anwendung verschiedener Faltungsoperationen.

5.2.2 Padding und Stride

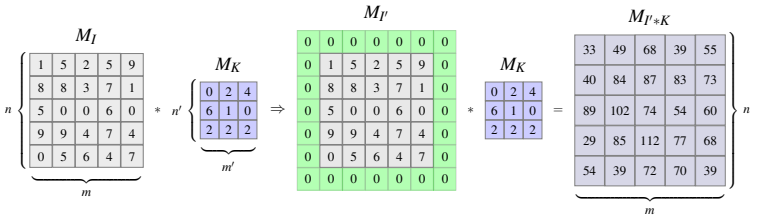
Der oben beschriebene Ansatz der Faltung durch Matrixoperationen erzeugt bei Eingabe einer Matrix $M_I \in \mathbb{R}^{n \times m}$ und einer gegebenen Kernelmatrix $M_K \in \mathbb{R}^{n' \times m'}$ eine Feature Map M_{I*K} der Größe $(n - n' + 1) \times (m - m' + 1)$. Üblicherweise sind n und m relativ große Werte und n' und m' im Verhältnis zu n und m relativ klein. Der „Verlust“ an Auflösung bei der Transformation von M_I zu M_{I*K} ist deshalb auch relativ gering. Dennoch ist es oft wünschenswert, dass die Auflösung bei Anwendung einer Faltungsoperation erhalten bleibt (oder manchmal sogar erhöht wird). Zum einen beinhalten CNNs typi-

scherweise mehrere Faltungsschichten und eine iterative Verkleinerung der Auflösung kann dann zu einem signifikanten Verlust an Ausdrucksstärke führen. Zum anderen ist der Einfluss der „Randpixel" auf die resultierende Feature Map beim obigen Ansatz geringer als die der „inneren" Pixel (da erstere in weniger Berechnungen benutzt werden als letztere). Dies kann dazu führen, dass sich Merkmale in Randnähe schlechter erkennen lassen. Um diese Probleme zu umgehen, gibt es die sogenannten *Padding*-Methoden, die die Eingabematrix M_I künstlich vergrößern. Neben dem schon bekannten *validen Padding* (das keinem Padding entspricht), gibt es noch das *halbe Padding* (engl. *half padding*, auch *same padding* genannt) und das *vollständige Padding* (engl. *full padding*). Beim halben Padding wird die Matrix M_I oben um $\lceil (n'-1)/2 \rceil$ und unten um $\lfloor (n'-1)/2 \rfloor$ Zeilen, sowie links um $\lceil (m'-1)/2 \rceil$ und rechts um $\lfloor (m'-1)/2 \rfloor$ Spalten erweitert. Dies führt dazu, dass die Matrix M_{I*K} wieder das Format $n \times m$ hat. Beim vollständigen Padding wird die Matrix M_I oben und unten um jeweils $n'-1$ Zeilen und rechts und links um jeweils $m'-1$ Spalten erweitert. Dies führt dazu, dass jede Zelle der Matrix M_I gleich oft in Berechnungen von M_{I*K} vorkommt (nämlich $m'n'$-oft), die resultierende Matrix M_{I*K} hat dann die Dimension $(n+n'-1) \times (m+m'-1)$. Jede dieser Padding-Varianten ist parametrisiert durch die Art, wie die Werte der neuen Zellen bestimmt sind. Die einfachste und gebräuchlichste Methode ist das *Zero-Padding*, bei dem alle neuen Zellen den Wert 0 erhalten. Eine andere Methode kann darin bestehen, die Werte aus den Randzeilen und -spalten zu kopieren und nur die neuen Eckbereiche mit Nullen aufzufüllen. Abbildung 90 veranschaulicht die verschiedenen Padding-Methoden.

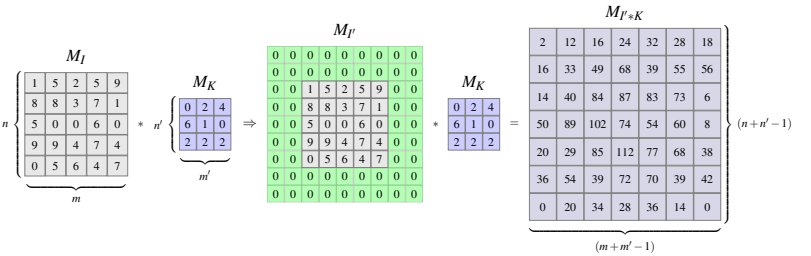
Ein weiterer Punkt, an dem eine Faltungsoperation parametrisiert werden kann, ist der *Stride* (dt. *Schrittweite*). Bei der normalen Faltungsoperation, siehe Abbil-



(a)



(b)



(c)

Abbildung 90: Verschiedene Padding-Methoden: (a) valides Padding (kein Padding), (b) Half-Zero-Padding, (c) Full-Zero-Padding

dung 88, wird die Matrix M_K bei der Berechnung der einzelnen Zellen von M_{I*K} jeweils um eine Zeile/Spalte in M_I „weiterbewegt", d. h., jede $n' \times m'$-Untermatrix von M_I wird bei der Berechnung von M_{I*K} verwendet. Dies ent-

spricht einem Stride von 1. Bei einem Stride von $k \in \mathbb{N}$, $k > 1$ wird die Matrix M_K stattdessen in jedem Schritt k Zeilen und/oder Spalten bewegt. Dies resultiert in einer kleineren Feature Map M_{I*K}. Abbildung 91 illustriert die Faltungsoperation mit verschiedenen Stride-Längen. Üblicherweise wählt man einen Stride von 1 oder 2, höhere Werte können aber insbesondere für ressourcenbeschränkte Aufgaben sinnvoll sein.

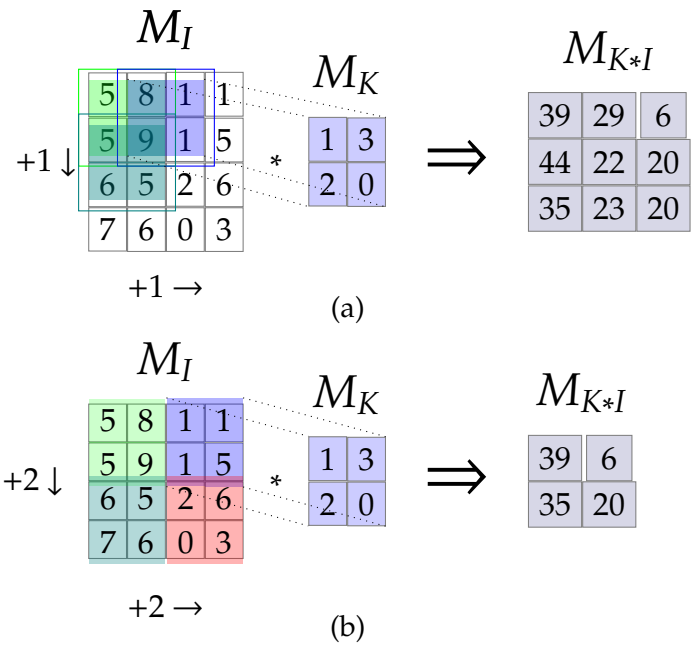


Abbildung 91: Die Faltungsoperation mit verschiedenen Stride-Längen: (a) Faltung mit 2×2-Kernelfunktion und Stride von 1, (2) Faltung mit 2×2-Kernelfunktion und Stride von 2

5.2.3 Die Pooling-Operation

Das Resultat M_{I*K} einer Faltungsoperation wird üblicherweise noch durch eine Aktivierungsfunktion geleitet, bevor es in einem CNN weitergereicht wird. Für CNNs hat sich hierbei die ReLU-Funktion h^{ReLU} mit $h^{\text{ReLU}}(x) = \max\{0, x\}$ aufgrund diverser guter Eigenschaften als Standard etabliert und es wird kaum eine andere Aktivierungsfunktion verwendet. Die Funktion wird komponentenweise auf M_{I*K} angewendet und erzeugt somit eine Matrix $\hat{M}_{I*K}$ mit identischer Dimension wie M_{I*K}. Die Matrix $\hat{M}_{I*K}$ wird dann anschließend in einer *Pooling*-Schicht weiterverarbeitet, die wir im Folgenden diskutieren werden.

Genau wie die Faltungsoperation können wir die Pooling-Operation als Matrixoperation verstehen. Anders als bei der Faltungsoperation ist es bei der Pooling-Operation erwünscht, dass die Auflösung der Ausgabematrix kleiner ist als die der Eingabematrix. Die Pooling-Operation dient dazu, die Informationen in der Eingabematrix zusammenzufassen, um anschließende Operationen effizienter zu gestalten und redundante Informationen in der Eingabematrix zu entfernen. Ein konkretes Beispiel (und auch die gebräuchlichste Ausprägung) für eine Pooling-Operation ist das *Max-Pooling*. Ähnlich wie bei der Faltungsoperation wird auch hier mit einem „Fenster" über verschiedene Ausschnitte der Eingabematrix gefahren und ein Wert berechnet, hier konkret das Maximum. Wie bei der Faltung nennen wir dieses Fenster auch *Filter* und dessen Größe *Filtergröße*. Typische Parameter beim Max-Pooling sind eine Filtergröße von 2×2 bei einem Stride von 2 (der genauso definiert ist wie bei der Faltung), Abbildung 92 zeigt eine exemplarische Anwendung eines solchen Max-Poolings. Die Motivation hinter dem (Max-)Pooling ist es, dass kleine Verschiebungen eines Bildes

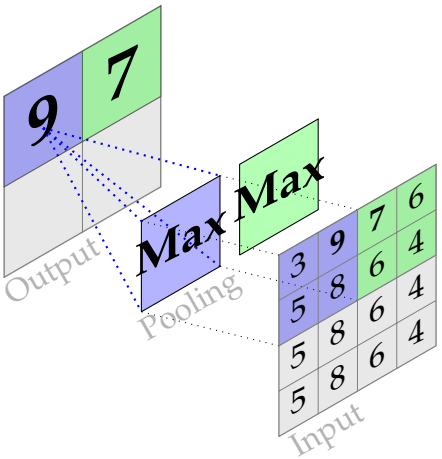


Abbildung 92: Max-Pooling mit Filtergröße 2×2 und Stride 2.

keine Auswirkung auf die Klassifikationsaufgabe haben sollten. Wird ein Bild um einen Pixel nach links oder rechts verschoben, so ändert dies nichts am dargestellten Inhalt. Das (Max-)Pooling implementiert diese Invarianz bzgl. geringer Verschiebungen (engl. *translation invariance*), indem innerhalb des Filters nur das stärkste Signal durchgereicht wird.

5.2.4 CNN-Architektur

Wir haben nun alle Grundbausteine zusammen, um die Gesamtarchitektur eines CNNs zu diskutieren. Eine einfache typische Beispielarchitektur ist in Abbildung 93 dargestellt. Die Eingabe dieses CNNs besteht hier aus einem Farbbild mit 96×96 Pixeln. Um die Farbe zu kodieren haben wir hier für jeden der Kanäle Rot, Grün und Blau jeweils eine 96×96 Pixel große Eingabe. In einem CNN wechseln sich zu Anfang Faltungs- und Poolingschich-

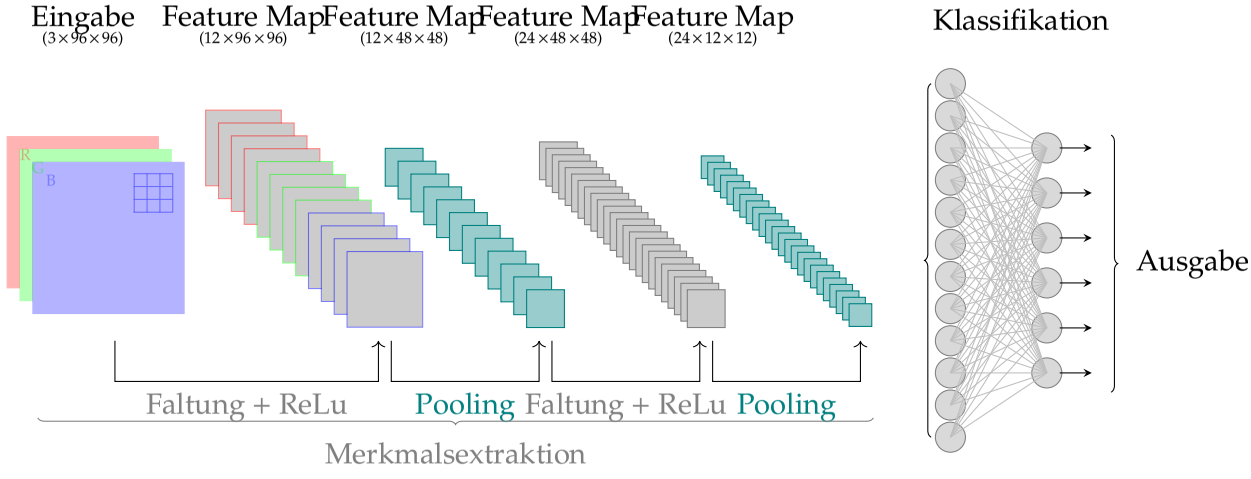


Abbildung 93: Beispielarchitektur eines CNNs.

ten ab. Im CNN in Abbildung 93 haben wir insgesamt zwei Faltungs- und Poolingschichten hintereinander. Im dargestellten CNN benutzen wir halbes Padding, um die Größe der Eingabe beizubehalten (und einen Stride von 1), und die erste Faltungsschicht wendet 4 verschiedene Faltungsoperationen gleichzeitig auf alle drei Eingabematrizen an. Dies resultiert in einer Gesamtzahl von 12 Feature Maps der Größe 96×96 (dies entspricht also 110592 Neuronen). Anschließend werden alle Werte zunächst durch die ReLU-Funktion und dann an die erste (Max-)Poolingschicht geleitet. Im Beispiel hat diese eine Filtergröße von 2×2 und einen Stride von 2. Dies resultiert in derselben Anzahl Feature Maps (12), die nun aber jeweils eine Größe von 48×48 haben. Es folgt eine weitere Faltungsschicht mit 2 verschiedenen Faltungsoperationen. Dies verdoppelt die Anzahl der Feature Maps auf 24, die Auflösung von 48×48 behalten wir allerdings bei (wir benutzen wieder halbes Padding mit Stride 1). Es folgt eine weitere (Max-)Poolingschicht (inklusive vorgelagerter ReLU-Anwendung) mit Filtergröße 4×4 und Stride 4, dies resultiert in 24 Feature Maps der Größe 12×12 (dies entspricht insgesamt 3456 Neuronen). Im letzten Teil des CNNs befindet sich noch ein vollständig vernetztes *Feedforward*-Netzwerk, das die eigentliche Klassifikationsaufgabe löst (die vorangegangen Schichten entsprechen prinzipiell nur der Merkmalsextraktion). Im Beispiel gibt es eine Schicht von 12 Neuronen, die mit ihrer Vorgängerschicht und der nachfolgenden Ausgabeschicht voll vernetzt ist. Die Ausgabeschicht in diesem Beispiel ist für die Mehrklassenklassifikation konzipiert und entspricht der Beschreibung aus Abschnitt 5.1.1.

Die Architektur des obigen CNNs dient nur der Veranschaulichung, reale CNNs können über weitaus mehr Faltungs- und Poolingschichten, als auch über mehr Schichten im vollvernetzen Teil, verfügen. Weiter-

hin ist die Anzahl der verschiedenen Faltungsoperationen üblicherweise höher. Nichtsdestotrotz ist ein CNN eine sehr effektive Struktur und hat im allgemeinen eine signifikant geringere Anzahl an Parametern als ein vollvernetztes neuronales Netzwerk ähnlicher Größe. Der Grund dafür sind die zuvor bereits erwähnten Annahmen zur Nichtlokalität von Merkmalen in (insbesondere) Bilddaten und die daraus resultierenden Designentscheidungen für CNNs:

- Im Gegensatz zu einem vollvernetzten *Feedforward*-Netzwerk gibt es nur relativ wenig Kanten zwischen den einzelnen Schichten eines CNNs (Stichwort engl. *sparse interaction*). Beispielsweise sind die Eingabepixel des Bildes in der oberen linken Ecke nur mit den Pixeln in der Nachbarschaft der oberen linken Ecke in den Feature Maps der ersten Faltungsschicht verbunden.

- Jede Faltungsoperation einer Schicht hat eine fixe Menge an Parametern, die für die Berechnung der Faltung auf dem gesamten Bild verwendet wird. Mit anderen Worten, es gibt eine ganze Menge an Kanten zwischen den einzelnen Schichten eines CNNs, die das gleiche Kantengewicht haben (Stichwort engl. *parameter sharing*).

Diese beiden Aspekte führen dazu, dass das CNN eine vergleichsweise geringe Anzahl an Parametern hat, die während des Lernens trainiert werden. Das Lernen dieser Parameter selbst geschieht nahezu identisch zu allgemeinen *Feedforward*-Netzwerken unter Verwendung von (beispielsweise) Stochastic Gradient Descent und dem Backpropagation-Algorithmus (siehe Abschnitt 5.1.3). Es muss einzig darauf geachtet werden, dass die Gewichte von verschiedenen Kanten, die sich ein Gewicht teilen,

nur einmal während eines Gradient Descent-Schrittes aktualisiert werden.

5.3 Rekurrente Neuronale Netzwerke

Rekurrente neuronale Netzwerke (RNNs) sind eine spezielle Form von Künstlichen neuronalen Netzwerken, die für die Verarbeitung von Sequenzen konzipiert sind.

5.3.1 Motivation und Grundlagen

Schauen wir uns dazu das prototypische Problem der *Wortvorhersage* an, das beispielsweise bei der Erstellung von Nachrichtentexten auf Smartphones auftaucht. Gegeben den Anfang eines Satzes wie „Die Katze jagt die. . . “, ist es die Aufgabe eines Algorithmus zur Wortvorhersage, das wahrscheinlichste nächste Wort (oder die wahrscheinlichsten nächsten Wörter) vorherzusagen, in diesem Beispiel also vermutlich „Maus“. Dieses Problem kann zunächst als Klassifikationsproblem modelliert werden. Ist $\Sigma = \{w_1, \ldots, w_n\}$ die Menge aller Wörter, so kann jedes Wort $w_i \in \Sigma$, $i = 1, \ldots, n$, als ein Vektor $x_{w_i} \in \mathbb{R}^n$, repräsentiert werden, bei dem alle Komponenten außer der i-ten 0 sind und die i-te Komponente 1 ist, also

$$x_{w_i} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \leftarrow i\text{-te Position}$$

Diese Art der Kodierung nennt man auch *One-Hot*-Kodierung und eignet sich üblicherweise für die Verarbeitung bei neuronalen Netzwerken besser als naive Kodierungsmethoden (wie beispielsweise die Kodierung von Wörtern durch verschiedene Zahlen). Die Eingabe des

Klassifikationsalgorithmus ist dann eine Sequenz $x = (x^{(1)}, \ldots, x^{(m)})$ solcher Vektoren und die Ausgabe ein Vektor $o \in \mathbb{R}^n$, der (im besten Fall) die Vorhersagewahrscheinlichkeit der einzelnen Wörter beinhaltet.

Beispiel 97. Sei $\Sigma_1 = \{\text{die}, \text{katze}, \text{jagt}, \text{maus}, \text{lampe}\}$ unser Vokabular (wir ignorieren hier Groß- und Kleinschreibung). Dann kann der Teilsatz „Die Katze jagt die…" repräsentiert werden als

$$x = ((1,0,0,0,0)^T, (0,1,0,0,0)^T, (0,0,1,0,0)^T, (1,0,0,0,0)^T)$$

und eine mögliche Ausgabe eines Klassifikationsalgorithmus wäre

$$o = (0, 0.1, 0, 0.85, 0.05)^T$$

und würde damit „Maus" die größte Wahrscheinlichkeit zuordnen (und „Katze" und „Lampe" auch kleine positive Wahrscheinlichkeiten, da diese grammatikalisch zumindest noch Sinn ergeben würden).

Für eine Anwendung klassischer *Feedforward*-Netzwerke ergibt sich aus der obigen Problembeschreibung direkt das erste Problem: die Eingabelänge, d. h., die Anzahl der Wörter in der gegebenen Teilsequenz, ist nicht notwendigerweise fest bestimmt und auch nicht notwendigerweise beschränkt. Die bisher betrachteten Architekturen für neuronale Netzwerke verfügten dagegen über eine fixe Anzahl an Eingaben. Eine weitere Herausforderung, die wir in ähnlicher Form bereits für CNNs in Unterkapitel 5.2 diskutiert haben, ist die flexible Handhabung von *Lokalität*. Die Vorhersage des fünften Wortes einer Sequenz sollte konzeptuell genauso gehandhabt werden wie die Vorhersage des achten Wortes. Die Parameter des Netzwerks sollten also auch in ähnlicher Weise wie bei den CNNs *geteilt* werden.

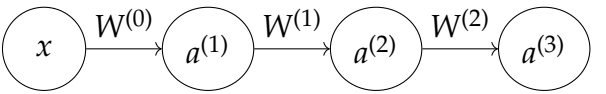


Abbildung 94: Berechnungsgraph für ein einfaches dreischichtiges *Feedforward*-Netzwerk.

Die Grundidee von rekurrenten neuronalen Netzwerken ist es, die Eingabesequenz wortweise abzuarbeiten. In jedem Schritt wird ein Wort gelesen und verarbeitet, sowie eine Ausgabe produziert und ein „Gedächtnis" aktualisiert. Sowohl das Gedächtnis als auch das nächste Wort werden wieder in das Netzwerk gespeist und der Prozess wird iteriert, bis die Sequenz vollständig gelesen wurde. Um diese Idee etwas zu formalisieren, benutzen wir das Konzept des *Berechnungsgraphen* (engl. *computational graph*), das die Funktionsweise von insbesondere neuronalen Netzwerken geeignet abstrahieren kann. Abbildung 94 zeigt beispielsweise einen Berechnungsgraphen für ein einfaches dreischichtiges *Feedforward*-Netzwerk, das dem Netzwerk aus Abbildung 8 in Unterkapitel 5.1 entspricht. Der Eingabevektor x wird hier zunächst mit der Gewichtsmatrix $W^{(0)}$ multipliziert und anschließend durch eine entsprechende Aktivierungsfunktion geleitet (die Anwendung der Aktivierungsfunktion ist hier allerdings nicht explizit dargestellt). Dies resultiert im Vektor $a^{(1)}$, der die Aktivierungswerte der ersten Schicht enthält. Die weitere Berechnung erfolgt analog. Zu beachten ist hier, dass insbesondere von der Anzahl der Neuronen in jeder Schicht abstrahiert wird.

Abbildung 95 zeigt den Berechnungsgraphen eines einfachen rekurrenten neuronalen Netzwerkes. Zur Spezifikation der Architektur eines Netzwerks wird dazu die

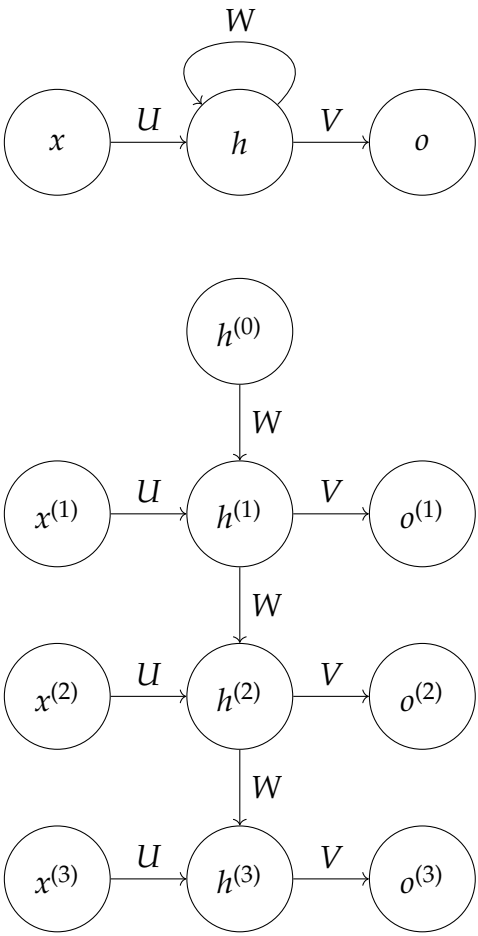


Abbildung 95: Berechnungsgraph für ein einfaches rekurrentes neuronales Netzwerk; oben ist die komprimierte Darstellung und unten die „entfaltete“ Darstellung bei Eingabe einer Sequenz $x = (x^{(1)}, x^{(2)}, x^{(3)})$.

in Abbildung 95 links dargestellte Repräsentation verwendet. Die Eingabe führt hier unter Anwendung einer Parametermatrix U zu der Bestimmung eines Vektors h.

Dieser Vektor modelliert den versteckten Zustand des Netzwerks (engl. *hidden state*, daher auch die Verwendung des Bezeichners h). Dieser wird zum einen zur Bestimmung der Ausgabe o (unter Verwendung einer weiteren Gewichtsmatrix V) verwendet, als auch (unter Verwendung einer Gewichtsmatrix W) zur Bestimmung des nächsten versteckten Zustands des Netzwerks benutzt. Abbildung 95 rechts zeigt die „entfaltete" Darstellung des Netzwerks bei Eingabe einer Sequenz $x = (x^{(1)}, x^{(2)}, x^{(3)})$. Ausgehend von einem initialen Zustandsvektor $h^{(0)}$ (eine übliche Initialisierung könnte hier aus dem Nullvektor bestehen) und dem ersten Element $x^{(1)}$ wird hier der erste versteckte Zustand $h^{(1)}$ bestimmt. Eine einfache Realisierung dieser Operation besteht darin, dass wir die Matrizen U und W und die Vektoren $x^{(1)}$ und $h^{(0)}$ einfach jeweils miteinander konkatenieren und dann multiplizieren.[45] Wir benutzen das Symbol $\circ$ für die spaltenweise Konkatenation zweier Matrizen und die zeilenweise Konkatenation zweier Spaltenvektoren. Genauer, sind $A \in \mathbb{R}^{n \times m}$ und $B \in \mathbb{R}^{n \times m'}$ zwei Matrizen mit gleicher Anzahl an Zeilen, so ist $A \circ B \in \mathbb{R}^{n \times (m+m')}$ die entsprechende Konkatenation:

$$A = \begin{pmatrix} a_{1,1} & \dots & a_{1,m} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \dots & a_{n,m} \end{pmatrix}$$

$$B = \begin{pmatrix} b_{1,1} & \dots & b_{1,m'} \\ \vdots & \ddots & \vdots \\ b_{n,1} & \dots & b_{n,m'} \end{pmatrix}$$

[45] An dieser Stelle ist noch zu erwähnen, dass wir in der gesamten folgenden Darstellung auf Bias-Neuronen verzichten, da diese die mathematische Darstellung verkomplizieren. Bei einer Implementierung sollten diese allerdings immer berücksichtigt werden.

$$A \circ B = \begin{pmatrix} a_{1,1} & \dots & a_{1,m} & b_{1,1} & \dots & b_{1,m'} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & \dots & a_{n,m} & b_{n,1} & \dots & b_{n,m'} \end{pmatrix}$$

Sind weiterhin $v \in \mathbb{R}^m$ und $w \in \mathbb{R}^{m'}$ zwei Spaltenvektoren (potentiell unterschiedlicher Länge), so ist $v \circ w \in \mathbb{R}^{m+m'}$:

$$\begin{aligned} v &= (v_1, \dots, v_m)^T \\ w &= (w_1, \dots, w_{m'})^T \\ v \circ w &= (v_1, \dots, v_m, w_1, \dots, w_{m'})^T \end{aligned}$$

Vergewissern Sie sich, dass für die obigen Definition gilt

$$(A \circ B)(v \circ w) = Av + Bw$$

Sei weiterhin act eine beliebige Aktivierungsfunktion (die bei Anwendung auf einen Vektor komponentenweise angewendet wird). Dann berechnet sich $h^{(1)}$ durch

$$h^{(1)} = \text{act}(Ux^{(1)} + Wh^{(0)})$$

Aus $h^{(1)}$ berechnen wir dann zunächst die erste Ausgabe $o^{(1)}$ via

$$o^{(1)} = \text{act}(Vh^{(1)})$$

Im Allgemeinen gilt für eine Eingabe $x = (x^{(1)}, \dots, x^{(m)})$

$$h^{(i)} = \text{act}(Ux^{(i)} + Wh^{(i-1)}) \tag{48}$$

$$o^{(i)} = \text{act}(Vh^{(i)}) \tag{49}$$

für $i = 1, \dots, m$. Zu beachten ist, dass diese Netzwerkarchitektur mit Eingaben beliebiger Länge umgehen kann, aber eine fixe Anzahl an Parametern besitzt (in den Matrizen U, V, W).

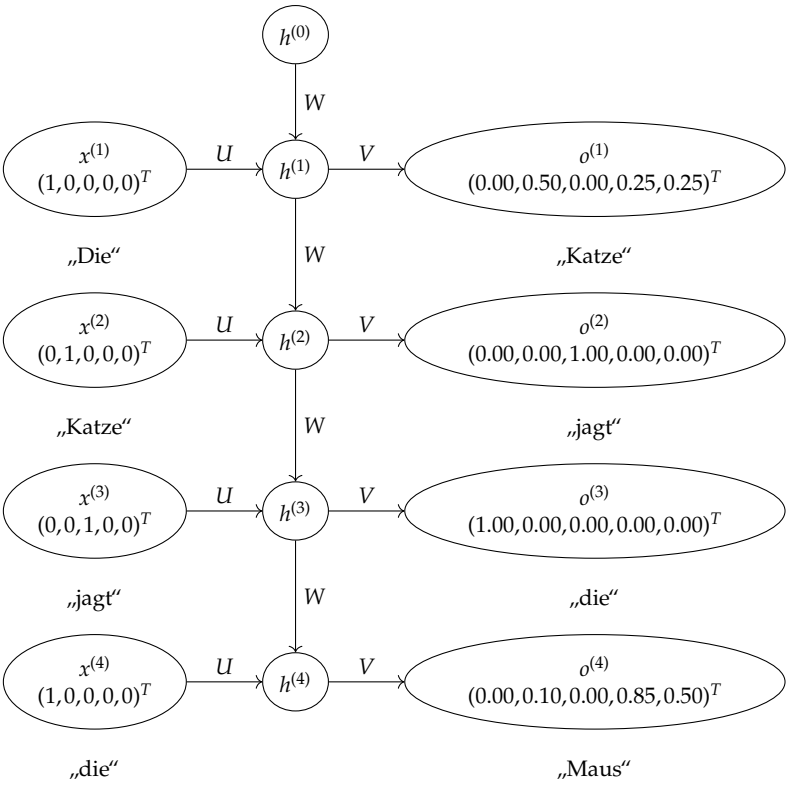


Abbildung 96: Wortvorhersage durch ein RNN bei Eingabe „Die Katze jagt die. . . “.

Beispiel 98. Wir führen Beispiel 97 fort. Abbildung 96 zeigt ein „entfaltetes“ trainiertes RNN für die Wortvorhersage bei Eingabe von „Die Katze jagt die. . . “, repräsentiert durch

$$x = ((1,0,0,0,0)^T, (0,1,0,0,0)^T, (0,0,1,0,0)^T, (1,0,0,0,0)^T)$$

Die letzte Ausgabeschicht $o^{(4)}$ beinhaltet dann die Vorhersage des nächsten Wortes nach der Eingabesequenz. Die Ausgabeschichten $o^{(1)}$ bis $o^{(3)}$ beinhalten die Vorhersagen

der entsprechenden Teilsequenzen, die bei der Anwendung für die Eingabe x ignoriert werden können.

Die Parameter eines rekurrenten neuronalen Netzwerks werden wie bei normalen *Feedforward*-Netzwerken durch den Backpropagationsalgorithmus und einen entsprechenden Optimierungsalgorithmus wie *Stochastic Gradient Descent* gelernt. Ein Datensatz $D = \{X_1, \ldots, X_k\}$ für das Wortvorhersageproblem ist dabei eine Menge von Sätzen $X_i = (x_i^{(1)}, \ldots, x_i^{(m_i)})$ und alle Teilsequenzen eines jeden X_i werden dabei genutzt, die Parameter zu trainieren. Dazu wird das zu lernende RNN entsprechend der um eins verkürzten Länge eines X_i „entfaltet", der Satz X_i an die Eingabe des RNNs angelegt (bis auf das letzte Wort) und die um eins verschobene Sequenz von X_i wird an der Ausgabe erwartet (in ähnlicher Weise wie in Abbildung 96 für den Satz „Die Katze jagt die Maus" dargestellt). Ist das RNN darauf ausgelegt, an der Ausgabe eine Wahrscheinlichkeitsverteilung über die vorherzusagenden Wörter auszugeben, so bietet sich als Kostenfunktion der negative Log-Likelihood an. Ist $o^{(i)}$ der entsprechende Ausgabevektor und das zu erwartende Wort ist x, so ist $(o^{(i)})^T x$ genau die Wahrscheinlichkeit von x in $o^{(i)}$. Sei weiterhin $\phi_{U,V,W}(X_i)^{(j)} = o^j$ für $j = 1, \ldots, m_i$ die Ausgabe des Netzwerkes. Daraus ergibt sich für die Kostenfunktion L^{log} bzgl. eines einzelnen Beispiels $X_i = (x_i^{(1)}, \ldots, x_i^{(m_i)})$:

$$L^{log}(X_i, \phi_{U,V,W}) = -\sum_{j=1}^{m_i - 1} \log\left((\phi_{U,V,W}(X_i)^{(j)})^T x_i^{(j+1)}\right)$$

Wie beim normalen *Backpropagation*-Algorithmus können nun die partiellen Ableitungen bzgl. aller Gewichte berechnet werden. Dabei ist, wie auch schon beim Lernen

von CNNs, allerdings Sorge zu tragen, dass die mehrfachen Verwendungen der Parameter in den Matrizen U, V und W entsprechend beachtet werden und nur einmal aktualisiert werden. Diese Variante der Backpropagation nennt man auch *back-propagation through time*, da die Backpropagation rückwärts durch die Sequenz geht.

5.3.2 Long short-term memory-Netzwerke

Eine Grundidee von RNNs ist, dass die versteckten Schichten $h^{(i)}$ eine Art „Gedächtnis“ für die Verarbeitung einer Sequenz repräsentieren. Für die Worterkennung wird in $h^{(i)}$ beispielsweise das Geschlecht des aktuellen Subjekts gespeichert werden. Beim Lesen eines neuen Wortes wird das Gedächtnis entsprechend nach (48) aktualisiert. Diese Aktualisierungsregel hat allerdings zwei entscheidende Nachteile, wenn die zu verarbeitenden Sequenzen sehr lang sind. Zum einen wird der Einfluss von sehr weit zurückliegenden versteckten Zuständen sehr gering, da (48) wiederholt den vorherigen Zustand mit der Matrix W multipliziert. Das heißt auch, dass sich das RNN schwer weit zurückliegende Informationen merken kann. Haben wir beispielsweise einen Satz wie „Die Katze, die Anna gestern auf der Straße zugelaufen ist, jagt die...“, so ist die korrekte Wortvorhersage hier für ein normales RNN signifikant schwieriger als bei einem Satz wie „Die Katze jagt die...“. Die Information, dass die Katze das Subjekt des Satzes ist, wird bei wiederholter Multiplikation mit W leicht „vergessen“. Zum anderen erschweren die wiederholten Multiplikationen mit W das Lernen via Backpropagation. Lange Sequenzen führen dazu, dass bei der Backpropagation durch die Verbindungen via W die Gradienten in frühen Schichten gegen Null gehen, siehe das Problem des verschwindenden Gradienten in Abschnitt 5.1.4.

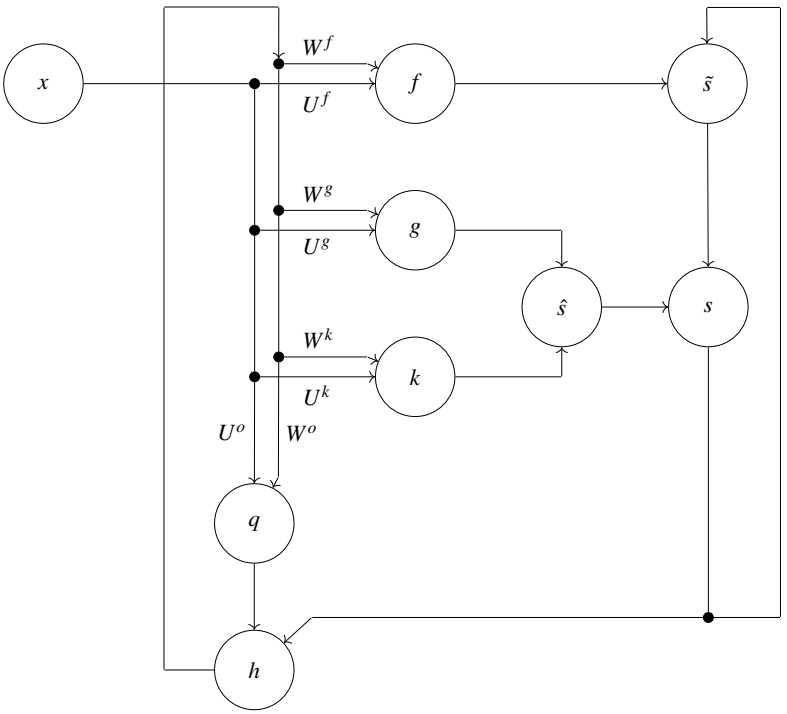


Abbildung 97: Berechnungsgraph eines LSTMs.

Long short-term memory-Netzwerke (LSTM-Netzwerke) lösen die obigen Probleme durch eine intelligentere Aktualisierungsregel. Der Berechnungsgraph eines LSTM-Netzwerks ist in Abbildung 97 dargestellt. Im Gegensatz zu normalen RNNs, gibt es bei LSTMs zwei Vektoren, die das „Gedächtnis" modellieren, den versteckten Zustand h und den *Zellzustand* s. Der Zellzustand s stellt das „Langzeitgedächtnis" dar und wird durch zwei Operationen modifiziert: eine Vergessensoperation (realisiert durch die Zwischenzustände f und $\tilde{s}$) und eine Lernoperation (realisiert durch die Zwischenzustände g, k und $\hat{s}$). Wir beschreiben die Aktualisierungsoperationen nun im

Detail. Eine LSTM ist parametrisiert durch Gewichtsmatrizen $W^f, U^f, W^g, U^g, W^k, U^k, W^o, U^o$. Sei $x = (x^{(1)}, \ldots, x^{(m)})$ eine Sequenz und $i \in \{1, \ldots, m\}$. Seien $h^{(i-1)}$ und $s^{(i-1)}$ die vorherigen Zustände (initial kann man diese als Nullvektoren definieren). Dann berechnen wir zunächst[46]

$$f^{(i)} = h^{\text{logit}}\left(U^f x^{(i)} + W^f h^{(i-1)}\right)$$

Die Idee hinter $f^{(i)}$ ist, dass dieser Vektor steuern soll, was aus dem Langzeitgedächtnis s vergessen werden soll ($f^{(i)}$ heißt auch *forget gate*). Da zur Bestimmung von $f^{(i)}$ die Aktivierungsfunktion h^{logit} benutzt wird, sind alle Komponenten von $f^{(i)}$ im Wertebereich $(0,1)$. Der Vektor $f^{(i)}$ wird als eine Art „Filter" auf den vorherigen Zustand gelegt, indem *komponentenweise* $f^{(i)}$ mit $s^{(i-1)}$ multipliziert wird:

$$\tilde{s}^{(i)} = f^{(i)} \cdot s^{(i-1)}$$

Die Vektoren $g^{(i)}$ und $k^{(i)}$ berechnen sich durch

$$g^{(i)} = h^{\text{logit}}\left(U^g x^{(i)} + W^g h^{(i-1)}\right)$$
$$k^{(i)} = h^{\tanh}\left(U^k x^{(i)} + W^k h^{(i-1)}\right)$$

Beachten Sie, dass zur Berechnung von $g^{(i)}$ und $k^{(i)}$ verschiedene Gewichtsmatrizen und Aktivierungsfunktionen benutzt werden. Der Vektor $g^{(i)}$ heißt auch *input gate* und steuert, welche Informationen aus $k^{(i)}$ (und damit aus $x^{(i)}$ und $h^{(i-1)}$) in das Langzeitgedächtnis aufgenommen werden sollen. Dies wird durch eine Addition des (komponentenweisen) Produkts von $g^{(i)}$ und $k^{(i)}$ zu $\tilde{s}^{(i)}$

[46] Wir verzichten der Einfachheit halber wieder auf eine explizite Darstellung der Bias-Neuronen.

realisiert:

$$s^{(i)} = \tilde{s}^{(i)} + \hat{s}^{(i)} \qquad (50)$$
$$\text{mit } \hat{s}^{(i)} = g^{(i)} \cdot k^{(i)}$$

Der Vektor $q^{(i)}$ heißt auch *output gate* und steuert, welche Information in die Ausgabe und den nächsten versteckten Zustand $h^{(i)}$ einfließt:

$$q^{(i)} = h^{\text{logit}}\left(U^o x^{(i)} + W^o h^{(i-1)}\right)$$

Schließlich berechnet sich $h^{(i)}$ via

$$h^{(i)} = h^{\tanh}(s^{(i)}) \cdot q^{(i)}$$

Beachten Sie wieder, dass die Anwendung der Funktion $h^{\tanh}$ und die Vektormultiplikation komponentenweise zu verstehen ist. Die Ausgabe eines LSTMs wird üblicherweise mit $h^{(i)}$ (oder einem Teil des Vektors) gleichgesetzt.

Die Kernidee hinter LSTMs liegt in der Definition des Zellzustands (50), hier noch einmal in vollständiger Form:

$$s^{(i)} = f^{(i)} \cdot s^{(i-1)} + g^{(i)} \cdot k^{(i)} \qquad (51)$$

Durch die Verwendung der Addition (anstelle einer Multiplikation) zur Hinzunahme neuer Informationen wird vermieden, dass bei langen Sequenzen weit zurückliegende Informationen immer weiter vergessen werden. Durch die Verwendung der *gates* $f^{(i)}$ und $g^{(i)}$ kann zu jedem Punkt der Sequenz gesteuert werden, wie viel Information erhalten bleibt und wie viel weitergegeben wird. Weiterhin erlaubt (51) eine robustere Weiterleitung des Gradienten bei der Backpropagation. Beispielsweise ist die Ableitung von $s^{(i)}$ nach $s^{(i-1)}$ einfach $f^{(i)}$ und wird nicht durch eine (potentiell hohe) Potenz einer Gewichtsmatrix modifiziert.

5.3.3 Weitere Architekturen und Anwendungen

Eine weitere wichtige Anwendung für RNNs ist die automatische Übersetzung. Abbildung 98 zeigt die Architektur eines einfachen RNNs, das einen Text in einer gegebenen Sprache in eine andere Sprache übersetzt (hier Übersetzung von Deutsch nach Englisch). Das RNN ist in zwei Teile geteilt, der erste Teil ist verantwortlich für das Lesen und „kodieren" des Eingabetexts in eine abstrakte Form mithilfe der Gewichtsmatrizen U und W^e. Beachten Sie, dass dieser Teil des RNNs keine Ausgabeneuronen besitzt. Über eine Schnittstelle via der Gewichtsmatrix W^i wird der innere Zustand an den zweiten Teil des RNNs geleitet, in dem mithilfe der Gewichtsmatrizen V und W^d die Ausgabe generiert („dekodiert") wird. Beachten Sie, dass der zweite Teil keinen direkten Zugriff auf die Eingabe hat. Anstelle der einfachen in Abbildung 98 dargestellten Architektur verwendet man auch LSTMs in einer ähnlich aufgebauten Weise.

Weitere wichtige Anwendungsgebiete für RNNs sind Sprach- und Schrifterkennung. Auch hier werden Architekturen wie in Abbildung 98 verwendet und zunächst die Eingabe (eine Reihe von digital repräsentierten Audioschnipseln oder Bilder von Buchstaben/Wörtern) eingelesen und im zweiten Teil entsprechend ausgegeben (beispielsweise als natürlichsprachlicher Satz). Gerade bei diesen Anwendungen ist aber auch eine Erweiterung der RNNs von besonderer Relevanz, nämlich die *bidirektionalen RNNs*. Abbildung 99 zeigt hier eine einfache Architektur eines bidirektionalen RNNs (für eine Anwendung in den zuvor genannten Bereichen würden dann die Architekturen in Abbildungen 98 und 99 miteinander kombiniert werden). Ein bidirektionales RNN verfügt über zwei versteckte Schichten h und g, die entgegengesetzt zueinander angeordnet sind. Während die

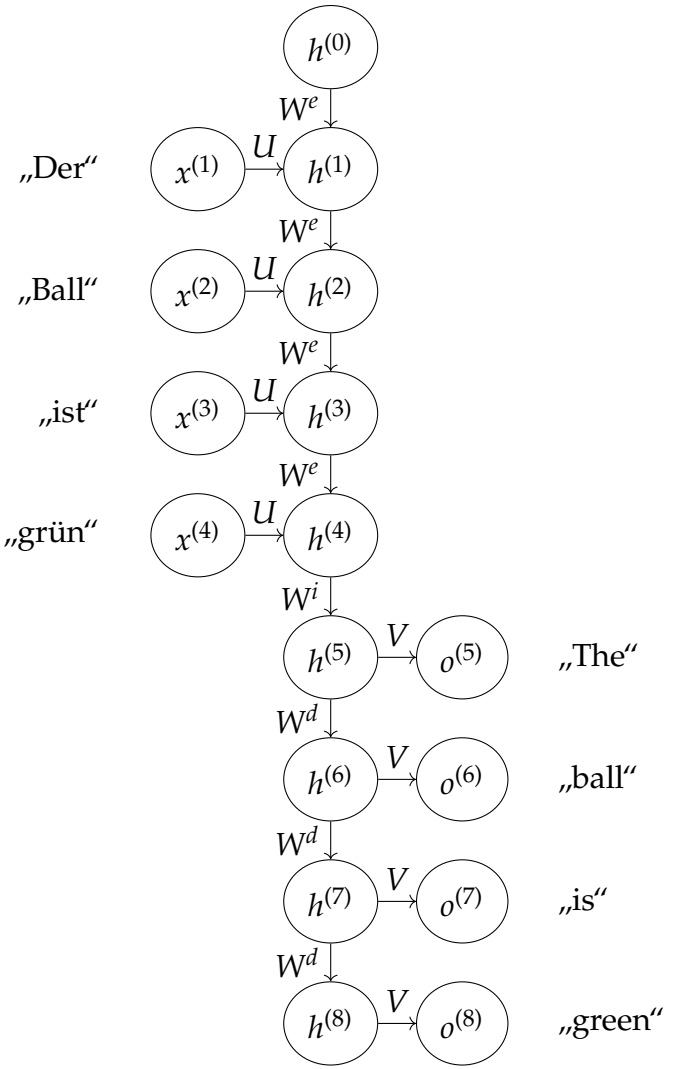


Abbildung 98: RNN für automatische Übersetzung.

Schicht h wie beim normalen RNN funktioniert, so lesen die einzelnen g-Schichten die Eingabe rückwärts und erlauben damit die Einbindung zukünftiger Informationen

in die Verarbeitung. Dies ist insbesondere dann wichtig, wenn zukünftige Informationen eine Verarbeitung vereinfachen. Bei der Handschrifterkennung ist es beispielsweise hilfreich, im Text vorauszuschauen, um zu sehen, wie beispielsweise Buchstaben „l" (klein-L) und „I" (groß-I) an anderen Stellen geschrieben werden, um diese korrekt zu erkennen. Die beiden Schichten verfügen über eigene Parametermatrizen U^f, W^f, V^f für die Vorwärtsrichtung bzw. U^b, W^b, V^b für die Rückwärtsrichtung.

Gerade bei Anwendungen im Bildbereich (wie bei der Schrifterkennung) werden RNNs auch mit CNNs (siehe Unterkapitel 5.2) kombiniert. Zwischen der Eingabe x und der versteckten Schicht h wird hier mithilfe eines CNNs zunächst die Eingabe „kodiert" und dann erst in h weiterverarbeitet. Auch bei der *automatischen Bildbeschriftung* (engl. *(automatic) image captioning*), bei der automatisch eine Beschreibung eines gegebenen Bildes generiert werden soll, findet eine Kombination von CNNs und RNNs Anwendung. Hier besteht das RNN nur aus dem zweiten Teil der in Abbildung 98 dargestellten Architektur und $h^{(5)}$ wird direkt mit der Ausgabe des CNNs gekoppelt.

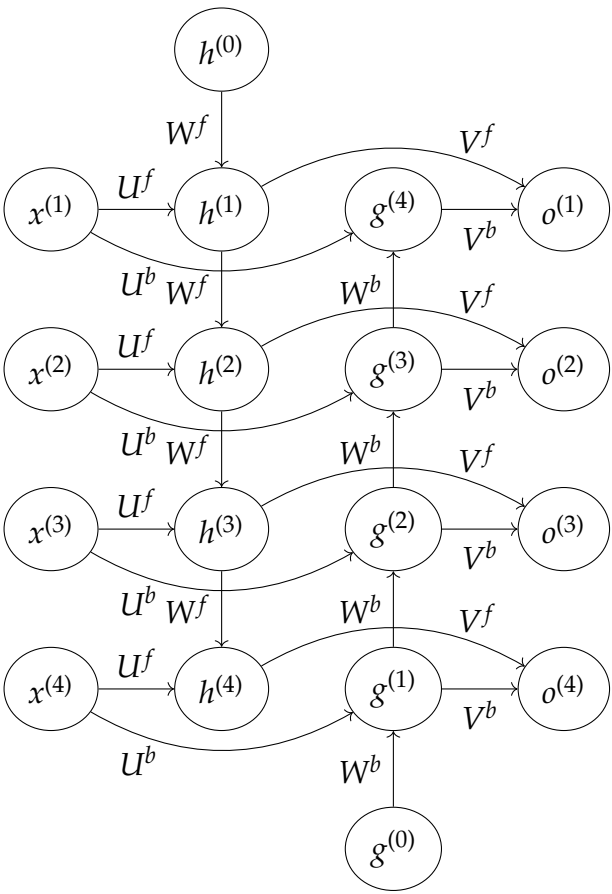


Abbildung 99: Bidirektionales RNN.

5.4 Lernen von Repräsentationen

Einer der großen Vorteile von *Deep Learning* ist die automatische Merkmalsbestimmung. Bei klassischen Ansätzen des maschinellen Lernens (siehe Kapitel 2–4) ist die Definition der Merkmale eine zentrale Herausforderung und entscheidet maßgeblich über den Erfolg eines Ansatzes für ein gegebenes Problem. Tiefe neuronale Netzwerke hingegen können direkt mit Rohdaten arbeiten, also beispielsweise Pixeldaten eines Bildes oder digitale (Audio-)Signaldaten. Die für das gegebene Problem entscheidenden Merkmale werden während des Lernens durch das Netzwerk selbst bestimmt. Dabei lernt ein tiefes Netzwerk eine kompakte Repräsentation der Verteilung der Eingabedaten, die für das jeweilige Problem ausreichend ist, um die Eingabe, zum Beispiel, entsprechend zu klassifizieren. Dies erlaubt es, moderne *Deep Learning*-Ansätze weitestgehend als *Black Box*-Ansätze anzusehen und anwenden zu können, ohne tiefes Expertenwissen im maschinellen Lernen zu besitzen. Dies bringt natürlich auch gewissen Nachteile mit sich, da *Black Box*-Ansätze im Allgemeinen schwer zu interpretieren sind. Dies gilt insbesondere für tiefe Netzwerke, die, beispielsweise, Bilddaten mit hoher Genauigkeit klassifizieren können (beispielsweise, ob ein Bild eine Katze oder einen Hund zeigt), es für den Anwender aber oft nicht nachvollziehbar ist, *warum* eine gewisse Klassifikation gegeben wird. Innerhalb des Forschungsgebiets der *Künstlichen Intelligenz* hat sich deswegen eine eigene Bewegung, die *Explainable Artificial Intelligence*-Bewegung (XAI) gebildet, die versucht, interpretier- und erklärbare tiefe Modelle für das maschinelle Lernen zu entwickeln. Wir werden uns in diesem Unterkapitel allerdings nicht mit diesem Aspekt beschäftigen, sondern uns stattdessen einige Netzwerkarchitekturen anschauen, die wei-

teren Nutzen aus der implizit gelernten Repräsentation ziehen. Dieser Unterkapitel ist zudem auch als Ausblick zu verstehen und wir werden uns die Architekturen nicht im Detail anschauen.

Wir werden im Folgenden die Architekturen von *Autoencodern* (Abschnitt 5.4.1) und *Generative Adversarial Networks* (Abschnitt 5.4.2) diskutieren.

5.4.1 Autoencoder

Ein *Autoencoder* ist ein *Feedforward*-Netzwerk, das darauf trainiert wird, die Eingabe zur Ausgabe zu kopieren. Es besteht aus drei Komponenten: dem Kodierer (engl. *encoder*), dem Dekodierer (engl. *decoder*) und dem Flaschenhals (engl. *bottleneck*). Der Kodierer und der Dekodierer sind (potentiell mehrschichtige) neuronale Netzwerke, die üblicherweise antisymmetrisch aufgebaut sind (die Anzahl der Eingabeneuronen n des Kodierers entsprechen der Anzahl der Ausgabeneuronen des Dekodierers und andersherum). Der Flaschenhals besteht aus einer Neuronenschicht, deren Anzahl $\hat{n}$ üblicherweise geringer ist als die Anzahl der Eingabeneuronen des Kodierers. Eine Darstellung eines Autoencoders ist in Abbildung 100 zu finden. Der Kodierer realisiert damit eine Funktion $f : \mathbb{R}^n \to \mathbb{R}^{\hat{n}}$, wohingegen der Dekodierer eine Funktion $g : \mathbb{R}^{\hat{n}} \to \mathbb{R}^n$ realisiert. Gegeben ein Datensatz $D = \{x^{(1)}, \ldots, x^{(m)}\}$ mit $x^{(i)} \in \mathbb{R}^n$, wird ein Autoencoder darauf trainiert[47], dass $g \circ f$ die Identitätsfunktion realisiert, d. h., die Kostenfunktion L^{auto} ist definiert als der quadratische Fehler zwischen dem Abstand von $x \in D$ und

[47] Das tatsächliche Training erfolgt wie bei normalen *Feedforward*-Netzwerken durch Backpropagation und Optimierung durch beispielsweise Stochastic Gradient Descent.

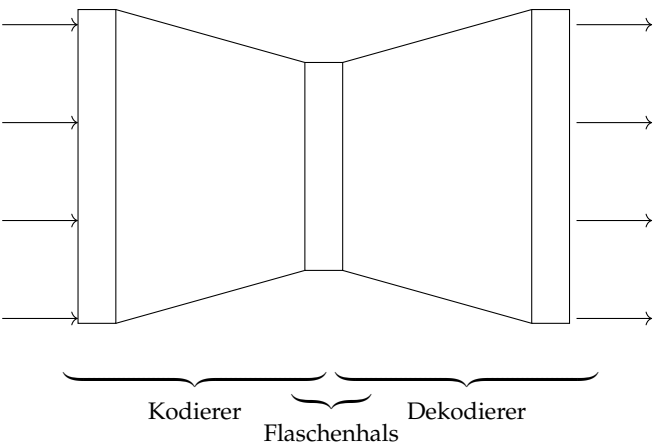


Abbildung 100: Allgemeine Architektur eines Autoencoders.

$(g \circ f)(x)$:

$$L^{\text{auto}}(D, g \circ f) = \sum_{i=1}^{m} \|x^{(i)} - (g \circ f)(x^{(i)})\|^2 \tag{52}$$

Für den Fall $n = \hat{n}$ minimieren f^{id} und g^{id}, definiert als $f^{\text{id}}(x) = g^{\text{id}}(x) = x$, die Kostenfunktion L^{auto} trivialerweise. Ist $\hat{n}$ kleiner als n, so müssen Kodierer und Dekodierer darauf trainiert werden, eine Eingabe x so zu komprimieren, dass sie bestmöglich wieder dekomprimiert werden kann. Ähnlich wie die Hauptkomponentenanalyse (siehe Unterkapitel 3.5) kann ein trainierter Autoencoder also dafür genutzt werden Daten zu komprimieren. Für die eigentliche Anwendung werden dann Kodierer und Dekodierer (mit ihren gelernten Gewichten) voneinander getrennt. Der Kodierer kann dann dazu genutzt werden, Daten zu komprimieren und der Dekodierer kann komprimierte Daten wieder dekomprimieren. Die Werte des

Flaschenhalses $f(x)$ zu einer Eingabe x nennt man auch den *Code* von x und für $\hat{n} < n$ heißt der entsprechende Autoencoder *undercomplete*. Sind alle Aktivierungsfunktionen linear, so kann gezeigt werden, dass der optimal trainierte Autoencoder tatsächlich auch die gleiche Kompressionstruktur erzeugt wie PCA. Werden nicht-lineare Aktivierungsfunktionen genutzt, so können komplexere Merkmale genutzt werden, um die Daten noch effektiver zu komprimieren.

Ein Autoencoder mit $\hat{n} > n$ heißt *overcomplete* und kann auch sinnvoll sein, wenn ein Regularisierungsterm in die Kostenfunktion (52) integriert wird. Ein *sparse* Autoencoder (SAE) ist *overcomplete* und wird trainiert durch Minimierung der Kostenfunktion L^{sae}, definiert als

$$L^{\text{sae}}(D, g \circ f) = L^{\text{auto}}(D, g \circ f) + \lambda \sum_{i=1}^{m} \|f(x^{(i)})\|$$

wobei λ der Regularisierungsparameter ist. Auch wenn bei $\hat{n} > n$ die Funktionen f^{id} und g^{id} den Term $L^{\text{auto}}(D, g^{\text{id}} \circ f^{\text{id}})$ minimieren, so kann es andere Repräsentation $f(x)$ für $x \in D$ geben, die den gesamten Term $L^{\text{sae}}(D, g \circ f)$ minimieren. Wie der Name *sparse* Autoencoder suggeriert, lernt dieser Repräsentationen, die im Flaschenhals nur spärlich besetzt sind, d. h., viele Komponenten in $f(x)$ sind Null oder nahe Null. Eine Variante des SAEs ist der *denoising* Autoencoder (DAE). Bei diesem wird zunächst zu einem Trainingsdatensatz $D = \{x^{(1)}, \ldots, x^{(m)}\}$ ein Rauschen hinzufügt, was zu einem neuen Trainingsdatensatz $\hat{D} = \{\hat{x}^{(1)}, \ldots, \hat{x}^{(m)}\}$ führt. Hierbei wird $\hat{x}^{(i)}$ aus $x^{(i)}$ gebildet, indem beispielsweise jede Komponente um einen kleinen Gauß-verteilten additiven Term modifiziert wird. Ein

DAE wird dann durch Minimierung der Kostenfunktion

$$L^{\text{dae}}(D, g \circ f) = \sum_{i=1}^{m} \|x^{(i)} - (g \circ f)(\hat{x}^{(i)})\|^2$$

trainiert. Der DAE erhält also als Eingabe die verrauschten Daten $\hat{x}$ und muss das Originaldatum x rekonstruieren. Dies führt ähnlich wie beim SAE dazu, dass der Code $f(x)$ eine nicht-triviale Repräsentation der Verteilung in D darstellt und für Aufgaben wie Klassifikation relevante Merkmale gelernt wurden.

Eine weitere wichtige Anwendung von Autoencodern ist die Datengenerierung. Der Dekodierer eines trainierten Autoencoders kann dazu genutzt werden, neue Daten, die der Verteilung des Trainingsdatensatzes folgen, zu generieren. Dazu wird dem Dekodierer eine zufällig generierte Eingabe eingegeben und die Ausgabe entspricht dann einem neuen Datum. Besondere Anwendung findet dies beispielsweise bei der künstlichen Bildgenerierung, bei der beispielsweise ein Autoencoder auf Bilder eines gewissen Künstlers trainiert wird, und neue Bilder durch den Dekodierer generiert werden können, die dem Stil des Künstlers entsprechen.

5.4.2 Generative Adversarial Networks

Eine weitere Architektur, die für die Aufgabe der (Bild-)Datengenerierung noch mehr Erfolg in jüngsten Jahren gezeigt hat, ist die der *Generative Adversarial Networks* (GANs). Ein GAN besteht aus zwei separaten *Feedforward*-Netzwerken: dem *Generator* und dem *Diskriminator*. Der Generator ist ein neuronales Netzwerk mit ähnlicher Struktur wie der Dekodierer bei Autoencodern. Als Eingabe nimmt er einen relativ niedrigdimensionalen Vektor zufällig generierter Zahlen entgegen (üblicherweise ist jede Komponente Gauß-verteilt)

und als Ausgabe produziert er ein synthetisches Beispiel (etwa ein Bild). Der Diskriminator ist ein neuronales Netzwerk, das als Eingabe ein Beispiel erhält und entscheiden soll, ob das Beispiel echt ist (d. h., aus dem gegebenen Trainingsdatensatz stammt) oder vom Generator erzeugt wurde. Diese beiden Netzwerke werden simultan trainiert und dies führt im besten Fall dazu, dass die final vom Generator erzeugten Beispiele nicht mehr vom Diskriminator von echten Bildern aus dem Trainingsdatensatz unterschieden werden können. Im Folgenden schauen wir uns den Lernprozess bei GANs etwas genauer an.

Sei $D = \{x^{(1)}, \ldots, x^{(m)}\}$ ein Datensatz (beispielsweise von Bildern) mit $x^{(i)} \in \mathbb{R}^n$, $i = 1, \ldots, m$. Der Generator realisiert eine Funktion $\Gamma : \mathbb{R}^k \to \mathbb{R}^n$, d. h., die Ausgabe von Γ ist von derselben Dimension wie die Daten in D. Üblicherweise ist k relativ klein (beispielsweise $k = 100$). Der Diskriminator realisiert eine Funktion $\Delta : \mathbb{R}^n \to [0,1]$, d. h., Δ nimmt als Eingabe ein Beispiel x aus D oder ein von Γ erzeugtes Beispiel und gibt als Ausgabe die Wahrscheinlichkeit $\Delta(x)$, dass x ein echtes Beispiel ist (also $\Delta(x) = 1$ gdw. $x \in D$ und $\Delta(x) = 0$ gdw. $x = \Gamma(p)$ für ein beliebiges $p \in \mathbb{R}^k$). Das Lernen der (hier nicht explizit genannten Gewichte) in den Funktionen Γ und Δ geschieht abwechselnd auf wenigen Beispielen. Sei $R \subseteq D$ eine zufällig ausgewählte Teilmenge mit $|R| = m'$ (üblicherweise $m' < 5$) aus echten Beispielen und S eine gleich große Menge an generierten Beispielen, d. h., $S = \{\Gamma(p_1), \ldots, \Gamma(p_{m'})\}$ und $p_1, \ldots, p_{m'}$ sind zufällig normalverteilt. Die Funktion L_Δ definiert durch

$$L_\Delta(R,S) = -\sum_{x \in R} \log(\Delta(x)) - \sum_{x \in S} \log(1 - \Delta(x))$$

gilt es zu *minimieren*: der Ausdruck $L_\Delta(R,S)$ ist nämlich genau dann minimal, wenn alle echten Beispiele als echt

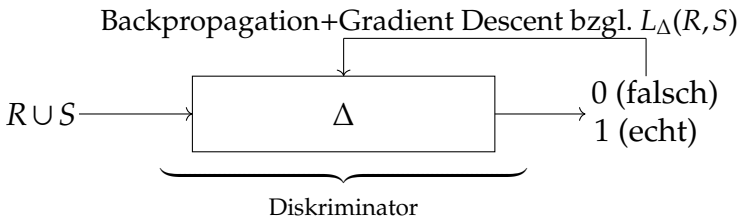


Abbildung 101: Ein Schritt im Lernprozess des Diskriminators bei GANs.

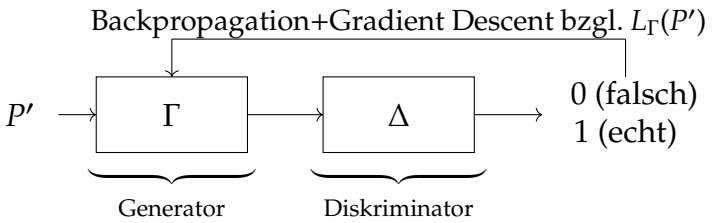


Abbildung 102: Ein Schritt im Lernprozess des Generators bei GANs.

erkannt werden ($\Delta(x) = 1$ für alle $x \in R$) und alle generierten Beispiele als falsch erkannt werden ($\Delta(x) = 0$ für alle $x \in S$). Das eigentliche Lernen wird realisiert durch Backpropagation und Gradient Descent auf der Kostenfunktion $L_\Delta(R,S)$. Abbildung 101 veranschaulicht diesen Lernschritt für den Diskriminator. Nach der einmaligen Aktualisierung von Δ wird im nächsten Lernschritt der Generator aktualisiert. Das Ziel des Generators ist es, den Diskriminator zu täuschen. Dazu generieren wir eine neue Menge $S' = \{\Gamma(p'_1), \ldots, \Gamma(p'_{m'})\}$ und *minimieren* die Funktion $L_\Gamma(S')$, definiert durch

$$L_\Gamma(S') = \sum_{x \in S'} \log(1 - \Delta(x))$$

Falls der Diskriminator durch die generierten Bilder getäuscht wird, d. h., $\Delta(x) = 1$ für alle $x \in S'$, so ist $L_\Gamma(S')$ minimal. Setzen wir $P' = \{p'_1, \ldots, p'_{m'}\}$ so schreiben wir L_Γ auch als

$$L_\Gamma(P') = \sum_{p' \in P'} \log(1 - \Delta(\Gamma(p')))$$

Mit anderen Worten, wir verbinden für diesen zweiten Lernschritt den Diskriminator mit dem Generator und verwenden Backpropagation und Gradient Descent, um die Kostenfunktion $L_\Gamma(P')$ zu minimieren. Abbildung 102 veranschaulicht diesen Lernschritt für den Generator. Zu beachten ist hier, dass auch wenn in diesem Schritt Backpropagation durch den Diskriminator stattfindet, nur die Gewichte im Generator aktualisiert werden. Die beiden oben beschriebenen Aktualisierungsschritte von Diskriminator und Generator werden abwechselnd durchgeführt, bis eine Konvergenz festgestellt werden kann.

6 Zusammenfassung und Ausblick

Dieses Kapitel schließt das Buch „Maschinelles Lernen“ mit einer kurzen Zusammenfassung und einem Ausblick ab.

6.1 Zusammenfassung

Dieses Buch behandelte nach einer kurzen Einführung die Grundkonzepte des maschinellen Lernens und diskutierte insbesondere überwachtes Lernen, unüberwachtes Lernen, *Reinforcement Learning* und schließlich *Deep Learning*.

In Kapitel 2 „Überwachtes Lernen“ haben wir uns einerseits mit einer Reihe klassischer Ansätze für eben dieses, als auch mit allgemeinen Grundkonzepten des maschinellen Lernens beschäftigt. Zu letzteren gehören neben allgemeinen Begriffen wie „Trainings-“ und „Testdatensatz“ auch die Themen der Über- und Unteranpassung, sowie die Methodik der Regularisierung. Mit der linearen Regression haben uns weiterhin mit einem elementaren Ansatz für das Lösen des Regressionsproblems, also der Vorhersage von funktional abhängigen Werten, beschäftigt. Mit der logistischen Regression haben wir uns eine ähnliche Methodik für das Lösen des Klassifikationsproblems, also der Vorhersage von (diskreten) Klassen, angeschaut. Auch *Support Vector Machines* adressieren das Problem der Klassifikation und sind im Gegensatz zur logistischen Regression in der Lage, durch Auswahl entsprechender Kernelfunktionen komplexere Entscheidungsgrenzen zu modellieren. Weiterhin haben wir mit der Nächste-Nachbarn-Klassifikation einen konzeptuell sehr einfachen Ansatz zur Klassifikation diskutiert, der kein Training im eigentlichen Sinne benötigt, aber laufzeittechnisch relativ anspruchsvoll ist

und komplexe Entscheidungsgrenzen nicht gut modellieren kann. Diesen Ansatz kann man allerdings auch einfach auf Regressionsprobleme anwenden. Im Zuge der Nächste-Nachbarn-Klassifikation/Regression haben wir uns auch mit dem allgemeinen Thema der Merkmalskalierung beschäftigt, die insbesondere für die Nächste-Nachbarn-Klassifikation/Regression von hoher Bedeutung ist. Mit dem (naiven) Bayes-Klassifikator haben wir uns mit einem weiteren konzeptuell recht einfachen Ansatz beschäftigt, der allerdings auf einer wahrscheinlichkeitstheoretischen formalen Grundlage basiert und vergleichsweise auch bei komplexeren Problemen gute Resultate liefert. Schließlich haben wir Entscheidungsbäume und insbesondere die Konstruktion von Entscheidungsbäumen durch den ID3- und den C4.5-Algorithmus diskutiert. Entscheidungsbäume liefern eine visuell sehr attraktive Darstellungsform des gelernten Modells und sind damit (im Vergleich zu vielen anderen Ansätzen des maschinellen Lernens) leicht interpretier- und nachvollziehbar.

Kapitel 3 behandelte „Unüberwachtes Lernen“, dessen wichtigstes Problem, die Clusteranalyse, durch die Vorstellung der Ansätze des *K-Means-Clusterings* und des *Hierarchical Clusterings* diskutiert wurde. *K-Means-Clustering* basiert auf einer iterativen Methodik zum Finden von „Zentroiden“, die die Mittelpunkte der zu bestimmenden Cluster definieren. *Hierarchical Clustering* unterscheidet zwischen agglomerativen und divisiven Ansätzen und ist prinzipiell parameterfrei bezüglich der Clusterzahl. Diese Ansätze konstruieren ein Dendrogramm, aus dem verschiedene konkrete Clusterings bei gegebener Zahl extrahiert werden können. Mit dem Single-Link-Clustering und dem DIANA-Ansatz haben wir uns Vertreter von agglomerativen bzw. divisen Ansätzen angeschaut. Ein weiteres Problem des

unüberwachten Lernens ist das Lernen von Assoziationsregeln, also Regeln, die Muster in den Daten modellieren. Mit dem Apriori- und dem FPGrowth-Algorithmus haben wir uns hier zwei Algorithmen zur Extraktion von Assoziationsregeln angeschaut. Ein weiteres Problem des unüberwachten Lernens ist die Anomalieerkennung, die wir kurz durch das Verfahren der Dichteabschätzung einer Normalverteilung kennengelernt haben. Schließlich haben wir uns mit der Hauptkomponentenanalyse eine Methode zur Dimensionsreduktion eines Datensatzes angeschaut.

Kapitel 4 führte in die Grundlagen des *Reinforcement Learnings* ein. Im Gegensatz zum überwachten und unüberwachten Lernen wird beim *Reinforcement Learning* nicht aus einem gegebenen Datensatz gelernt. Vielmehr wird der Lerner in eine Umgebung eingebettet und lernt aus seinen eigenen Aktionen und den daraus resultierenden Beobachtungen. Wir haben uns dazu zunächst mit den Grundlagen zu Markov-Entscheidungsprozessen beschäftigt, die eine einfache Repräsentation einer Umgebung modellieren. Insbesondere haben wir uns Methoden angeschaut, die bei vollständiger Kenntnis des Markov-Entscheidungsprozesses die Nutzen der Zustände und die optimale Strategie bestimmen können (sogenannte Offline-Verfahren). Anschließend haben wir Online-Verfahren diskutiert, also Verfahren, die initial keine Kenntnis über die Dynamik der Umgebung haben. Hier haben wir zunächst Verfahren zum Lernen der Zustandsnutzen (passive Verfahren) kennengelernt, wie etwa adaptive dynamische Programmierung und *Temporal Difference Learning*. Schließlich haben wir uns mit „aktiven" Verfahren zum *Reinforcement Learning* beschäftigt, die die optimale Strategie in unbekannten Umgebungen lernen. Ein wichtiges Problem in diesem Kontext war das *exploration vs. exploitation*-Dilemma, das wir durch das all-

gemeine *epsilon-greedy Learning* behandelt haben. Kombiniert mit dem *Q-Learning* haben wir damit einen prototypischen Vertreter für das aktive *Reinforcement Learning* diskutiert.

Kapitel 5 behandelte die Familie der *Deep Learning*-Ansätze zum maschinellen Lernen, die grundsätzlich für Probleme des überwachten und unüberwachten Lernens, sowie des *Reinforcement Learnings* anwendbar sind (wobei wir uns auf die Anwendung im überwachten Kontext fokussiert haben). Diese Ansätze beruhen auf dem Modell der künstlichen neuronalen Netzwerke, deren Architektur und Bausteine wir uns zunächst detailliert angeschaut haben. Wegen der großen Parameterzahl neuronaler Netzwerke ist der Lernprozess hier sehr anspruchsvoll und bedarf spezieller Methoden. Der Backpropagation-Algorithmus bietet hier eine elegante Lösung zur Berechnung der für die Optimierung nötigen partiellen Ableitungen. Anschließend haben wir uns einige spezielle Architekturen von künstlichen neuronalen Netzwerken angeschaut. *Convolutional Neural Networks* sind auf die Verarbeitung von Bilddaten spezialisiert und nutzen *Faltungsoperationen* mit geteilten Parametern, um ein uniforme Behandlung verschiedener Bildregionen zu realisieren. *Recurrent Neural Networks* sind auf die Verarbeitung von Sequenzdaten wie natürlichsprachliche Sätze ausgelegt. Sie beinhalten Komponenten für ein „Gedächtnis" und sind rekursiv definiert, um Elemente an verschiedenen Positionen einer Eingabesequenz in gleicher Weise zu verarbeiten. Schließlich haben wir uns mit Autoencodern und *Generative Adversarial Networks* Ansätze zum Lernen von Repräsentationen angeschaut.

6.2 Ausblick

Das Buch „Maschinelles Lernen“ bietet einen breiten Einstieg in das Thema des maschinellen Lernens, aber kann natürlich nicht alle Aspekte und Ansätze beleuchten. Weiterhin fokussiert dieses Buch auch auf die praktischen Aspekte des maschinellen Lernens. Theoretische Aspekte, insbesondere zum Thema der algorithmischen Lerntheorie, wurden hier nur wenig diskutiert.[48] Gerade in Kapitel 4 und 5 haben wir uns auf eine Darstellung der Grundlagen zu *Reinforcement Learning* und *Deep Learning* beschränkt. Einen tieferen Einstieg zu *Reinforcement Learning* bietet [14]. Eine Kombination von *Reinforcement Learning* und *Deep Learning* ist *Deep Reinforcement Learning*. Der *Deep Q-Learning*-Ansatz [9, 10] benutzt *Convolutional Neural Networks* zur Repräsentation der *Q*-Funktion und ist dadurch signifikant skalierbarer als normales *Q-Learning*. Der Bereich des *Deep Learnings* hat in den letzten Jahren sehr viele neue und vielversprechende Ansätze hervorgebracht. Zu nennen ist hier beispielsweise das Transformermodell [16], das insbesondere im Bereich der Sprachverarbeitung Modelle basierend auf *Recurrent Neural Networks* abgelöst hat. Aktuelle Systeme basierend auf dem Transformermodell wie ChatGPT[49] zeigen hier eindrucksvoll die Möglichkeiten dieser Technologie.

[48] Einen guten Einstieg zur algorithmischen Lerntheorie bietet beispielsweise Kapitel 5 in [8].

[49] `http://chat.openai.com/chat`

Literatur

[1] Charu C. Aggarwal. *Neural Networks and Deep Learning*. Springer International Publishing AG, 2018.

[2] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'94)*, pages 487–499, 1994.

[3] Christoph Beierle and Gabriele Kern-Isberner. *Methoden wissensbasierter Systeme: Grundlagen, Algorithmen, Anwendungen*. Springer Vieweg, 6. edition, 2019.

[4] Andriy Burkov. *The Hundred-Page Machine Learning Book*. Andriy Burkov, 2019.

[5] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[6] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. *ACM SIGMOD Record*, 29(2):1–12, 2000.

[7] Leonard Kaufman and Peter J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley & Sons, Inc., 2005.

[8] Tom M. Mitchell. *Machine Learning*. McGraw-Hill Science, 1997.

[9] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[10] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[11] Andrew Ng. Supervised machine learning: Regression and classification, 2022. `https://www.coursera.org/learn/machine-learning`.

[12] Andrew Ng. Unsupervised learning, recommenders, reinforcement learning, 2022. `https://www.coursera.org/learn/unsupervised-learning-recommenders-reinforcement-learning`.

[13] Stuart Russel and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, fourth edition, 2020.

[14] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 2018.

[15] Sergios Theodoridis and Konstantinos Koutroumbas. *Pattern Recognition*. Elsevier Inc., 2009.

[16] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems*

2017, December 4-9, 2017, Long Beach, CA, USA, pages 5998–6008, 2017.

Stichwortverzeichnis